Is "Better Data" Better Than "Better Data Miners"?

On the Benefits of Tuning SMOTE for Defect Prediction

Amritanshu Agrawal
Department of Computer Science
North Carolina State University
Raleigh, NC, USA
aagrawa8@ncsu.edu

Tim Menzies
Department of Computer Science
North Carolina State University
Raleigh, NC, USA
tim@menzies.us

ABSTRACT

We report and fix an important systematic error in prior studies that ranked classifiers for software analytics. Those studies did not (a) assess classifiers on multiple criteria and they did not (b) study how variations in the data affect the results. Hence, this paper applies (a) multi-performance criteria while (b) fixing the weaker regions of the training data (using SMOTUNED, which is an auto-tuning version of SMOTE). This approach leads to dramatically large increases in software defect predictions when applied in a 5*5 cross-validation study for 3,681 JAVA classes (containing over a million lines of code) from open source systems, SMOTUNED increased AUC and recall by 60% and 20% respectively. These improvements are independent of the classifier used to predict for defects. Same kind of pattern (improvement) was observed when a comparative analysis of SMOTE and SMOTUNED was done against the most recent class imbalance technique.

In conclusion, for software analytic tasks like defect prediction, (1) data pre-processing can be more important than classifier choice, (2) ranking studies are incomplete without such pre-processing, and (3) SMOTUNED is a promising candidate for pre-processing.

KEYWORDS

Search based SE, defect prediction, classification, data analytics for software engineering, SMOTE, imbalanced data, preprocessing

ACM Reference Format:
Amritanshu Agrawal and Tim Menzies. 2018. Is "Better Data" Better Than "Better Data Miners"?: On the Benefits of Tuning SMOTE for Defect Prediction . In *ICSE '18: ICSE '18: 40th International Conference on Software Engineering , May 27-June 3, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3180155.3180197

1 INTRODUCTION

Software quality methods cost money and better quality costs exponentially more money [16, 66]. Given finite budgets, quality assurance resources are usually skewed towards areas known to be most safety critical or mission critical [34]. This leaves "blind spots": regions of the system that may contain defects which may be missed. Therefore, in addition to rigorously assessing critical areas, a parallel activity should be to *sample the blind spots* [37].

To sample those blind spots, many researchers use *static code defect predictors*. Source code is divided into sections and researchers annotate the code with the number of issues known for each section. Classification algorithms are then applied to learn what static code attributes distinguish between sections with few/many issues. Such static code measures can be automatically extracted from the code base with little effort even for very large software systems [44].

One perennial problem is what classifier should be used to build predictors? Many papers report *ranking studies* where a quality measure is collected from classifiers when they are applied to data sets [13, 15–18, 21, 25–27, 29, 32, 33, 35, 40, 53, 62, 67]. These ranking studies report which classifiers generate best predictors.

Research of this paper began with the question *would the use of data pre-processor change the rankings of classifiers?* SE data sets are often imbalanced, i.e., the data in the target class is overwhelmed by an over-abundance of information about everything else except the target [36]. As shown in the literature review of this paper, in the overwhelming majority of papers (85%), SE research uses SMOTE to fix data imbalance [7] but SMOTE is controlled by numerous parameters which usually are tuned using engineering expertise or left at their default values. This paper proposes SMOTUNED, an automatic method for setting those parameters which when assessed on defect data from 3,681 classes (over a million lines of code) taken from open source JAVA systems, SMOTUNED outperformed both the original SMOTE [7] as well as state-of-the-art method [4].

To assess, we ask four questions:

- **RQ1**: *Are the default "off-the-shelf" parameters for* SMOTE *appropriate for all data sets?*

Result 1
SMOTUNED *learned different parameters for each data set, all of which were very different from default* SMOTE.

- **RQ2**: *Is there any benefit in tuning the default parameters of* SMOTE *for each new data set?*

Result 2
Performance improvements using SMOTUNED *are dramatically large, e.g., improvements in AUC up to 60% against* SMOTE.

In those results, we see that while no learner was best across all data sets and performance criteria, SMOTUNED was most often seen in the best results. That is, creating better training data might be more important than the subsequent choice of classifiers.

ICSE '18, May 27-June 3, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5638-1/18/05…$15.00
https://doi.org/10.1145/3180155.3180197

- **RQ3**: *In terms of runtimes, is the cost of running* SMOTUNED *worth the performance improvement?*

Result 3
SMOTUNED *terminates in under two minutes, i.e., fast enough to recommend its widespread use.*

- **RQ4**: *How does* SMOTUNED *perform against the recent class imbalance technique?*

Result 4
SMOTUNED *performs better than a very recent imbalance handling technique proposed by Bennin et al. [4].*

In summary, the contributions of this paper are:

- The discovery of an important systematic error in many prior ranking studies, i.e., all of [13, 15–18, 21, 25–27, 29, 32, 33, 35, 40, 53, 62, 67].
- A novel application of search-based SE (SMOTUNED) to handle class imbalance that out-performs the prior state-of-the-art.
- Dramatically large improvements in defect predictors.
- Potentially, for any other software analytics task that uses classifiers, a way to improve those learners as well.
- A methodology for assessing the value of pre-processing data sets in software analytics.
- A reproduction package to reproduce our results then (perhaps) to improve or refute our results (Available to download from http://tiny.cc/smotuned).

The rest of this paper is structured as follows: Section 2.1 gives an overview on software defect prediction. Section 2.2 talks about all the performance criteria used in this paper. Section 2.3 explains the problem of class imbalance in defect prediction. Assessment of the previous ranking studies is done in Section 2.4. Section 2.5 introduces SMOTE and discusses how SMOTE has been used in literature. Section 2.6 provides the definition of SMOTUNED. Section 3 describes the experimental setup of this paper and above research questions are answered in Section 4. Lastly, we discuss the validity of our results and a section describing our conclusions.

Note that the experiments of this paper only make conclusions about software analytics for defect prediction. That said, many other software analytics tasks use the same classifiers explored here: for non-parametric sensitivity analysis [41], as a pre-processor to build the tree used to infer quality improvement plans [31], to predict Github issue close time [55], and many more. That is, potentially, SMOTUNED is a sub-routine that could improve many software analytics tasks. This could be a highly fruitful direction for future research.

2 BACKGROUND AND MOTIVATION

2.1 Defect Prediction

Software programmers are intelligent, but busy people. Such busy people often introduce defects into the code they write [20]. Testing software for defects is expensive and most software assessment budgets are finite. Meanwhile, assessment effectiveness increases exponentially with assessment effort [16]. Such exponential costs exhaust finite resources so software developers must carefully decide what parts of their code need most testing.

A variety of approaches have been proposed to recognize defect-prone software components using code metrics (lines of code, complexity) [10, 38, 40, 45, 58] or process metrics (number of changes, recent activity) [22]. Other work, such as that of Bird et al. [5], indicated that it is possible to predict which components (for e.g., modules) are likely locations of defect occurrence using a component's development history and dependency structure. Prediction models based on the topological properties of components within them have also proven to be accurate [71].

The lesson of all the above is that the probable location of future defects can be guessed using logs of past defects [6, 21]. These logs might summarize software components using static code metrics such as McCabes cyclomatic complexity, Briands coupling metrics, dependencies between binaries, or the CK metrics [8] (which is described in Table 1). One advantage with CK metrics is that they are simple to compute and hence, they are widely used. Radjenović et al. [53] reported that in the static code defect prediction, the CK metrics are used twice as much (49%) as more traditional source code metrics such as McCabes (27%) or process metrics (24%). The static code measures that can be extracted from a software is shown in Table 1. Note that such attributes can be automatically collected, even for very large systems [44]. Other methods, like manual code reviews, are far slower and far more labor intensive.

Static code defect predictors are remarkably fast and effective. Given the current generation of data mining tools, it can be a matter of just a few seconds to learn a defect predictor (see the runtimes in Table 9 of reference [16]). Further, in a recent study by Rahman et al. [54], found no significant differences in the cost-effectiveness of (a) static code analysis tools FindBugs and Jlint, and (b) static code defect predictors. This is an interesting result since it is much slower to adapt static code analyzers to new languages than defect predictors (since the latter just requires hacking together some new static code metrics extractors).

2.2 Performance Criteria

Formally, defect prediction is a binary classification problem. The performance of a defect predictor can be assessed via a confusion matrix like Table 2 where a "positive" output is the defective class under study and a "negative" output is the non-defective one.

