Applying Novel Resampling Strategies To Software Defect Prediction*

Lourdes Pelayo (SA)
Department of Electrical and Computer Engineering
University of Alberta
Edmonton, AB. T6G 2V4 Canada.
pelayo@ualberta.ca

Scott Dick
Department of Electrical and Computer Engineering
University of Alberta
Edmonton, AB. T6G 2V4 Canada.
dick@ece.ualberta.ca

***Abstract* - Due to the tremendous complexity and sophistication of software, improving software reliability is an enormously difficult task. We study the software defect prediction problem, which focuses on predicting which modules will experience a failure during operation. Numerous studies have applied machine learning to software defect prediction; however, skewness in defect-prediction datasets usually undermines the learning algorithms. The resulting classifiers will often never predict the faulty (minority0 class. This problem is well known in machine learning and is often referred to as learning from imbalanced datasets. We examine stratification, a widely used technique for learning imbalanced data that has received little attention in software defect prediction. Our experiments are focused on the SMOTE technique, which is a method of over-sampling minority-class examples. Our goal is to determine if SMOTE can improve recognition of defect-prone modules, and at what cost. Our experiments demonstrate that after SMOTE resampling, we have a more balanced classification. We found an improvement of at least 23% in the average geometric mean classification accuracy on four benchmark datasets.**

I. INTRODUCTION

Accurate defect prediction is enormously important, because of the huge economic impact of faulty software. According to Jim Johnson, chairman of the Standish Group, "Faulty software costs businesses $78 billion per year"[2]. Also from the $42 billion that the USA Department of Defense spends for the development and maintenance of their computer systems, only 17% is spent buying hardware[3]. It is generally believed that repairing failures after a software system is deployed is 100 times as expensive as repairing those faults during development.

In software, a common rule of thumb is that 80% of the problems reside in only 20% of the modules. This skewness is one of the main difficulties in software defect prediction; when we try to predict the occurrence of faults in software where the majority of modules are fault-free, the classifier is often unable to detect the faulty modules. This is a well-known problem in machine learning, often referred to as learning from imbalanced datasets. A data set that is heavily skewed toward the majority class will sometimes generate classifiers that never predict the minority class. In other cases the minority class will be predicted, but with a much higher rate of error than the majority class. This bias often makes the classifier highly accurate (it always correctly classifies the majority class examples), and completely useless at the same time.

Our goal in this research is to explore the use of stratification-based resampling in software defect prediction. Our experiments focus on the SMOTE technique [1], which is a method of oversampling minority-class examples by generating synthetic examples of the minority class, rather than replicating existing examples. SMOTE works by creating a new minority-class sample at a random point on the line connecting a minority class example with its nearest neighbor (of the same class) in feature space. We also simultaneously under-sample the majority class through uniform sampling without replacement.

The four benchmark datasets we used in this paper come from several NASA projects and are available at the PROMISE Repository of Software Engineering Databases [4]. Those datasets contain 21 software metrics based on the product's size, complexity and vocabulary, along with a binary class variable (module experienced a failure). We used a tenfold cross-validation experimental design, repeated for each choice of resampling rates in the majority and minority classes (hereafter the resampling strategy). Each training set was resampled as per the resampling strategy; the test sets were not resampled. All experiments used the c4.5 decision tree generator, with its default parameters. As overall accuracy is a poor measure of classifier performance on imbalanced data, we used the geometric mean of the individual class accuracies as our performance measure on the test data. After resampling the classes we have a more balanced class distribution, and the geometric mean accuracy increased by at least 23% in all four datasets.

The remainder of this paper is arranged as follows. We start in section II with related work in learning from imbalanced datasets, cost-sensitive classification and software defect prediction. In section III we describe our experimental methodology and performance evaluation. Following in section IV are our experimental results, and finally we offer a summary and discussion of future work in section V.

* This research was supported in part by NSERC under grant no. G121210906

II. RELATED WORK

Learning from imbalanced datasets is a current, active topic of research in the machine learning and data mining communities. We will briefly review major results in this area, and then provide an overview of software defect prediction.

A. *Imbalanced datasets*

The imbalanced dataset problem is present whenever the number of samples for one class in the dataset outnumbers the samples for the other class(es). This is common in application domains such as such as software defect prediction [4], credit card fraud detection [5], and breast cancer detection [6] where the number of faulty, fraudulent or positive samples, respectively, is considerably reduced in comparison with the majority class. However, the minority class samples are the ones of greatest interest to an analyst.

