Three-Step Intelligent Pruning for Data Classification in Just-in-Time Software Defect Prediction

Nan Luo [1], Ying Ma [1]

[1] *Department of Computer and Information Engineering, Xiamen University of Technology, Xiamen, China*

Abstract

Just-in-time software defect prediction technology is a defect prediction method that enables defect prediction of software change levels. The difficulties of learning classifiers from imbalanced data is demonstrated in a variety of real-world applications,especially in this era of big data, which has generated more classification tasks. Researchers have taken many existing JIT-SDP efforts to assume that the features of software releases remain costant over time. However, the researchers did not consider that JIT-SDP may be affected by the gradual evolution of class imbalance. Specifically, class imbalance (that is, the number of changes caused by defects is not adequately represented) has been changing over time, and the number of clean class changes and defect class changes may both increase or decrease, so here In this case, the existing JIT-SDP method becomes inapplicable. Taking these factors into consideration, we propose a new imbalanced classification framework, which aims to achieve data class balance by applying a new three-step smart pruning strategy, i.e., first undersampling the majority class, then undersampling the minority class. Oversampling is performed, since the minority class becomes the majority class after oversampling, as a result, the final stage is to intelligently undersample the minority group that eventually becomes the dominant group. Through these three steps, data balance is achieved before classification. Experiments show that this new framework is very computationally efficient, leading to better performance even under highly imbalanced distributions of clean and defective data. At the same time, our proposed framework can also be easily adapted to most existing learning methods to improve their performance on imbalanced data.

Keywords

Machine Learning, JIT-SDP, Class Imbalance, Artifical Intelligence

1. Introduction

It is well known that reducing the number of software defects is a challenging problem, and the process of software debugging requires high labor and material costs, especially when testing resources are limited and software teams are often under intense pressure to deliver quickly. Therefore, researchers have come up with many machine learning methods to predict if there are any flaws in the source code of software, these machine learning methods can allocate more attention to software components that may contain defects by rationally distributing testing and inspection efforts. Just-in-Time (JIT) SDP is a special type of SDP method that, as soon as a software change occurs, identifies the change that caused the defect (ie Just-in-Time).

In the current big data environment, most classifiers and learning techniques cannot handle the issue of class imbalance well. Therefore, the issue of class imbalance is also an important factor to be considered in instant defect prediction research. Among the traditional methods of dealing with imbalanced data, several common algorithms include upsampling oversample for the minority class and downsampling undersample for the majority class[1], and artificially synthesized minority class

ISCIPT2022@7th International Conference on Computer and Information Processing Technology, August 5-7, 2022, Shenyang, China
EMAIL: luonan@stu.xmut.edu.cn.com (Nan Luo); maying@xmut.edu.cn (Ying Ma)

© 2022 Copyright for this paper by its authors.
Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).
CEUR Workshop Proceedings (CEUR-WS.org)

CEUR-WS.org/Vol-3206/paper19.pdf

proposed by Chawla et al.[2]Oversampling (SMOTE). Borderline SMOTE[3], ADASYN[4] (Adaptive Synthetic Sampling), and Majority Weighted Minority Oversampling (MWMOTE)[5] are some popular smart sampling strategies.

A technique is used in our job,SSO-SMOTE-SSO is applied instead of oversampling rate boosting (ORB). The purpose of intelligently pruning training data is achieved by combining undersampling and oversampling in layers. The SSO algorithm is in charge of intelligent undersampling of majority class data (expressed in the first and third steps of the algorithm), while the SMOTE algorithm is in charge of minority class oversampling. Because it prunes both the majority and minority classes and keeps only sample information that is useful for the classification task, such a sequential combination provides an efficient solution to the class imbalance problem in the instant software defect prediction task. Our paper is organized as follows: Section II contains a literature survey related to related work, Section III has the precise procedures for the suggested strategy for dealing with imbalanced data, Section IV analyzes the experimental setup and result analysis, and the last section presents general conclusions are drawn.

2. Related work

In this section, we first briefly introduce the meaning and common evaluation metrics of instant defect prediction, then introduce class imbalance learning and validation delay in defect prediction, and finally introduce the methods involved in machine learning to solve class imbalance.