Table 2: Results Matrix

	Actual	
Prediction	false	true
defect-free	*TN*	*FN*
defective	*FP*	*TP*

Further, "false" means the learner got it wrong and "true" means the learner correctly identified a fault or non-fault module. Hence, Table 2 has four quadrants containing, e.g., *FP* which denotes "false positive".

From this matrix, we can define performance measures like:

- **Recall** = $pd = TP/(TP + FN)$
- **Precision** = $prec = TP/(TP + FP)$
- **False Alarm** = $pf = FP/(FP + TN)$
- **Area Under Curve (AUC)**, which is the area covered by an ROC curve [11, 60] in which the X-axis represents, false positive rate and the Y-axis represents true positive rate.

As shown in Figure 1, a typical predictor must "trade-off" between false alarm and recall. This is because the more sensitive the detector, the more often it triggers and the higher its recall. If a

Table 1: OO CK code metrics used for all studies in this paper. The last line shown, denotes the dependent variable.

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
lcom	lack of cohesion in methods	number of pairs of methods that do not share a reference to an case 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 $lcom3 = ((\frac{1}{a}\sum_j \mu(a,j)) - m)/(1-m)$.
loc	lines of code	
max, cc	maximum McCabe	maximum McCabe's cyclomatic complexity seen in class
mfa	functional abstraction	no. of methods inherited by a class plus no. 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	
nDefects	raw defect counts	numeric: number of defects found in post-release bug-tracking systems.
defects present?	boolean	if *nDefects* > 0 then *true* else *false*

detector triggers more often, it also raises more false alarms. Hence, when increasing recall, we should expect the false alarm rate to increase (ideally, not by very much).

There are many more ways to evaluate defect predictors besides the four listed above. Previously, Menzies et al. catalogued dozens of them (see Table 23.2 of [39]) and even several novel ones were proposed (balance, G-measure [38]).

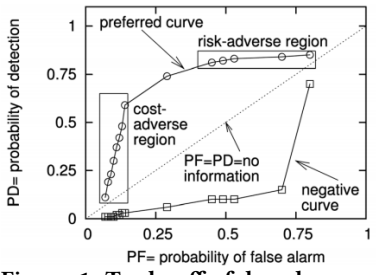

Figure 1: Trade-offs false alarm vs recall (probability of detection).

But no evaluation criteria is "best" since different criteria are appropriate in different business contexts. For e.g., as shown in Figure 1, when dealing with safety-critical applications, management may be "risk adverse" and hence many elect to *maximize recall*, regardless of the time wasted exploring false alarm. Similarly, when rushing some non-safety critical application to market, management may be "cost adverse" and elect to *minimize false alarm* since this avoids distractions to the developers.

In summary, there are numerous evaluation criteria and numerous business contexts where different criteria might be preferred by different local business users. In response to the cornucopia of evaluation criteria, we make the following recommendations: a) do evaluate learners on more than one criteria, b) do not evaluate learners on all criteria (there are too many), and instead, apply the criteria widely seen in the literature. Applying this advice, this paper evaluates the defect predictors using the four criteria mentioned above (since these are widely reported in the literature [16, 17])) but not other criteria that have yet to gain a wide acceptance (i.e., balance and G-measure).

2.3 Defect Prediction and Class Imbalance

Class imbalance is concerned with the situation in where some classes of data are highly under-represented compared to other classes [23]. By convention, the under-represented class is called the *minority* class, and correspondingly the class which is over-represented is called the *majority* class. In this paper, we say that class imbalance is *worse* when the ratio of minority class to majority *increases*, that is, *class-imbalance of 5:95* is worse than *20:80*. Menzies et al. [36] reported SE data sets often contain class imbalance. In their examples, they showed static code defect prediction data sets with class imbalances of 1:7; 1:9; 1:10; 1:13; 1:16; 1:249.

The problem of class imbalance is sometimes discussed in the software analytics community. Hall et al. [21] found that models based on C4.5 under-perform if they have imbalanced data while Naive Bayes and Logistic regression perform relatively better. Their general recommendation is to not use imbalanced data. Some researchers offer preliminary explorations into methods that might mitigate for class imbalance. Wang et al. [67] and Yu et al. [69] validated the Hall et al. results and concluded that the performance of C4.5 is unstable on imbalanced data sets while Random Forest and Naive Bayes are more stable. Yan et al. [68] performed fuzzy logic and rules to overcome the imbalance problem, but they only explored one kind of learner (Support Vector Machines). Pelayo et al. [49] studied the effects of the percentage of oversampling and undersampling done. They found out that different percentage of each helps improve the accuracies of decision tree learner for defect prediction using CK metrics. Menzies et al. [42] undersampled the non-defect class to balance training data and reported how little information was required to learn a defect predictor. They found that throwing away data does not degrade the performance of Naive Bayes and C4.5 decision trees. Other papers [49, 50, 57] have shown the usefulness of resampling based on different learners.

We note that many researchers in this area [19, 67, 69] refer to the SMOTE method explored in this paper, but only in the context of future work. One rare exception to this general pattern is the recent paper by Bennin et al. [4], which we explored as part of RQ4.

2.4 Ranking Studies

A constant problem in defect prediction is what classifier should be applied to build the defect predictors? To address this problem, many researchers run *ranking studies* where performance scores

Table 3: Classifiers used in this study. Rankings from Ghotra et al. [17].

RANK	LEARNER	NOTES
1 "best"	RF= random forest	Random forest of entropy-based decision trees.
	LR=Logistic regression	A generalized linear regression model.
2	KNN= K-means	Classify a new instance by finding "k" examples of similar instances. Ghortra et al. suggested $K = 8$.
	NB= Naive Bayes	Classify a new instance by (a) collecting mean and standard deviations of attributes in old instances of different classes; (b) return the class whose attributes are statistically most similar to the new instance.
3	DT= decision trees	Recursively divide data by selecting attribute splits that reduce the entropy of the class distribution.
4 "worst"	SVM= support vector machines	Map the raw data into a higher-dimensional space where it is easier to distinguish the examples.

Table 4: 22 highly cited Software Defect prediction studies.

Ref	Year	Citations	Ranked Classifiers?	Evaluated using multiple criteria?	Considered Data Imbalance?
[38]	2007	855	✓	2	✗
[32]	2008	607	✓	1	✗
[13]	2008	298	✓	2	✗
[40]	2010	178	✓	3	✗
[18]	2008	159	✓	1	✗
[30]	2011	153	✓	2	✗
[53]	2013	150	✓	1	✗
[25]	2008	133	✓	1	✗
[67]	2013	115	✓	1	✓
[35]	2009	92	✓	1	✗
[33]	2012	79	✓	2	✗
[28]	2007	73	✗	2	✓
[49]	2007	66	✗	1	✓
[27]	2009	62	✓	3	✗
[29]	2010	60	✓	1	✓
[17]	2015	53	✓	1	✗
[26]	2008	41	✓	1	✗
[62]	2016	31	✓	1	✗
[61]	2015	27	✗	2	✓
[50]	2012	23	✗	1	✓
[16]	2016	15	✓	1	✗
[4]	2017	0	✓	3	✓

are collected from many classifiers executed on many software defect data sets [13, 16–18, 21, 25–27, 29, 32, 33, 35, 40, 53, 62, 67]. This section assesses those ranking studies. We will say a ranking study is "good" if it compares multiple learners using multiple data sets and multiple evaluation criteria while at the same time doing something to address the data imbalance problem.

In July 2017, we searched scholar.google.com for the conjunction of "software" and "defect prediction" and "OO" and "CK" published in the last decade. This returned 231 results. We only selected OO and CK keywords since CK metrics are more popular and better than process metrics for software defect prediction [53]. From that list, we selected "highly-cited" papers, which we defined as having more than 10 citations per year. This reduced our population of papers down to 107. After reading the titles and abstracts of those papers, and skimming the contents of the potentially interesting papers, we found 22 papers of Table 4 that either performed ranking studies (as defined above) or studied the effects of class imbalance on defect prediction. In the column "evaluated using multiple criteria", papers scored more than "1" if they used multiple performance scores of the kind listed at the end of Section 2.2.

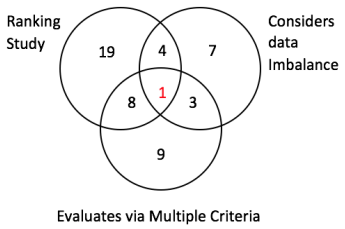

Figure 2: Summary of Table 4.

