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, 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, Differential Evolution, CART, Random Forest, SMOTE.**

I. 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 and 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 be 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 measures, we can make use of classification tools form machine learning 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 a very well known issue faced by several machine learning experts and is referred to as class imbalanced in datasets. A data set that is heavily skewed toward the majority class will

- Statistical classifiers: Linear discriminant analysis, Quadratic discriminant analysis, Logistic regression, Naive Bayes, Bayesian networks, Least-angle regression, Relevance vector machine,
- *Nearest neighbor methods:* k-nearest neighbor, K-Star
- *Neural networks:* Multi-Layer Perceptron, Radial bias functions,
- *Support vector machine-based classifiers:* Support vector machine, Lagrangian SVM Least squares SVM, Linear programming, Voted perceptron,
- *Decision-tree approaches:* C4.5, CART, Alternating DTs
- *Ensemble methods:* **Random Forest**, Logistic Model Tree.

Figure 1: Software Defect Predictors

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 issues 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 is not necessarily true and that brings us to the research questions that this paper tries to answer:

RQ1: Can over/under sampling techniques such as SMOTE be used to improve prediction accuracy for defect prediction?
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?
RQ4: Is the SMOTE*ing* technique limited only to defect prediction?

The rest of this paper is organized as follows—Section II highlights the underlying principles used in this paper. Section III introduces the datasets used. Section IV presents the experimental setup followed by section V whichents the experimental results and discuss each one. Section VI contains concluding remarks.

II. Background Notes

This section briefly highlights the essential tools and methodologies discussed in this paper. Topics include Defect Prediction and Tuning with Differential Evolution.

A. Defect Prediction

The previous section presented key findings from literature that seem to conclude that defect prediction is quite important. This section introduces the idea of using instance based approach to identify defects. Assessing the quality of solutions to a real-world problem can be extremely hard [8]. In several software engineering applications, researchers have models that can emulate the problem, for instance there is the COCOMO effort model [9, p29-57], the COQUALMO defect model [9, p254-268], Madachys schedule risk model [9, p284-291], to name a few. Using these models it is possible to examine several scenarios in a short period of time, and this can be done in a reproducible manner. However, models aren't always the solution, as we shall see.

There exist several problems where models are hard to obtain, or the input and output are related by complex connections that simply cannot be modeled in a reliable manner, or generation of reliable models take prohibitively long [10]. Software defect prediction is an excellent example of such a case. Models that incorporate all the intricate issues that may lead defects in a product is extremely hard to come by. Moreover, it has been shown that models for different regions within the same data can have very different properties [11]. This makes it extremely hard for one to design planning systems that are capable of mitigating these defects.

An alternative is the use of an instance based approach, an subset of case based reasoning strategy, instead of the conventional model based approach. Instance based approaches are used extensively by the effort estimation community. For more reference, see [12]–[16]. This approach has been proposed as an alternative to closed form mathematical models or other modeling methods such as regression [12]. There are several other reasons for instance based approaches being a useful tool, see [13]. As pointed out by [14] it can be used with

partial knowledge of the target project at an early stage which could be a very useful tool in preventing software defects. Instance based approaches are also rather robust in handling cases with sparse samples [17]. All these features are desirable and suggest that instance-based approach is a useful adjunct to traditional model based approach.

A recent IEEE TSE paper by Lessmann et al. [18] compared 21 different learners for software defect prediction, listed in figure 1. They concluded that Random Forrest was the best method, CART being the worst. As a result of this conclusion, this paper used Random Forest and CART as the classifiers to verify how handling class imbalance helps improve the prediction accuracy.

Random Forest is an ensemble learning scheme that constructs a number of decision trees at the training time, for a test instance it outputs the mode of the classes of individual tree. It's patent from how random forest operates that the prediction will suffer if there is an imbalance in classes during the training. Unfortunately, the data sets explored here do suffer from severe skewness, as highlighted in Figure 4.

SMOTE: A study conducted by Pelayo and Dick [19] inspected this issue. They showed that the SMOTE technique [20] can be used to improve recognition of defect-prone modules. SMOTE is an over-sampling technique in which the minority class is over-sampled by creating synthetic examples. This over sampling technique works by introducing synthetic examples for each minority class sample along the plane connecting any (all) of the k minority class nearest neighbors. This is followed by randomly removing samples from the majority class population until there is set number of samples. Finally, the classifiers are learned on the datasets prepared by SMOTE*ing* the minority class and decimating the majority class.

Algorithm 1 Pesudocode for SMOTE

Algorithm *SMOTE*(T, N, k)
Input: Number of minority class samples T; Amount of SMOTE $N\%$; Number of nearest neighbors k
Output: $(N/100)$ * T synthetic minority class samples
(∗ *If N is less than 100%, randomize the minority class samples as only a random percent of them will be SMOTEd.* ∗)
if $N < 100$
  then Randomize the T minority class samples
    $T = (N/100) * T$
    $N = 100$
endif
$N = (int)(N/100)$ (∗ *The amount of SMOTE is assumed to be in integral multiples of 100.* ∗)
k = Number of nearest neighbors
$numattrs$ = Number of attributes
$Sample[\,][\,]$: array for original minority class samples
$newindex$: keeps a count of number of synthetic samples generated, initialized to 0
$Synthetic[\,][\,]$: array for synthetic samples
(∗ *Compute k nearest neighbors for each minority class sample only.* ∗)
for $i \leftarrow 1$ **to** T
    Compute k nearest neighbors for i, and save the indices in the $nnarray$
    Populate(N, i, $nnarray$)
endfor