Estabrooks discusses in [7] the possibility of undersampling the majority class, oversampling the minority class via replication, or both. After several experiments with different datasets they reached the conclusion that undersampling is not better than oversampling or vice versa, but a combination of both is the best option. They propose a 3-stage meta-classifier that first stratifies the dataset at 20 different rates (10 are undersampled, 10 are oversampled), and trains a c4.5 decision tree for each one. In the second stage, two decision modules (one for undersampled trees, one for oversampled trees) output the "best" classification from stage 1. The third stage is another decision module that selects one of the two outputs from the second stage, and returns it as the final decision.

Kaminsky and Boetticher in [8] utilize genetic programming for software defect prediction. To compensate for data skewness they apply "equalized learning" considering the distribution of training set; they replicate instances in the minority class so that the training data will contain an equal distribution of instances (a simple oversampling technique, criticized in [1]). The dataset is finally shuffled. The performance measure reported is the fitness function; this is derived from the classification accuracy (measured on the training data).

Batista, Prati and Monard propose in [9] two methods to deal with the imbalanced dataset problem: SMOTE + Tomek and SMOTE + ENN. These are extensions of the SMOTE algorithm in [1]. According to their results, oversampling methods perform better in contrast with undersampling methods, with the area under the ROC curve (AUC) as their performance measure. This differs sharply from the recommendation in [10], where the authors assert that oversampling does not provide useful improvements over undersampling, while also increasing the time and memory requirements for learning algorithms. The authors of [10] reach this conclusion after experiments on over 20 datasets, which were first undersampled to create controlled class distributions.

B. *Cost-sensitive classification.*

Numerous algorithms in machine learning assume that all errors in classification are equally important. However the cost of misclassification errors is often higher when the errors are located in the minority class in contrast with the majority class [10]. The reason is that, when the minority class samples are the interesting ones, mistaking a minority class sample for a majority class sample is more costly to the end users of the classification. For example, mistaking a fault-prone software module (minority class) for a fault-free module (majority class) can permit faults to slip through into field use, where they are vastly more expensive to fix than they were in the testing phase of development. Cost-sensitive classifiers explicitly consider these differential costs, and will minimize the total expected cost of errors, rather than just the number of errors as in most classifiers. Elkan introduces in [11] a theorem showing how to change the proportion of negative examples in a training set in order to make an optimal cost-sensitive classification; however it has little effect when using Bayesian and decision tree learning algorithms. Turney introduces in [12] a new algorithm for cost sensitive classification called ICET that is a hybrid of genetic algorithms and decision trees. Ting and Zheng explore in [13] boosting techniques for cost-sensitive classification focused on decision tree learning; their algorithm has an extra computation cost because it needs to create new classifiers every time the costs of misclassification change. Domingos proposes a method for a cost-sensitive classification called MetaCost in [14]. It is based on relabeling training examples with their minimal-cost classes based on their class probabilities instead of their "optimal" class, and re-applying the classifier to the new training set.

C. *Software Defect Prediction*

Software metrics have been used for software defect prediction for decades. They include McCabe's complexity [15] and Halsted's metrics [16], along with product's size, complexity and vocabulary. For Object Oriented programming there are six best-known metrics developed in [17] known as the CK metrics: Weighted Methods per Class (WMC), Depth of Inheritance Tree (DIT), Number of Children (NOC), Response For a Class (RFC), Coupling Between Object Classes (CBO), and Lack of Cohesion of Methods (LCOM). Research in defect prediction consistently shows a moderately strong linear correlation between almost every metric and the number of defects per module; however, no parametric model has ever been developed that accurately forecasts the number or occurrence of faults in a software module.

There are also different classes of software metrics. The McCabe, Halstead and CK metrics all represent "code" metrics, computed directly from source code. There are also requirements metrics, test metrics, etc. Rosenberg, Hammer and Shaw[21] discuss the integrated use of requirements, code and testing metrics at NASA's Software Assurance technology Center (circa 1998).

Fenton and Neil [18] provide a critical review of software-defect prediction research up to 1999. They

highlighted numerous statistical models (e.g. Akiyama's [19] and Ferdinand's [20]) developed for this domain, and concluded that a variety of methodological problems rendered many of them incomparable. Bayesian belief networks were proposed in [18] as an alternative to parametric models.

Following the same rationale as [18], a number of authors have investigated the use of machine-learning techniques as non-parametric models for software defect prediction. Some examples include neural networks [26], genetic programming [8], fuzzy clustering [27] and decision trees [28]. Imbalanced class distributions have been significant problems in these studies. Recently, Seliya and Khoshgoftaar [22] proposed a semi-supervised clustering method to detect failures in software modules. Instead of working with the individual modules on software, they group modules and label them as fault prone or not fault prone. Clustering algorithms -in this case the k-means algorithm- are supplemented with a set of labeled program modules, resulting in a semi-supervised method. With this the domain expert is helped to label further clusters as either fault prone or not fault prone.