We find that, in those 22 papers from Table 4, numerous classifiers have used AUC as the measure to evaluate the software defect predictor studies. We also found that majority of papers (from last column of Table 4, 6/7=85%) in SE community has used SMOTE to fix the data imbalance [4, 28, 49, 50, 61, 67]. This also made us to propose SMOTUNED. As noted in [17, 32], no single classification technique always dominates. That said, Table IX of a recent study by Ghotra et al. [17] ranks numerous classifiers using data similar to what we use here (i.e., OO JAVA systems described using CK metrics). Using their work, we can select a range of classifiers for this study ranking from "best" to "worst': see Table 3.

The key observation to be made from this survey is that, as shown in Figure 2, the overwhelming majority of prior papers in our sample *do not satisfy* our definition of a "good" project (the sole exception is the recent Bennin et al. [4] which we explore in RQ4). Accordingly, the rest of this paper defines and executes a "good" ranking study, with an additional unique feature of an auto-tuning version of SMOTE.

2.5 Handling Data Imbalance with SMOTE

SMOTE handles class imbalance by changing the frequency of different classes of the training data [7]. The algorithm's name is short for "synthetic minority over-sampling technique". When applied to data, SMOTE sub-samples the majority class (i.e., deletes some examples) while super-sampling the minority class until all classes have the same frequency. In the case of software defect data, the minority class is usually the defective class.

Figure 3 shows how SMOTE works. During super-sampling, a member of the minority class finds k nearest neighbors. It builds an artificial member of the minority class at some point in-between itself and one of its random nearest neighbors. During that process, some distance function is required which is the *minkowski_distance* function.

```
def SMOTE(k=2, m=50%, r=2): # defaults
  while Majority > m do
    delete any majority item # random
  while Minority < m do
    add something_like(any minority item)

def something_like(X0):
  relevant = emptySet
  k1 = 0
  while(k1++ < 20 and size(found) < k) {
     all = k1 nearest neighbors
     relevant += items in "all" of X0 class}
  Z = any of found
  Y = interpolate (X0, Z)
  return Y

def minkowski_distance(a,b,r):
  return (Σ_i abs(a_i − b_i)^r)^(1/r)
```

Figure 3: Pseudocode of SMOTE

SMOTE's control parameters are (a) k that selects how many neighbors to use (defaults to $k = 5$), (b) m is how many examples of

each class which need to be generated (defaults to $m = 50\%$ of the total training samples), and (3) r which selects the distance function (default is $r = 2$, i.e., use Euclidean distance).

In the software analytics literature, there are contradictory findings on the value of applying SMOTE for software defect prediction. Van et al. [64], Pears et al. [47] and Tan et al. [61] found SMOTE to be advantageous, while others, such as Pelayo et al. [49] did not.

Further, some researchers report that some learners respond better than others to SMOTE. Kamei et al. [28] evaluated the effects of SMOTE applied to four fault-proneness models (linear discriminant analysis, logistic regression, neural network, and decision tree) by using two module sets of industry legacy software. They reported that SMOTE improved the prediction performance of the linear and logistic models, but not neural network and decision tree models. Similar results, that the value of SMOTE was dependent on the learner, was also reported by Van et al. [64].

Recently, Bennin et al. [4] proposed a new method based on the chromosomal theory of inheritance. Their MAHAKIL algorithm interprets two distinct sub-classes as parents and generates a new synthetic instance that inherits different traits from each parent and contributes to the diversity within the data distribution. They report that MAHAKIL usually performs as well as SMOTE, but does much better than all other class balancing techniques in terms of recall. Please note, that work did not consider the impact of parameter tuning of a preprocessor so in our RQ4 we will compare SMOTUNED to MAHAKIL.

2.6 SMOTUNED = auto-tuning SMOTE

One possible explanation for the variability in the SMOTE results is that the default parameters of this algorithm are not suited to all data sets. To test this, we designed SMOTUNED, which is an auto-tuning version of SMOTE. SMOTUNED uses different control parameters for different data sets.

SMOTUNED uses DE (differential evolution [59]) to explore the parameter space of Table 5. DE is an optimizer useful for functions that may not be smooth or linear. Vesterstrom et al. [65] find DE's optimizations to be competitive with other optimizers like particle swarm optimization or genetic algorithms. DEs have been used before for parameter tuning [2, 9, 14, 16, 46]) but this paper is the first attempt to do DE-based class re-balancing for SE data by studying multiple learners for multiple evaluation criteria.

In Figure 4, DE evolves a *frontier* of candidates from an initial population which is driven by a goal (like maximizing recall) evaluated using a fitness function (shown in line 17). In the case of SMOTUNED, each candidate is a randomly selected value for SMOTE's k, m and r parameters. To evolve the frontier, within each generation, DE compares each item to a *new* candidate generated by combining three other frontier items (and better *new* candidates replace older items). To compare them, the *better* function (line 17) calls *SMOTE* function (from Figure 3) using the proposed *new* parameter settings. This pre-processed training data is then fed into a classifier to find a particular measure (like recall). When our DE terminates, it returns the best candidate ever seen in the entire run.

Table 6 provides important terms of SMOTUNED when exploring SMOTE's parameter ranges, shown in Table 5. To define the parameters, we found the range of used settings for SMOTE and

```
def DE( n=10, cf=0.3, f=0.7): # default settings                          1
  frontier = sets of guesses (n=10)                                      2
  best = frontier.1 # any value at all                                   3
  lives = 1                                                              4
  while(lives-- > 0):                                                    5
    tmp = empty                                                          6
    for i = 1 to |frontier|: # size of frontier                          7
      old = frontier_i                                                   8
      x,y,z = any three from frontier, picked at random                  9
      new= copy(old)                                                     10
      for j = 1 to |new|: # for all attributes                           11
        if rand() < cf # at probability cf...                            12
          new.j = x.j + f * (z.j - y.j) # ...change item j               13
      # end for                                                          14
      new = new if better(new,old) else old                              15
      tmp_i = new                                                        16
      if better(new,best) then                                           17
        best = new                                                       18
        lives++ # enable one more generation                             19
      end                                                                20
    # end for                                                          21
    frontier = tmp                                                       22
  # end while                                                          23
  return best                                                            24
```

Figure 4: SMOTUNED uses DE (differential evolution).

Table 5: SMOTE parameters

Para	Defaults used by SMOTE	Tuning Range (Explored by (SMOTUNED)	Description
k	5	[1,20]	Number of neighbors
m	50%	{50, 100, 200, 400}	Number of synthetic examples to create. Expressed as a percent of final training data.
r	2	[0.1,5]	Power parameter for the Minkowski distance metric.

Table 6: Important Terms of SMOTUNED Algorithm

Keywords	Description
Differential weight ($f = 0.7$)	Mutation power
Crossover probability ($cf = 0.3$)	Survival of the candidate
Population Size ($n = 10$)	Frontier size in a generation
Lives	Number of generations
Fitness Function ($better$)	Driving factor of DE
Rand() function	Returns between 0 to 1
Best (or Output)	Optimal configuration for SMOTE

distance functions in the SE and machine learning literature. To avoid introducing noise by overpopulating the minority samples we are not using m as percentage rather than number of examples to create. Aggarawal et al. [1] argue that with data being highly dimensional, r should shrink to some fraction less than one (hence the bound of $r = 0.1$ in Table 5).

3 EXPERIMENTAL DESIGN

This experiment reports the effects on defect prediction after using MAHAKIL or SMOTUNED or SMOTE. Using some data $D_i \in D$, performance measure $M_i \in M$, and classifier $C_i \in C$, this experiment conducts the 5*5 cross-validation study, defined below. Our data sets D are shown in Table 7. These are all open source JAVA OO systems described in terms of the CK metrics. Since, we are comparing these results for imbalanced class, only imbalanced class data sets were selected from SEACRAFT (http://tiny.cc/seacraft).

Our performance measures M were introduced in Section 2.2 which includes AUC, precision, recall, and the false alarm. Our

classifiers C come from a recent study [17] and were listed in Table 3. For implementations of these learners, we used the open source tool Scikit-Learn [48]. Our cross-validation study [56] is defined as follows:

(1) We randomized the order of the data set D_i five times. This reduces the probability that some random ordering of examples in the data will conflate our results.
(2) Each time, we divided the data D_i into five bins;
(3) For each bin (the test), we trained on four bins (the rest) and then tested on the test bin as follows.
   (a) The training set is pre-filtered using either No-SMOTE (i.e., do nothing) or SMOTE or SMOTUNED.
   (b) When using SMOTUNED, we further divide those four bins of training data. 3 bins are used for training the model, and 1 bin is used for validation in DE. DE is run to improve the performance measure M_i seen when the classifier C_i was applied to the training data. Important point: *we only used* SMOTE *on the training data, leaving the testing data unchanged.*
   (c) After pre-filtering, a classifier C_i learns a predictor.
   (d) The model is applied to the test data to collect performance measure M_i.
   (e) We print the *relative performance delta* between this M_i and another M_i generated from applying C_i to the raw data D_i (i.e., compare the learner without any filtering). We finally report median on the 25 repeats.

