On Strategies for Improving Software Defect Prediction

Rahul Krishna, *Dept. of Electrical and Computer Engineering*
North Carolina State University, Email: rkrish11@ncsu.edu

Abstract—Programming inherently introduces defects into programs, as a result software systems can crash or fail to deliver an important functionality. It is very important to test a software throughly before it can be used. But an extensive testing can be prohibitively expensive or may take too much time to conduct This necessitates the use of automated software defect prediction tools. Although numerous machine learning algorithms are available to detect defects in software, but several factors undermine the accuracy of such algorithm. This paper uses Classification and Regression Trees (CART) and Random Forests to examines two approaches to counter the aforementioned problem. The first approach involves the use Synthetic Minority Oversampling Technique (also known as SMOTE). The second approach attempts to use a metaheursitic algorithm such as differential evolution to find the right set of parameters that can change the performance of the predictor.

Index Terms—Defect Prediction, Machine Learning, Differential Evolution, CART, Random Forest.

1 Introduction

Defect prediction is the study of identifying which software *modules* are defective. Modules refer to some premitive units of an operating systems, like funtions or classes. It needs to be pointed out that early identification of possible defects can lead to a significant reduction in construction costs. No software is developed in a single day, or by just one person, rather it is constructed over time with old modules being extensively reused. Therefore, the sooner defects can be detected and fixed, the less rework is required for development. Boehm and Papaccio [1] for instance mention that reworking software early in its life-cycle is far more cost effective (*by a factor of almost 200*) than doing so later in it's life cycle. This effect has also been reported by several other studies. In their study, Shull *et al.* [2] report that finding a repairing severe software defects is often hudereds of times cheaper if done during the requirements and design phase than doing so after the release. In fact, they claim that *"About 40-50% of user programs enter use with nontrivial defects."*. Authur *et al.* [3] conducted a controlled study with a few engineers at NASA's Langley Research Center, they found the group with a specialized verification team found, (a) More issues, (b) Found them early, which directly translates to lower costs to fix, see [4].

All this leads use to one key conclusion: *Find Bugs Early!* For this we need efficient code analysis measures. We also want them to be generic in that they must be applicable across several projects. Moreover, platforms such as github has over 9 million users, hosting over 21.1 million repositories. Faced with such a massive code base we need these to also be easy to compute. Static code measure is one such tool, it can automatically extracted from a code base with very little effort even for very large software systems [5]. Such measures reduce the effort required for defect prediction. As [6] and [7] have shown, if inspection teams used defect predictors to identify issues, they can find 80% to 88% of the defects after inspecting only 20% to 25% of the code.

With these code analysis measure, we can make use of classification tools form machine learning, such as CART and Random Forest, to detect the presence of defects. However, notice the skewness in the above result, *80% of the problems reside in only 20% of the modules*. This is a key difficulty often faced in software defect prediction. In other words we are trying to predict the occurrence of a defect in a software most of whose modules work just fine. Therefore the classification tool that is used is quite often unable to detect the faulty modules. This is, however, a very well known issue faced by several machine learning experts. In that context, this issue is referred to as class imbalanced in datasets. A data set that is heavily skewed toward the majority class will sometimes generate classifiers that never predict the minority class. In software defect prediction, this bias often makes the classifier highly accurate in predicting non-defects, and totally useless for predicting defects.

One of the other issue in data mining is the choice of parameters that run these classification tools. The parameters of these data miners are rarely tested for the application they are being applied for. A common notion among it's users is that the space of options for these parameters has been well explored by experts and the best settings have been used.

This brings us to the research question that this paper tries to answer.

- **RQ1: Can Over/under sampling techniques such as SMOTE to improve prediction accuracy for defect prediction?** It has been established to be a useful tool.
- **RQ2: Does Tuning a data miner improve it's prediction accuracy?**
- **RQ3: Is tuning performed in conjunction with SMOTE any better than either one performed alone?**

The rest of this paper is organized as follows— Section **??** offers a small ilustration of the impact of SMOTE and tuning on the accuracy of the predictor. Section 2 highlights the underlying principles used in this paper. Section **??** presents the experimental setup followed by section **??** which presents the experimental results and discuss each one. Section **??** contains concluding remarks and finally section **??** talks about the future work.

