



Addressing Class Imbalance in Software Defect Prediction by Averaging

Moses A. Agebure¹, Peter A. Agbedemna²

Department of Computer Science
University for Development Studies
Navrongo Campus, Navrongo
GHANA

Abstract: *The class imbalance problem is common in most software defect datasets and as a result imbalance learning techniques proposed in general machine learning domain are normally adapted to minimise its effects in software defect prediction. In this paper a simple software data centred approach is proposed and examined using twelve software defect datasets and three classification algorithms. Empirical results showed that models built after using the proposed technique to reduce the number of non-defective modules in training datasets statistically outperform models built without altering the training dataset in all test cases using AUC and G-Mean measures. It also recorded higher mean performance in all test cases and is better than some well-known data sampling techniques.*

Keywords: *Class imbalance; software defect prediction; classification; machine learning; data sampling*

I. Introduction

The application of machine learning techniques in software defect prediction using software metric data spans over decades. This application comes in two folds; to determine the number of defects that resides in a software module [1], or to predict if a module is defective or not [2], [3], [4]. Software defect prediction using machine learning techniques is very useful in software engineering processes particularly in the testing phase as its output serves as a guide as to where software testers should focus their attention and limited resources. Software testing is a tedious task that has the potential of consuming up to 50% of the total project cost without adding any basic functionality to the end product [5]. Shull et al. [6] hold the view that finding and fixing software defects in the course of development is 100% less expensive as compared to fixing the defects after the software have been deployed. That is, if software quality issues are not handled efficiently in the course of development will certainly affect the reliability and hence the maintenance cost of the end product.

However, due to the general nature of software systems, software metric data generated for the purpose of building classification models often suffer from what is commonly known in the machine learning community as class imbalance problem. It is common knowledge that about 80% of software defects reside in approximately 20% of software modules [7]. This implies that only a smaller percentage of the total software modules normally contain defects. This skewness in the number of defective and non-defectives modules is demonstrated in literature to negatively affect the ability of classification models to correctly classify the defective modules, which the processes are often set to identify. In most cases, such models achieve higher accuracies while classifying all or most of the defective modules as non-defective modules.

To address the above stated challenge, several imbalanced learning methods have been proposed in the general machine learning discipline and adapted in software defect prediction to improve the ability of classifiers to correctly classify defective modules. One class of such robust methods are the data level or data sampling techniques [8], [9], [10], [11]. These techniques focus on altering the distribution of datasets by either increasing the number of defective modules or reducing the number of non-defective modules to produce a fairly balanced dataset for training. Although these sampling techniques improve the performance of most classification algorithms, some of them, particularly the random undersampling technique (RUS) do not take into consideration the internal relationship of the non-defective modules (majority class instances) during undersampling, which may lead to loss of information. The goal of this research is therefore to represent a simple method for reducing the size of the non-defective modules while preserving their internal distribution, thus minimizing information loss.

The remaining of the paper is organized as follows. In section II, related work in learning from imbalanced datasets in software defect prediction is discussed. Section III describes the proposed method, and section IV follows with the methodology. The results are presented and discussed in section V, and finally conclusion and recommendations in section VI.

II. Related Work

In this section, some data sampling techniques and their application in software defect prediction is briefly reviewed.

A. Data Sampling Techniques

The two main approaches to data sampling are the majority class undersampling (e.g. Random Undersampling (RUS), Tomek links (TL) [10], Condensed Nearest Neighbor Rule (CNN) [11], and One Sided Selection (OSS) [9]) and minority class oversampling techniques (e.g. Synthetic Minority Oversampling Technique (SMOTE) [8] and Random Oversampling (ROS)). In some cases, undersampling and oversampling techniques are combined to form hybrid methods such as SMOTE+Tomek links [13], and SMOTE+Edited Nearest Neighbor Rule (SMOTE+ENN) [14].