Note that the above rig tunes SMOTE, but not the control parameters of the classifiers. We do this since, in this paper, we aim to document the benefits of tuning SMOTE since as shown below, they are very large indeed. Also, it would be very useful if we can show that a single algorithm (SMOTUNED) improves the performance of defect prediction. This would allow subsequent work to focus on the task of optimizing SMOTUNED (which would be a far easier task than optimizing the tuning of a wide-range of classifiers).

3.1 Within- vs Cross-Measure Assessment

We call the above rig as the *within-measure assessment rig* since it is biased in its evaluation measures. Specifically, in this rig, when SMOTUNED is optimized for (e.g.,) AUC, we do not explore the effects on (e.g.,) the false alarm. This is less than ideal since it is known that our performance measures are inter-connected via the Zhang's equation [70]. Hence, increasing (e.g.,) recall might potentially have the adverse effect of driving up (e.g) the false alarm rate. To avoid this problem, we also apply the following *cross-measure assessment rig*. At the conclusion of the *within-measure assessment rig*, we will observe that the AUC performance measure will show the largest improvements. Using that best performer, we will re-apply steps 1,2,3 abcde (listed above) but this time:

- In step 3b, we will tell SMOTUNED to optimize for AUC;
- In step 3d, 3e we will collect the performance delta on AUC as well as precision, recall, and false alarm.

In this approach, steps 3d and 3e collect the information required to check if succeeding according to one performance criteria results in damage to another. We would also want to make sure that our model is not over-fitted based on one evaluation measure. And since SMOTUNED is a time expensive task, we do not want to tune for each measure which will quadruple the time. The results of within- vs cross-measure assessment is shown in Section 4.

Table 7: Data set statistics. Data sets are sorted from low percentage of defective class to high defective class. Data comes from the SEACRAFT repository: http://tiny.cc/seacraft

Version	Dataset Name	Defect %	No. of classes	lines of code
4.3	jEdit	2	492	202,363
1.0	Camel	4	339	33,721
6.0.3	Tomcat	9	858	300,674
2.0	Ivy	11	352	87,769
1.0	Arcilook	11.5	234	31,342
1.0	Redaktor	15	176	59,280
1.7	Apache Ant	22	745	208,653
1.2	Synapse	33.5	256	53,500
1.6.1	Velocity	34	229	57,012
		total:	3,681	1,034,314

3.2 Statistical Analysis

When comparing the results of SMOTUNED to other treatments, we use a statistical significance test and an effect size test. Significance test are useful for detecting if two populations differ merely by random noise. Also, effect sizes are useful for checking that two populations differ by more than just a trivial amount.

For the significance test, we used the Scott-Knott procedure [17, 43]. This technique recursively bi-clusters a sorted set of numbers. If any two clusters are statistically indistinguishable, Scott-Knott reports them both as one group. Scott-Knott first looks for a break in the sequence that maximizes the expected values in the difference in the means before and after the break. More specifically, it splits l values into sub-lists m and n in order to maximize the expected value of differences in the observed performances before and after divisions. For e.g., lists l, m and n of size ls, ms and ns where $l = m \cup n$, Scott-Knott divides the sequence at the break that maximizes:

$$E(\Delta) = ms/ls * abs(m.\mu - l.\mu)^2 + ns/ls * abs(n.\mu - l.\mu)^2$$

Scott-Knott then applies some statistical hypothesis test H to check if m and n are significantly different. If so, Scott-Knott then recurses on each division. For this study, our hypothesis test H was a conjunction of the A12 effect size test (endorsed by [3]) and non-parametric bootstrap sampling [12], i.e., our Scott-Knott divided the data if *both* bootstrapping and an effect size test agreed that the division was statistically significant (99% confidence) and not a "small" effect ($A12 \geq 0.6$).

4 RESULTS

RQ1: Are the default "off-the-shelf" parameters for SMOTE appropriate for all data sets?

As discussed above, the default parameters for SMOTE, k, m and r are 5, 50% and 2. Figure 5 shows the range of parameters found by SMOTUNED across nine data sets for the 25 repeats of our cross-validation procedure. All the results in this figure are *within-measure assessment* results, i.e., here, we SMOTUNED on a particular performance measure and then we only collect performance for that performance measure on the test set.

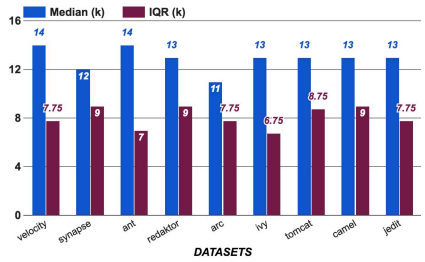

Figure 5a: Tuned values for k (default: $k = 5$).

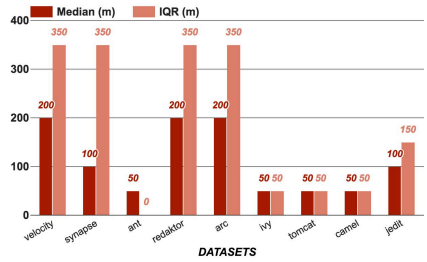

Figure 5b: Tuned values for m (default: $m = 50\%$).

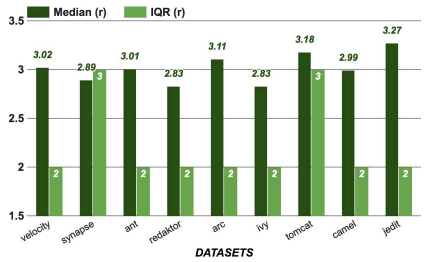

Figure 5c: Tuned values for r (default: $r = 2$).

Figure 5: Data sets vs Parameter Variation when optimized for recall and results reported on recall. "Median" denotes 50th percentile values seen in the 5*5 cross-validations and "IQR" shows the intra-quartile range, i.e., (75-25)th percentiles.

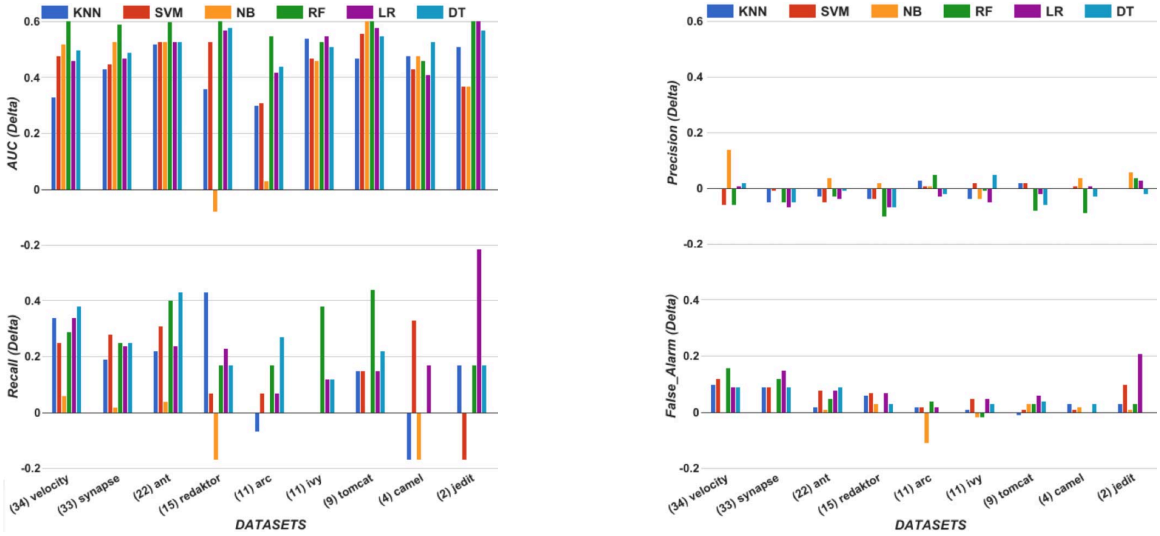

Figure 6: SMOTUNED **improvements over** SMOTE. **Within-Measure assessment (i.e., for each of these charts, optimize for performance measure M_i, then test for performance measure M_i). For most charts, *larger* values are *better*, but for false alarm, *smaller* values are *better*. Note that the corresponding percentage of minority class (in this case, defective class) is written beside each data set.**

In Figure 5, the *median* is the 50th percentile value and *IQR* is the (75-25)th percentile (variance). As can be seen in Figure 5, most of the learned parameters are far from the default values: 1) Median k is never less than 11; 2) Median m differs according to each data set and quite far from the actual; 3) The r used in the distance function was never 2, rather, it was usually 3. Hence, our answer to **RQ1** is "no": the use of off-the-shelf SMOTE should be deprecated.

