Noname manuscript No.
(will be inserted by the editor)

Handling Class Imbalance: How Far Have we Come? (Case Studies in Defect Prediction and Vulnerability Detection)

Xiao Ling · Tim Menzies · Christopher Hazard · Jack Shu · Jacob Beel

the date of receipt and acceptance should be inserted later

Abstract Defect prediction and vulnerability detection become more challenging when the target class is rare (since when distributions from the majority class dominate the data sampling, the learner can no longer recognize the minority target). To address this challenge, a recent MSR'22 paper by Shu et al. recommends a deep learning approach called GAN.

When we try to reproduce Shu et al.'s results, we encountered numerous limitations in their analysis, along with a data corruption issue. Hence, this paper aims to reproduce Shu et al.'s study with (a) cleaner data, (b) more case studies, (c) a border set of learners (SVM, RF, LR, LGBM, KNN, GBDT, DT, AdaB), and (d) additional data synthesis methods (GAN, several SMOTE variants, ADASYN, DataSynthesizer, two different Synthetic Data Vault generators, Recursive Random Projections, and a Laplacian method from our industrial partner called Howso AI Engine).

We find that better data synthesis methods take care to adjust what the "distance between examples" means, before performing synthesis. Two such methods explored in this study are the Howso Engine and the SMOTUNED which autonomously fine-tunes its internal parameters. Given that the SMOTUNED runs five times slower than the Howso Engine, we recommend the Howso Engine.

Also, contrary to the conclusion drawn in Shu et al.'s study, we cannot recommend GAN for data synthesis to address the class imbalance problem in defect prediction and vulnerability detection due to its needlessly slowness, and low performance when training the learners on the synthetic data generated by GAN.

To support open science, all materials are available at `https://github.com/Anonymity941212/SyntheticOversampling`.

X. Ling and T. Menzies
North Carolina State University, Raleigh, USA
E-mail: lingxiaohzsz3ban@gmail.com, timm@ieee.org

C. Hazard, J. Shu, and J. Beel
Howso, Raleigh, USA
E-mail: hazard@howso.com, jshu@howso.com, jbeel@howso.com

Keywords Class Imbalance Problem · Machine Learning Classification · Synthetic Data Generation · Security Vulnerability Prediction · Defect Prediction · Generative Adversarial Networks · Random Projections

1 Introduction

Software analytics becomes challenging when the target class is rare. For example, in some of the data explored here, defects or vulnerabilities occur only once in 32 instances, or even more drastically, once in 88 examples. In such imbalanced data, a learner may become insensitive to the minority targets, as distributions from the majority class dominate data sampling. This problem is well-known as *class imbalance problem* (Japkowicz and Stephen, 2002; Longadge and Dongre, 2013).

One of the promising solution is called the *data level solution* which performs action on the data instances. Prior literature mainly explores (a) over-sampling, (b) under-sampling, and (c) hybrid-sampling. Over-sampling and under-sampling aim to balance the class ratio by generating synthetic minority samples and removing majority samples correspondingly, while the hybrid-sampling is the mix of the over-sampling and the under-sampling. Our study, due to the highly imbalanced ratio, explores the over-sampling solution.

A recent state-of-the-art study from Shu et al. recommend the use of hyper-parameter tuning on generative adversarial networks (GAN) (Shu et al, 2022). GAN employs two deep learners to synthesize more examples of the minority class. Within GAN, the *generator* artificially produces fake data, while the *discriminator* attempts to identify which outputs have been artificially synthesized. Feedback from the discriminator is used to guide and enhance the generator.

In our experiment, we significantly extend Shu et al.'s study by (a) exploring much more over-sampling synthetic data generation algorithms and (b) empirically evaluating these algorithms on more software analytic tasks. With more experimental resources, we find different results as Shu et al. presented in their study. More specifically, our contributions can be concluded as

- We carefully inspect the data and find software analytic datasets could contain duplicated data points. In our study, we remove those duplicates to provide a much cleaner training datasets.
- We explore four more synthetic over-sampling algorithms comparing to Shu et al.'s study. DataSynthesizer and Synthetic Data Vault are highly cited synthetic data generation algorithms; Howso engine comes from our industrial partner Howso; and Recursive Random Projection algorithm is proposed by ourselves and have been proved success in the synthetic data generation task (Ling et al, 2023).
- We include both software defect prediction datasets and software vulnerability detection datasets.
- We get different conclusions comparing to Shu et al.'s study. Moreover, through our empirical study, we find that Howso engine from our industrial partner is both effective and efficiency than other state-of-the-art methods.

