Software Analytics without Tuning Considered Harmful: A Case Study with SMOTE

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

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

***Abstract*—When data is highly imbalanced, it is hard for data miners to find the target class. Such imbalanced datasets are common in software engineering, particularly in the field of defect prediction.**

One way to handle imbalance datasets is using the SMOTE procedure (Synthetic Minority Oversampling Technique) which throws away random examples of the majority class while synthesizing more examples of the minority class. The standard SMOTE, referred to as SMOTE1, uses a set of default parameters. Our tuned version of SMOTE1, referred to as SMOTE2, uses parameters found by an optimization technique called Differential Evolution (DE).

In studies with SMOTE1 and SMOTE2, performed on 9 defect datasets, we observe that SMOTE1 may or may not be useful depending on the evaluation goal. We observed mixed results for precision (sometimes better sometimes worse). On the other hand, SMOTE2 lead to dramatic improvements in AUC (pf, recall) and modest improvements were seen in recall and these were much better than SMOTE1.

We conclude that (a) SMOTE is useful but (b) it is best *not* to use the default parameters of standard SMOTE1. More generally, we caution all researchers to tune their analytics tools before trusting the results they generate.

Keywords*—*Software repository mining, search based software engineering, defect prediction, classification, data analytics for software engineering, SMOTE, imbalanced data, preprocessing

1. Introduction

Since time and manpower are finite resources, it makes sense to assign personnel and/or resources to areas of a software system with a higher probable quantity of defects. Currently researchers working in defect prediction focuses on (i) estimating the number of defects remaining in software systems, (ii) discovering defect associations, and (iii) classifying the defect-proneness of software components, typically into two classes defect-prone and not defect-prone.

This paper deals with the third type of problem for code metrics (especially, the CK object-oriented metrics [10]). Our task is to classify the defect-proneness of software components, typically into two classes, defective and not defective.

One issue with this kind of research is deciding which data mining algorithm is best for a particular dataset. In a comparative study of numerous learners for defect prediction, Ghotra et al. [18] recommended six learners: Naive Bayes, Logistic regression, Support Vector Machines, Nearest Neighbor, decision trees and Random forest. One of the limitations of their analysis was the effects of class imbalance for such models was not explored. This is an important property since most software systems have less percentage of defective classes compared to non-defective class. Accordingly, this paper explores the Ghotra et al. learners using datasets that have an increasingly large imbalances in the frequency of the defect class. For example, here we explore datasets where the percent of defective classes range down to 2%. Handling class imbalance is important since it is known to adversely affect many machine learning algorithms such as decision tress, neural networks or support vectors machines [26].

Accordingly, we evaluate the performance of learners with imbalanced data using a class balancing technique. We examine a standard off-the-shelf SMOTE, called SMOTE1, which uses the default parameters for that algorithm. The results were not promising, so we applied a hyperparameter optimization method called differential evolution, or DE [60] to tune the parameters of SMOTE1. We refer this as tuned version SMOTE2. Hyperparameter optimization chooses a set of hyperparameters for a learning algorithm, usually with the goal of optimizing a measure of the algorithm's performance on an independent dataset.

Using SMOTE1 and SMOTE2, we explore the following questions:

RQ1: **Is standard "off-the-shelf" SMOTE1 preprocessing method useful for defect prediction?**

Result 1
For defect data, SMOTE1 has adverse effect on precision, modest improvements for AUC (pf, recall) and large improvements in recall.

RQ2: **Can SMOTE2 achieve better results?**

Result 2
For defect data, SMOTE2 offered improvements over SMOTE1 for recall and dramatic improvements for AUC (pf, recall).

RQ3: **Does hyperparameter optimization lead to different optimal configurations for different datasets?**

Result 3
Yes. DE finds different "best" parameter settings for SMOTE on different datasets.

This is an important result since it means reusing tunings suggested by any other previous study for a dataset different from the one under study is *not* recommended. Instead, it is better to use automatic tuning methods to find the best tuning parameters for the under study dataset.

Note that before we demand that tuning should be a standard for all analytics task, we must assess the practicality of that proposal. This leads to the next question:

RQ4: **Is tuning impractically slow?**

Result 4
Tuning using DE makes training runtimes four to five times slower, but given the large performance improvements, the extra effort is justifiable.

The rest of this paper is structured as follows: §2-A gives an overview of software defect prediction. §2-B explains the importance of balancing the minority class. §2-C introduces SMOTE and discusses how SMOTE has been used in literature. Advantages of tuning is mentioned in §2-D and the experimental setup of this paper is discussed in §3 We have answered above research questions in §4 This is followed by a discussion on the validity of our results and a section describing our conclusions and future work.

2. Related Work

A. Defect Prediction

Much prior work has estimated number of defects remaining in software systems [22] using statistical approaches, capture-recapture (CR) models, and detection profile methods (DPM) [58] or association rule mining [59]. A variety of approaches have been proposed to tackle the problem of classifying the defect-proneness of software components. It heavily relied on diverse information, such as code metrics (lines of code, complexity) [11], [39], [43], [57], [40], process metrics (number of changes, recent activity) [23] or previous defects [31].

Bird et al. [5] indicate that it is possible to predict which components (for example modules) are likely locations of defect occurrence using a component's development history, and dependency structure. Two key properties of software components in large systems are dependency relationships (which components depend on or are dependent on by others), and development history (who made changes to the components and how many times). Thus, we can link software components to other components i) in terms of their dependencies, and also ii) in terms of the developers that they have in common. Prediction models based on the topological properties of components within them have proven to be quite accurate [76].

By keeping change logs of the most recently or frequently changed files are the most probable source of future defects [22], [7]. These mentioned papers compared various code metrics like CK metrics suite, McCabes cyclomatic complexity, Briands coupling metrics, code metrics, dependencies between binaries. The CK metrics aim at measuring whether a piece of code follows OO principles. It contains a check of these OO design attributes which are explained in Figure 1. CK metrics are a set of metrics and it has added advantage than other OO and static code attributes metrics [11].

There is other added advantage that come with CK metrics is they are simple to compute. CK metrics are endorsed by much of the community since it has been used in past covering research papers about twice (49%) as more traditional source code metrics (27%) or process metrics (24%) [53].

There has been vast amount of studies done to find the best defect prediction performing model. But literature suggests [33], [18], that no single prediction technique dominates and making an informed decision is challenging since the techniques are usually applied on datasets with different levels of imbalance, which are preprocessed differently. The level of imbalance is not the only factor that affects the performance of the classifiers, but other factors such as the degree of data overlapping (represented as duplicates) among the classes, dataset shift (training and test data follow different distributions), small disjuncts, the lack of density or small sample size, the correct management of borderline examples or noisy data adversely affect the performance of defect predictors [36], [37]. Many of these problems are related to how to measure these data characteristics and the quality of data. For instance, Van et al. [66] explored into how the level of noise in data (quality) impact the performance of the classifiers. This makes a necessary argument to study preprocessing filters and how it can affect the performance of classifiers.

Different data preprocessing has been proved to improve the performance of defect prediction models by Menzies et al. [39]. Jiang et al. [28] evaluate the impact of log transformation and discretization on the performance of defect prediction models, and report different modeling techniques "prefer" different transformation techniques. For instance, Naive Bayes achieves better performance on discretized data, while logistic regression achieves better performance for both. Peters et al. [52] propose different filters; and Li et al. [35] propose to use sampling. Nam et al. [44] transformed both training and testing data to the same latent feature space, and build models on the latent feature space. Feature Selection is a common method that can reduce features and sampling can balance the diversity of class instance numbers [74], in turn improving the performance of defect prediction. In this paper we only tackle the class imbalance problem.