Populate(N, i, $nnarray$) (∗ *Function to generate the synthetic samples.* ∗)
while $N \neq 0$
    Choose a random number between 1 and k, call it nn. This step chooses one of the k nearest neighbors of i.
    for $attr \leftarrow 1$ **to** $numattrs$
      Compute: $dif = Sample[nnarray[nn]][attr] - Sample[i][attr]$
      Compute: gap = random number between 0 and 1
      $Synthetic[newindex][attr] = Sample[i][attr] + gap * dif$
    endfor
    $newindex$++
    $N = N - 1$
endwhile
return (∗ *End of Populate.* ∗)

B. Parameter Tuning

The classifiers (referred to as *predictors* henceforth) being used in this paper are both tree based learners. The recurse on their splits. Their output is a boolean value `[True/False]`. But make this decision, they use a lot of parameters. Figure 3 shows all the parameters that matter for defect prediction. In the usual case, the user is expected to supply these values. The goal of **RQ2** is simply to use a metaheuristic[1] algorithm to automate this tuning process. There are very many techniques to do this otherwise, without using Metaheuristics, like using gradient descent optimizers [21]. There are also simpler techniques like Simulated annealing which is often used in search-based SE e.g. [22], [23]. Another popular technique is Differential Evolution [24]; there are a lot more methods.

This paper uses make an engineering decision to use DE. In fact, DEs have been applied before for parameter tuning (e.g. see [25], [26]). A recent, as of yet unpublished, paper by Fu et al. [27] also makes a case for DE as a tuner for defect prediction.

Differential Evolution: The psuedocode for differential evolution is shown in Algorithm 2. Note that, as the algorithm is described, any superscript number denotes a line in that algorithm. DE is an evolutionary algorithm where the next *Generation* is learnt from the current *Population*. If the new is not any better than the current instance, then a *life* is lost (terminating when lives is zero)[5]. Each candidate solution in the *Population* is a pair of *(Tunings,*

[1] http://en.wikipedia.org/wiki/Metaheuristic

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).

Learner Name	Parameters	Default	Tuning Range	Description
CART	threshold	0.5	[0,1]	The value to determine defective or not.
	max_feature	None	[0.01,1]	The number of features to consider when looking for the best split.
	min_sample_split	2	[2,20]	The minimum number of samples required to split an internal node.
	min_samples_leaf	1	[1,20]	The minimum number of samples required to be at a leaf node.
	max_depth	None	[1, 50]	The maximum depth of the tree.
Random Forests	threshold	0.5	[0.01,1]	The value to determine defective or not.
	max_feature	None	[0.01,1]	The number of features to consider when looking for the best split.
	max_leaf_nodes	None	[1,50]	Grow trees with max_leaf_nodes in best-first fashion.
	min_sample_split	2	[2,20]	The minimum number of samples required to split an internal node.
	min_samples_leaf	1	[1,20]	The minimum number of samples required to be at a leaf node.
	n_estimators	100	[50,150]	The number of trees in the forest.

Figure 3: List of parameters to be tuned.

Scores). Tunings are selected from 3 and *Scores* come from training a learner using those parameters and applying it test data[23−28].

The premise of DE is that the best way to mutate existing tunings is to *Extrapolate*[29] between current solutions. Three solutions a, b, c are selected at random. For each tuning parameter i, at some probability *cr*, we replace the old tuning x_i with y_i:

- (For numerics) $y_i = a_i + f \times (b_i - c_i)$ where f is a parameter controlling the cross-over amount. The *trim* function[39] limits the new value to the legal range min..max of that parameter.
- (For booleans) $y_i = \neg x_i$ (see line 37).

The main loop of DE[7] runs over the *Population*, replacing old items with new *Candidate*s (if the new candidate is better than the old item). This means that, as the loop progresses, the *Population* is full of increasiningly more valuable solutions. This, in turn, also improves the candidates, which are generated from the *Population*.

For the experiments of this paper, we collect performance values from a data miner, from which a *Goal* function extracts one performance value[27] (so this code is rerun many times, each time with a different *Goal*[2]). Technically, this makes a *single objective* DE (and for notes on multi-objective DEs, see [28]–[30]).

Algorithm 2 Pesudocode for DE with Early Termination

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

```
1:
2: function DE(np, f, cr, life, Goal)
3:     Population ← InitializePopulation(np)
4:     S_best ←GetBestSolution(Population)
5:     while life > 0 do
6:         NewGeneration ← ∅
7:         for i = 0 → np − 1 do
8:             S_i ← Extrapolate(Population[i], Population, cr, f)
9:             if Score(S_i)≥Score(Population[i]) then
10:                NewGeneration.append(S_i)
11:            else
12:                NewGeneration.append(Population[i])
13:            end if
14:        end for
15:        Population ← NewGeneration
16:        if ¬ Improve(Population) then
17:            life− = 1
18:        end if
19:        S_best ← GetBestSolution(Population)
20:    end while
21:    return S_best
22: end function
23: function SCORE(Candidate)
24:    set tuned parameters according to Candidate
25:    model ←TrainLearner()
26:    result ←TestLearner(model)
27:    return Goal(result)
28: end function
29: function EXTRAPOLATE(old, pop, cr, f)
30:    a, b, c ← threeOthers(pop, old)
31:    newf ← ∅
32:    for i = 0 → np − 1 do
33:        if cr < random() then
34:            newf.append(old[i])
35:        else
36:            if typeof(old[i]) == bool then
37:                newf.append(not old[i])
38:            else
39:                newf.append(trim(i,(a[i] + f ∗ (b[i] − c[i]))))
40:            end if
41:        end if
42:    end for
43:    return newf
44: end function
```