RUS reduces the number of majority class instances in a training dataset by randomly discarding some of the majority class instances; this however, may lead to information loss. TL, CNN, and OSS contrary to RUS remove noise, borderline and/or redundant majority class instances from the training dataset. The TL technique removes all majority class instances that participate in Tomek links. Two instances I_i, I_j belonging to different classes are said to participate in a Tomek link if there is no other instance, I_k such that the distance, $d(I_i, I_k) < d(I_i, I_j)$ or $d(I_j, I_k) < d(I_i, I_j)$. If two instances form a Tomek link, either one of them is noise or both are borderline instances. Also, in the CNN technique, all minority class instances in the original dataset, S are copied into a dataset, S' with only one randomly selected majority class instance. S is classified with 1-NN rule using the instances in S' and all misclassified instance are moved from S to S' . CNN is susceptible to noise since noisy instances are more likely to be misclassified and hence added to the training data. OSS on the other hand combines CNN and TL. It applies CNN on the dataset followed by TL to remove noisy and borderline instances.

Oversampling minimizes the effects of the imbalance problem by increasing the size of the minority class instances. ROS achieves this by randomly replicating minority instances in the training dataset. This random replication of instances does not add new information to the training data, hence may result to overfitting and also the increase in size of the training dataset may prolong training time [15]. SMOTE attempts to solve the overfitting problem associated with ROS by generating synthetic minority instances. For each minority instance, its k nearest neighbours is determined and synthetic instances are generated in the direction of all or some of the neighbours depending on the oversampling percentage required [8].

The hybrid methods such as SMOTE+Tomek links and SMOTE+ENN, uses SMOTE to oversample the minority class instances followed by Tomek Links to remove both majority and minority instances that participate in Tomek Links or ENN to remove both minority and majority class instances whose class labels differ from at least two of its three nearest neighbours.

B. Software Defect Prediction

Static code attributes are basic software metric data that is extracted directly from source code and often carry information of the source code such as its complexity, size and vocabulary. There are basically three categories of static code metrics namely; line of code (LOC) counts, Halstead's metrics [16], and McCabe's metrics [17].

Though these metrics have been extensively used in software defect prediction research, Fenton & Pfleeger [18] and Shepperd & Ince [19] criticized the usefulness of some of these metrics. Fenton & Pfleeger [18] demonstrated how the same software functionality is achieved using different programming language constructs, which yield different static measurements for the module. Based on this example they argue the uselessness of static code attributes. Also, Shepperd & Ince [19] in their work presented empirical evidence that suggest that McCabe metrics offer nothing more than uninformative metrics such as lines of code. They remarked that "for a large class of software it (cyclomatic complexity) is no more than a proxy for, and in many cases outperformed by lines of code."

Menzies et al. [20] however, refute the above claims by Fenton & Pfleeger and Shepperd & Ince stating that, "the supposedly better static code attributes, such as Halstead and McCabe, should perform no better than just simple thresholds on lines of code, and the performance of a predictor learned by a data miner should be very poor." They note that none of these is true for at least the datasets they used for their study. They showed that, their study found defect predictors with a probability of detection of about 71% which is evidently higher as compared to a conclusion arrived at by a panel at IEEE Metrics 2002 [5] that manual software reviews can find approximately 60% of defects. This conclusion further confirms the reason for the wide use of static code metrics in software defect prediction research.

Another problem that plagues most software metric datasets is class imbalance. To mitigate its effects, researchers adapted and assessed the efficiency of most class imbalance learning techniques in software defect prediction. For example, Seiffert et al. [15] examined how data sampling techniques and boosting can improve the performance of C4.5 decision tree algorithm. RUS, SMOTE, ROS, borderline-synthetic minority oversampling (BSMOTE), and Wilson's Editing (WE) were examined. Using Kolmogorov-Smirnov (K-S) statistic and Area Under the receiver-operating-characteristic Curve (AUC) as performance measures, they concluded that all the sampling techniques improved the performance of the C4.5 decision tree with RUS outperforming most of the other methods.

Riquelme et al. [21], analysed the impact of SMOTE and resampling on two learning algorithms; Naïve Bayes and C4.5 decision tree algorithm and reported that sampling did improve the AUC of the classification models, that is, they classified better the minority class instances, which are of interest. In a similar work, Pelayo & Dick [22] examined stratification, and focused the experiment on SMOTE with the goal of determining if data stratification and SMOTE can improve the learning algorithms ability to recognise defect-prone modules. The authors demonstrated that after SMOTE resampling, an improvement of at least 23% in average geometric mean of the classification accuracy on four benchmark datasets was realised.