To tackle the variations available among classifiers, some studies suggested to tune the learners to find the best parameter settings [63], [17]. Both the studies report how performance of a defect predictor is dependent on the parameter settings of the predictors and recommends to use automated ways to tune the predictors. Recent studies [17], [2] endorse DE as a method to perform automated tuning (also known as hyperparameter optimization). This was the motivation to use an automatic method like DE to tune the settings of SMOTE.

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 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	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	
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*	

Fig. 1: OO code metrics used for all studies in this paper. Last line, shown in denote the dependent variable.

In terms of the paper with most influential work, our experimental methods to select which learners to pick are informed by Ghotra et al. [18] on "Revisiting the impact of classification techniques on the performance of defect prediction models". To compare the performance of defect prediction models, they used the Area Under the Receiver Operating Characteristic (ROC) Curve (AUC), which plots false positive rate against true positive rate. We denote this as $AUC\ (pf, recall)$. They ran the Scott-Knott test [27] to group classification techniques into statistically distinct ranks. After running these evaluation criteria, they concluded that Naive Bayes, Logistic regression, Support Vector Machines, Nearest Neighbor, decision tree and Random forest performed the best depending on various datasets. Accordingly, we use these learners in this paper.

B. Class Imbalance

Class imbalance learning refers to learning from datasets that exhibit significant imbalance among or within classes. Class imbalance is concerned with the situation in where some classes of data are highly under-represented compared to other classes [24]. 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 ratio of minority class to majority *increases*, that is, *class-imbalance of 5:95* is worse than *20:80*.

Misclassifying an example from the minority class is usually more expensive. In software defect prediction, due to the nature of the problem, the defect case is much less likely to happen than the non-defect case. The failure of finding a defect could degrade software quality greatly (since more bugs are delivered to the client). Hence, our learning objective can be described as *obtaining a classifier that will provide higher performances for the minority class*.

Numerous methods have been proposed to tackle class imbalance problems at data and algorithm levels. Data-level methods manipulate the training data to select some samples for training. This include a variety of resampling techniques, random oversampling, random undersampling, and SMOTE [15]. Algorithm-level methods address class imbalance by modifying their training mechanism directly with the goal of better accuracy on the minority class, including cost-sensitive learning algorithms [24]. Algorithm-level methods require specific treatments for different kinds of learning algorithms, which hinders their use in many applications, because we do not know in advance which algorithm would be the best choice in most cases. In addition to the aforementioned data-level and algorithm-level solutions, ensemble learning has become another promising category of approaches to handle imbalanced data by combining multiple classifiers, such as SMOTEBoost [9], and AdaBoost.NC [69]. To the best of our knowledge, none of these methods have thoroughly investigated the class imbalance problem.

Hall et al. [22] found that models based on C4.5 underperform if they have imbalanced data while Naive Bayes and Logistic regression perform relatively better. Their general recommendation is to not use imbalanced data.

Yu et al. [75] validated the Hall et al. results and concluded that the performance of C4.5 is unstable on imbalanced datasets. They studied the stability issues due to class imbalance and found out that Random Forest and Naive Bayes are the most stable among the learners explored. They generated synthetic datasets from the original one to build class imbalance datasets. This could be affected by random sampling and heance may not be the ideal way to build imbalanced datasets. They also used default set of parameters but wanted to study the effects of tuning classifiers as well as mentioned to use techniques like SMOTE for class imbalance in their future work.

Wang et al. [70] studied various undersampling and oversampling technique and compared the results with Naive Bayes and Random Forest. Other interesting finding from Wang et al. was to use AdaBoost.NC, which has better performance than the rest while others like Gray et al. [21] is planning to use SMOTE in future studies. Yan et al. [72] performed fuzzy logic and rules to overcome the imbalance problem only to work with Support Vector Machines.

Pelayo et al. [49] studied the effects of 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. [41] undersampled the non-defect class to balance training data, and report 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, and instead improves the performance of C4.5. Some other papers also showed the usefulness of resampling based on different learners [49], [50], [55].

C. On the Value of SMOTE for SE

This section lists some of the results achieved by SMOTE for SE datasets. Some authors [67], [62] found SMOTE to be advantageous, others [49] do not. This difference in opinion was one of the main motivations for this paper.

SMOTE works by creating a new minority-class sample at a random point along the line segments joining any/all of the k minority class nearest neighbors. Depending upon the amount (m, number of samples) of over-sampling required, neighbors from the k nearest neighbors are randomly chosen. To find nearest neighbors different distance metric is used. Most common distance metric Minkowski [12] which has power (r) in its formula.

Now coming to the use cases of SMOTE in defect prediction, Pears et al. [47] used SMOTE to study software build outcomes. They observed that classification accuracy steadily improves after creating approximately 900 instances of builds that have been fed to the classifier. Tan et al. [62] investigated online defect prediction for imbalance data. They studied resampling techniques to remove imbalance improving precision by 6.4 - 34.8%.

Pelayo [49] found out that by using SMOTE, there was no improvement in defect prediction but other resampling strategies like trial-and-error, showed promising results by arriving at the highest geometric mean accuracies. Kamei et al. [30] evaluated the effects of SMOTE applied to only four fault-proneness models (linear discriminant analysis, logistic regression analysis, neural network and classification tree) by using two module sets of industry legacy software. They reported that SMOTE improved the prediction performance of the linear and logistic models, while neural network and classification tree models did not benefit from it. Van et al. [67] identified that classifier performance is improved with SMOTE, but individual learners respond differently on sampling. We will be studying many more models and will show that SMOTE does help in all prediction models depending on tuning goal.

D. Importance of Tuning

The previous section documented a strange difference of opinion about the value of SMOTE. One explanation for these differing results is that SMOTE runs with different control parameters. The impact of tuning a learner's control parameters is well understood in the theoretical machine learning literature [4]. When we tune a data miner for its performance, what we are really doing is changing how a learner applies its heuristics. This means tuned data miners use different heuristics, which means they ignore different possible models, which means they return different models; i.e. *how* we learn changes *what* we learn.

Yet issues relating to tuning are poorly addressed in the software analytics literature. Fu et al. [17] surveyed a few of recent SE papers in the area of software defect prediction from static code attributes and found that SE authors, with the exception of [64] do not take steps to explore tunings. For example, Elish et al. [14] compared support vector machines to other data miners for the purposes of defect prediction. They tested different "off-the-shelf" data miners on the same dataset, without adjusting the parameters of each individual learner. Similar comparisons of data miners in SE, with no or minimal pre-tuning study, can be found in the work on Lessmann et al. [34] and, most recently, in Yang et al [73].

We choose to use DE after a literature search on search-based SE methods. DE is a method that optimizes a problem by iteratively trying to improve a candidate solution with regard to a given measure of quality. In the ast researchers have used many optimizers like: simulated annealing [16], [38]; various genetic algorithms [20] augmented by techniques such as DE (differential evolution [60]), tabu search and scatter search [19], [3], [42], [45]; particle swarm optimization [46]; numerous decomposition approaches that use heuristics to decompose the total space into small problems, then apply a response surface methods [32], [77]. Of these, we use DE for two reasons. Firstly, it has been proven useful in prior SE tuning studies [17], [2]. Secondly, our reading of the current literature is that there are many advocates for differential evolution.

Fu et al. [17] showed that using DE to tune the parameters of software defect prediction learners results in large improvements, and the tuning was really simple. Similarly Agrawal et al. [2] tuned the parameters of LDA (Latent Dirichlet Allocation [6]) and reports how tuning can provide better model stability. This shows how advantageous it is to tune the parameters of a data miner.