Data	Symbol	Training Version	Testing Version	Training Samples	Bugs	% Defective
Ant	ant	1.5, 1.6	1.7	644	124	19.25
Ivy	ivy	1.1, 1.4	2.0	352	79	22.44
Jedit	jed	4.1, 4.2	4.3	679	127	18.70
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 4: Attributes of the defect data sets

III. DATA SETS

The following section describes the experimental rig and the experiments used to measure the performance of the defect predictors on 8 data sets. And, 1 security flaw dataset.

A. Defect Data Set

The data for defect prediction was obtained from the PROMISE repository[2]. For the defect data, this work investigated 24 releases from 8 open source Java projects. These projects are characterized by the metrics highlighted in figure 2. The datasets include *Apache Ant* (1.5 – 1.7), *Apache Camel* (1.2 – 1.6), *Apache Ivy* (1.1 – 2.0), *JEdit* (4.1 – 4.3), *Apache Log4j* (1.0 – 1.2), *Apache Lucene* (2.0 – 2.2), *PBeans* (1.0 and 2.0), *Apache POI* (2.0 – 3.0), *Apache Synapse* (1.0 – 1.2), *Apache Velocity* (1.4 – 1.6), and *Apache Xalan-Java* (2.5 – 2.7).

Given the empirical nature of the data, it is important to design an experiment such that the prediction phase uses only the *past* data to learn trends which can then be applied to the *future* data. Thus for the experiment data sets that have at least two consecutive releases are used.

- To predict for defects in release i, the predictor uses releases releases $(i-1)$ and $(i-2)$.
- The tuning uses releases $(i-1)$ and $(i-2)$ to tune. Therein, the paper uses $(i-2)$ as training and $(i-2)$ for testing the scheme.

The attributes of the datasets being used have been summarized in figure 4.

[2]Promise Repository: http://openscience.us/repo

B. Bugzilla Flaw Dataset

To perform replication studies, there is a need to collect a large number of stack traces from the product, which in this case is firefox. Unfortunately, because of the requirement to use a historical set of traces due to security data availability, we could not make use of Mozillas primary stack trace data website, Mozilla Crash Reports[3]. Instead, the historical dumps[4] were used. This historical dataset contains a random sampling of stack trace data (approximately 10% of the crashes seen by the crash reporting system), sorted by day. The analysis were performed on crashes occurring from May 2010 to March 2012 due to the available security data. These dumps do not contain the entirety of the stack trace; rather, only the topmost filename is included in each trace. We further pruned the dataset to only crashes on the first of every month in the time period. In the end, 1,013,770 occurrences of files in stack traces were recorded.

IV. Experimental Setup

A. The Rig

The experimental rig uses an *oracle* to determine weather a certain test case is defective or not. The oracle has 2 components namely, the trainer and the prediction tool. The trainer is either SMOTE, or DE, or both.

B. Statistical Measures

Let $\{A, B, C, D\}$ denote the true negatives, false negatives, false positives, and true positives (respectively) found by a binary detector. Certain standard measures can be computed from A, B, C, D:

$$\begin{array}{ll} pd = recall = & D/(B+D) \\ pf = fallout = & C/(A+C) \\ prec = precision = & D/(D+C) \\ F = & 2 * pd * prec/(pd + prec) \\ G = & 2 * pd * (1-pf)/(1+pd-pf) \end{array}$$

All the above vary from zero to one. Following this need to be highlighted:

- For pf (*Fallout*), the *better* scores are *smaller*.
- For all other scores, the *better* scores are *larger*.
- *prec*, *G*, and *F* refer to *both* the defect and non-defective modules. This is different to *pf* and *recall* which only refer to either non-defective or defective modules (repsectively).

In order to assess the performance of the prediction scheme for the defect data set, we use the G measure from above. This measure is particularly useful because it gives a unified number, a combination of recall and fallout, that characterizes the performance of the predictor.

Notice that G is a harmonic mean of sensitivity and specificity, which are given by:

$$\begin{aligned} \text{Recall} &= \frac{\text{Number of true positives}}{\text{Total no. of defective modules}} \\ &= \text{probability of a non-defect, given that the prediction is negative} \end{aligned}$$

$$\begin{aligned} \text{specificity} &= \frac{\text{Number of true negatives}}{\text{Total non-defective modules}} \\ &= \text{probability of defect, given that the prediction is positive} \end{aligned}$$

In the context of our application, we want to have both high sensitivity and high specificity at the same time. And is why G is an appropriate measure. Since it is a harmonic mean, it is always less than the least among sensitivity and specificity. Therefore, a high G implies that both sensitivity and specificity are higher than G itself, as a result simplifying the analysis.

In addition to the above, we rank the different variants of the planning scheme to identify the best approach. We make use of the Scott-Knott procedure, recommended by Mittas & Angelis in their 2013 IEEE TSE paper [31], to compute the ranks. It works as follows: A list of treatments *l* is sorted by the Medianian score. The list *l* is then split into sub-lists m, n in order to maximize the expected value of the differences in the observed performance before and after division. A statistical hypothesis test *H* is applied on the splits *m, n* to check if they are statistically different. If so, Skott-Knott then recurses on each division.