III. Proposed Approach

The proposed approach unlike the other sampling techniques which are generic is centred on software defect datasets. It attempts to exploit the relationship or similarities that often exist between software modules. It is prudent to state that, given sufficiently large software, there would be a number of modules that exhibit similar programming constructs and static code metric data generated from these modules would either have the same attributes or differ in a few attributes. Instead of discarding some of these in the course of data sampling, the closest pair of these instances are transformed into single instances by averaging. This we suggest would to a greater extent minimize information loss that may occur in data undersampling. The proposed approach is summarized as follows:

1. Given a training dataset D containing n majority class instances.
2. Randomly select a majority class instance, I_i .
3. Find the nearest neighbour (majority class instance) I_j of I_i such that; $d(I_i, I_j) < d(I_i, I_{k:k=1..n-2, i \neq k})$ i.e. the distance from I_i to I_j should always be less than the distance from I_i to any other instance I_k in the dataset.
4. Compute the average for each corresponding attribute of I_i and I_j to form a new instance.
5. Remove I_i and I_j from D and add the new instance to D .
6. Repeat step 2 through 5 until the desired size of training data is realized.

IV. Experimental Setup

A. Datasets

The datasets used for all experiments in this study are static code metric datasets created at National Aeronautics Space Administration (NASA) from various software products under their Metric Data Program (MDP) and are available at the Promise Repository [12].

B. Classification Algorithms

Four classification algorithms are considered in this study to assess the efficiency of the proposed approach. These include a C4.5 Decision tree [23], Support Vector Machines [24], and a proportional rule based learner [25], implemented in the Weka Machine learning tool [26] as J48, SMO, and JRip respectively. The default parameters of these classifiers specified in the Weka tool are used for all experiments except otherwise stated. This is because the focus of the study is to assess the extent to which the proposed approach improves the ability of classification algorithms to correctly classify defective modules. These classifiers are chosen because they have been widely used in software defect prediction research and also represent different approaches to learning.

C. Data Sampling

In order to assess the efficiency of the proposed approach, experiments are conducted using four well-known data sampling techniques (RUS, ROS, SMOTE, and OSS). These techniques are used to sample the training data prior to building classification models. Also, experiments are conducted without sampling the training data and these are considered the based experiments. For the proposed approach and each sampling technique, the size of the training data is varied using several parameters prior to training and the parameter that yields the highest classification performance given a dataset and classification algorithm is recorded. The following parameters are used for the proposed approach and RUS; 5, 10, 25, 50, 75, 90, and 95. Also, 50, 100, 200, 300, 500, 750, and 1000 are used for ROS and SMOTE. If a parameter, 75 is used for RUS and the proposed approach it means that the number of majority instances in the training data is reduced by 75%, likewise, oversampling by 200 means the number of minority instances in the training data is increased by 200%. These parameters are used in several studies including [15].

D. Experimental Process

Stratified ten (10) fold cross validation process is adapted for all experiments. That is, a dataset is divided into 10 approximately equal folds and for each of the ten runs one fold is held as the testing data and remaining nine folds combined to form the training data. Each fold must be used once as testing data in this process. The average performance of the classifier for the ten runs is calculated and considered its performance. The cross

validation process is repeated ten (10) times and the performance further averaged and this is recorded as the final performance of the classifier under consideration. The repetition is done to eliminate any biasness that might be introduced by the data sampling or stratification process. For all source of randomness, a random seed of 10 is used unless otherwise stated.

E. Performance Metrics

As recommended in [15] and [20], the Area Under receiver operating characteristic Curve (AUC) and Geometric Mean (G-mean) performance metrics are used to measure the performance of classifiers in this research due to the imbalanced nature of the datasets. The AUC measure is derived from Receiver-Operating Characteristic (ROC) curve and its value range between 0 and 1, with 1 being best and 0 worse performance. It is calculated using True Positive Rate (TPR) and False Positive Rate (FPR) as:

$$AUC = \frac{1 + TPR - FPR}{2} \quad (1)$$

The G-mean is calculated using the true positive rate (TPR) and true negative rate (TNR) of a classifier as follows:

$$G - mean = \sqrt{TPR * TNR} \quad (2)$$

V. Results and Discussion

The experimental results of the three classification algorithms over twelve (12) software defect datasets given the various data sampling techniques are presented and discussed in this section. The results of classification models built without applying data sampling techniques represent the base model labelled as Base. The results of the proposed approach (AVG) are first compared to the results of the base models and subsequently with the results of the models built after applying the various data sampling techniques. Table II shows the results of the classification models based on AUC performance measure and table III shows the results based on G-Mean measure.