3. Experimental Setup

All the data, source code and results can be found online[1]. We employ 3 techniques which we have named as, (a) Baseline learner results without SMOTE (No-SMOTE), (b) Standard SMOTE with default parameters (SMOTE1) and (c) SMOTE1 tuned using DE (SMOTE2). Our implementation currently uses 5 nearest neighbors for SMOTE1 method and for SMOTE2, DE finds the best k. Synthetic samples are generated in the following way:

- Take the difference between the feature vector (sample) under consideration and its nearest neighbor.
- Multiply this difference by a random number between 0 and 1, and add it to the feature vector under consideration.
- This causes the selection of a random point along the line segment between two specific features

To reduce the variation, each of the above techniques is performed with a 5-fold stratified cross validation [54] which means:

- We randomized the order of the training set five times;
- Each time, we divided the data into five bins;

[1] Blinded for peer review

- For each bin (test), we trained on 4 bins (rest) and then tested on the test bin as follows:
  - On the training set, SMOTE's super-sampling selects instances from the minority class and finds "k" nearest neighbors for each instance and then creates new instances using the selected instances and their neighbors until we have "m" numbers of minority class samples. In our data, the minority class is considered the Defective class and all the datasets used here have less than 50% minority samples.
  - On the training set, SMOTE's sub-sampling eliminates instances from the majority class (selected at random) until we have "m" remaining samples.
  - On the test set, we do nothing (i.e. no SMOTE; i.e. we only SMOTE the training samples (leaving the testing data in its natural form).

A. *Data*

We used the datasets available in PROMISE repository[2] [1]. In total, 9 imbalanced datasets are used, which were collected by Jureczko et al. [29]. Statistics of these datasets can be found in Table I. Datasets are sorted with low percentage of defective class to high defective class. These datasets have already been used in various defect prediction case studies by various researchers [25], [52], [51], [65] making these datasets more important.

Version	Dataset Name	Defect %	Non-Defect %
4.3	jEdit	2	98
1.0	Apache Camel	4	96
6.0.3	Apache Tomcat	9	91
2.0	Apache Ivy	11	89
1.0	Arcilook	11.5	88.5
1.0	Redaktor	15	85
1.7	Apache Ant	22	78
1.2	Apache Synapse	33.5	66.5
1.6.1	Apache Velocity	34	66

TABLE I: Dataset Statistics

B. *Preprocessing*

We ignored string columns in the data and assumed that the last column in the dataset is always the target class. Originally, the target class contains number of defects, which we converted to binary, i.e if target class has defect then it represents 1 otherwise it denotes 0. The code assumes user has preprocessed the data before passing it to the learners.

C. *Classifiers*

We used six classifiers which are mentioned in the baseline paper [18] using the default parameters suggested by Ghotra et al.

- **Support Vector Machines (SVM - Linear Kernel)** are supervised learning models with associated learning algorithms that analyze data used for classification and regression analysis. Given a set of training examples, each marked as belonging to one or the other of two categories, an SVM training algorithm builds a model that assigns new examples to one category or the other, making it a non-probabilistic binary linear classifier.
- **Linear Regression (LR)** is an approach for modeling the relationship between a scalar dependent variable y and one or more explanatory variables (or independent variables) denoted x.
- **Naive Bayes (NB)** classifiers are a family of simple probabilistic classifiers based on applying Bayes theorem with strong (naive) independence assumptions between the features.
- **K Nearest Neighbors (KNN -** $K = 8$**)** is a non-parametric method used for classification and regression. In both cases, the input consists of the k closest training examples in the feature space. Since k is even, for tie breaking a random result is selected.
- **Decision Trees (DT - CART, Split Criteria=Entropy)** are a decision support tool that use a tree-like graph or model of decisions and their possible consequences, including chance event outcomes, resource costs, and utility. It is one way to display an algorithm.
- **Random Forests (RF - Split Criteria=Entropy)** are an ensemble learning method for classification, regression and other tasks, that operate by constructing a multitude of decision trees at training time and outputting the class that is the mode of the classes (classification) or mean prediction (regression) of the individual trees.

Similar Ghotra et al. [18], we used $k = 8$ for k Nearest Neighbours. Also for Decision trees and Random Forest we used Entropy as split criteria. Most of these implementations are provided in Scikit-Learn [48] and available open source.

D. *Tuning SMOTE using DE (SMOTE2)*

DE adjusts the parameters of SMOTE given in Table II. Most of these parameters are explained below.

Parameters	Defaults	Tuning Range	Description
k	5	[1,20]	Number of neighbors in SMOTE
m	50%	50,100,200,400	Number of synthetic examples to create.
r	2	[0.1,5]	Power parameter for the Minkowski distance metric.

TABLE II: List of parameters tuned by this paper

In Table II, k controls how many neighbors to choose in minority target class such that synthetic examples can be created between them. It is important to select the number of synthetic examples to create (m) and how much undersampling (m) of majority class needs to be done. In this case number of oversampling and undersampling are the same. To select neighbors, power (r) of Minkowski distance metric is also tunable.

Algorithm 1 is a pseudocode of generic DE. In the following description, superscript numbers denote lines in the pseudocode. DE evolves a *NewGeneration* of candidates from a current population. Each candidate solution in the population is a pair of (Tunings, Scores). Tunings are selected from Table II and one of the scores come from §3-E. Here, the runtimes comes from $iter * np$ evaluations of tuned experiment. The goal of this DE is to either maximize one to these measures like precision, recall, and AUC scores or to minimize the false alarm score. This goal is evaluated on rest data (4 bins). DE is terminated after 3 generations of population which is recommended by Agrawal et al [2].

[2]http://openscience.us/repo/defect/ck/

The main loop of DE[9] runs over the *Population*, replacing old items with new Candidates (if new candidate is better). DE generates *new Candidates* via extrapolating[23] between current solutions in the frontier. Three solutions a, b and 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 booleans, we use $y_i = x_i$ [30]. For numerics, $y_i = a_i + f \times (b_i - c_i)$ where f is a parameter controlling crossover. The trim function[33] limits the new value to the legal range min..max of that parameter.

Algorithm 1 Pseudocode for DE with a constant number of iterations

```
Input: np = 10, f = 0.7, cr = 0.3, iter = 3, Goal ∈ Finding maximum/minimum score
Output: Score, final_generation
function DE(np, f, cr, iter, Goal)
    Cur_Gen ← ∅
    Population ←InitializePopulation(np)
    for i = 0 to np − 1 do
        Cur_Gen.add(Population[i],score)
    end for
    for i = 0 to iter do
        NewGeneration ← ∅
        for j = 0 to np − 1 do
            S_i ←Extrapolate(Population[j],Population,cr,f,np)
            if score(S_i) ≥ Cur_Gen[j][1] then
                NewGeneration.add(S_i,score(S_i))
            else
                NewGeneration.add(Cur_Gen[j])
            end if
        end for
        Cur_Gen ← NewGeneration
    end for
    Score ← GetBestSolution(Cur_Gen)
    final_generation ← Cur_Gen
    return Score, final_generation
end function
function EXTRAPOLATE(old, pop, cr, f, np)
    a, b, c ← threeOthers(pop, old)
    newf ← ∅
    for i = 0 to np − 1 do
        if cr ≤ random() then
            newf.add(old[i])
        else
            if typeof(old[i])== bool then then
                newf.add(not old[i])
            else
                newf.add(trim(i,(a[i]+f∗(b[i] − c[i]))))
            end if
        end if
    end for
    return newf
end function
```

The loop invariant of DE is that, after the i-th iteration[7], the *Population* contains examples that are better than at least one other candidate. As the looping progresses, the *Population* is full of increasingly more valuable solutions which, in turn, also improve the candidates, which are extrapolated from the Population. Hence, Vesterstrom et al. [68] found DE to be competitive with particle swarm optimization and other GAs.

E. Evaluation Measures

Since, this is a binary classification problem, we represent the predictions using a confusion matrix where a 'positive' output is the defective class under study and a 'negative' output is the non defective class. The confusion matrix is shown in Figure 2.