The research conducted by Shepperd and Mac-

[3]https://crash-stats.mozilla.com/home/products/Firefox
[4]https://crash-analysis.mozilla.com/crash_analysis/

Donell [32], Kampenes [33] and Kocaguenli et al. [34], highlighted that an "effect size" in lieu of a mere hypothesis test is required in order to verfiy if two populations are "significantly" different. An ICSE'11 paper by Arcuri [35] endorsed the use of Vargha and Delaney's A12 effect size for reporting results in software engineering. Thus, for hypothesis testing H in Skott-Knott, we use the A12 test and a non-parametric bootstrap sampling [36].

V. Experimental Results

This section discusses the experimental results for the experiments. In particular, it is broken down into 2 experiments. Experiment 1, deals with the defect dataset, an answers **RQ1, RQ2, RQ3**, while Experiment 2 deals with the security flaw data set and answers **RQ4**. The **Research Questions *(RQs)*** can be found in section I.

Experiment 1: Defect Dataset

The defect dataset was prepared as mentioned in the previous section. In total there were 8 datasets. There were 23 columns in each dataset. The first 22 columns each correspond to the CK and OO metrics, highlighted in Figure 2, the last column was the defects. The data is split into training and testing, figure 4 shows the version used in each data set for this. Following this, each dataset was processed with the following treatments:

- CART was trained with the training data and defects were predicted from the testing data. All statistical measures, see section IV-B, were obtained by comparing the original with the prediction.
- Likewise, Random Forest was used to do the same process.
- Differential Evolution was used to tune the attributes of RF and CART. Then the predictions were obtained using the tuned predictors.
  Note: tuning was done with only the training dataset, the test dataset remains unseen by the predictors.
- Then, for the *untuned* CART and Random Forest, the training data was treated by SMOTE*ing* and the statistical measures were obtained.
- Lastly, the predictors were *Tuned* and training data was SMOTE*ed* prior to using the predictors.

It is worth mentioning at this point that the above process was repeated more than 20 times for all cases in order to overcome any measurement biases. The results of all the statistical measures are expressed as median and interquartile ranges obtained from the 20 repetitions for each the 8 data, refer to Appendix A.

The results are rather voluminous, with over 10 columns. This makes analysis prohibitively hard. Therefore, as suggested by section IV-B, using only the G scores could aid the analysis. It needs to be reiterated that since the G measure comprises of both the sensitivity and specificity, it summarizes the predictor's ability to predict both defects and non-defects accurately.

The G scores are summarized in figure 5. They are presented in a tabular format, with Skott-Knott ranks as the first column, and the quartile charts as the last. *The larger the Rank, the better the Performance*. It needs to be noted that sometimes, two or more treatments are ranked the same even if they are unequal. This is because they are not *statistically different* and they must be treated as equal by the reader.

RQ1: Can over/under sampling techniques such as SMOTE to improve prediction accuracy for defect prediction? *Yes.*

In 6 out of the 8 datasets, RF proved to be better than CART as was reported by Lessmann et al. [18]. In 4 out of the 6 datasets where RF was the better predictor, SMOTE*ing* ranked better than raw training data. Thus, *SMOTEing does improve prediction accuracy*

RQ2: Does Tuning a data miner improve it's prediction accuracy? *Not always. Works in only a few datasets.*

Just tuning the predictor seems only to help in 4 out of the 8 datasets. This was to be expected however because merely tuning a predictor is not going to change the nature of the dataset. If the dataset is skewed, it is going to pose the same problem to the predictor regardless of it being tuned or not.

ant

Rank	Treatment	Median	IQR
1	CART(SMOTE)	59.0	3.0
2	RF	61.0	0.0
2	CART	63.0	0.0
3	CART(SMOTE,Tune)	67.0	0.0
3	RF(Tune)	68.0	2.0
4	RF(SMOTE,Tune)	70.0	1.0
5	CART(Tune)	71.0	0.0
5	RF(SMOTE)	71.0	1.0

ivy

Rank	Treatment	Median	IQR
1	CART(SMOTE)	38.0	0.0
1	CART(Tune)	39.0	0.0
1	RF(Tune)	39.0	0.0
2	RF	46.0	7.0
3	RF(SMOTE,Tune)	54.0	6.0
3	CART	56.0	0.0
3	RF(SMOTE)	56.0	0.0
3	CART(SMOTE,Tune)	59.0	7.0

jedit

Rank	Treatment	Median	IQR
1	CART(SMOTE,Tune)	60.0	11.0
2	RF(SMOTE,Tune)	66.0	1.0
2	CART(SMOTE)	67.0	0.0
2	RF	67.0	0.0
2	RF(Tune)	67.0	0.0
3	RF(SMOTE)	69.0	0.0
4	CART(Tune)	72.0	0.0
5	CART	85.0	0.0

lucene

Rank	Treatment	Median	IQR
1	CART(Tune)	53.0	0.0
2	CART(SMOTE)	57.0	0.0
2	CART	58.0	0.0
3	RF(SMOTE)	59.0	2.0
3	RF(SMOTE,Tune)	59.0	1.0
3	CART(SMOTE,Tune)	60.0	5.0
4	RF	61.0	1.0
5	RF(Tune)	65.0	1.0