The results in tables II and III show that using the proposed approach, AVG to reduce the size of majority class instances prior to building models improve the performance of all three classification algorithms across the twelve (12) datasets considered as compared to results of the base models. The percentage of improvement varies from one dataset to the other depending on the classification algorithm and the skewness of the dataset under consideration. This improvement ranges from 2.97% to 61.25% and 1.69% to 180% for the J48 classifier using AUC and G-Mean measures respectively. For the jRip classifier, the improvement in AUC values range from 3.95% to 60.65% and G-Mean values from 21.79% to 536.22%. The SMO classifier recorded improvements ranging from 3.34% to 69.74% of AUC measures and 21.37% to 790.03% of G-Mean measures. It is worth noting that the base models of J48 and JRip recorded 0.00 G-Mean values in the PC2 dataset and SMO recorded 0.00 G-Mean values in six out of the twelve datasets (CM1, MC1, MW1, PC1, PC2, and PC3) considered. This implies that, these base models failed to correctly classify at least a single defective module in the respective datasets. The improvement in AUC and G-Mean measures suggest that the proposed technique has the tendency of significantly improving the performance of classification algorithms. However, a definite conclusion cannot be drawn without further comparison with the results of well-known sampling techniques and statistical test.

The AUC and G-Mean measures of the classification models when the datasets are sampled prior to training are shown in column 2 through 5 of table II and III respectively. It is evident from these tables that AVG yielded results that are better or comparable to that of the four well-known sampling techniques. In the J48 learner, AVG outperformed the four sampling techniques in seven out of the twelve datasets using AUC measure and six out of twelve using G-Mean measure. SMOTE produced better AUC values in six out of twelve datasets in the JRip learner with AVG yielding better results in four out of the remaining six datasets. Considering the G-Mean values of the JRip learner, AVG outperformed the sampling techniques in six datasets out of the twelve considered. For the SMO learner, AVG performed better than the four sampling techniques in only two of the twelve datasets considered using both AUC and G-Mean measures as shown in table II and III respectively.

To ascertain whether the variation in performance of the classification algorithms across the data sampling techniques and base models are statistically significant, an ANOVA analysis is carried out for each of the six (6) test cases shown in table II and III. At $\alpha = 0.05$ the ANOVA analysis suggest that there is significant difference in the performance of the classification algorithms given the various data sampling techniques for all six test cases. To determine which technique(s) resulted to the statistically different performance of the classification algorithms; a Tukey's HSD post-hoc test is carried out for all test cases at $\alpha = 0.05$.