We define the measures as

- **Area Under Curve** is the area covered by an ROC curve [61], [13] in which the X-axis represents:

$$\%FP = \frac{FP}{FP+TN}$$

and the Y-axis represents:

$$\%TP = \frac{TP}{TP+FN}$$

		Actual	
		p	n
Predicted	p'	TP	FP
	n'	FN	TN

Fig. 2: Confusion Matrix

- **Recall** is the fraction of relevant instances that are retrieved.

$$Recall(pd) = \frac{TP}{TP+FN}$$

- **Precision** is the fraction of retrieved instances that are relevant.

$$Precision(prec) = \frac{TP}{TP+FP}$$

- **False Alarm** is the ratio of false positive to predicted negative total.

$$False\ alarm(pf) = \frac{FP}{FP+TN}$$

4. Results

A. *RQ1: Is standard "off-the-shelf" SMOTE1 preprocessing method useful for defect prediction?*

Figure 3 represents the improvement of SMOTE1 (median value) against the No-SMOTE (median value) results and Figure 5 represents the improvement of SMOTE2 (median value) against SMOTE1 (median value) results. Each figure contains subfigures for all our 4 evaluation measures (mentioned in §3-E) compared with 6 learners (mentioned in §3-C).

For subfigures (AUC, Recall and Precision) in each Figure 3 and 5:

- *Larger* y-values are *better*
- If the y-value goes *negative*, then the corresponding learner trained on SMOTE1/SMOTE2 data is *worse* than learner learnt on raw/SMOTE1 data respectively.

For the false alarms, the plots must be interpreted differently:

- *Larger* y-values are *worse*;
- If the y-value goes *positive* then the corresponding learner trained on SMOTE1/SMOTE2 data is *worse* than learner learnt on raw/SMOTE1 data respectively.

In both the figures, the X-axis shows all 9 datasets in the decreasing percentage of defective classes from left to right. The corresponding percentage of minority class (defective class) is written beside each dataset. According to the proponents of SMOTE, SMOTE's improvements should *increase* as we move left to right across those plots.

For many of the results in Figure 3, the changes resulting from applying SMOTE1 are very modest. (often, less than 20%). The two consistent exceptions to that pattern are:

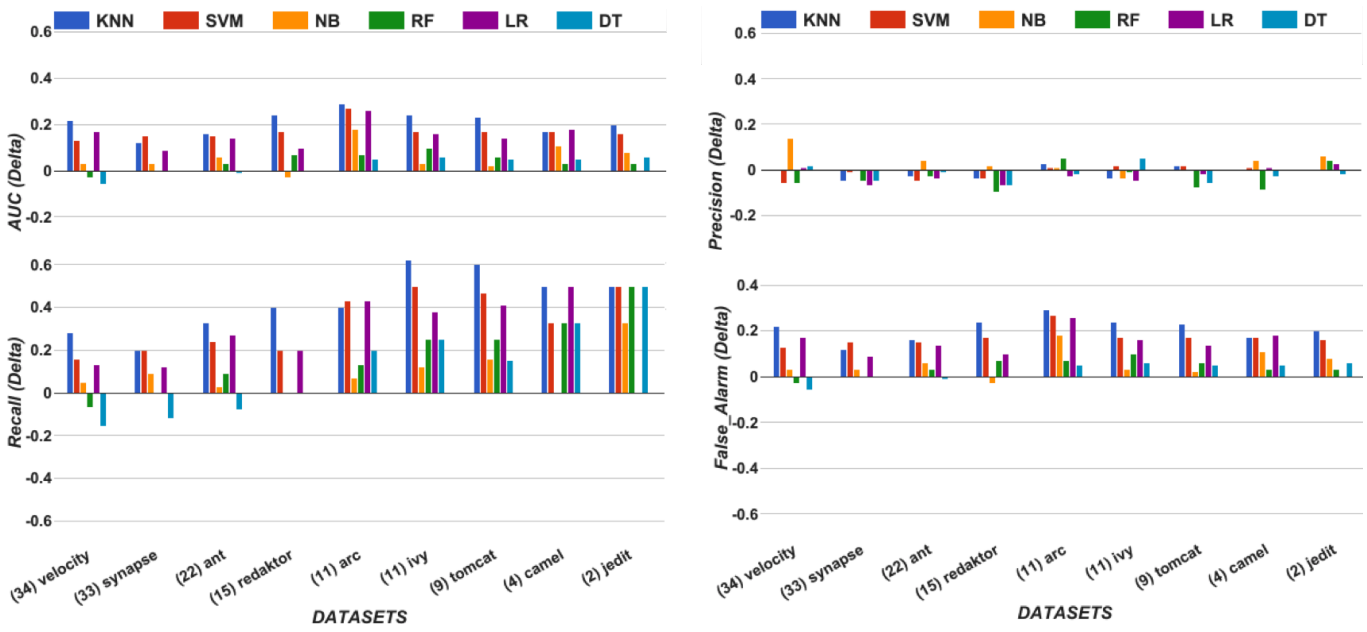

Fig. 3: SMOTE1 improvement over No-SMOTE. Legends represent the classifiers mentioned in §3-C

For subfigures (AUC, Recall and Precision): larger y-values are *better*, if the y-value goes *negative*, then the corresponding learner trained on SMOTE1 data performs *worse* than learner learnt on raw data. For false alarms, the plot must be interpreted differently: larger y-values are *worse*, if the y-value goes *positive*, then the corresponding learner trained on SMOTE1 data performs *worse* than learner learnt on raw data. The corresponding percentage of minority class (in this case, defective class) is written beside each dataset.

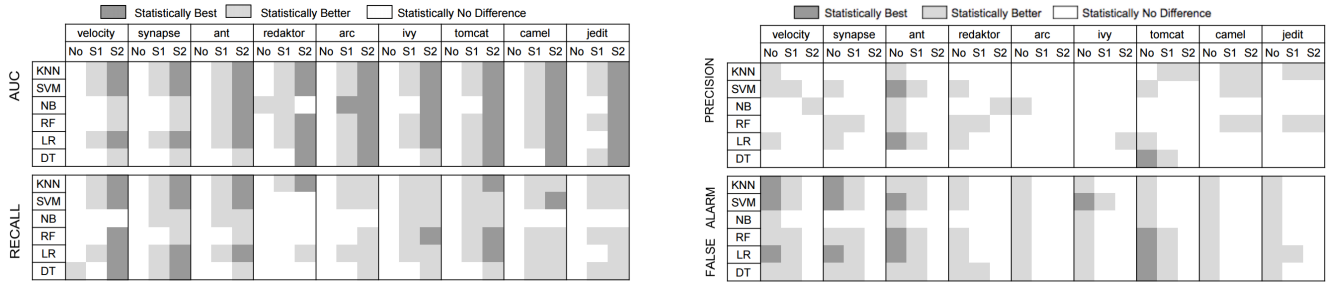

Fig. 4: Scott Knott analysis of No-SMOTE, SMOTE1 and SMOTE2. The column headers are denoted as No for No-SMOTE, S1 for SMOTE1 and S2 for SMOTE2.

- KNN seems to be effected by SMOTE1 more than any other learner.
- Occasionally, SMOTE1 leads to dramatic drops in precision but in many cases, they are not statistically significant different as show in Figure 4.
- Looking left-to-right across Figure 3c, we see more large positive increases on the right-hand-side than the left.

We are not seeing improvement in precision is because of the way in which SMOTE works. False positives (from Figure 2) do not get reduced after applying SMOTE. SMOTE generates synthetic examples within defective data samples and its neighbors making more denser region for positive samples. So if there are any actual negative samples surrounding these denser regions will now get wrongly classified as positive by learners.

Summarizing the above:

Result 1
We recommend SMOTE1 for improving recall, but SMOTE1 can, sometimes, adversely affect precision.