poi

Rank	Treatment	Median	IQR
1	CART	46.0	0.0
2	CART(SMOTE)	56.0	4.0
2	CART(SMOTE,Tune)	56.0	6.0
2	RF(SMOTE)	60.0	4.0
3	RF	62.0	3.0
3	CART(Tune)	63.0	0.0
3	RF(Tune)	63.0	8.0
4	RF(SMOTE,Tune)	65.0	3.0

synapse

Rank	Treatment	Median	IQR
1	RF(Tune)	42.0	2.0
2	RF	45.0	0.0
3	CART(SMOTE)	48.0	0.0
4	CART	52.0	0.0
4	CART(Tune)	52.0	0.0
5	RF(SMOTE)	56.0	0.0
6	RF(SMOTE,Tune)	57.0	2.0
6	CART(SMOTE,Tune)	58.0	0.0

velocity

Rank	Treatment	Median	IQR
1	CART(SMOTE)	42.0	4.0
2	CART	45.0	0.0
3	CART(SMOTE,Tune)	48.0	0.0
3	CART(Tune)	50.0	0.0
4	RF(SMOTE)	52.0	1.0
4	RF(SMOTE,Tune)	52.0	3.0
5	RF(Tune)	55.0	2.0
5	RF	56.0	1.0

xalan

Rank	Treatment	Median	IQR
1	CART(Tune)	57.0	0.0
1	RF(Tune)	57.0	0.0
2	RF	59.0	1.0
2	CART(SMOTE)	60.0	3.0
3	CART(SMOTE,Tune)	64.0	6.0
3	CART	65.0	0.0
4	RF(SMOTE)	69.0	2.0
4	RF(SMOTE,Tune)	69.0	1.0

Figure 5: G-Scores for the Defect Prediction Datasets

With respect to the 4 datasets where it does work, there is a possibility that the data is not only skewed other factors affect the prediction accuracy. And this needs to be studied.

RQ3: Is tuning performed in conjunction with SMOTE any better than either one performed alone? *Yes.*

Before SMOTE:

Rank	Name	Median	IQR	
1	Accuracy	99.00	2.00	•
1	Recall	100.00	3.00	•
1	**Specificity**	**8.00**	**4.00**	•
1	Precision	99.00	3.00	•
1	F	100.00	2.00	•
1	**G**	**15.00**	**3.00**	•

After SMOTE:

Rank	Name	Median	IQR	
1	Accuracy	80.00	2.00	•
1	Recall	77.00	3.00	•
1	**Specificity**	**81.00**	**4.00**	•
1	Precision	78.00	3.00	•
1	F	77.00	2.00	•
1	**G**	**80.00**	**3.00**	•

Figure 6: Performance Scores Before and After SMOTE for the Bugzilla Security Flaw Dataset

This is a particularly interesting result. In 5 out of the 8 dataset, the use of both as beneficial. In ant RF+SMOTE+Tune is almost as good as RF+SMOTE, and definitely better than plain RF. Similarly, in ivy, poi, synapse and xalan the results hold true.

Note that, with the exception of jedit, using tools such as DE and SMOTE to prepare both the predictor and the training data is beneficial.

A. Experiment 2: Bugzilla Secturity Flaw Dataset

This was one contiguous data set with 60000 instance of which only 391 were security flaws. This a classic example of skewness in the data.

Since there is only one dataset in this category, this paper tested the statistical validity of the predictors by performing a 5×5 cross validation study. For this, the data was randomly grouped into sets of 5, one of the set was used as testing and other 4 are used as training, then the same is repeated for the 4 remaining datasets. This entire process is repeated 5 more times. Therefore, a total of 25 evaluations of the stats are performed. They are then tabulated using the median and interquartile spread.

Although the SMOTE algorithm was applied to this dataset, it was slightly altered to fit the application. In usual SMOTE, as previously discussed, minority classes are synthetically oversampled and the majority class is randomly undersampled by a certain fraction. Due to the nature of the minority class in this dataset, it was reasoned that creating new rows would translate to inserting artificial security flaw which might not be fair. Therefore, for this dataset, only the majority class was under sampled and the minority class was unaltered. Moreover, it was ensured that the number of majority and minority class samples are the same.

RQ4: Can SMOTE*ing* technique be applied to other Software Engineering paradigms? Yes.

As was reported in the previous papers, a simple Random Forest on the dataset without any sort of preprocessing produce true appalling results, see Figure 6.

However, once the data was preprocessed by undersampling the majority class, the prediction performance improved drastically. In fact, the specificity, which measures the probability of security flaw given the predictor tests positive, rose from just 0.08 (80%) before sampling to 0.81 (81%) after sampling. That's an order of magnitude better.

VI. Conclusions

This work has shown that the use of off-the-shelf predictors is not a wise approach for software defect prediction. The results show that one must reflect on the nature of the data they are handling and the reason a specific predictor is being used. In general, result have seem to conclude that one of the tree based classifier RF is better than the other. In addition, use of a preprocessing tool to prepare the dataset and a mechanism to determine the attributes of a classifier are important. More importantly, the use of both in general is also quite beneficial. It is true that the data sets being handled here are open-source projects that may not have adhered to any strict development strategy, this adds another dimension to what causes defects. This can't be handled by machine learning. It is therefore advisable that one conduct as test such as this to see what the best approach to defect prediction. A strong statistical reasoning is also pertinent to making managerial

decision regarding a project.

Another purpose of this paper was to test if the sampling helps improve security flaws. The results are very positive and further affirm the need for data preprocessing before any sort of predictive analytics is performed.