We note that many of the settings in Figure 5 are very similar; for e.g., median values of $k = 13$ and $r = 3$ seems to be a common result irrespective of data imbalance percentage among the datasets. Nevertheless, we do *not* recommend replacing the defaults of SMOTE with the findings of Figure 5. Also, IQR bars are very large. Clearly, SMOTUNED's decisions vary dramatically depending on what data is being processed. Hence, we strongly recommend that SMOTUNED be applied to each new data set.

RQ2: Is there any benefit in tuning the default parameters of SMOTE for each new data set?

Figure 6 shows the performance delta of the *within-measure assessment rig*. Please recall that when this rig applies SMOTUNED, it optimizes for performance measure, $M_i \in \{recall, precision, false\ alarm, AUC\}$ after which it uses the *same* performance measure M_i when evaluating the test data. In Figure 6, each subfigure shows that DE is optimized for each M_i and results are reported against the same M_i. From the figure 6, it is observed that SMOTUNED achieves large AUC (about 60%) and recall (about 20%) improvements relatively without damaging precision and with only minimal changes to false alarm. Another key observation here that can be made is that improvements in AUC with SMOTUNED is constant whether imbalance is of 34% or 2%. Another note should be taken of the AUC improvements, that these are the largest improvements we have yet seen, for any prior treatment of defect prediction data. Also, for the raw AUC values, please see http://tiny.cc/raw_auc.

Figure 7 offers a statistical analysis of different results achieved after applying our three data pre-filtering methods: 1) ***NO*** = do nothing, 2) ***S1*** = use default SMOTE, and 3) ***S2*** = use SMOTUNED. For any learner, there are three such treatments and *darker* the cell,

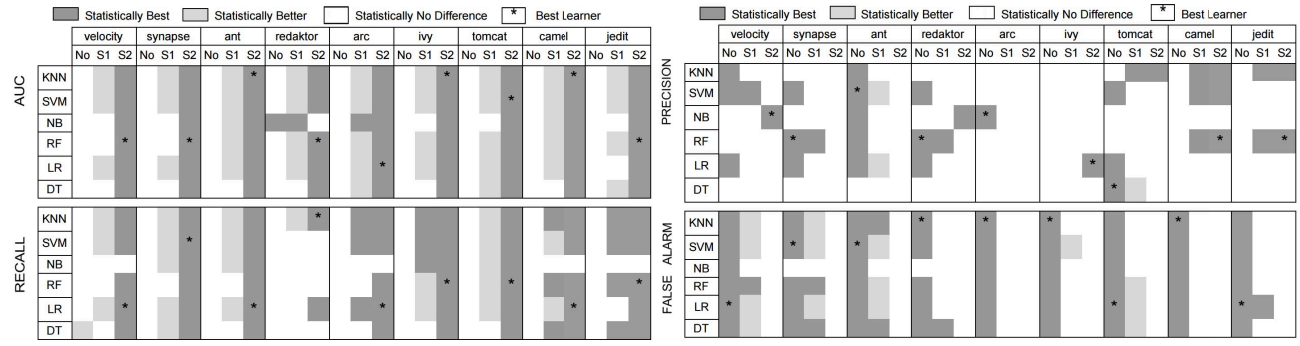

Figure 7: Scott Knott analysis of No-SMOTE, SMOTE and SMOTUNED. The column headers are denoted as No for No-SMOTE, S1 for SMOTE and S2 for SMOTUNED. (∗) Mark represents the best learner combined with its techniques.

better the performance. In that figure, cells with the same color are either not statistically significantly different or are different only via a *small effect* (as judged by the statistical methods described in Section 3.2).

As to what combination of pre-filter+learner works better for any data set, that is marked by a '*'. Since we have three pre-filtering methods and six learners providing us with in-total 18 treatments, and '*' represents the best learner picked with highest median value.

In the AUC and recall results, the best "*" cell always appears in the S2=SMOTUNED column, i.e., SMOTUNED is always used by the best combination of pre-filter+learner .

As to precision results, at first glance, the results in Figure 7 look bad for SMOTUNED since, less than half the times, the best "*" happens in S2=SMOTUNED column. But recall from Figure 6 that the absolute size of the precision deltas is very small. Hence, even though SMOTUNED "losses" in this statistical analysis, the pragmatic impact of that result is negligible. But if we can get feedback from domain/expert, we can change between SMOTE and SMOTUNED dynamically based on the measures and data miners.

As to the false alarm results from Figure 7, as discussed above in Section 2.2, the cost of increased recall is to also increase the false alarm rate. For e.g., the greatest *increase* in the recall was 0.58 seen in the *jEdit* results. This increase comes at a cost of *increasing* the false alarm rate by 0.20. Apart from this one large outlier, the overall pattern is that the recall improvements range from +0.18 to +0.42 (median to max) and these come at the cost of much smaller false alarm *increase* of 0.07 to 0.16 (median to max).

In summary, the answer to **RQ2** is that our AUC and recall results strongly endorse the use of SMOTUNED while the precision and false alarm rates show there is little harm in using SMOTUNED.

Before moving to the next research question, we note that these results offer an interesting insight on prior ranking studies. Based on the Ghotra et al. results of Table 3, our expectation was that Random Forests (RF) would yield the best results across this defect data. Figure 7 reports that, as predicted by Ghotra et al., RF earns more "stars" than any other learner, i.e., it is seen to be "best" more often than anything else. That said, RF was only "best" in 11/36 of those results, i.e., even our "best" learner (RF) fails over half the time.

It is significant to note that SMOTUNED was consistently used by whatever learner was found to be "best" (in recall and AUC). Hence, we conclude prior ranking study results (that only assessed different learners) have missed a much more general effect; i.e. it can be more useful to reflect on data pre-processors than algorithm selection. To say that another way, at least for defect prediction, "better data" might be better than "better data miners".

RQ3: In terms of runtimes, is the cost of running SMOTUNED worth the performance improvement?

Figure 8 shows the mean runtimes for running a 5*5 cross-validation study for six learners for each data set. These runtimes were collected from one machine running CENTOS7, with 16 cores. Note that they do not increase monotonically with the size of the data sets– a result we can explain with respect to the internal structure of the data. Our version of SMOTE uses ball trees to optimize the nearest neighbor calculations. Hence, the runtime of that algorithm is dominated by the internal topology of the data sets rather than the number of classes. Also, as shown in Figure 3, SMOTUNED explores the local space until it finds k neighbors of the same class. This can take a variable amount of time to terminate.

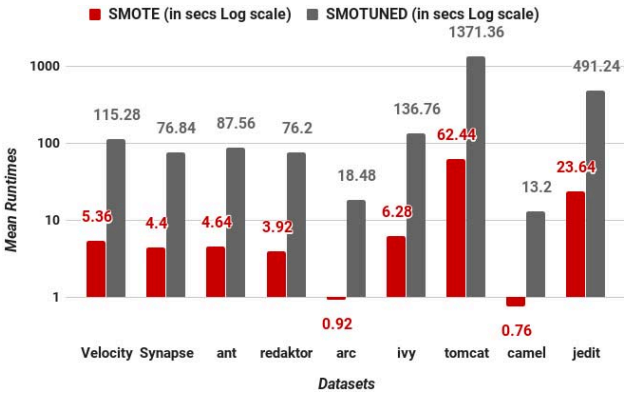

Figure 8: Data sets vs Runtimes. Note that the numbers shown here are the mean times seen across 25 repeats of a 5*5 cross-validation study.