2.1. JIT-SDP

The software defect prediction technology mainly includes module-level, file-level and change-level defect prediction according to different prediction granularities. The change-level defect prediction aims at predicting whether the introduced code has defects when the developer submits the code. , so it is also called just-in-time defect prediction. Kim et al.[6] were the first to investigate JIT-SDP. They classified changes into clean and defective changes based on software change features such as adding and removing deltas, directory/file names, metrics complexity, and so on. Several other research has looked into the features of software changes that lead to defects and the underlying metrics (i.e., input characteristics) used to predict them, Shihab et al.[7] investigated dangerous(defect-causing) changes, including the day of the week[8] and time of day[9] when the change was committed. Lines of code had been introdduced, and flaws were touching files, they discovered (i.e. ratio of bug fixes to total changes touching files), number of bug reports associated with commits, and developer experience were the top indicators of risky changes. Kamei et al.[10] conducted one of the largest JIT-SDP studies. They used a number of factors gleaned from commits and bug reports, which are thought to be good markers of software modifications that result in problems. They demonstrated that the indicators they utilised in their research were highly predictive for both open source and commercial applications. As a result, we employ the same measures in this study. The general just-in-time software defect prediction model is shown in Figure 1.

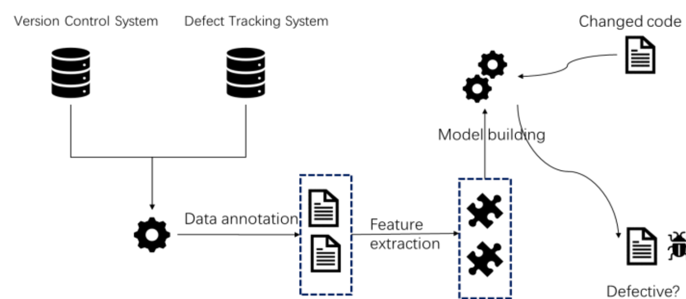

Figure 1: Just-in-time defect prediction frame-work in general

2.2. Verification Latency in JIT-in-SDP

The fact that the labels of training samples may come later than their input features is referred to as validation latency. Tan et al.[11] discovered that neglecting validation delays results in unduly optimistic predictions of prediction performance, so they propose storing additional batches of training data over time and using all batches received to develop a JIT -SDP classifier. After a predetermind waiting time has elapsed, training examples are only available for building fresh batches. Their research did not look into how long it takes to identify problems in the real world, and their proposed solution assumes no change in class imbalance. Unlike their work, this study explores the impact of class imbalance evolution on the JIT-SDP classifier's prediction performance over time, provides techniques to better handle class imbalance evolution, and investigates how long software changes normally take to be identified as generating the defect class.

2.3. SDP Class Imbalance Learning

Class imbalance refers to the fact that the number of instances from different classes is not the same, or even varies greatly. This is a common occurrence in a variety of real-world applications, such as in fraud detection (normal vs. fraudulent), medicine (healthy vs. sick), software changes (clean vs. defective). Mahmood et al.[12] showed that as the data became more imbalanced, the predictive performance of the SDP classifier (according to the Mathews correlation coefficient) became worse; Wang and Yao[13] did not Balanced learning techniques have been comprehensively studied, including resampling, threshold shifting, and ensemble; Bennin et al.[14] presented a synthetic oversampling approach based on genetic chromosome theory. Kamei et al.[15] studied the application of four resampling strategies for fault-prone module detection. However, these methods adopt a fixed resampling rate and consider the imbalance rate to be fixed over time, i.e., there is no need to contemplate the growth of a class imbalance. Specifically, rather of allowing the resampling rate to dynamically adjust to the current level of imbalance in the data, their parameter tuning procedure locks the resampling rate utilised across the dataset to a single value. Uneven distribution of data brings great difficulty to applying canonical learning algorithms on unbalanced data only. Although such problems have been extensively studied, the existing models' performance still needs to be enhanced.