B. *RQ2: Can SMOTE2 achieve better results?*

Figure 5 shows the results after applying SMOTE2. All these plots show the *delta* between the results obtained by SMOTE1 and SMOTE2. Similar to Figure 3 before, *positive* values are better for AUC, recall and precision wile *negative* values are better for false alarms.

The benefit of SMOTE2's tunings is clearly evident in the AUC results of Figure 5a: all learners show large performance improvements. And from the Scott-Knott test shown in Figure 4, SMOTE2 always have significantly best results for AUC. For recall, in most learners either SMOTE1 or SMOTE2 are significantly the same, or SMOTE2 performed better. Better yet, as shown in Figure 4, for precision and false alarm, SMOTE2 does not adversely effect false alarm and precision as most of the times No-SMOTE, SMOTE1, and SMOTE2 are significantly the same.

At first glance, SMOTE2's effects on recall seem strange since they are *better* for the more balanced left-hand-side datasets of Figure 5c. But recall from the above that SMOTE1 had less of an effect on those left-hand-side datasets. That is,

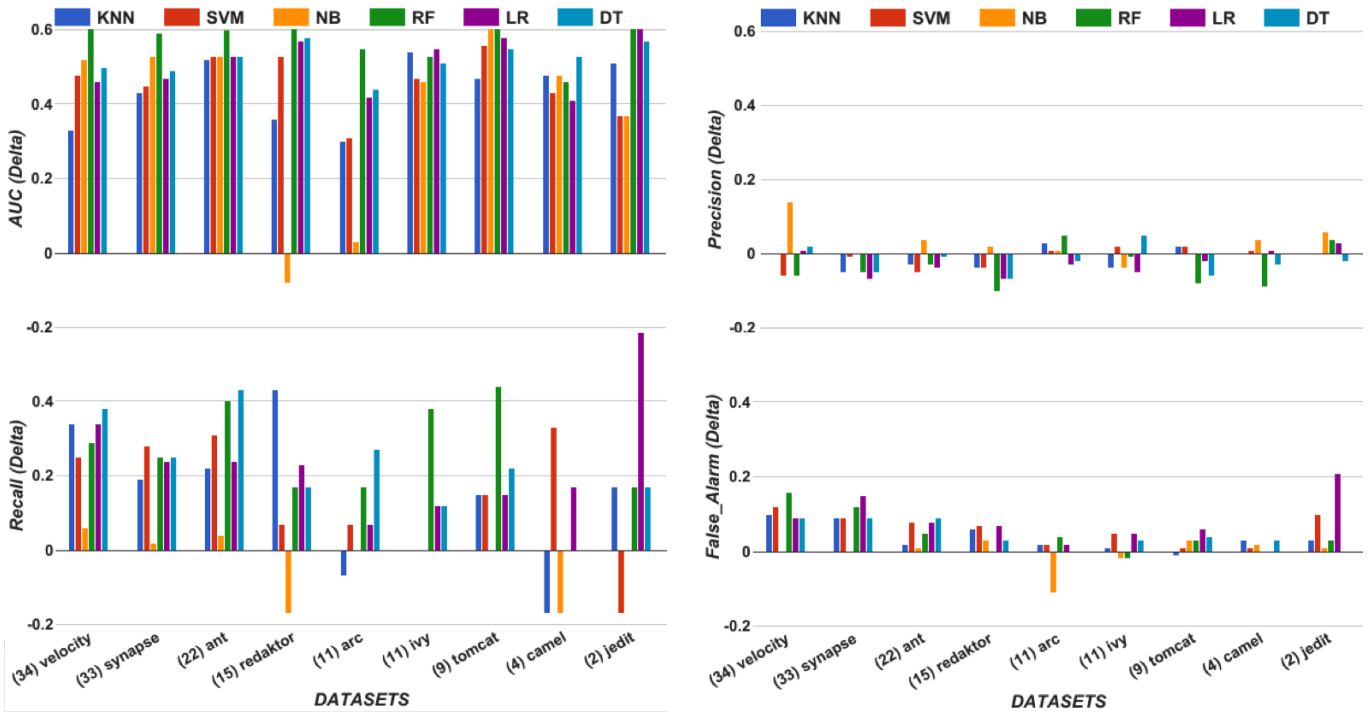

Fig. 5: SMOTE2 improvement over SMOTE1. Legends represent the classifiers mentioned in §3-C.

For subfigures (AUC, Recall and Precision): larger y-values are *better*, if the y-value goes *negative*, then the corresponding learner trained on SMOTE2 data performs *worse* than learner learnt on SMOTE1 data. For false alarms, the plot must be interpreted differently: larger y-values are *worse*, if the y-value goes *positive*, then the corresponding learner trained on SMOTE2 data performs *worse* than learner learnt on SMOTE1 data. The corresponding percentage of minority class (in this case, defective class) is written beside each dataset.

what we are seeing here is that:

- SMOTE2 works often as well as SMOTE1 for imbalanced datasets;
- SMOTE2 offers additional benefits for datasets that are not greatly imbalanced.

In summary:

Result 2
For defect data, SMOTE2 offers some improvements over SMOTE1 for recall and dramatic improvements for AUC (pf, recall). The effects on precision and false alarm are similar to SMOTE1.

C. *RQ3: Does hyperparameter optimization lead to different optimal configurations for different datasets?*

Figure 6 represents the parameter variations when we tuned to maximize the recall. These parameter settings are found by each learner for that dataset. On display in each set of vertical bars are the median values generated across 25 evaluations. Also, shown are the inter-quartile range (IQR) of those tunings (the IQR is the 75th-25th percentile values and is a non-parametric measure of variation around the median value). Note that in Figure 6b, IQR=0 for ant dataset where tuning always converged on the same final value. That shows that the maximum score of recall is reached when $m = 50$. No other parameter setting found out to be useful by DE.

These figures show how tuning selects the different ranges of parameters. Some of the above numbers are far from the standard values; e.g. Chawla et al. [8] recommend using $k = 5$ neighbors yet in our datasets, best results were seen using $k \approx 13$. On other hand it was suggested to use $m = 900$ by [47]. Clearly, best results from tuning vary with each dataset.

Clearly:

Result 3
Yes. DE finds different "best" parameter settings for SMOTE2 on different datasets.

That is, reusing tunings suggested by any other previous study for any dataset is <u>not</u> recommended. Instead, it is better to use automatic tuning methods to find the best tuning parameters for the current dataset.

D. *RQ4: Is tuning impractically slow?*

Search-based SE methods can be very slow. Wang et al. [71] once needed 15 years of CPU time to find and verify the tunings required for software clone detectors. Sayyad et al. [56] routinely used 10^6 evaluations (or more) of their models in order to extract products from highly constrained product lines. Hence, before recommending any search-based method, it is wise to consider the runtime cost of that recommendation.

Figure 8 shows, in circle and square markers, the runtimes required to run SMOTE1 and SMOTE2 respectively. The longer runtimes (in square) include the times required for DE to find the tunings. There is a disadvantage with SMOTE2. Figure 8 shows SMOTE1 reporting all the measures in the shown time. On the other hand, here SMOTE2 reports runtimes

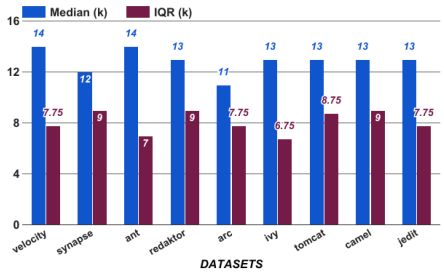

Figure 6a: Tuned values for k.

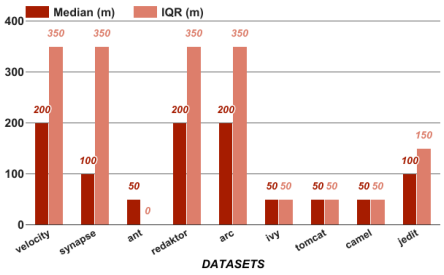

Figure 6b: Tuned values for m.