As expected, SMOTUNED is an order of magnitude slower than SMOTE since it has to run SMOTE many times to assess different parameter settings. That said, those runtimes are not excessively slow. SMOTUNED usually terminates in under two minutes and never more than half an hour. Hence, in our opinion, we answer **RQ3**

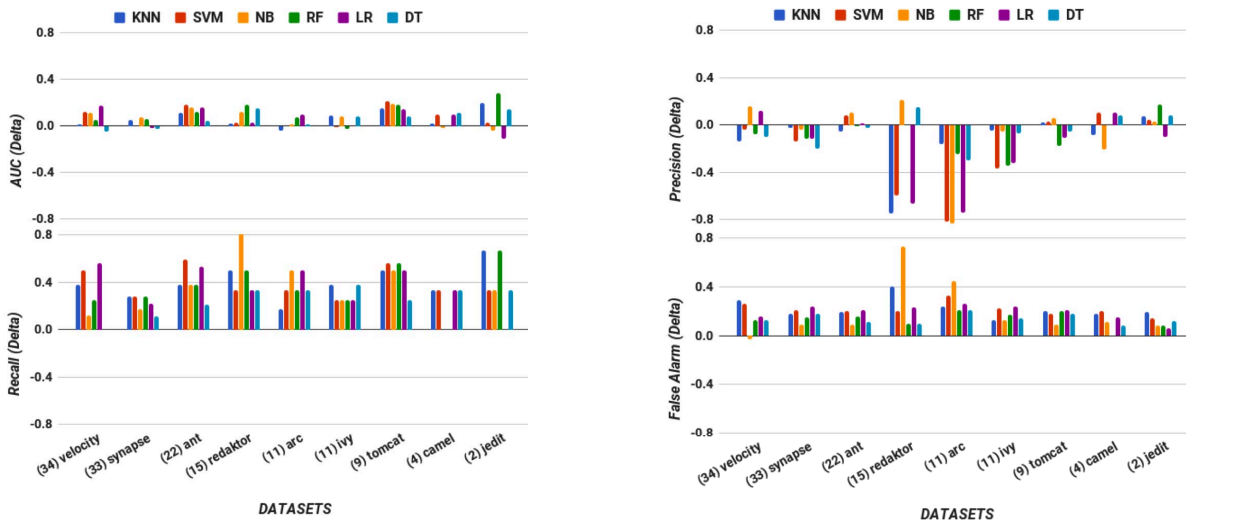

Figure 9: SMOTUNED improvements over MAHAKIL [4]. Within-Measure assessment (i.e., for each of these charts, optimize for performance measure M_i, then test for performance measure M_i). Same format as Figure 6.

as "yes" since the performance increment seen in Figure 6 is more than to compensate for the extra CPU required for SMOTUNED.

RQ4: How does SMOTUNED perform against more recent class imbalance technique?

All the above work is based on tuning the original 2002 SMOTE paper [7]. While that version of SMOTE is widely used in the SE literature, it is prudent to compare SMOTUNED with more recent work. Our reading of the literature is that the MAHAKIL algorithm of Bennin et al. [4] represents the most recent work in SE on handling class imbalance. At the time of writing of this paper (early August 2017), there was no reproduction package available for MAHAKIL so we wrote our own version based on the description in that paper (Available on http://tiny.cc/mahakil). We verified our implementation on their datasets, and achieved close to their values ± 0.1. The difference could be due to different random seed.

Figure 9 compares results from MAHAKIL with those from SMOTUNED. These results were generated using the same experimental methods as used for Figure 6 (those methods were described in Section 3.1). The following table repeats the statistical analysis of Figure 7 to report how often SMOTE, SMOTUNED, or MAHAKIL achieves best results across nine data sets. Note that, in this following table, *larger* values are *better*:

	number of wins			
Treatments	AUC	Recall	Precision	False Alarm
MAHAKIL	1/9	0/9	**6/9**	**9/9**
SMOTE	0/9	1/9	0/9	0/9
SMOTUNED	**8/9**	**8/9**	3/9	0/9

These statistical tests tell us that the differences seen in Figure 9 are large enough to be significant. Looking at Figure 9, there are 9 datasets on x-axis, and the differences in precision are so small in 7 out of those 9 data sets that the pragmatic impact of those differences is small. As to AUC and recall, we see that SMOTUNED generated larger and better results than MAHAKIL (especially for recall). SMOTUNED generates slightly larger false alarms but, in 7/9 data sets, the increase in the false alarm rate is very small.

According to its authors [4], MAHAKIL was developed to reduce the false alarm rates on SMOTE and on that criteria it succeeds (as seen in Figure 9, since SMOTUNED does lead to slightly higher false alarm rates). But, as discussed above in section 2.2, the downside on minimizing false alarms is also minimizing our ability to find defects which is measured in terms of AUC and recall, SMOTUNED does best. Hence, if this paper was a comparative assessment of SMOTUNED vs MAHAKIL, we would conclude that by recommending SMOTUNED.

However, the goal of this paper is to defend the claim that "better data" could be better than "better data miners", i.e., data preprocessing is more effective than switching to another data miner. In this regard, there is something insightful to conclude if we combine the results of *both* MAHAKIL and SMOTUNED. In the MAHAKIL experiments, the researchers spent some time on tuning the learner's parameters. That is, Figure 9 is really a comparison of two treatments: tuned data miners+adjust data against just using SMOTUNED to adjust the data. Note that SMOTUNED still achieves better results even though the MAHAKIL treatment *adjusted both data and data miners*. Since SMOTUNED performed so well without tuning the data miners, we can conclude from the conjunction of these experiments that "better data" is better than using "better data miners".

Of course, there needs to be further studies done in other SE applications to make the above claim. There is also one more treatment *not* discussed in the paper: tuning *both* the data pre-processor *and* the data miners. This is a very, very large search space so while we have experiments running to explore this task, at this time we have not definitive conclusions to report.

5 THREATS TO VALIDITY

As with any empirical study, biases can affect the final results. Therefore, any conclusions made from this work must consider the following issues in mind.

Order bias: With each data set how data samples are distributed in training and testing set is completely random. Though there

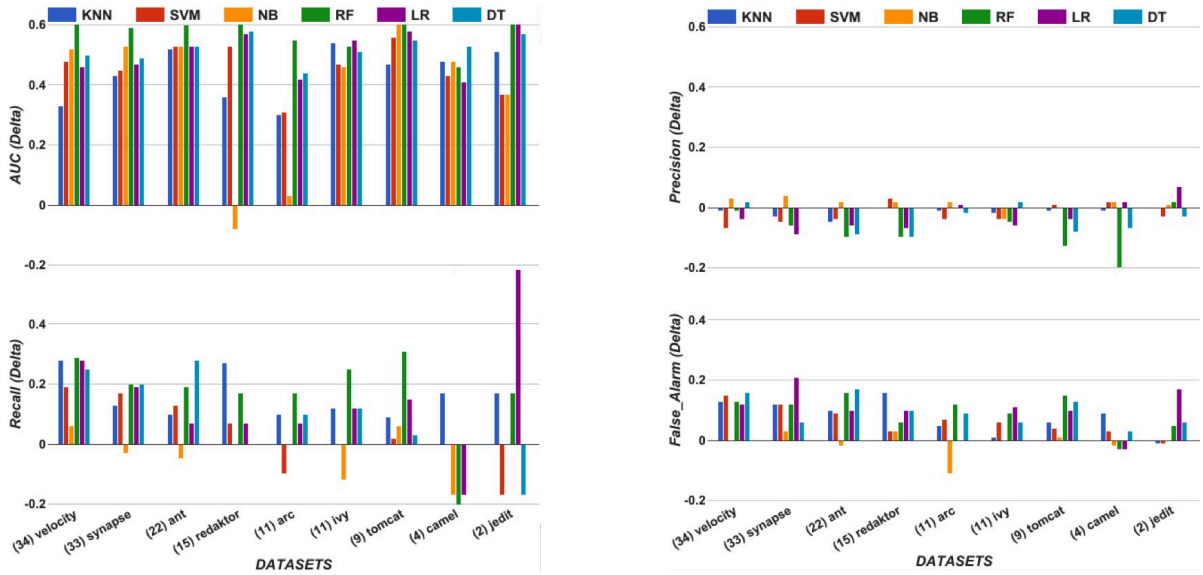

Figure 10: SMOTUNED improvements over SMOTE. Cross-Measure assessment (i.e., for each of these charts, optimize for AUC, then test for performance measure M_i). Same format as Figure 6.

could be times when all good samples are binned into training and testing set. To mitigate this order bias, we run the experiment 25 times by randomly changing the order of the data samples each time.

Sampling bias threatens any classification experiment, i.e., what matters there may not be true here. For e.g., the data sets used here comes from the SEACRAFT repository and were supplied by one individual. These data sets have used in various case studies by various researchers [24, 51, 52, 63], i.e., our results are not more biased than many other studies in this arena. That said, our nine open-source data sets are mostly from Apache. Hence it is an open issue if our results hold for proprietary projects and open source projects from other sources.

Evaluation bias: In terms of evaluation bias, our study is far less biased than many other ranking studies. As shown by our sample of 22 ranking studies in Table 4, 19/22 of those prior studies used *fewer* evaluation criteria than the four reported here (AUC, recall, precision and false alarm).