III. Experimental methodology

The datasets used for our experiments are referred to as CM1, KC1, KC2 and PC1. They come from several NASA projects and are available at the PROMISE Repository of Software Engineering Databases [4]. CM1 is made up of 20,000 lines of C code, KC1 contains 43,000 lines of C++ code, KC2 contains over 43,000 lines of C++ code, and PC1 consists of 40,000 lines of C code. Each dataset contain 21 software metrics based on the product size, complexity and vocabulary, along with a binary class variable (module experienced a failure). These datasets were also used in [8].

We used the well-known C4.5 [23] decision tree classifier for our experiments. It provides a simple and useful environment to perform classification on datasets. Several recent studies of stratification [7], [9], [10] use C4.5 and its default parameters to measure classification performance after stratification; we adopt this same approach. We utilized a 10-fold cross validation design to test each stratification "strategy". A "strategy" is a class-by-class specification of the degree of undersampling or oversampling to be applied [24]. We stratify the training set for each fold, and evaluate the classifier on the unaltered test dataset.

Each fold was resampled with the SMOTE algorithm [1]. SMOTE is a proposed oversampling approach in which the minority class is oversampled by creating "synthetic" examples rather than by oversampling with replacement. To create a synthetic example, SMOTE searches for the nearest neighbors (having the same class label) of a minority-class example. A new synthetic example is then created at a random point on the line connecting the two genuine samples (this assumes that each dimension of feature space belongs forms a ratio scale), and class label for the new example is the minority class. Different synthetic examples are based on different neighbor pairs. SMOTE also uses uniform selection without replacement to undersample the majority class.

We calculated the geometric mean of the class accuracies from the confusion matrix for each fold. The geometric mean is highly sensitive to imbalance in the individual class accuracies. It has been used as a performance measure for learning in imbalanced datasets in [25]. We determine the average and standard deviation of the geometric mean accuracy across all ten folds for each dataset.

IV. Experimental results

We first used SMOTE to achieve a uniform class distribution for each dataset. However, having approximately the same number of samples per class did not yield the highest geometric mean accuracy. We explored other resampling strategies using trial-and-error, and finally arrived at the highest geometric mean accuracies. The percentage of over-sampling and under-sampling was according the behavior of each dataset, confirming the proposal of Weiss and Provost in [10], that the resampling rates will be determined according to each dataset. We also observed that over-sampling the minority class more than 300% was not useful. We present our experimental results in Table 1.

TABLE I
Experimental results

Dataset	Under-sample majority class (%)	Over-sample minority class (%)	Average Geometric Mean Accuracy (%)	Standard Deviation
CM1	0 (original)	0	13.69	0.22
	25	200	55.34	0.11
	50	300	47.87	0.19
	75	300	**62.20**	0.14
KC1	0	0	52.30	0.07
	50	300	69.20	0.08
	85	100	69.39	0.03
	75	300	**71.25**	0.04
KC2	0	0	60.13	0.12
	50	100	72.55	0.08
	50	200	73.33	0.04
	75	200	**74.09**	0.07
PC1	0	0	47.00	0.12
	75	200	70.00	0.10
	75	300	68.24	0.09
	85	300	**74.26**	0.09

As we can see in Table 1, there are substantial improvements reflected in higher average geometric mean accuracy. The most dramatic example is where the dataset CM1 showed 0% classification accuracy for the minority class in several folds. Stratification improved the average geometric mean accuracy of CM1 from 13.69% to 62.20%. Across all four datasets, we were able to improve the geometric mean accuracy by at least 23%. We also note that the standard deviations were simultaneously reduced, indicating more consistent classification results.

V. Summary and future work

We applied the SMOTE algorithm to software defect prediction. This resampling technique allowed us to create

more balanced training datasets, and the geometric mean classification accuracy (using c4.5 decision trees) on the unmodified test sets was substantially improved in our tenfold cross-validation experiments. Imbalanced class distributions are a major problem in applying machine learning techniques to software defect prediction, and there has been very little investigation of stratification as a solution to this very important problem.

In our future work, we will seek to quantify the contribution of SMOTE and undersampling, in response to Weiss & Provost's assertion that undersampling alone is sufficient. We will use a full factorial design with over-sampling and undersampling as the two factors to test Weiss and Provost's claim against a larger set of software defect prediction datasets. We will also investigate the application of Response Surface Methodology as a structured methodology to identify the best resampling strategy for a given dataset.

REFERENCES

[1] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "SMOTE: Syntethic Minority Over-Sampling Technique," *Journal of Artificial Intelligence Research,* vol. 16, pp. 321-357, June 2002.

[2] J. Johnson, "Let's Stop Wasting $78 Billion per Year," *CIO Magazine,* October 15, 2001 2001.

[3] N. Brown, "Industrial-strength management strategies," *IEEE Software,* vol. 13, pp. 94-103, July 1996.