Table II: Classifier Performance Using AUC

Tech/ Dataset	AVG	RUS	SMOTE	ROS	OSS	Base
J48						
CM1	0.671	0.609	0.696	0.654	0.544	0.602
JM1	0.677	0.665	0.682	0.594	0.635	0.657
KC1	0.765	0.732	0.720	0.680	0.685	0.675
KC3	0.670	0.645	0.640	0.634	0.596	0.646
MC1	0.881	0.864	0.878	0.843	0.782	0.779
MC2	0.651	0.662	0.704	0.660	0.611	0.603
MW1	0.672	0.666	0.646	0.621	0.660	0.487
PC1	0.763	0.762	0.758	0.689	0.690	0.698
PC2	0.834	0.679	0.669	0.606	0.603	0.517
PC3	0.699	0.735	0.694	0.692	0.694	0.608
PC4	0.825	0.835	0.789	0.778	0.800	0.750
PC5	0.928	0.922	0.813	0.718	0.859	0.743
JRip						
CM1	0.678	0.674	0.711	0.680	0.550	0.512
JM1	0.649	0.616	0.693	0.654	0.617	0.562
KC1	0.729	0.704	0.773	0.686	0.662	0.588
KC3	0.668	0.629	0.737	0.668	0.638	0.618
MC1	0.831	0.653	0.763	0.815	0.731	0.653
MC2	0.643	0.649	0.714	0.672	0.674	0.618
MW1	0.641	0.679	0.646	0.636	0.687	0.568
PC1	0.751	0.714	0.703	0.686	0.590	0.564
PC2	0.800	0.578	0.690	0.521	0.498	0.498
PC3	0.731	0.764	0.757	0.693	0.648	0.572
PC4	0.840	0.781	0.865	0.799	0.768	0.718
PC5	0.931	0.917	0.805	0.826	0.869	0.742
SMO						
CM1	0.681	0.674	0.708	0.696	0.495	0.500
JM1	0.619	0.616	0.624	0.627	0.568	0.503
KC1	0.713	0.704	0.717	0.722	0.631	0.518
KC3	0.611	0.629	0.703	0.705	0.544	0.522
MC1	0.656	0.653	0.639	0.639	0.500	0.500
MC2	0.667	0.649	0.672	0.684	0.658	0.629
MW1	0.690	0.679	0.710	0.695	0.625	0.500
PC1	0.724	0.714	0.775	0.774	0.515	0.499
PC2	0.517	0.578	0.500	0.500	0.500	0.500
PC3	0.763	0.764	0.757	0.763	0.502	0.500
PC4	0.795	0.781	0.826	0.819	0.639	0.555
PC5	0.920	0.917	0.890	0.902	0.764	0.542

Table III: Classifier Performance Using G-Mean

Tech/ Dataset	AVG	RUS	SMOTE	ROS	OSS	Base
J48						
CM1	0.527	0.522	0.474	0.421	0.450	0.324
JM1	0.490	0.518	0.518	0.564	0.593	0.482
KC1	0.706	0.696	0.645	0.574	0.628	0.490
KC3	0.660	0.561	0.607	0.541	0.585	0.437
MC1	0.762	0.772	0.596	0.715	0.634	0.498
MC2	0.593	0.644	0.675	0.615	0.571	0.499
MW1	0.662	0.574	0.454	0.470	0.422	0.236
PC1	0.717	0.725	0.624	0.595	0.647	0.407
PC2	0.681	0.418	0.140	0.100	0.000	0.000
PC3	0.691	0.708	0.632	0.602	0.599	0.492
PC4	0.828	0.832	0.782	0.762	0.808	0.725
PC5	0.926	0.919	0.767	0.717	0.832	0.646
JRip						
CM1	0.626	0.629	0.597	0.666	0.270	0.098
JM1	0.559	0.548	0.607	0.625	0.541	0.391
KC1	0.714	0.689	0.659	0.657	0.610	0.452
KC3	0.599	0.571	0.712	0.607	0.583	0.427
MC1	0.819	0.541	0.713	0.789	0.667	0.541
MC2	0.587	0.566	0.689	0.653	0.642	0.468
MW1	0.527	0.565	0.540	0.506	0.570	0.248
PC1	0.712	0.675	0.658	0.628	0.433	0.312
PC2	0.704	0.236	0.435	0.070	0.000	0.000
PC3	0.721	0.755	0.646	0.667	0.591	0.404
PC4	0.818	0.778	0.787	0.777	0.747	0.671
PC5	0.924	0.916	0.782	0.808	0.863	0.697
SMO						
CM1	0.647	0.629	0.698	0.668	0.000	0.000
JM1	0.553	0.548	0.564	0.570	0.409	0.062
KC1	0.695	0.689	0.715	0.720	0.555	0.174
KC3	0.571	0.571	0.691	0.670	0.271	0.068
MC1	0.556	0.541	0.526	0.526	0.000	0.000
MC2	0.619	0.566	0.656	0.669	0.641	0.510
MW1	0.676	0.565	0.685	0.676	0.402	0.000
PC1	0.720	0.675	0.771	0.769	0.078	0.000
PC2	0.070	0.236	0.000	0.000	0.000	0.000
PC3	0.754	0.755	0.753	0.752	0.027	0.000
PC4	0.791	0.778	0.819	0.816	0.527	0.304
PC5	0.920	0.916	0.887	0.901	0.733	0.289