To structure this analysis, we ask two questions:

RQ1: Which synthetic oversampling technique can generate the most informative minority samples, leading to the highest performance for

learners trained on the original dataset combined with synthetic samples? *Contrary to* the conclusion drawn by Shu et al., we find GAN is needlessly slow since its deep learning based. Also, when combined with a wide-range of learners, some other methods can build better classifiers. We find better data synthesis methods take care to adjust the meaning of "distance between examples" before synthesize the data (where all worse methods do not).

RQ2: When generating the synthetic samples, which approach has the highest efficiency (measured in terms of runtimes)? Two well performed methods from RQ1 are (a) the Howso Engine and (b) the SMOTUNED, which combines SMOTE with hyperparameter optimization to find good local settings for a learner. SMOTUNED is five times slower than Howso engine, so we recommend the latter.

The rest of this paper is constructed as follows. Section 2 offers some background. Section 3 introduces all over-sampling algorithms from literature, as well as our proposed random projection algorithm and the Howso engine from our industrial partner. Section 4 introduces all our experimental setups. Section 5 shows our experimental results and our analysis to the results. We discuss the threat to validity in Section 6 and make the conclusion in Section 7. One of those threats to validity is important enough to state here:

- Studies that fiddle with data need to ensure that they *do not* fiddle with the test data.
- Accordingly, we assert that while we train classifiers from over-sampled data, we evaluate those models on test set that have the *original* (not the oversampled) distributions.

2 Background

2.1 Software Vulnerability Prediction

The early detection of software vulnerability is an essential task. Failure to identify vulnerable files can not only increase the cost on fixing them, but also cause the loss of reputation and damage litigation for a software firm (Hovsepyan et al, 2012).

To mitigate this issue, prior works have proposed different strategies for identifying vulnerable files. Deep learning and machine learning is one of the well-explored strategies (Li et al, 2018c, 2021; Russell et al, 2018; Mazuera-Rozo et al, 2021). In this context, most studies fully utilize the source code and transform it into metrics or tokens (e.g. line of code, number of functions, and total external calls) (Shu et al, 2022). On the other hand, there are some studies to utilize the natural language text mining technologies to identify the keywords that will reveal the security issues (Scandariato et al, 2014; Li et al, 2018a; Jiang et al, 2020; Peters et al, 2017).

One perennial problem in software vulnerability prediction is class imbalance in the available training data. For instance, in the study of critical security vulnerabilities in software products, Peters et al. (Peters et al, 2017) caution that only 0.8% of bug reports are known to be security bug reports in their study.

2.2 Software Defect Prediction

Software defect prediction is the technique used to identify or predict the defective files during the CI/CD development. Previous literature has employed the machine learning classification to achieve this task (Challagulla et al, 2008; Alsaeedi and Khan, 2019; Iqbal et al, 2019; Khan et al, 2021). The training data for defect prediction can be obtained through (a) version control systems and (b) issue tracking systems (Li et al, 2018b). Specifically, version control systems contain metrics related to source code, developer information, and commit information (e.g. line of code add, number of class modified, developer experience, and commit message), while issue tracking systems track defect information (e.g. whether the file is defective or not). In our study, the class imbalance defect prediction datasets is characterized by three metrics: source code metrics (Chidamber and Kemerer, 1994), network metrics (Zimmermann and Nagappan, 2008), and process metrics (Moser et al, 2008).

Similar to vulnerability detection, generating adequate labeled training data for defect prediction can be prohibitively expensive. Hence, we routinely encounter situations where defect prediction is requested despite having very few examples of the defects to be predicted. There are some fundamental reasons for this challenge. Automatic labeling methods can be highly error-prone (Tu et al, 2020), while manual methods are slow and costly. For example, Tu et al. have studied approximately 714 software projects, which totally including 476K commit files (Tu et al, 2020). After an extensive analysis, they proposed a cost model for labeling such data. Assuming two people checking per commit, it would require three years of effort to label the entire dataset.