2.4. Machine Learning to Tackle Cla-ss Imbalance Evolution

To cope with class imbalance evolution, Wang et al.[16] suggested two online class imbalance learning methods: enhanced undersampling online bagging UOB (Undersampling Online Bagging) and improved oversampling online bagging OOB (Oversampling Online Bagging) (Oversampling Online Bagging). These approaches keep track of the present rate of imbalance, i.e. the ratio of examples $\rho_c^{(t)}$belonging to each class $c \in \{0,1\}$ as follows:

$$\rho_c^{(t)} = \theta' \rho_c^{(t-1)} + (1-\theta')(y^{(t)} == c), \tag{1}$$

where t represents the current time step; each time step corresponds to the algorithm being presented with a new training example;$(y^{(t)} == c)$ represents if the training sample at time t is class c, it returns 1, otherwise it returns 0; $0 \ll \theta' \leq 1$, θ' is a predefined parameter, which is emphasized for adjusting the latest data. A smaller θ' is used for the current data, $\rho_c^{(t)}$ can reflect the change of the imbalance rate faster, but noise may have a greater impact. Tracking the evolution of class imbalances entails tracking (but not yet resolving) variations in imbalance rates. For the first time, this work investigates the class imbalance evolution learning method under the condition of JIT-SDP, based on UOB and OOB.

3. Proposed Method

In this section, SSO and SMOTE are the essential components of our suggested hybrid SSOMaj-SMOTE-SSOMin. We present information regarding SSO, SMOTE, and the proposed variant three-

step sampling approach, as well as the related pseudo-code introduction.

3.1. Verification Latency Learning C-lassification Framework

Because we have no way of knowing whether a new software change will produce a bug at the moment it is submitted, we consider that within Ω (wait time) days after the change submission, once the change is found to cause a defect, the change will be marked as causing a defect Defective class changes that would otherwise be marked as clean class changes. This waiting time Ω can be set by the software administrator. After many experiments, it is found that it is more appropriate to set the waiting time value to 90 days. This framework can also be applied to other classifiers.

3.2. A Three-step intelligent prun-ing strategy:SSO-SMOTE-SSO

To better address the problem of class imbalance, we use a three-step smart pruning technique to replace the ORB[17] algorithm. Inspired by the work on oversampling and undersampling methods for dealing with imbalanced classes, we try to stack several sampling methods in steps, i.e. perform a smart pruning process for imbalanced classes through specific consecutive three processes. : 1. First, use the sample subspace optimization algorithm (SSO) to undersample the majority class. SSO is a strategy for locating the most representative majority class samples through intelligent majority class undersampling, and then use these samples with the minority class. Class combination to provide distinguishing information between the two; 2. Oversample the minority class using the SMOTE algorithm. SMOTE's core strategy entails analysing minority class samples and artificially synthesising new samples based on the minority class samples, which are then added to the data set; 3. The SSO algorithm is used again to undersample the minority class after oversampling, so this strategy is called SSO-SMOTE-SSO. Figures 2 to 4 summarize the pseudocode of the three algorithms involed.

Algorithm 1: SampleSubsetOptimization preprocessing procedure.
Input: Original Imbalanced training dataset D_I Output: Optimized sample subsets D_{major}
1 Step 1 Assign a dimension in particle space to each sample in the majority class 2 Step 2 Iterate over each particle and extract samples based on the indicator function 3 Step 3 Use the selected majority class samples and all minority class samples to train the classifier 4 Step 4 Use the test samples to calculate the fitness of the trained classifier 5 Step 5 Update particle velocity and position based on fitness value End for

Figure 2: Pseudo-code for SSO

Algorithm 2: SyntheticMinorityOversamplingTechnique preprocessing procedure.
Input: Original Imbalanced training dataset D_I Output: Balanced dataset D_B
1 Step 1 Find the K nearest neighbors of a sample Xi in the minority class 2 Step 2 Randomly select a neighbor $x_i(n,n)$ of x_i, and randomly generate a number between 0 and 1 to synthesize a new sample 3 Step 3 Repeat step 2 for each randomly selected neighbor of Xi 4 Step 4 Repeat the above steps for each sample of the minority class RETURN Balanced dataset D_B