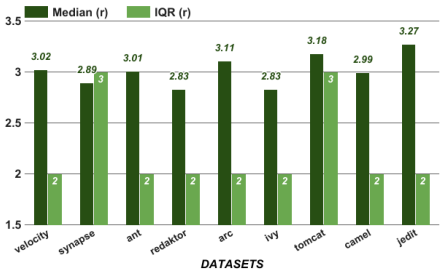

Figure 6c: Tuned values for r.

Fig. 6: Datasets vs Parameter Variation

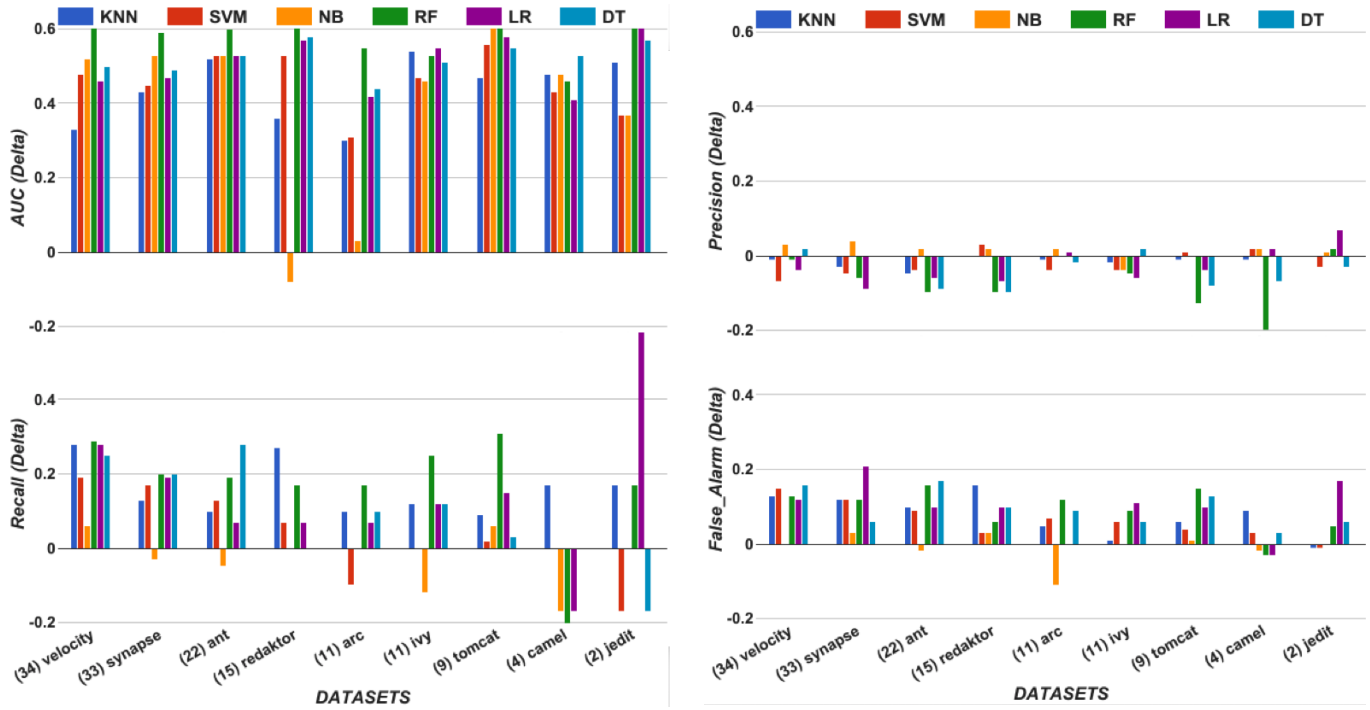

Fig. 7: SMOTE2 improvement over SMOTE1 when tuning goal is to maximize AUC.

For subfigures (AUC, Recall and Precision): larger y-values are *better*, if the y-value goes *negative*, then the corresponding learner trained on SMOTE2 data performs *worse* than learner learnt on SMOTE1 data. For false alarms, the plot must be interpreted differently: larger y-values are *worse*, if the y-value goes *positive*, then the corresponding learner trained on SMOTE2 data performs *worse* than learner learnt on SMOTE1 data. The corresponding percentage of minority class (in this case, defective class) is written beside each dataset.

when trying to maximize recall. If it has to provide all the other 3 measures as well, it would take 3 times more than the above number which is not recommended in a real world scenario.

The work around would be is to try maximizing SMOTE2 for AUC tuning goal. AUC represents the area under the curve for pf (false alarm) and recall. When AUC improves, the heuristic is that false alarm must be getting lower and recall must be improving. And if false alarm is getting lower then precision is improving as well [39]. We report results of AUC, Precision, Recall and False alarm in Figure 7 when we tried maximizing only AUC. We are seeing a similar conclusions as answered in our RQ2.

This additional result shows that, tuning slows down the training by a factor of up to five for 1 goal (which is very close to our theoretical prediction) but the improvements achieved are quite advantageous.

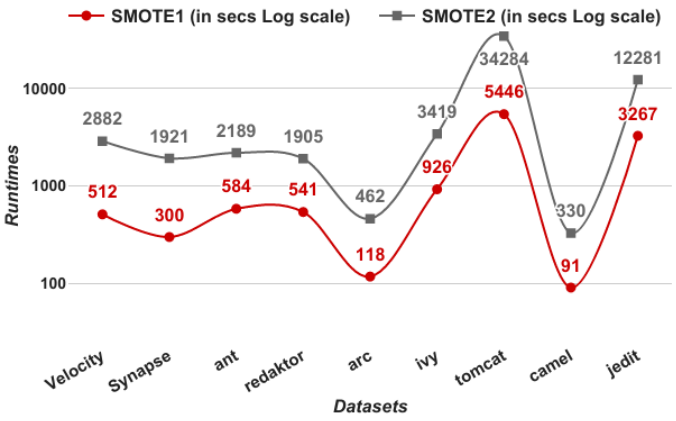

Fig. 8: Datasets vs Runtimes

Result 4
Tuning with DE makes training four to five times slower, but the improvements in performance with respect to AUC and recall justifies the extra time required for training.

5. Threats to Validity

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

Sampling bias threatens any classification experiment; i.e., what matters there may not be true here. For example, the datasets used here comes from the PROMISE [1] repository and were supplied by one individual. But these datasets have used in various case studies by various researchers [25], [52], [51], [65] making our results more conclusive. Though we have used 9 open-source datasets for Software Defect prediction (Table I) which are mostly from Apache, and some are proprietary projects, there could be other datasets for which our results could be wrong.

Learner bias: For building the defect predictors in this study, we selected each learner with default parameters like k=8 in k-NN, entropy as split criteria in RF, and DT. The above predefined parameters have been used in the conclusions made by other studies [18], [63]. But classification is a large and active field and any single study can only use a small subset of the known classification algorithms.

Evaluation bias: Most highly cited papers [18], [63] have only concluded using $AUC\ (pf, recall)$ whereas we went ahead and are reporting 3 other measures. There are other measures used in software engineering which includes Accuracy, Fscore. But we still performed SMOTE2 on Fscore and Accuracy and did not find any improvement (Figure 9). Anyway, accuracy is not the best measure to report for defect prediction as we do not want any true negative numbers included. Fscore is the harmonic mean of precision and recall and if you recall our SMOTE2 results, we see increment in recall as well as decrement in precision making Fscore negligible.

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

6. Conclusion

Based on the above, we offer a specific and general conclusion. Most specifically, we recommend.