2.3 Class Imbalance and Data Synthesis

Much prior work has explored class imbalance (Abd Elrahman and Abraham, 2013; Japkowicz, 2000; Japkowicz and Stephen, 2002; Longadge and Dongre, 2013) (notation: the class with overwhelmed instances is called the *majority* , and the other classes is called the *minority*). When training machine learning or data mining models directly with imbalance datasets, standard classification algorithms often produce biased results (Japkowicz et al, 2000; Abd Elrahman and Abraham, 2013). This is because these models tend to learn biased information from rich positive samples, and hence misclassifying the information from the minority samples. Furthermore, Elrahman et al. pointed out that the cost of biased prediction results caused by class imbalance problem is particularly high, especially in some domains such as cancer prediction and fraud prediction (Abd Elrahman and Abraham, 2013).

To mitigate the class imbalance problem, numerous solutions have been proposed over the past decades. These solutions can be categorized into (a) *data level solutions*, (b) *algorithmic solutions*, and (c) *ensemble learning based solutions* (Devi et al, 2020). In this study, we focus on *data level solutions*, with a specific focus on over-sampling methods to mitigate the class imbalance problem in software engineering datasets.

Our analysis of the literature suggests that the most cost-effective approach to addressing class imbalance problem is the synthesis of artificial examples through *data-level approaches*, which utilize various "sampling" techniques such as

- **Over-sampling** adds minority class examples.
- **Under-sampling** prunes majority class examples.
- **Hybrid-sampling** combines both **Over-sampling** and **Under-sampling**.

In the realm of vulnerability and defect prediction, over-sampling is more favored over the other two techniques for several reasons:

- Firstly, in some datasets, the imbalance ratio is very high (e.g. a pos:neg ratio of 1:86 in one of our case studies). Under-sampling the majority class in such situation might remove too many instances which significantly reducing the informativeness of the dataset.
- Secondly, all data points in the software analytics domain are important. Under-sampling or hybrid-sampling may remove informative data points.

Additionally, **Under-sampling** and **hybrid-sampling** are typically designed for large training data, as they can reduce the effort on generating synthetic samples and decrease the runtime of training the machine learning or data mining algorithms. However, in our case studies, we do not have large training data. Thus, our study is more suitable for **over-sampling** technique.

2.4 On the Generality of Data Synthesis

We predict that data synthesis will become a widely used technique in software analytics in the very near future. As mentioned earlier, data synthesis is useful for addressing class imbalance when training classifiers for defect prediction and vulnerability detection. But there are many other applications of this technology (Menzies and Hazard, 2023).

For example, synthetic data techniques can create all the data needed to satisfy the needs of data-hungry machine learning algorithms (Menzies and Hazard, 2023). One approach to this is that synthetic data can exploit current trends in the data, thus supporting forecasting (Menzies et al, 2007b). Data synthesis also serves as a method to obtain the data needed to stress test a system (Gao et al, 2020). In addition, synthetic data can adjust the data (e.g., by removing data discrimination (Chakraborty et al, 2021); or impute missing information into existing data. Further, data synthesis can be used to enable data sharing, without incurring the wrath of legislative bodies. This enables organizations to share insights more safely, thus helping in scientific reasoning (for Adolescent Depression Trials Study Team including: et al, 2013; Menzies and Hazard, 2023). Lastly, increased reliance on synthetic data, as opposed to real data, enhances data security.

For these reasons, we think it is very important to explore data synthesis.

3 Synthetic Oversampling Algorithms

In this section, we will introduce the synthetic over-sampling algorithms we studied in this paper. These algorithms were selected as follows. Firstly, we added all

the methods from Shu et al.'s MSR'22 paper. Secondly, from the cooperation of our industrial partner (i.e. the Howso algorithm in §3.6). Next, we did a Google search on over-sampling in the last ten years, and looked for highly cited papers. The papers were divided into "families" of algorithms (in which a paper shared assumptions with others) and we selected the most cited of each family. This lead us to the SDV method as described in §3.4 and the DataSynthesizer method in §3.5.

For theoretical and systems reasons, we did not explore the R-package SYNTHPOP (Nowok et al, 2016). This package generates values for features conditionally on the values selected previously, which is an approach analogous to the SDV method in §3.4. Also, everything else in this study was Python-based and this would have been the only R-based tool.