The HSD results revealed that, the AUC and G-Mean measures of all classifications models built using AVG on the datasets prior to learning significantly outperformed all the corresponding base models. This however, does not hold true for the other sampling techniques as RUS failed to outperform the base models of J48 and JRip classifiers using AUC measure. SMOTE also outperformed the base models of all test cases except that of the J48 classifier using both AUC and G-Mean. OSS on the other hand failed to outperform the base models in all cases considered. ROS only performed significantly better than the base models of the JRip classifier using G-Mean measure and SMO classifier using both AUC and G-Mean. According to the HSD test, the performances of the J48 classification algorithm do not differ significantly when the various data sampling techniques (AVG, RUS, SMOTE, ROS, and OSS) are applied to the training datasets prior to training and this holds for both AUC and G-Mean measures. Similarly, no data sampling technique resulted to a significantly different performance of the JRip algorithm using both AUC and G-Mean measures. The performance of the SMO is significantly different (worst) when OSS is used to sample the training data prior to training. That is, AVG, RUS, SMOTE, and ROS yielded significantly better performance than OSS. The SMO classifier's performance using both AUC and G-Mean measure is however not statistically different when AVG, RUS, SMOTE, and ROS are used to sample the training data. In a summary, the proposed technique recorded the highest mean performance across all test cases for each of the classification algorithms hence it is a viable alternative to data sampling in software defect prediction.

VI. Conclusion

Software defect prediction is widely studied because of the immense benefits it presents to software developers or practitioners. However, most software datasets are often plagued with the class imbalance problem, which negatively affects the performance of classification. Generic imbalance learning techniques such as data sampling methods are often used to address this problem in software defect datasets. In this paper, a data

reduction method that rely on the internal relationship of software modules is proposed and examined. The proposed technique combines non-defective modules that share similar characteristics to form single modules thereby reducing their population in the training dataset.

The proposed technique is examined empirically using three machine learning algorithms (C4.5 decision tree, Jrip learner, and Support vector machine) and twelve (12) real-world software defect datasets. The performance of the models built after applying AVG to the training datasets are compared to models built using other techniques (NONE, RUS, SMOTE, ROS, and OSS). HSD post-hoc test at $\alpha = 0.05$ showed that the mean performance of models built after applying AVG to the training data is significantly better than the mean performance of models built without altering the training. The HSD test also revealed that the mean performance of AVG is statistically same a RUS, SMOTE, and ROS for all algorithms using both AUC and G-Mean measures. AVG however performed statistically better than OSS in the SMO algorithm using both AUC and G-Mean measures.

It can therefore be concluded that the proposed technique is a promising alternative to data sampling techniques in software defect prediction. Future research will therefore focus on carrying out an empirical analysis on the datasets to determine if indeed the increase in performance of classification algorithms are merely due to the reduction of fault-prone modules in the training datasets or the averaging of the modules indeed positively influenced classifier performance.