- Any study [18] that has reported AUC without studying the effects of class imbalance needs to be analyzed again by applying SMOTE. We say this since SMOTE2 showed big improvements in AUC.
- Unlike the advise of Chawla et al. [8] of using $k = 5$ for SMOTE, we should have automatic methods like DE to tune the parameters of SMOTE and find the best parameter settings.
- Do not use someone else's pre-tuned SMOTE, since, as shown here, the best tunings vary from dataset to dataset.

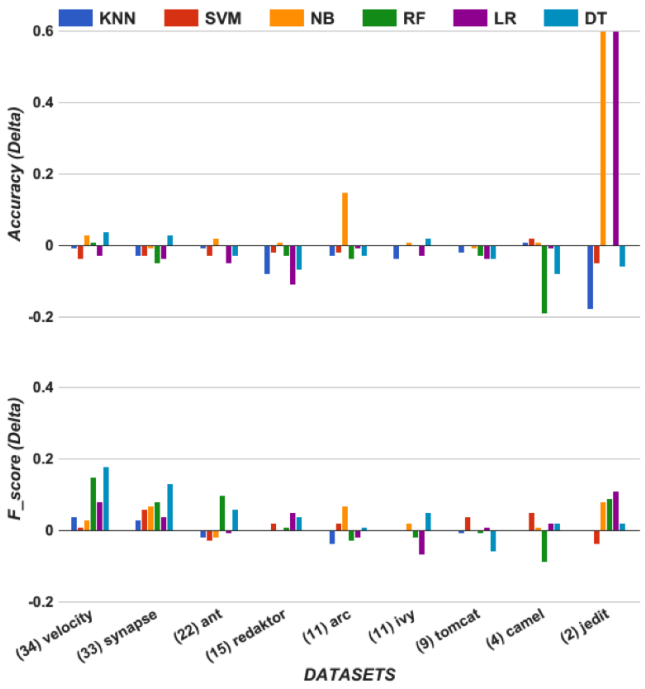

Fig. 9: SMOTE2 improvement over SMOTE1. Legends represent the classifiers mentioned in §3-C

Based on our runtime results, we can say that list last recommendation (to tune before applying analytics) is not a particular onerous demand. The additional cost of tuning is relatively minor while the benefits of the SMTOE2 tunings are very large (see the AUC and recall results shown above).

References

[1] The promise repository of empirical software engineering data, 2015.

[2] A. Agrawal, W. Fu, and T. Menzies. What is wrong with topic modeling?(and how to fix it using search-based se). *arXiv preprint arXiv:1608.08176*, 2016.

[3] R. P. Beausoleil. MOSS multiobjective scatter search applied to nonlinear multiple criteria optimization. *European Journal of Operational Research*, 169(2):426–449, 2006.

[4] J. Bergstra and Y. Bengio. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13(Feb):281–305, 2012.

[5] C. Bird, N. Nagappan, H. Gall, B. Murphy, and P. Devanbu. Putting it all together: Using socio-technical networks to predict failures. In *2009 20th International Symposium on Software Reliability Engineering*, pages 109–119. IEEE, 2009.

[6] D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent dirichlet allocation. In *the Journal of machine Learning research*, volume 3, pages 993–1022. JMLR. org, 2003.

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

[8] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer. Smote: synthetic minority over-sampling technique. *Journal of artificial intelligence research*, 16:321–357, 2002.

[9] N. V. Chawla, A. Lazarevic, L. O. Hall, and K. W. Bowyer. Smoteboost: Improving prediction of the minority class in boosting. In *European Conference on Principles of Data Mining and Knowledge Discovery*, pages 107–119. Springer, 2003.

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

[11] M. D'Ambros, M. Lanza, and R. Robbes. An extensive comparison of bug prediction approaches. In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, pages 31–41. IEEE, 2010.

[12] M. M. Deza and E. Deza. Encyclopedia of distances. In *Encyclopedia of Distances*, pages 1–583. Springer, 2009.

[13] R. O. Duda, P. E. Hart, and D. G. Stork. *Pattern classification*. John Wiley & Sons, 2012.

[14] K. O. Elish and M. O. Elish. Predicting defect-prone software modules using support vector machines. *Journal of Systems and Software*, 81(5):649–660, 2008.

[15] A. Estabrooks, T. Jo, and N. Japkowicz. A multiple resampling method for learning from imbalanced data sets. *Computational intelligence*, 20(1):18–36, 2004.

[16] M. S. Feather and T. Menzies. Converging on the optimal attainment of requirements. In *Requirements Engineering, 2002. Proceedings. IEEE Joint International Conference on*, pages 263–270. IEEE, 2002.

[17] W. Fu, T. Menzies, and X. Shen. Tuning for software analytics: Is it really necessary? *Information and Software Technology*, 76:135–146, 2016.

[18] B. Ghotra, S. McIntosh, and A. E. Hassan. Revisiting the impact of classification techniques on the performance of defect prediction models. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 789–800. IEEE Press, 2015.

[19] F. Glover and C. McMillan. The general employee scheduling problem. an integration of ms and ai. *Computers & operations research*, 13(5):563–573, 1986.

[20] A. T. Goldberg. *On the complexity of the satisfiability problem*. PhD thesis, New York University, 1979.

[21] D. Gray, D. Bowes, N. Davey, Y. Sun, and B. Christianson. 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*, pages 223–234. Springer, 2009.

[22] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell. A systematic literature review on fault prediction performance in software engineering. *IEEE Transactions on Software Engineering*, 38(6):1276–1304, 2012.

[23] A. E. Hassan. Predicting faults using the complexity of code changes. In *Proceedings of the 31st International Conference on Software Engineering*, pages 78–88. IEEE Computer Society, 2009.

[24] H. He and E. A. Garcia. Learning from imbalanced data. *IEEE Transactions on knowledge and data engineering*, 21(9):1263–1284, 2009.

[25] Z. He, F. Shu, Y. Yang, M. Li, and Q. Wang. An investigation on the feasibility of cross-project defect prediction. *Automated Software Engineering*, 19(2):167–199, 2012.

[26] N. Japkowicz and S. Stephen. The class imbalance problem: A systematic study. *Intelligent data analysis*, 6(5):429–449, 2002.

[27] E. G. Jelihovschi, J. C. Faria, and I. B. Allaman. Scottknott: a package for performing the scott-knott clustering algorithm in r. *TEMA (São Carlos)*, 15(1):3–17, 2014.

[28] Y. Jiang, B. Cukic, and T. Menzies. Can data transformation help in the detection of fault-prone modules? In *Proceedings of the 2008 workshop on Defects in large software systems*, pages 16–20. ACM, 2008.

[29] M. Jureczko and L. Madeyski. Towards identifying software project clusters with regard to defect prediction. In *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*, page 9. ACM, 2010.

[30] Y. Kamei, A. Monden, S. Matsumoto, T. Kakimoto, and K.-i. Matsumoto. The effects of over and under sampling on fault-prone module detection. In *First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007)*, pages 196–204. IEEE, 2007.

[31] S. Kim, T. Zimmermann, E. J. Whitehead Jr, and A. Zeller. Predicting faults from cached history. In *Proceedings of the 29th international conference on Software Engineering*, pages 489–498. IEEE Computer Society, 2007.

[32] J. Krall, T. Menzies, and M. Davies. Gale: Geometric active learning for search-based software engineering. *Software Engineering, IEEE Transactions on*, 41(10):1001–1018, 2015.

[33] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch. Benchmarking classification models for software defect prediction: A proposed framework and novel findings. *IEEE Transactions on Software Engineering*, 34(4):485–496, 2008.

[34] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch. Benchmarking classification models for software defect prediction: A proposed framework and novel findings. *IEEE Transactions on Software Engineering*, 34(4):485–496, July 2008.

[35] M. Li, H. Zhang, R. Wu, and Z.-H. Zhou. Sample-based software defect prediction with active and semi-supervised learning. *Automated Software Engineering*, 19(2):201–230, 2012.