As to commercial tools, it turns out that there is a burgeoning international synthetic data market, with many recent commercial offerings. Increasingly, data aggregation is contingent upon maintaining anonymity. Data synthesis methods can be useful here since once we can generate fake representative examples. We tried to include some of those commercial tools in this study, but were foiled by proprietary factors (e.g. we do not know the internal details of the Gretel[1] and MostlyAI[2] systems) or by public offerings of those tools with limits on their usage (e.g. MostlyAI's public offering cannot generate more than 100k instances per day). In the future, we expect those limits to disappear (e.g. Howso Engine was a proprietary tool but now is freely available within the Python package ecosystem).

3.1 RANDOM

Random synthesis just adds copies of minority class instances (selected at random) to the data, until the classes are balanced. Our results will not recommend this method (but we include it here, just as a baseline).

3.2 SMOTE

Synthetic minority over-sampling technique (SMOTE) is proposed by Chawla et al. (Chawla et al, 2002) in 2002. SMOTE firstly finds k nearest neighbors for each minority class data point. After that, for each minority point, SMOTE randomly picks one of the k neighbors, and uses following linear interpolation to generate a new case:

$$p^i_{new} = p^i_{old} + r * (p^i_{nn} - p^i_{old}) \tag{1}$$

where

- p^i_{old} is the i^{th} attribute of the original minority point.
- p^i_{nn} is the i^{th} attribute of one of the nearest neighbor minority points to the original minority point p^i_{old}.
- r is a uniformly generated float number $\in (0, 1)$.

[1] https://gretel.ai
[2] https://mostly.ai

This repeats till the ratio of minority and majority classes equalizes.

At the time of this writing, the original SMOTE paper has over 26,000 citations. Many researchers have proposed variations to this algorithm (see below).

3.2.1 ADASYN

ADASYN (Adaptive Synthetic Sampling) is proposed by He et al. in 2008 (He et al, 2008). Different to SMOTE that each minority point will generate k synthetic samples where k is the number of nearest neighbors, ADASYN controls the number of generated synthetic samples for each minority point by the population density of its nearest neighbors. More specifically, ADASYN calculates a ratio r_j for each minority point by

$$r_j = N_{mjr}/k \tag{2}$$

where

- k = number of nearest neighbors to explore.
- N_{mjr} = number of majority points in k nearest neighbors.

After all r_j are calculated, ADASYN will then normalize them to make all r_j to a density distribution. Such density distribution can control the number of synthetic samples to generate by

$$N_{syn}^{j} = r_j * G \tag{3}$$

where G is the total number of synthetic samples that need to be generated. The linear interpolation in SMOTE is then used to each minority point with N_{syn}^{j} calculated above (He et al, 2008).

3.2.2 Borderline-SMOTE

Borderline-SMOTE is proposed by Han et al. in 2005 (Han et al, 2005), which only explores the minority points that in the "dangerous" region. More specifically, for the k nearest neighbors of each minority point, if there are more majority points than minority points in the k nearest neighbors, that minority point will be marked in the borderline (Han et al, 2005). After that, for each minority point in the borderline, Borderline-SMOTE will find its another k nearest neighbors that only come from minority region. The SMOTE linear interpolation will then be used to generate synthetic samples for those minority points in the borderline.

3.2.3 SVM-SMOTE

SVM-SMOTE is proposed by Nguyen et al. (Nguyen et al, 2011) in 2011. The design idea of SVM-SMOTE is similar to Borderline-SMOTE which explores the minority points in the "dangerous" region. However, different to the Borderline-SMOTE, SVM-SMOTE relies on the support vectors from SVM model that is trained on the original dataset. In the linear interpolation, Nguyen et al. also claimed that using both interpolation and extrapolation is better than only using the interpolation (Nguyen et al, 2011). Thus, for each support vector of minority class, SVM-SMOTE creates synthetic samples by using either linear interpolation or linear extrapolation depending on whether there are more majority points than minority points or more minority points than majority points correspondingly in the nearest neighbors.

3.2.4 SMOTUNED

SMOTUNED is proposed by Agrawal et al. (Agrawal and Menzies, 2018) in 2018. It auto-tunes the parameters that used in the SMOTE by using the differential evolution algorithm. Differential evolution algorithm finds the global best optimal by iteratively improving a candidate solution by using mutation and crossover operations. Agrawal et al. showed that different parameters are found with different datasets, and thus tuning is more important than using the default parameters in SMOTE (Agrawal and Menzies, 2018).