Figure 3: Pseudo-code for SMOTE

Algorithm 3: SSO-SMOTE-SSO's training procedure.
Input: Original Imbalanced training dataset *I* Output: Area Under Curve (AUC) by classification with Balanced dataset *B*
1 Step 1 Undersampling the majority class with SSO method 2 Step 2 Minority class oversampling with SMOTE method 3 Step 3 Use the SSO method to undersample the data processed in step 2 4 OPT_TRAIN2←10-fold PSO (OVERSAMP_TRAIN) with J.48 5 AUC=CLASSIFIER (OPT_TRAIN 2, VALID) RETURN AUC

Figure 4: Pseudo-code for SSO-SMOTE-SSO

4. Experiments & Analysis

To analyse the performance of just-in-time (JIT) models, we employ two well-known software projects, QT and OPENSTACK. Developed by The Qt Company, Qt is a cross-platform application framework that allows individual developers and organizations to contribute. OPENSTACK is an open-source cloud computing software platform that is delivered as an infrastructure-as-a-service, giving clients access to their resources. To obtain software changes that cause defect classes, we use Commit Guru[18], a tool that evaluates and delivers change-level analysis, which provides change-level indicators: (1) the size of the change; (2) what was the file changed; (3) the proliferation of changes; (4) the developers' experience in making the adjustments; (5) the reason for the changes. The datasets used in our work are simply summarised in Table I. Mc Intosh and Kamei [19] originally gathered and curated this dataset. After final processing, Table 1 shows the relevant information from the two project datasets, with the QT dataset having 23,912 commits and the OPENSTACK dataset having 22,757 commits.

Table 1

Information of the dataset used in this work

Dataset	Timespan		Commits			Imbalance ratio
	start	end	clean	defect	total	Clean:defect
QT	06/2011	03/2014	20330	3582	23912	5.676:1
OPENSTACK	01/2011	02/2014	16830	5927	22757	2.840:1

In the research of immediate defect prediction, the AUC score is often used as the evaluation index of the model. AUC stands for Area under the Receiver Operating Characteristic Curve, and it refers to the area beneath the curve of the receiver operating characteristic (ROC), which is mainly used for Investigates performance on imbalanced class datasets, with values ranging from 0 to 1. The suggested approach is used to analyse the QT and OPENSTACK data sets, and the ROC curve's area under the curve (AUC) is given in Table 2. The results of multiple experiments show that for the treatment of class imbalance problems, SSO-SMOTE is used. - The effect obtained by SSO processing the dataset is more significant than that obtained by using only a single SMOTE method.

Table 2

Area Under Curve (AUC) from ROC curve analysis for various datasets

Methods	QT	OPENSTACK
SMOTE	0.742	0.758
SSO-SMOTE-SSO	0.765	0.803

5. Conclusion

In this paper, we identify and predict approxi-mately 50,000 modifications from two open source projects using an innovative methodology that combines oversampling and undersampling methods to finish the processing of imbalance classes in on-the-fly software defect prediction.This study investigates the evolution of class imbalance in JIT-SDP, demonstrating that class imbalance is a significant issue in JIT-SDP by verifying the delay architecture, after that, a three-step intelligent sampling for class imbalance dataset was used. The method is used in a model that predicts software defects in real time, and the correction of unbalanced data is completed before the classification process, and the balanced data set is obtained to complete the defect prediction. In real datasets, the proposed mixed sampling strategy provides an effective solution to the imbalanced number of clean and faulty class changes (i.e. QT and OPENSTACK). Our future research will focus on how to handle class-imbalanced data distributions more quickly and accurately to produce an on-the-fly software defect prediction model with shorter run times and more accurate prediction outputs.

6. References

[1] Japkowicz, Nathalie, and Shaju Stephen. "The class imbalance problem: A systematic study." Intelligent data analysis 6, no. 5 (2002): 429-449.