ACKNOWLEDGMENTS

The author would like to thank Mr. Chris Theisen for providing the Bugzilla security flaw dataset.

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.

[8] T. Menzies, "Xomo: Understanding development options for autonomy."

[9] B. W. Boehm, C. Abts, A. W. Brown, S. Chulani, B. K. Clark, E. Horowitz, R. Madachy, D. J. Reifer, and B. Steece, "Software cost estimation with cocomo ii," 2009.

[10] J. Ludewig, "Models in software engineering - an introduction," *Software and Systems Modeling*, vol. 2, no. 1, pp. 5–14, 2003. [Online]. Available: http://link.springer.com/10.1007/s10270-003-0020-3

[11] T. Menzies, A. Butcher, D. Cok, A. Marcus, L. Layman, F. Shull, B. Turhan, and T. Zimmermann, "Local versus global lessons for defect prediction and effort estimation," *Software Engineering, IEEE Transactions on*, vol. 39, no. 6, pp. 822–834, June 2013.

[12] J. W. Keung, B. A. Kitchenham, and D. R. Jeffery, "Analogy-x: providing statistical inference to analogy-based software cost estimation," *Software Engineering, IEEE Transactions on*, vol. 34, no. 4, pp. 471–484, 2008.

[13] T. Menzies, A. Brady, J. Keung, J. Hihn, S. Williams, O. El-Rawas, P. Green, and B. Boehm, "Learning project management decisions: A case study with case-based reasoning versus data farming," *Software Engineering, IEEE Transactions on*, vol. 39, no. 12, pp. 1698–1713, Dec 2013.

[14] F. Walkerden and R. Jeffery, "An empirical study of analogy-based software effort estimation," *Empirical software engineering*, vol. 4, no. 2, pp. 135–158, 1999.

[15] M. Shepperd and C. Schofield, "Estimating software project effort using analogies," *Software Engineering, IEEE Transactions on*, vol. 23, no. 11, pp. 736–743, 1997.

[16] E. Kocaguneli, G. Gay, T. Menzies, Y. Yang, and J. W. Keung, "When to use data from other projects for effort estimation," in *Proceedings of the IEEE/ACM international conference on Automated software engineering*. ACM, 2010, pp. 321–324.

[17] I. Myrtveit, E. Stensrud, and M. Shepperd, "Reliability and validity in comparative studies of software prediction models," *Software Engineering, IEEE Transactions on*, vol. 31, no. 5, pp. 380–391, May 2005.

[18] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch, "Benchmarking classification models for software defect prediction: A proposed framework and novel findings," *Software Engineering, IEEE Transactions on*, vol. 34, no. 4, pp. 485–496, July 2008.

[19] L. Pelayo and S. Dick, "Applying novel resampling strategies to software defect prediction," pp. 69–72, June 2007.

[20] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "Smote: synthetic minority over-sampling technique," *Journal of artificial intelligence research*, vol. 16, no. 1, pp. 321–357, 2002.

[21] A. Saltelli, K. Chan, E. M. Scott *et al.*, *Sensitivity analysis*. Wiley New York, 2000, vol. 1.

[22] M. Feather and T. Menzies, "Converging on the optimal attainment of requirements," in *IEEE Joint Conference On Requirements Engineering ICRE'02 and RE'02, 9-13th September, University of Essen, Germany*, 2002, available from http://menzies.us/pdf/02re02.pdf.

[23] T. Menzies, O. El-Rawas, J. Hihn, M. Feather, B. Boehm, and R. Madachy, "The business case for automated software engineerng," in *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. New York, NY, USA: ACM, 2007, pp. 303–312, available from http://menzies.us/pdf/07casease-v0.pdf.

[24] R. Storn and K. Price, "Differential evolution–a simple and efficient heuristic for global optimization over continuous spaces," *Journal of global optimization*, vol. 11, no. 4, pp. 341–359, 1997.

[25] I. Chiha, J. Ghabi, and N. Liouane, "Tuning pid controller with multi-objective differential evolution," in *Communications Control and Signal Processing (ISCCSP), 2012 5th International Symposium on*. IEEE, 2012, pp. 1–4.

[26] M. G. Omran, A. P. Engelbrecht, and A. Salman, "Self-adaptive barebones differential evolution," in *Evolutionary Computation, 2007. CEC 2007. IEEE Congress on*. IEEE, 2007, pp. 2858–2865.

[27] W. Fu and T. Menzies, "Analytics Without Parameter Tuning Considered Harmful?"

[28] T. Robic and B. Filipic, "Demo: Differential evolution for multiobjective optimization," in *Evolutionary Multi-Criterion Optimization*, ser. Lecture Notes in Computer Science, C. Coello, A. Hernndez Aguirre, and E. Zitzler, Eds. Springer Berlin Heidelberg, 2005, vol. 3410, pp. 520–533. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-31880-4_36

[29] Q. Zhang and H. Li, "Moea/d: A multiobjective evolutionary algorithm based on decomposition," *Trans. Evol. Comp*,

vol. 11, no. 6, pp. 712–731, Dec. 2007. [Online]. Available: http://dx.doi.org/10.1109/TEVC.2007.892759

[30] W. Huang and H. Li, "On the differential evolution schemes in moea/d," in *Natural Computation (ICNC), 2010 Sixth International Conference on*, vol. 6, Aug 2010, pp. 2788–2792.

[31] N. Mittas and L. Angelis, "Ranking and clustering software cost estimation models through a multiple comparisons algorithm," *Software Engineering, IEEE Transactions on*, vol. 39, no. 4, pp. 537–551, 2013.