3.3 GAN

GAN (Generative Adversarial Nets) was first proposed by Goodfellow et al. (Goodfellow et al, 2014) in 2014. The basic structure of GAN is two multi-layer perceptron deep neural networks called generator and discriminator. Generator is responsible for generating fake samples, and discriminator distinguishes fake samples and real samples. More specifically, as Goodfellow et al. described, generator and discriminator are playing the *two-player minimax game*, which discriminator is trained to maximize the probability to predict fake samples and real samples correctly, while generator is trained to produce the instances that can lower the performance of discriminator (Goodfellow et al, 2014). If we notate G as generator and D as discriminator, then the loss function of GAN can be described as follow:

$$\min_G \max_D L(D,G) = E_{x_r}[\log D(x_r)] + E_z[\log(1 - D(G(z)))] \tag{4}$$

where E_{x_r} is the expected value cross all real data x_r that comes from the training data set. $G(z)$ is the output of generator G given the randomly generated noise input z, and $D(G(z))$ calculates the probability to predict the fake samples as the real samples. Discriminator tends to minimize the probability from $D(G(z))$, so that generator is trained to minimize $\log(1 - D(G(z)))$. The value of loss function will feedback to both discriminator and generator during backpropagation to adjust the weights in these two models.

3.3.1 WGAN

Normal GAN model has shown to have issues with *diminished gradient* and *mode collapse*, which can cause the convergence problem during the training. *Diminished gradient* means that the gradient becomes vanishingly small during the backpropogation step, which prevents the update of weights. In the GAN model, this will happen when the performance of discriminator is very good. Fedus et al. (Fedus et al, 2017) pointed out that if the discriminator is perfect (i.e. $D(G(z)) = 0$), then the loss of the term $E_z[\log(1 - D(G(z)))]$ is also 0, which causes the vanishing gradient problem. On the other hand, *mode collapse* means that the generator simply memorizes some training examples and only outputs those samples. This happens during the training when the generator finds a set of examples that can fool the discriminator (Saatci and Wilson, 2017).

To handle the above problems, Arjovsky et al. (Arjovsky et al, 2017) proposed Wasserstein GAN (WGAN). WGAN uses the different distance calculation called

Earth-Mover (EM) distance or Wasserstein distance (Arjovsky et al, 2017), which presented as follow

$$\delta_W(\mathcal{P}, \mathcal{Q}) = \inf_{\gamma \in \Pi(\mathcal{P}, \mathcal{Q})} \mathbb{E}_{(p,q)\sim\gamma}[||p - q||] \tag{5}$$

where $\Pi(\mathcal{P}, \mathcal{Q})$ is the set of joint distributions $\gamma(p, q)$ which marginals are respectively $\mathcal{P}$ and $\mathcal{Q}$; $||p - q||$ calculates the cost that transferring p to q so that the distribution $\mathcal{P}$ can be transferred to the distribution $\mathcal{Q}$; and the overall Wasserstein distance is the cost of the most optimal (infimum) transformation.

3.3.2 cWGAN-GP

Conditional Wasserstein GAN with Gradient Penalty (cWGAN-GP) is proposed by Zheng et al. in 2020. Firstly, conditional GAN is used to generate data with specific labels, which both generator and discriminator not only input the sample data, but also input the label. Thus the output of generator is data with specific conditional label (Zheng et al, 2020). Moreover, cWGAN-GP added a term of gradient norm penalty to ensure that the gradient of the discriminator D is limited to 1 to achieve Lipschitz continuity (Zheng et al, 2020). Such design can train a better generator model by providing more gradient information to the generator instead of the discriminator. The loss function of cWGAN-GP can be expressed as follow

$$L_{\text{WGAN-GP}}(D, G) = L_{\text{WGAN}}(D, G) + \lambda \mathbb{E}_{p\sim\mathcal{P}}[(||\nabla_p D(p)||_2 - 1)^2] \tag{6}$$