[2] Chawla, Nitesh V., Kevin W. Bowyer, Lawrence O. Hall, and W. Philip Kegelmeyer. "SMOTE: synthetic minority over-sampling technique." *Journal of artificial intelligence research* 16 (2002): 321-357

[3] Han, Hui, Wen-Yuan Wang, and Bing-Huan Mao. "Borderline-SMOTE: a new over-sampling method in imbalanced data sets learning." In *International Conference on Intelligent Computing*, pp. 878-887. Springer, Berlin, Heidelberg, 2005.

[4] He, Haibo, Yang Bai, Edwardo A. Garcia, and Shutao Li. "ADASYN: Adaptive synthetic sampling approach for imbalanced learning." In *Neural Networks, 2008. IJCNN 2008.(IEEE World Congress on Computational Intelligence). IEEE International Joint Conference on*, pp. 1322-1328. IEEE, 2008.

[5] Barua, Sukarna, Md Monirul Islam, Xin Yao, and Kazuyuki Murase. "MWMOTE--majority weighted minority oversampling technique for imbalanced data set learning." *IEEE Transactions on Knowledge and Data Engineering* 26, no. 2 (2014): 405-425.

[6] S. Kim, E. J. W. Jr., and Y. Zhang, "Classifying software changes: Clean or buggy?" *IEEE Transactions on Software Engineering (TSE)*, vol. 34, no. 2, pp. 181–196, 2008.

[7] E. Shihab, A. E. Hassan, B. Adams, and Z. M. Jiang, "An industrial study on the risk of software changes," in *Proceedings of the 20th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*, 2012, pp. 1–11.

[8] J. Sliwerski, T. Zimmermann, and A. Zeller, "When do changes induce ´ fixes?" in *Proceedings of the 17th International Workshop on Mining Software Repositories*, ser. MSR '05, 2005, pp. 1–5.

[9] J. Eyolfson, L. Tan, and P. Lam, "Do time of day and developer experience affect commit bugginess?" in *Proceedings of the 8th Working Conference on Mining Software Repositories (MSR)*, 2011, pp. 153–162.

[10] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi, "A large-scale empirical study of just-in-time quality assurance," *IEEE Transactions on Software Engineering (TSE)*, vol. 39, no. 6, pp. 757–773, 2013.

[11] M. Tan, L. Tan, S. Dara, and C. Mayeux, "Online defect prediction for imbalanced data," in *Proceedings of the 37th International Conference on Software Engineering (ICSE)*, 2015, pp. 99–108.

[12] Z. Mahmood, D. Bowes, P. Lane, and T. Hall, "What is the impact of imbalance on software defect prediction performance?" in *Proceedings of the 11th International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE)*, 2015, pp. 4.1–4.4.

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

[14] K. E. Bennin, J. Keung, P. Phannachitta, A. Monden, and S. Mensah, "Mahakil: Diversity based oversampling approach to alleviate the class imbalance issue in software defect prediction," *IEEE Transactions on Software Engineering (TSE)*, vol. 44, no. 6, pp. 534–550, June 2018.

[15] Y. Kamei, A. Monden, S. Matsumoto, T. Kakimoto, and K. Matsumoto, "The effects of over and under sampling on fault-prone module detection," in *Proceedings of the 1st International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2007, pp. 196–204.

[16] S. Wang, L. L. Minku, and X. Yao, "Resampling-based ensemble methods for online class imbalance learning," *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, vol. 27, no. 5, pp. 1356–1368, 2015.

[17] Cabral G G, Minku L L, Shihab E, et al. Class imbalance evolution and verification latency in just-in-time software defect prediction[C]//2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). IEEE, 2019: 666-676.

[18] M. D. Zeiler and R. Fergus, "Stochastic pooling for regularization of deep convolutional neural networks," *arXiv preprint arXiv:1301.3557*, 2013.

[19] S. McIntosh and Y. Kamei, “Are fifix-inducing changes a moving target?: A longitudinal case study of just-in-time defect prediction,” in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE ’18. New York, NY, USA: ACM, 2018, pp. 560– 560. [Online]. Availa-ble:http://doi.acm.org/10.1145/3180155.3182514.