[32] M. J. Shepperd and S. G. MacDonell, "Evaluating prediction systems in software project estimation," *Information & Software Technology*, vol. 54, no. 8, pp. 820–827, 2012.

[33] V. B. Kampenes, T. Dybå, J. E. Hannay, and D. I. K. Sjøberg, "A systematic review of effect size in software engineering experiments," *Information & Software Technology*, vol. 49, no. 11-12, pp. 1073–1086, 2007.

[34] E. Kocaguneli, T. Zimmermann, C. Bird, N. Nagappan, and T. Menzies, "Distributed development considered harmful?" in *ICSE*, 2013, pp. 882–890.

[35] A. Arcuri and L. Briand, "A practical guide for using statistical tests to assess randomized algorithms in software engineering," in *ICSE'11*, 2011, pp. 1–10.

[36] B. T. Efron, "Rj (1993).≪an introduction to the bootstrap≫," *Monograps on Statistics and Applied Probability*, vol. 57.

APPENDIX: A

Ant:

Treatment	TP		FP		FN		TN		Accuracy		Recall		Fallout		Precision		F		G	
	med	irq	med	irq	med	irq	med	irq	med	irq	med	irq	med	irq	med	irq	med	irq	med	irq
RF	16	1	35	2	16	1	226	2	82	1	50	3	13	1	30	2	38	2	63	2
RF(SMOTE)	22	0	63	6	10	0	198	6	75	2	69	0	24	2	26	2	38	2	72	1
RF(Tune)	17	2	29	3	15	2	232	3	85	0	53	6	11	1	37	2	44	3	67	5
RF(SMOTE, Tune)	21	1	74	9	10	1	186	9	71	4	67	3	28	4	22	2	34	2	70	2
CART	16	0	40	0	16	0	221	0	81	0	50	0	15	0	29	0	36	0	63	0
CART(SMOTE)	17	0	88	0	15	0	173	0	65	0	53	0	34	0	16	0	25	0	59	0
CART(Tune)	20	0	49	0	12	0	212	0	79	0	63	0	19	0	29	0	40	0	71	0
CART(SMOTE, Tune)	23	0	99	0	9	0	162	0	63	0	72	0	38	0	19	0	30	0	67	0

Ivy:

Treatment	TP		FP		FN		TN		Accuracy		Recall		Fallout		Precision		F		G	
	med	irq	med	irq	med	irq	med	irq	med	irq	med	irq	med	irq	med	irq	med	irq	med	iqr
RF	4	1	22	1	12	1	203	1	86	1	25	6	10	0	15	2	19	3	39	7
RF(SMOTE)	7	0	48	0	9	0	177	0	76	0	44	0	21	0	13	0	20	0	56	0
RF(Tune)	3	1	21	7	12	1	204	7	85	3	22	6	9	3	14	5	17	5	35	8
RF(SMOTE, Tune)	9	1	56	7	7	1	168	7	73	3	56	6	25	3	12	3	20	4	62	6
CART	7	0	47	0	9	0	178	0	77	0	44	0	21	0	13	0	20	0	56	0
CART(SMOTE)	4	0	55	0	12	0	170	0	72	0	25	0	24	0	7	0	11	0	38	0
CART(Tune)	4	0	36	0	12	0	189	0	80	0	25	0	16	0	10	0	14	0	39	0
CART(SMOTE, Tune)	9	0	84	0	7	0	141	0	62	0	56	0	37	0	10	0	17	0	59	0

Jedit:

Treatment	TP		FP		FN		TN		Accuracy		Recall		Fallout		Precision		F		G	
	med	irq	med	irq	med	irq	med	irq	med	irq	med	irq	med	irq	med	irq	med	irq	med	irq
RF	6	0	67	3	5	0	414	3	85	1	55	0	14	0	8	0	14	0	67	0
RF(SMOTE)	7	0	129	7	4	0	352	7	73	1	64	0	27	2	5	0	10	0	68	1
RF(Tune)	6	0	69	3	5	0	412	3	85	0	55	0	14	1	8	0	14	0	67	0
RF(SMOTE, Tune)	7	0	166	5	4	0	315	5	65	1	64	0	34	1	4	0	8	0	64	1
CART	10	0	100	0	1	0	381	0	79	0	91	0	21	0	9	0	17	0	85	0
CART(SMOTE)	7	0	139	0	4	0	342	0	71	0	64	0	29	0	5	0	9	0	67	0
CART(Tune)	7	0	78	0	4	0	403	0	83	0	64	0	16	0	8	0	15	0	72	0
CART(SMOTE, Tune)	4	2	109	57	7	2	372	57	76	11	36	19	23	12	4	1	6	1	49	11

Lucene:

Treatment	TP		FP		FN		TN		Accuracy		Recall		Fallout		Precision		F		G	
	med	irq	med	irq	med	irq	med	irq	med	irq	med	irq	med	irq	med	irq	med	irq	med	iqr
RF	129	3	59	2	73	3	77	2	61	0	64	1	43	2	68	1	66	0	60	1
RF(SMOTE)	110	6	49	5	93	6	88	5	59	1	54	3	36	3	69	2	61	2	59	2
RF(Tune)	132	3	47	3	71	3	89	3	65	2	65	1	34	2	73	2	69	1	65	2
RF(SMOTE, Tune)	113	10	46	10	89	10	90	10	60	2	56	5	34	8	70	5	62	3	60	3
CART	113	0	54	0	90	0	83	0	58	0	56	0	39	0	68	0	61	0	58	0
CART(SMOTE)	110	2	54	1	93	2	82	1	57	1	54	1	39	1	67	1	60	1	57	0
CART(Tune)	146	0	80	0	57	0	57	0	60	0	72	0	58	0	65	0	68	0	53	0
CART(SMOTE, Tune)	97	28	41	22	106	28	96	22	57	2	48	14	30	16	70	5	57	8	57	5