The analysis done in RQ4 could be affected by some other settings which we might not have considered since the reproduction package was not available from the original paper [4]. That said, there is another more subtle evaluation bias arises in the Figure 6. The four plots of that figure are four *different* runs of our *within-measure assessment rig* (defined in Section 3.1). Hence, it is reasonable to check what happens when (a) one evaluation criteria is used to control SMOTUNED, and (b) the results are assessed using all four evaluation criteria. Figure 10 shows the results of such a *cross-measure assessment rig* where AUC was used to control SMOTUNED. We note that the results in this figure are very similar to Figure 6, e.g., the precision deltas aver usually tiny, and false alarm increases are usually smaller than the associated recall improvements. But there are some larger improvements in Figure 6 than Figure 10. Hence, we recommend cross-measure assessment only if CPU is critically restricted. Otherwise, we think SMOTUNED should be controlled by whatever is the downstream evaluation criteria (as done in the within-measure assessment rig of Figure 6.)

6 CONCLUSION

Prior work on ranking studies tried to improve software analytics by selecting better learners. Our results show that there may be *more* benefits in exploring data pre-processors like SMOTUNED because we found that no learner was usually "best" across all data sets and all evaluation criteria. On one hand, across the same data sets, SMOTUNED was consistently used by whatever learner was found to be "best" in the AUC/recall results. On the other hand, for the precision and false alarm results, there was little evidence against the use of SMOTUNED. That is, creating better training data (using techniques like SMOTUNED) may be more important than the subsequent choice of a classifier. To say that another way, at least for defect prediction, "better data" is better than "better data miners".

As to specific recommendations, we suggest that any prior ranking study which did not study the effects of data pre-processing needs to be analyzed again. Any future such ranking study should include a SMOTE-like pre-processor. SMOTE should not be used with its default parameters. For each new data set, SMOTE should be used with some automatic parameter tuning tool in order to find the best parameters for that data set. SMOTUNED is one of the examples of parameter tuning. Ideally, SMOTUNED should be tuned using the evaluation criteria used to assess the final predictors. However, if there is not enough CPU to run SMOTUNED for each new evaluation criteria, SMOTUNED can be tuned using AUC.

REFERENCES

[1] Charu C Aggarwal, Alexander Hinneburg, and Daniel A Keim. 2001. On the surprising behavior of distance metrics in high dimensional space. In *International Conference on Database Theory*. Springer, 420–434.

[2] Amritanshu Agrawal, Wei Fu, and Tim Menzies. 2016. What is wrong with topic modeling?(and how to fix it using search-based se). *arXiv preprint*

arXiv:1608.08176 (2016).

[3] Andrea Arcuri and Lionel Briand. 2011. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *Software Engineering (ICSE), 2011 33rd International Conference on*. IEEE, 1–10.

[4] Kwabena Ebo Bennin, Jacky Keung, Passakorn Phannachitta, Akito Monden, and Solomon Mensah. 2017. MAHAKIL: Diversity based Oversampling Approach to Alleviate the Class Imbalance Issue in Software Defect Prediction. *IEEE Transactions on Software Engineering* (2017).

[5] Christian Bird, Nachiappan Nagappan, Harald Gall, Brendan Murphy, and Premkumar Devanbu. 2009. Putting it all together: Using socio-technical networks to predict failures. In *2009 20th ISSRE*. IEEE, 109–119.

[6] Cagatay Catal and Banu Diri. 2009. A systematic review of software fault prediction studies. *Expert systems with applications* 36, 4 (2009), 7346–7354.

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

[8] Shyam R Chidamber and Chris F Kemerer. 1994. A metrics suite for object oriented design. *IEEE Transactions on software engineering* 20, 6 (1994), 476–493.

[9] I. Chiha, J. Ghabi, and N. Liouane. 2012. Tuning PID controller with multi-objective differential evolution. In *ISCCSP ’12*. IEEE, 1–4.

[10] Marco D’Ambros, Michele Lanza, and Romain Robbes. 2010. An extensive comparison of bug prediction approaches. In *2010 7th IEEE MSR)*. IEEE, 31–41.

[11] Richard O Duda, Peter E Hart, and David G Stork. 2012. *Pattern classification*. John Wiley & Sons.

[12] Bradley Efron and Robert J Tibshirani. 1994. *An introduction to the bootstrap*. Chapman and Hall, London.

[13] Karim O Elish and Mahmoud O Elish. 2008. Predicting defect-prone software modules using support vector machines. *JSS* 81, 5 (2008), 649–660.

[14] Wei Fu and Tim Menzies. 2017. Easy over Hard: A Case Study on Deep Learning. *arXiv preprint arXiv:1703.00133* (2017).

[15] Wei Fu and Tim Menzies. 2017. Revisiting unsupervised learning for defect prediction. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 72–83.

[16] Wei Fu, Tim Menzies, and Xipeng Shen. 2016. Tuning for software analytics: Is it really necessary? *IST* 76 (2016), 135–146.

[17] Baljinder Ghotra, Shane McIntosh, and Ahmed E Hassan. 2015. Revisiting the impact of classification techniques on the performance of defect prediction models. In *37th ICSE-Volume 1*. IEEE Press, 789–800.

[18] Iker Gondra. 2008. Applying machine learning to software fault-proneness prediction. *Journal of Systems and Software* 81, 2 (2008), 186–195.

[19] David Gray, David Bowes, Neil Davey, Yi Sun, and Bruce Christianson. 2009. Using the support vector machine as a classification method for software defect prediction with static code metrics. In *International Conference on Engineering Applications of Neural Networks*. Springer, 223–234.

[20] Philip J Guo, Thomas Zimmermann, Nachiappan Nagappan, and Brendan Murphy. 2011. Not my bug! and other reasons for software bug report reassignments. In *Proceedings of the ACM 2011 conference on Computer supported cooperative work*. ACM, 395–404.

[21] Tracy Hall, Sarah Beecham, David Bowes, David Gray, and Steve Counsell. 2012. A systematic literature review on fault prediction performance in software engineering. *IEEE TSE* 38, 6 (2012), 1276–1304.

[22] Ahmed E Hassan. 2009. Predicting faults using the complexity of code changes. In *31st ICSE*. IEEE Computer Society, 78–88.

[23] Haibo He and Edwardo A Garcia. 2009. Learning from imbalanced data. *IEEE Transactions on knowledge and data engineering* 21, 9 (2009), 1263–1284.

[24] Zhimin He, Fengdi Shu, Ye Yang, Mingshu Li, and Qing Wang. 2012. An investigation on the feasibility of cross-project defect prediction. *Automated Software Engineering* 19, 2 (2012), 167–199.

[25] Yue Jiang, Bojan Cukic, and Yan Ma. 2008. Techniques for evaluating fault prediction models. *Empirical Software Engineering* 13, 5 (2008), 561–595.

[26] Yue Jiang, Bojan Cukic, and Tim Menzies. 2008. Can data transformation help in the detection of fault-prone modules?. In *Proceedings of the 2008 workshop on Defects in large software systems*. ACM, 16–20.

[27] Yue Jiang, Jie Lin, Bojan Cukic, and Tim Menzies. 2009. Variance analysis in software fault prediction models. In *Software Reliability Engineering, 2009. ISSRE’09. 20th International Symposium on*. IEEE, 99–108.

[28] Yasutaka Kamei, Akito Monden, Shinsuke Matsumoto, Takeshi Kakimoto, and Ken-ichi Matsumoto. 2007. The effects of over and under sampling on fault-prone module detection. In *ESEM 2007*. IEEE, 196–204.

[29] Taghi M Khoshgoftaar, Kehan Gao, and Naeem Seliya. 2010. Attribute selection and imbalanced data: Problems in software defect prediction. In *Tools with Artificial Intelligence (ICTAI), 2010 22nd IEEE International Conference on*, Vol. 1. IEEE, 137–144.

[30] Sunghun Kim, Hongyu Zhang, Rongxin Wu, and Liang Gong. 2011. Dealing with noise in defect prediction. In *Software Engineering (ICSE), 2011 33rd International Conference on*. IEEE, 481–490.

[31] Rahul Krishna, Tim Menzies, and Lucas Layman. 2017. Less is More: Minimizing Code Reorganization using XTREE. *Information and Software Technology* (2017).

[32] Stefan Lessmann, Bart Baesens, Christophe Mues, and Swantje Pietsch. 2008. Benchmarking classification models for software defect prediction: A proposed framework and novel findings. *IEEE TSE* 34, 4 (2008), 485–496.