[36] V. López, A. Fernández, and F. Herrera. On the importance of the validation technique for classification with imbalanced datasets: Addressing covariate shift when data is skewed. *Information Sciences*, 257:1–13, 2014.

[37] V. López, A. Fernández, J. G. Moreno-Torres, and F. Herrera. Analysis of preprocessing vs. cost-sensitive learning for imbalanced classification. open problems on intrinsic data characteristics. *Expert Systems with Applications*, 39(7):6585–6608, 2012.

[38] T. Menzies, O. Elrawas, J. Hihn, M. Feather, R. Madachy, and B. Boehm. The business case for automated software engineering. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 303–312. ACM, 2007.

[39] T. Menzies, J. Greenwald, and A. Frank. Data mining static code attributes to learn defect predictors. *IEEE transactions on software engineering*, 33(1):2–13, 2007.

[40] T. Menzies, Z. Milton, B. Turhan, B. Cukic, Y. Jiang, and A. Bener. Defect prediction from static code features: currentresults, limitations, new approaches. *Automated Software Engineering*, 17(4):375–407, 2010.

[41] T. Menzies, B. Turhan, A. Bener, G. Gay, B. Cukic, and Y. Jiang. Implications of ceiling effects in defect predictors. In *Proceedings of the 4th international workshop on Predictor models in software engineering*, pages 47–54. ACM, 2008.

[42] J. Molina, M. Laguna, R. Martí, and R. Caballero. Sspmo: A scatter tabu search procedure for non-linear multiobjective optimization. *INFORMS Journal on Computing*, 19(1):91–100, 2007.

[43] N. Nagappan, T. Ball, and A. Zeller. Mining metrics to predict component failures. In *Proceedings of the 28th international conference on Software engineering*, pages 452–461. ACM, 2006.

[44] J. Nam, S. J. Pan, and S. Kim. Transfer defect learning. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 382–391. IEEE Press, 2013.

[45] A. J. Nebro, F. Luna, E. Alba, B. Dorronsoro, J. J. Durillo, and A. Beham. Abyss: Adapting scatter search to multiobjective optimization. *Evolutionary Computation, IEEE Transactions on*, 12(4):439–457, 2008.

[46] H. Pan, M. Zheng, and X. Han. Particle swarm-simulated annealing fusion algorithm and its application in function optimization. In *Computer Science and Software Engineering, 2008 International Conference on*, volume 1, pages 78–81. IEEE, 2008.

[47] R. Pears, J. Finlay, and A. M. Connor. Synthetic minority oversampling technique (smote) for predicting software build outcomes. *arXiv preprint arXiv:1407.2330*, 2014.

[48] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, et al. Scikit-learn: Machine learning in python. *Journal of Machine Learning Research*, 12(Oct):2825–2830, 2011.

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

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

[51] F. Peters, T. Menzies, L. Gong, and H. Zhang. Balancing privacy and utility in cross-company defect prediction. *IEEE Transactions on Software Engineering*, 39(8):1054–1068, 2013.

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

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

[54] P. Refaeilzadeh, L. Tang, and H. Liu. Cross-validation. In *Encyclopedia of database systems*, pages 532–538. Springer, 2009.

[55] J. Riquelme, R. Ruiz, D. Rodríguez, and J. Moreno. Finding defective modules from highly unbalanced datasets. *Actas de los Talleres de las Jornadas de Ingeniería del Software y Bases de Datos*, 2(1):67–74, 2008.

[56] A. S. Sayyad, J. Ingram, T. Menzies, and H. Ammar. Scalable product line configuration: A straw to break the camel's back. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 465–474. IEEE, 2013.

[57] M. Shepperd, D. Bowes, and T. Hall. Researcher bias: The use of machine learning in software defect prediction. *IEEE Transactions on Software Engineering*, 40(6):603–616, 2014.

[58] Q. Song, Z. Jia, M. Shepperd, S. Ying, and J. Liu. A general software defect-proneness prediction framework. *IEEE Transactions on Software Engineering*, 37(3):356–370, 2011.

[59] Q. Song, M. Shepperd, M. Cartwright, and C. Mair. Software defect association mining and defect correction effort prediction. *IEEE Transactions on Software Engineering*, 32(2):69–82, 2006.

[60] R. Storn and K. Price. Differential evolution–a simple and efficient heuristic for global optimization over continuous spaces. *Journal of global optimization*, 11(4):341–359, 1997.

[61] J. A. Swets. Measuring the accuracy of diagnostic systems. *Science*, 240(4857):1285, 1988.

[62] M. Tan, L. Tan, S. Dara, and C. Mayeux. Online defect prediction for imbalanced data. In *Proceedings of the 37th International Conference on Software Engineering-Volume 2*, pages 99–108. IEEE Press, 2015.

[63] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto. Automated parameter optimization of classification techniques for defect prediction models. In *Proceedings of the 38th International Conference on Software Engineering*, pages 321–332. ACM, 2016.

[64] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto. Automated Parameter Optimization of Classification techniques for Defect Prediction Models. In *Proc. of the International Conference on Software Engineering (ICSE)*, pages 321–332, 2016.

[65] B. Turhan, A. T. Mısırlı, and A. Bener. Empirical evaluation of the effects of mixed project data on learning defect predictors. *Information and Software Technology*, 55(6):1101–1118, 2013.

[66] J. Van Hulse and T. Khoshgoftaar. Knowledge discovery from imbalanced and noisy data. *Data & Knowledge Engineering*, 68(12):1513–1542, 2009.

[67] J. Van Hulse, T. M. Khoshgoftaar, and A. Napolitano. Experimental perspectives on learning from imbalanced data. In *Proceedings of the 24th international conference on Machine learning*, pages 935–942. ACM, 2007.

[68] J. Vesterstrøm and R. Thomsen. A comparative study of differential evolution, particle swarm optimization, and evolutionary algorithms on numerical benchmark problems. In *Evolutionary Computation, 2004. CEC2004. Congress on*, volume 2, pages 1980–1987. IEEE, 2004.

[69] S. Wang, H. Chen, and X. Yao. Negative correlation learning for classification ensembles. In *The 2010 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 2010.

[70] S. Wang and X. Yao. Using class imbalance learning for software defect prediction. *IEEE Transactions on Reliability*, 62(2):434–443, 2013.

[71] T. Wang, M. Harman, Y. Jia, and J. Krinke. Searching for better configurations: a rigorous approach to clone evaluation. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 455–465. ACM, 2013.

[72] Z. Yan, X. Chen, and P. Guo. Software defect prediction using fuzzy support vector regression. In *International Symposium on Neural Networks*, pages 17–24. Springer, 2010.

[73] Y. Yang, Y. Zhou, J. Liu, Y. Zhao, H. Lu, L. Xu, B. Xu, and H. Leung. Effort-aware just-in-time defect prediction: Simple unsupervised models could be better than supervised models. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 157–168, New York, NY, USA, 2016. ACM.

[74] H. Yin and K. Gai. An empirical study on preprocessing high-dimensional class-imbalanced data for classification. In *High Performance Computing and Communications (HPCC), 2015 IEEE 7th International Symposium on Cyberspace Safety and Security (CSS), 2015 IEEE 12th International Conferen on Embedded Software and Systems (ICESS), 2015 IEEE 17th International Conference on*, pages 1314–1319. IEEE, 2015.

[75] Q. YU, S. JIANG, and Y. ZHANG. The performance stability of defect prediction models with class imbalance: An empirical study.

[76] T. Zimmermann and N. Nagappan. Predicting defects using network analysis on dependency graphs. In *Proceedings of the 30th international conference on Software engineering*, pages 531–540. ACM, 2008.

[77] M. Zuluaga, G. Sergent, A. Krause, and M. Püschel. Active learning for multi-objective optimization. In *Proceedings of the 30th International Conference on Machine Learning*, pages 462–470, 2013.