Poi:

Treatment	TP		FP		FN		TN		Accuracy		Recall		Fallout		Precision		F		G	
	med	irq	med	irq	med	irq	med	irq	med	irq	med	irq	med	irq	med	irq	med	irq	med	iqr
RF	152	14	42	5	129	14	118	5	61	2	54	5	26	3	78	3	64	3	62	2
RF(SMOTE)	151	14	51	5	129	14	110	5	59	5	54	5	32	4	74	3	63	4	60	5
RF(Tune)	150	27	41	18	131	27	120	18	61	6	53	10	25	11	78	5	64	6	62	6
RF(SMOTE, Tune)	150	35	47	10	130	35	113	10	60	7	53	13	29	7	77	4	63	8	60	7
CART	93	0	37	0	188	0	124	0	49	0	33	0	23	0	72	0	45	0	46	0
CART(SMOTE)	151	60	71	8	130	60	89	8	54	13	53	21	44	5	67	7	60	16	53	10
CART(Tune)	162	0	51	0	119	0	110	0	62	0	58	0	32	0	76	0	66	0	63	0
CART(SMOTE, Tune)	157	21	60	6	123	21	101	6	57	7	56	8	37	3	71	5	62	7	57	7

Synapse:

Treatment	TP		FP		FN		TN		Accuracy		Recall		Fallout		Precision		F		G	
	med	irq	med	irq	med	irq	med	irq	med	irq	med	irq	med	irq	med	irq	med	irq	med	iqr
RF	19	0	34	3	41	0	128	3	66	2	32	0	21	2	36	2	34	1	45	1
RF(SMOTE)	25	0	27	0	35	0	135	0	72	0	42	0	17	0	48	0	45	0	56	0
RF(Tune)	18	1	27	3	42	1	135	3	69	2	30	2	17	2	40	5	34	3	44	2
RF(SMOTE, Tune)	27	4	39	5	32	4	122	5	67	1	46	7	24	4	40	2	43	2	57	3
CART	23	0	31	0	37	0	131	0	69	0	38	0	19	0	43	0	40	0	52	0
CART(SMOTE)	22	0	46	0	38	0	116	0	62	0	37	0	28	0	32	0	34	0	48	0
CART(Tune)	24	0	39	0	36	0	123	0	66	0	40	0	24	0	38	0	39	0	52	0
CART(SMOTE, Tune)	31	0	55	0	29	0	107	0	62	0	52	0	34	0	36	0	42	0	58	0

Velocity:

Treatment	TP		FP		FN		TN		Accuracy		Recall		Fallout		Precision		F		G	
	med	irq	med	irq	med	irq	med	irq	med	irq	med	irq	med	irq	med	irq	med	irq	med	iqr
RF	71	4	16	2	76	4	33	2	54	2	48	3	33	4	81	2	61	3	56	2
RF(SMOTE)	59	0	14	1	88	0	35	1	48	1	40	0	29	2	81	2	54	1	51	2
RF(Tune)	69	7	14	4	78	7	34	4	52	5	47	5	30	9	82	3	59	4	55	5
RF(SMOTE, Tune)	59	3	15	1	87	3	33	1	47	4	40	2	32	2	79	4	53	3	50	3
CART	58	0	23	0	89	0	26	0	43	0	39	0	47	0	72	0	51	0	45	0
CART(SMOTE)	61	2	27	8	86	2	21	8	43	3	41	1	56	16	69	5	52	1	43	6
CART(Tune)	60	0	17	0	87	0	32	0	47	0	41	0	35	0	78	0	54	0	50	0
CART(SMOTE, Tune)	57	4	12	2	90	4	37	2	48	2	39	3	24	4	83	1	53	3	51	2

Xalan:

Treatment	TP		FP		FN		TN		Accuracy		Recall		Fallout		Precision		F		G	
	med	irq	med	irq	med	irq	med	irq	med	irq	med	irq	med	irq	med	irq	med	irq	med	iqr
RF	95	1	332	4	15	1	280	4	52	1	86	1	54	1	22	0	35	0	60	1
RF(SMOTE)	81	8	217	26	28	8	395	26	65	3	74	7	35	4	27	1	39	1	68	1
RF(Tune)	101	3	384	7	8	3	228	7	46	1	92	3	63	1	21	0	34	1	53	1
RF(SMOTE, Tune)	81	6	225	22	29	6	388	22	65	3	74	5	36	4	27	3	39	2	68	2
CART	77	0	237	0	33	0	376	0	63	0	70	0	39	0	25	0	36	0	65	0
CART(SMOTE)	71	8	262	33	38	8	351	33	59	4	65	7	43	6	22	2	33	3	61	4
CART(Tune)	89	0	346	0	21	0	267	0	49	0	81	0	56	0	20	0	33	0	57	0
CART(SMOTE, Tune)	78	16	264	55	31	16	349	55	59	6	71	15	43	9	23	4	34	6	62	6