2 Background Notes

2.1 Defect Prediction

2.2 SMOTE

Data	Symbol	Training	Testing	Training Samples	Defective	% Defective
Ant	ant	1.5, 1.6	1.7	644	124	19.25
Camel	cam	1.2, 1.4	1.6	1480	361	24.39
Ivy	ivy	1.1, 1.4	2.0	352	79	22.44
Jedit	jed	4.1, 4.2	4.3	679	127	18.70
Log4j	log	1.0, 1.1	1.2	244	71	29.09
Lucene	luc	2.0, 2.2	2.4	442	235	53.16
Poi	poi	2.0, 2.5	3.0	699	285	40.77
Synapse	syn	1.0, 1.1	1.2	379	76	20.05
Velocity	vel	1.4, 1.5	1.6	410	289	70.48
Xalan	xal	2.5, 2.6	2.7	1688	798	47.27

Figure 1: Attributes of the defect data sets

amc	average method complexity	e.g. number of JAVA byte codes
avg_cc	average McCabe	average McCabe's cyclomatic complexity seen in class
ca	afferent couplings	how many other classes use the specific class.
cam	cohesion amongst classes	summation of number of different types of method parameters in every method divided by a multiplication of number of different method parameter types in whole class and number of methods.
cbm	coupling between methods	total number of new/redefined methods to which all the inherited methods are coupled
cbo	coupling between objects	increased when the methods of one class access services of another.
ce	efferent couplings	how many other classes is used by the specific class.
dam	data access	ratio of the number of private (protected) attributes to the total number of attributes
dit	depth of inheritance tree	
ic	inheritance coupling	number of parent classes to which a given class is coupled (includes counts of methods and variables inherited)
lcom	lack of cohesion in methods	number of pairs of methods that do not share a reference to an instance variable.
locm3	another lack of cohesion measure	if m, a are the number of $methods, attributes$ in a class number and $\mu(a)$ is the number of methods accessing an attribute, then $lcom3 = ((\frac{1}{a}\sum_j^a \mu(a_j)) - m)/(1-m)$.
loc	lines of code	
max_cc	maximum McCabe	maximum McCabe's cyclomatic complexity seen in class
mfa	functional abstraction	number of methods inherited by a class plus number of methods accessible by member methods of the class
moa	aggregation	count of the number of data declarations (class fields) whose types are user defined classes
noc	number of children	
npm	number of public methods	
rfc	response for a class	number of methods invoked in response to a message to the object.
wmc	weighted methods per class	
defect	defect	Boolean: where defects found in post-release bug-tracking systems.

Figure 2: OO measures used in our defect data sets. Last line is the dependent attribute (whether a defect is reported to a post-release bug-tracking system).

Rank	Treatment	Med	IQR	
1	RF	41.0	3.0	
2	CART	44.0	3.0	
3	CART (SMOTE)	70.0	2.0	
4	RF (SMOTE)	78.0	1.0	

(a) ant

Rank	Treatment	Med	IQR	
1	RF	39.0	1.0	
2	CART	43.0	2.0	
3	CART (SMOTE)	56.0	2.0	
4	RF (SMOTE)	60.0	2.0	

(b) Camel

Rank	Treatment	Med	IQR	
1	RF (SMOTE)	0.0	0.0	
1	CART (SMOTE)	15.0	15.0	
2	RF	50.0	1.0	
3	CART	56.0	1.0	

(c) Ivy

Rank	Treatment	Med	IQR	
1	RF	0.0	0.0	
1	CART (SMOTE)	84.0	1.0	
1	RF (SMOTE)	88.0	1.0	
1	CART	93.0	0.0	

(d) Jedit

Rank	Treatment	Med	IQR	
1	CART	36.0	3.0	
1	RF	40.0	4.0	
2	RF (SMOTE)	53.0	6.0	
2	CART (SMOTE)	54.0	4.0	

(e) POI

Rank	Treatment	Med	IQR	
1	RF (SMOTE)	2.0	2.0	
2	CART (SMOTE)	14.0	5.0	
3	RF	22.0	2.0	
4	CART	41.0	2.0	

(f) Log4j

Rank	Treatment	Med	IQR	
1	CART	47.0	1.0	
2	RF	51.0	1.0	
2	CART (SMOTE)	50.0	4.0	
3	RF (SMOTE)	56.0	3.0	