VII. References

- [1] T. M. Khoshgoftaar and N. Seliya, "Fault Prediction Modelling for Software Quality Estimation: Comparing Commonly Used Techniques," *Empirical Software Engineering*, vol. 8, pp. 255-283, 2003.
- [2] S. Wang and X. Yao, "Using Class Imbalance Learning for Software Defect Prediction," in *IEEE Transactions on Reliability* 62(2): 434-443, June 2013.
- [3] G. Czibula, Z. Marian, I. G. Czibula, "Software defect prediction using relational association rule mining," *Information Sciences* 264 (2014) 260–278.
- [4] C. Seiffert, T. M. Khoshgoftaar, J. V. Hulse, and A. Folleco, "An empirical study of the classification performance of learners on imbalanced and noisy software quality data," *Inform. Sci.* (2011), doi:10.1016/j.ins.2010.12.016.
- [5] G. Whyte and D. L. Mulder, "Mitigating the Impact of Software Test Constraints on Software Testing Effectiveness," *The Electronic Journal Information Systems Evaluation*, Vol. 14 Issue 2, 2011, (pp254-270), available online at www.ejise.com.
- [6] F. Shull, V. Basili, B. Boehm, A. Brown, P. Costa, M. Lindvall, D. Port, I. Rus, R. Tesoriero, and M. Zelkowitz, "What we have learned about fighting defects", *Proc. Eighth IEEE Symposium on Software Metrics*, pp. 249-258, 2002.
- [7] L. Pelayo and S. Dick, "Applying Novel Resampling Strategies to Software Defect Prediction," *NAFIDS 2007-2007 Annual Meeting of the North American Fuzzy Information Processing Society*, San Diego, CA, 2007, pp. 69-72.
- [8] N. V. Chawla, L. O. Hall, K. W. Bowyer, and W. P. Kegelmeyer, "SMOTE: Synthetic minority oversampling technique," *Journal of Artificial Intelligence Research*, vol. 16, pp. 321-357, 2002.
- [9] M. Kubat and S. Matwin, "Addressing the curse of imbalanced training sets: One sided selection," in *Proc. 14th Int. Conf. Mach. Learn.*, 1997, pp. 179–186.
- [10] I. Tomek, "Two Modifications of CNN," *IEEE Transactions on Systems Man and Communications SMC-6* (1976), pp. 769-772
- [11] P. E. Hart, "The Condensed Nearest Neighbor Rule," *IEEE Transactions on Information Theory IT-14* (1968), 515-516
- [12] Sayyad Shirabad, J. and Menzies, T. J. (2005) The PROMISE Repository of Software Engineering Datasets. School of Ottawa, canada. Available: <http://promise.site.uottawa.ca/SERepository>.
- [13] G. Batista, B. Bazzan, and M. Monard, "Balancing Training Data for Automated Annotation of Keywords: a Case Study," in *WOB*, 10-18, 2003.
- [14] G. Batista, R. C. Prati, and M. C. Monard, "A Study of the Behaviour of Several Methods for Balancing Machine Learning Data," *ACM Sigkdd Explorations Newsletter* 6(1), 20-29, 2004.
- [15] C. Seiffert, T. M. Khoshgoftaar, and J. V. Hulse, "Improving software-quality predictions with data sampling and boosting," *IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans*, vol. 39, no. 6, November 2009.
- [16] M. H. Halstead, *Elements of Software Science*, New York: Elsevier North - Holland, 1977.
- [17] T. McCabe, "A Complexity Measure," *IEEE Transactions on Software Engineering*, Vol. 2, No. 4, pp. 308-320, December 1976.
- [18] N.E. Fenton and S. Pfleeger, *Software Metrics: A Rigorous and Practical Approach*, International Thompson Press, 1997.
- [19] M. Shepperd and D. Ince, "A Critique of Three Metrics," *J. Systems and Software*, vol. 26, no. 3, pp. 197-210, Sept. 1994.
- [20] T. Menzies, J. Greenwald, and A. Frank, "Data Mining Static Code Attributes to Learn Defect Predictors," *IEEE Trans. Software Eng.*, vol. 33, no. 1, pp. 2-13, Jan. 2007.
- [21] J. C. Riquelme, R. Ruiz, D. Rodriguez, and J. Moreno, "Finding defective modules from highly unbalanced datasets," *Actas de los Talleres de las Jornadas de Ingenier'ia del Software y Bases de Datos*, vol. 2, no. 1, pp. 67–74, 2008.
- [22] L. Pelayo, and S. Dick, "Applying Novel Resampling Strategies to Software Defect Prediction," In *proceeding of: Fuzzy Information Processing Society*, 2007. ISBN: 1-4244-1213-7.
- [23] J. R. Quinlan. *C4.5: programs for machine learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.
- [24] Vapnik V. N. *Statistical learning theory*, New York: Wiley, 1998.
- [25] W. W. Cohen, "Fast Effective Rule Induction," In *Twelfth International Conference on Machine Learning*, 115-123, 1995.
- [26] I. H. Witten, F. Eibe, and M. A. Hall, *Data Mining: Practical Machine Learning Tools and Techniques*, Morgan Kaufmann Series in Data Management Systems, 3rd ed., Burlington, USA, Morgan Kaufmann Publishers Inc., 2011.