[33] Ming Li, Hongyu Zhang, Rongxin Wu, and Zhi-Hua Zhou. 2012. Sample-based software defect prediction with active and semi-supervised learning. *Automated Software Engineering* 19, 2 (2012), 201–230.

[34] Michael Lowry, Mark Boyd, and Deepak Kulkami. 1998. Towards a theory for integration of mathematical verification and empirical testing. In *Automated Software Engineering, 1998. Proceedings. 13th IEEE International Conference on*. IEEE, 322–331.

[35] Thilo Mende and Rainer Koschke. 2009. Revisiting the evaluation of defect prediction models. In *Proceedings of the 5th International Conference on Predictor Models in Software Engineering*. ACM, 7.

[36] Tim Menzies, Alex Dekhtyar, Justin Distefano, and Jeremy Greenwald. 2007. Problems with Precision: A Response to" comments on’data mining static code attributes to learn defect predictors’". *IEEE TSE* 33, 9 (2007).

[37] Tim Menzies and Justin S. Di Stefano. 2004. How Good is Your Blind Spot Sampling Policy. In *Proceedings of the Eighth IEEE International Conference on High Assurance Systems Engineering (HASE’04)*. IEEE Computer Society, Washington, DC, USA, 129–138. http://dl.acm.org/citation.cfm?id=1890580.1890593

[38] Tim Menzies, Jeremy Greenwald, and Art Frank. 2007. Data mining static code attributes to learn defect predictors. *IEEE TSE* 33, 1 (2007), 2–13.

[39] Tim Menzies, Ekrem Kocaguneli, Burak Turhan, Leandro Minku, and Fayola Peters. 2014. *Sharing data and models in software engineering*. Morgan Kaufmann.

[40] Tim Menzies, Zach Milton, Burak Turhan, Bojan Cukic, Yue Jiang, and Ayşe Bener. 2010. Defect prediction from static code features: current results, limitations, new approaches. *Automated Software Engineering* 17, 4 (2010), 375–407.

[41] Tim Menzies and Erik Sinsel. 2000. Practical large scale what-if queries: Case studies with software risk assessment. In *Automated Software Engineering, 2000. Proceedings ASE 2000. The Fifteenth IEEE International Conference on*. IEEE, 165–173.

[42] Tim Menzies, Burak Turhan, Ayse Bener, Gregory Gay, Bojan Cukic, and Yue Jiang. 2008. Implications of ceiling effects in defect predictors. In *Proceedings of the 4th international workshop on Predictor models in software engineering*. ACM, 47–54.

[43] Nikolaos Mittas and Lefteris Angelis. 2013. Ranking and clustering software cost estimation models through a multiple comparisons algorithm. *IEEE Transactions on software engineering* 39, 4 (2013), 537–551.

[44] Nachiappan Nagappan and Thomas Ball. 2005. Static analysis tools as early indicators of pre-release defect density. In *Proceedings of the 27th international conference on Software engineering*. ACM, 580–586.

[45] Nachiappan Nagappan, Thomas Ball, and Andreas Zeller. 2006. Mining metrics to predict component failures. In *Proceedings of the 28th international conference on Software engineering*. ACM, 452–461.

[46] M. Omran, A. P. Engelbrecht, and A. Salman. 2005. Differential evolution methods for unsupervised image classification. In *IEEE Congress on Evolutionary Computation ’05*, Vol. 2. 966–973.

[47] Russel Pears, Jacqui Finlay, and Andy M Connor. 2014. Synthetic Minority oversampling technique (SMOTE) for predicting software build outcomes. *arXiv preprint arXiv:1407.2330* (2014).

[48] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, and others. 2011. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research* 12, Oct (2011), 2825–2830.

[49] Lourdes Pelayo and Scott Dick. 2007. Applying novel resampling strategies to software defect prediction. In *NAFIPS 2007-2007 Annual Meeting of the North American Fuzzy Information Processing Society*. IEEE, 69–72.

[50] Lourdes Pelayo and Scott Dick. 2012. Evaluating stratification alternatives to improve software defect prediction. *IEEE Transactions on Reliability* 61, 2 (2012), 516–525.

[51] Fayola Peters, Tim Menzies, Liang Gong, and Hongyu Zhang. 2013. Balancing privacy and utility in cross-company defect prediction. *IEEE Transactions on Software Engineering* 39, 8 (2013), 1054–1068.

[52] Fayola Peters, Tim Menzies, and Andrian Marcus. 2013. Better cross company defect prediction. In *Mining Software Repositories (MSR), 2013 10th IEEE Working Conference on*. IEEE, 409–418.

[53] Danijel Radjenović, Marjan Heričko, Richard Torkar, and Aleš Živkovič. 2013. Software fault prediction metrics: A systematic literature review. *Information and Software Technology* 55, 8 (2013), 1397–1418.

[54] Foyzur Rahman, Sameer Khatri, Earl T. Barr, and Premkumar Devanbu. 2014. Comparing Static Bug Finders and Statistical Prediction *(ICSE)*. ACM, New York, NY, USA, 424–434. DOI: http://dx.doi.org/10.1145/2568225.2568269

[55] Mitch Rees-Jones, Matthew Martin, and Tim Menzies. 2017. Better Predictors for Issue Lifetime. *CoRR* abs/1702.07735 (2017). http://arxiv.org/abs/1702.07735

[56] Payam Refaeilzadeh, Lei Tang, and Huan Liu. 2009. Cross-validation. In *Encyclopedia of database systems*. Springer, 532–538.

[57] JC Riquelme, R Ruiz, D Rodríguez, and J Moreno. 2008. Finding defective modules from highly unbalanced datasets. *Actas de los Talleres de las Jornadas de Ingeniería*

del Software y Bases de Datos 2, 1 (2008), 67–74.
[58] Martin Shepperd, David Bowes, and Tracy Hall. 2014. Researcher bias: The use of machine learning in software defect prediction. *IEEE Transactions on Software Engineering* 40, 6 (2014), 603–616.
[59] Rainer Storn and Kenneth Price. 1997. Differential evolution–a simple and efficient heuristic for global optimization over continuous spaces. *Journal of global optimization* 11, 4 (1997), 341–359.
[60] John A Swets. 1988. Measuring the accuracy of diagnostic systems. *Science* 240, 4857 (1988), 1285.
[61] Ming Tan, Lin Tan, Sashank Dara, and Caleb Mayeux. 2015. Online defect prediction for imbalanced data. In *ICSE-Volume 2*. IEEE Press, 99–108.
[62] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E Hassan, and Kenichi Matsumoto. 2016. Automated parameter optimization of classification techniques for defect prediction models. In *ICSE 2016*. ACM, 321–332.
[63] Burak Turhan, Ayşe Tosun Mısırlı, and Ayşe Bener. 2013. Empirical evaluation of the effects of mixed project data on learning defect predictors. *Information and Software Technology* 55, 6 (2013), 1101–1118.
[64] Jason Van Hulse, Taghi M Khoshgoftaar, and Amri Napolitano. 2007. Experimental perspectives on learning from imbalanced data. In *Proceedings of the 24th international conference on Machine learning*. ACM, 935–942.
[65] Jakob Vesterstrom and Rene Thomsen. 2004. A comparative study of differential evolution, particle swarm optimization, and evolutionary algorithms on numerical benchmark problems. In *Evolutionary Computation, 2004. CEC2004. Congress on*, Vol. 2. IEEE, 1980–1987.
[66] Jeffrey M. Voas and Keith W Miller. 1995. Software testability: The new verification. *IEEE software* 12, 3 (1995), 17–28.
[67] Shuo Wang and Xin Yao. 2013. Using class imbalance learning for software defect prediction. *IEEE Transactions on Reliability* 62, 2 (2013), 434–443.
[68] Zhen Yan, Xinyu Chen, and Ping Guo. 2010. Software defect prediction using fuzzy support vector regression. In *International Symposium on Neural Networks*. Springer, 17–24.
[69] Qiao Yu, Shujuan Jiang, and Yanmei Zhang. 2017. The Performance Stability of Defect Prediction Models with Class Imbalance: An Empirical Study. *IEICE TRANSACTIONS on Information and Systems* 100, 2 (2017), 265–272.
[70] Hongyu Zhang and Xiuzhen Zhang. 2007. Comments on Data Mining Static Code Attributes to Learn Defect Predictors. *IEEE Transactions on Software Engineering* 33, 9 (2007), 635–637.
[71] Thomas Zimmermann and Nachiappan Nagappan. 2008. Predicting defects using network analysis on dependency graphs. In *ICSE*. ACM, 531–540.