(g) Lucene

Rank	Treatment	Med	IQR	
1	RF	51.0	0.0	
1	CART	53.0	0.0	
1	CART (SMOTE)	56.0	10.0	
1	RF (SMOTE)	56.0	1.0	

(h) PBeans

Rank	Treatment	Med	IQR	
1	CART (SMOTE)	63.0	1.0	
2	RF (SMOTE)	68.0	2.0	
3	CART	70.0	2.0	
3	RF	70.0	2.0	

(i) Velocity

Rank	Treatment	Med	IQR	
1	RF	24.0	1.0	
2	CART	52.0	18.0	
2	CART (SMOTE)	59.0	2.0	
2	RF (SMOTE)	60.0	1.0	

(j) Xalan

Table 1: Performance scores (g values) for the data sets.

Algorithm 1 Pesudocode for DE with Early Termination

Require: $np = 10,\ f = 0.75,\ cr = 0.3,\ life = 5,\ Goal \in \{pd, f, ...\}$
Ensure: S_{best}

```
function DE(np, f, cr, life, Goal)
    Population ← InitializePopulation(np)
    S_best ← GetBestSolution(Population)
    while life > 0 do
        NewGeneration ← ∅
        for i = 0 → np − 1 do
            S_i ← Extrapolate(Population[i], Population, cr, f)
            if Score(S_i) ≥ Score(Population[i]) then
                NewGeneration.append(S_i)
            else
                NewGeneration.append(Population[i])
            end if
        end for
        Population ← NewGeneration
        if ¬ Improve(Population) then
            life −= 1
        end if
        S_best ← GetBestSolution(Population)
    end while
    return S_best
end function
function SCORE(Candidate)
    set tuned parameters according to Candidate
    model ← TrainLearner()
    result ← TestLearner(model)
    return Goal(result)
end function
function EXTRAPOLATE(old, pop, cr, f)
    a, b, c ← threeOthers(pop, old)
    newf ← ∅
    for i = 0 → np − 1 do
        if cr < random() then
            newf.append(old[i])
        else
            if typeof(old[i]) == bool then
                newf.append(not old[i])
            else
                newf.append(trim(i,(a[i] + f ∗ (b[i] − c[i]))))
            end if
        end if
    end for
    return newf
end function
```

2.3 The Classifiers

2.4 Differential Evolution

3 Experimental Setup

3.1 Data Sets

4 Experimental Results

4.1 SMOTE*ing* improves Prediction Accuracy

4.2 Tuning Also improves Prediction Accuracy (?)

4.3 SMOTE*ing*+Tuning improves Prediction Accuracy (?)

5 Conclusions

References

[1] B. W. Boehm and P. N. Papaccio, "Understanding and controlling software costs," *IEEE Trans. Softw. Eng.*, vol. 14, no. 10, pp. 1462–1477, Oct. 1988. [Online]. Available: http://dx.doi.org/10.1109/32.6191

[2] F. Shull, V. Basili, B. Boehm, A. W. Brown, P. Costa, M. Lindvall, D. Port, I. Rus, R. Tesoriero, and M. Zelkowitz, "What we have learned about fighting defects," in *Software Metrics, 2002. Proceedings. Eighth IEEE Symposium on*. IEEE, 2002, pp. 249–258.

[3] J. D. Arthur, M. K. Groner, K. J. Hayhurst, and C. M. Holloway, "Evaluating the effectiveness of independent verification and validation," *Computer*, vol. 32, no. 10, pp. 79–83, 1999.

[4] J. Dabney, G. Barber, and D. Ohi, "Predicting software defect function point ratios using a bayesian belief network," in *Proceedings of the PROMISE workshop*, vol. 2006, 2006.

[5] N. Nagappan and T. Ball, "Static analysis tools as early indicators of pre-release defect density," in *Proceedings of the 27th international conference on Software engineering*. ACM, 2005, pp. 580–586.

[6] A. Tosun, A. B. Bener, and R. Kale, "Ai-based software defect predictors: Applications and benefits in a case study." in *IAAI*, 2010.

[7] T. J. Ostrand, E. J. Weyuker, and R. M. Bell, "Where the bugs are," in *ACM SIGSOFT Software Engineering Notes*, vol. 29, no. 4. ACM, 2004, pp. 86–96.