3.3.3 Dazzle

Dazzle is proposed by Shu et al. in 2022 (Shu et al, 2022). As cWGAN-GP has shown success in solving some class imbalance problems, Shu et al. claimed that cWGAN-GP would benefit from an automated hyper-parameter optimization technique (Shu et al, 2022). In the space of hyper-parameter optimization, the model trained with optimal parameters will have better performance than those models trained with other configurations. However, hyper-parameter optimization for deep neural network is not a trivial work since (a) a single execution of a deep neural network is far more lengthy than basic machine learning algorithms, (b) the parameter space to explore in deep neural network is larger, and (c) different configurations may qualify for different datasets. Thus, running a brute-force hyper-parameter exploration search (e.g. random search or grid search) will be an expensive task. To mitigate this problem, Shu et al. implemented Bayesian optimization (Shu et al, 2022).

Bayesian optimization method consists of two main functions: (a) a Bayesian statistical model for modeling the objective function, and (b) an acquisition function to choose the next sample position (Frazier, 2018). More specifically, with limited resource, the Bayesian optimization builds a surrogate function, which is the probability representation of the objective function, based on few known data points $\{(x_1, y_1), (x_2, y_2), \cdots, (x_n, y_n)\}$. After that, the acquisition function f is involved, and the next sampling position will be the position i such that $f(x_i)$ is maximum. The truth value of (x_i, y_i) will then be obtained by the objective function, and be added into the list of known data points. After that, Bayesian

Algorithm 1 Gaussian Copula: The Gaussian Copula algorithm for analyzing the distribution and covariance of the dataset. Return the covariance matrix Σ.

Require: $\mathcal{D} = \{d_1, d_2, \cdots, d_p\}$, $\mathcal{F} = \{f_1, f_2, \cdots, f_n\}$
1: **for** d_i in $\mathcal{D}$ **do**
2: $\quad Y_i = [\Phi^{-1}(f_0(d_{i0})), \Phi^{-1}(f_1(d_{i1})), \cdots, \Phi^{-1}(f_n(d_{in}))]$
3: **end for**
4: $\Sigma =$ **computeCovariance**$(\{Y_1, Y_2, \cdots, Y_n\})$

optimization will repeat the following step that the new surrogate function will then be generated and the acquisition function will be updated.

The

Dazzle collects all values from the loss function during the iterations of Bayesian optimization, and chooses the set of parameters that has the lowest loss function as the optimal choice. Shu et al. utilized *G-score* as the optimization goal which is calculated as follow

$$\text{G\_score} = 2 * \text{recall} * (1 - \text{fpr})/(\text{recall} + 1 - \text{fpr}) \tag{7}$$

3.4 Generation via Covariance: The Synthetic Data Vault

Synthetic Data Vault (SDV hereafter) is developed by Patki et al. in 2016 (Patki et al, 2016). It is a system that designed to generate synthetic data with generative models. The SDV system can work on both standalone table and relational tables, and in this study, we will focus on the standalone table data generation. SDV analyzes the data with the *distribution* of each feature, and the *covariance* of how the value in a column affects the value in another column given the same row. It uses the multivariate version of the Gaussian Copula, which can remove any bias that the distribution shape may induce (Patki et al, 2016). The detailed algorithm of SDV is presented in Algorithm 1. In this algorithm, $\mathcal{D}$ is the dataset with rows $\{d_1, d_2, \cdots, d_p\}$, and $\mathcal{F}$ is the cumulative distribution function for each column $\{f_1, f_2, \cdots, f_n\}$. The Φ^{-1} is the inverse cumulative distribution function of the Gaussian distribution applied to the cumulative distribution function of the original distribution. The distribution for each feature and the covariance matrix Σ calculated in Algorithm 1 form the generative model for generating the synthetic samples (Patki et al, 2016).

We explore two variants of SDV:

- SDV-GC uses the Gaussian Copula method described above.
- SDV-GAN is an experimental extension that augments SDV-GC with GAN.

All these two variants of SDV is available in their online public API package[3].