[4] J. Sayyad Shirabad and T. J. Menzies, "The PROMISE Repository of Software Engineering Databases. School of Information Technology and Engineering," University of Ottawa, 2005.

[5] T. Fawcett and F. Provost, "Adaptive Fraud Detection," *Data Mining and Knowledge Discovery,* vol. 1, September 1997.

[6] D. J. Newman, S. Hettich, C. L. Blake, and C. J. Merz, "UCI Repository of machine learning databases," Irvine, CA: University of California, Department of Information and Computer Science, 1998.

[7] A. Estrabooks, T. Jo, and N. Japkowicz, "A Multiple Resampling Method for Learning from Imbalanced Datasets," *Computational Intelligence,* vol. 20, 2004.

[8] K. Kaminsky and G. D. Boetticher, "Better Software Defect Prediction using Equalized Learning with Machine Learners," in *Proceeding (430) Knowledge Sharing and Collaborative Engineering* St. Thomas, US Virgin Islands.

[9] G. Batista, R. Prati, and M. C. Monard, "A study of the Behavior of Several Methods for Balancing Machine Learning Training Data," *Sigkdd Explorations,* vol. 6, pp. 20-29.

[10] G. Weiss and F. Provost, "Learning when training data are costly: The effect of class distribution on tree induction," *Journal of Artificial Intelligence Research,* vol. 19, pp. 315 - 354, 2003.

[11] C. Elkan, " The Foundations of Cost-Sensitive Learning " in *Proceedings of the Seventeenth International Joint Conference in Artificial Intelligence*, 2001.

[12] P. D. Turney, "Cost-Sensitive Classification: Empirical Evaluation of a Hybrid Genetic Decision Tree Induction Algorithm," *Journal of Artificial Intelligence Research 2,* pp. 369-409, 1995.

[13] K. M. Ting and Z. Zheng, "Boosting trees for cost-sensitive classifications," in *Proc. 10th European Conf. on Machine Learning*, Chemnitz, Germany, 1998, pp. 191-195.

[14] P. Domingos, "MetaCost: A General Method for Making Classifiers Cost-Sensitive," in *Proceedings of the Fifth International Conference on Knowledge Discovery and Data Mining (KDD-99)*, 1999.

[15] T. J. McCabe, "A Complexity Measure," *IEEE Trans. Software Eng,* vol. 2, pp. 308 - 320, 1976.

[16] M. H. Halstead, *Elements of Software Science*. North-Holland: Elsevier, 1975.

[17] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *Software Engineering, IEEE Transactions on,* vol. 20, pp. 476-493, 1994.

[18] N. E. Fenton and M. Neil, "A Critique of Software Defect Prediction Models," *IEEE Transactions on Software Engineering,* vol. 25, September/October 1999, 675-689.

[19] F. Akiyama, "An Example of Software System Debugging," *Information Processing,* vol. 71, pp. 353-379, 1971.

[20] A. E. Ferdinand, "A Theory of System Complexity," *Int'l J. General Systems,* vol. 1, pp. 19-33, 1974.

[21] L. Rosenberg, T. Hammer, and J. Shaw, "Software Metrics and Reliability," in *9th International Symposium on Software Reliability Engineering*, 1998.

[22] N. Seliya and T. M. Khoshgoftaar, "Software Quality Analysis of Unlabeled Program Modules With Semisupervised Clustering," *Systems, Man and Cybernetics, Part A, IEEE Transactions on,* vol. 37, pp. 201-211, 2007.

[23] J. R. Quinlan, *C4.5: Programs for machine learning*. San Mateo, CA: Morgan Kaufmann Publishers, 1993.

[24] S. Dick and A. Kandel, "Computational Intelligence in Software Quality Assurance," M. Last, A. Kandel, Eds., *Artificial Intelligence Methods in Software Testing,* World Scientific Press 2004.

[25] M. Kubat, R. Holte, and S. Matwin, "Machine Learning for the Detection of Oil Spills in Satellite Radar Images," *Machine Learning,* vol. 30, pp. 195-215, 1997.

[26] N. Karunanithi, Y.K. Malaiya, "Neural networks for software reliability engineering," in M.R. Lyu, Ed., Handbook of Software Reliability Engineering, New York: McGraw-hill, 1996, pp. 699-728.

[27] S. Dick, A. Sadia, " Fuzzy Clustering of Open-Source Software Quality Data: A Case Study of Mozilla," in *Proceedings of IJCNN 2006,* Vancouver, BC, Canada, pp. 4089-4096.

[28] T.M. Khoshgoftaar, E.B. Allen, W.D. Jones, J.P. Hudepohl, "Classification-tree models of software-quality over multiple releases," *IEEE Transactions on Reliability*, v. 49 no. 1, Mar. 2000, pp. 4-11.