3.5 Generation via Causal Models: The Data Synthesizer

DataSynthesizer is proposed by Ping et al. on 2017 (Ping et al, 2017) which contains three modules. "DataDescriber" investigates the data type for each feature, as well as its correlation and distribution. Moreover, "DataDescriber" can add

[3] https://github.com/sdv-dev/SDV/tree/main

Algorithm 2 GreedyBayes: The GreedyBayes algorithm for finding the attributes correlationships (Ping et al, 2017). Return the collection $\mathcal{N}$

Require: $\mathcal{D}$, $\mathcal{A}$, k
$\mathcal{N} = \varnothing$, $\mathcal{S} = \varnothing$ ▷ Initialize
Randomly pick X_1 from $\mathcal{A}$
$\mathcal{N}$**.append(**$(X_1, \varnothing)$**)**, $\mathcal{S}$**.append(**X_1**)**
for $i = 2, 3, \cdots,$ **size(**$\mathcal{A}$**) do**
  $\Omega = \varnothing$
  $p = \min(k,$ **size(**$\mathcal{S}$**))**
  for each $X \in A \setminus S$ and each $\Pi \in \binom{\mathcal{S}}{p}$ **do**
    Ω**.append(**(X, Π)**)**
  end for
  $MI_i =$ **MutualInfo(**Ω_i**)** ▷ Calculate mutual info for each pair in Ω
  $(X_{\max}, \Pi_{\max}) = \max(MI)$ ▷ Greedy select with max mutual info
  $\mathcal{N}$**.append(**$(X_{\mathbf{max}}, \Pi_{\mathbf{max}})$**)**, $\mathcal{S}$**.append(**$X_{\mathbf{max}}$**)**
end for

noise to the data distribution to preserve privacy. "DataGenerator", which samples the synthetic data from the summary computed by DataDescriber. Furthermore, "DataInsepctor" inspects the synthetic data generated by the generator (Ping et al, 2017) (which we will not use in this study).

More specifically, the *DataDescriber* module will firstly detect the data type of each feature automatically. After that, if the *DataDescriber* is set to the independent attribute mode, it will perform the frequency-based estimation of the unconditioned probability distributions of each attribute (Ping et al, 2017). The controlled noise $Lap(\frac{1}{n\epsilon})$ is added to the distributions to preserve the privacy. In the above formula, n is the shape of the input, and $\epsilon = 0.1$ by default. On the other hand, if the *DataDescriber* is set to the correlated attribute mode, then it utilizes GreedyBayes algorithm to construct the Bayesian networks for the correlated attributes. Algorithm 2 shows the psudocode for the GreedyBayes which takes the input of the dataset $\mathcal{D}$, a set of attributes $\mathcal{A}$, and the maximum number of parents k. Similar to the independent attribute mode, the controlled noise $Lap(\frac{4(\lceil\mathcal{A}\rceil-k)}{n\cdot\epsilon})$ is added to the distributions.

After *DataDescriber* generates the probability distribution or the correlation distribution, *DataGenerator* will then generate synthetic samples based on these distributions. More specifically, for the independent attributes mode, DataGenerator will generate synthetic samples from the distribution of each feature, and for the correlated attribute mode, DataGenerator will firstly sample the root attribute, and then sample remaining attributes by using the causal graph generated by the GreedyBayes introduced above.

3.6 Howso Engine

Howso Engine is based on k-NN with several improvements to synthesize data using both global and local distributions. The algorithm is controlled by three parameters:

- A loop control parameter $l = 6$;
- The number of items k in each neighborhood;=
- and the Minkowski coefficient p.

where k and p are selected using a grid search ($5 \leq k \leq 22$ and $0.1 \leq p \leq 2.0$) to maximize accuracy, precision, and recall.

The distance metric d used is Minkowski distance:

$$d(x, y, p, \Delta) = \left(\sum \Delta(x_i, y_i)^p \right)^{1/p} \tag{8}$$

.

The Δ used is the Łukaszyk–Karmowski metric (LK metric) for the Laplace distribution (Hazard et al, 2019):

$$\Delta(x_i, y_i) = |x_j - x_i| + \frac{1}{2} e^{-\frac{|x_j - x_i|}{b}} (3b + |x_j - x_i|) \tag{9}$$

.

The Laplace distribution is chosen because, like the Gaussian distribution, it makes entropy-minimizing assumptions about the underlying data. However, unlike the Gaussian distribution, the LK metric is much more performant. Howso Engine calculates b for each feature via a looping procedure that (a) first builds approximate neighborhoods (using traditional Minkowski distance) then (b) refines those neighborhoods at loop l using the value of b computed in iteration $l - 1$.

In loop $l = 1$, Minkowski distance ($\Delta = |x-y|$)[4] is applied to find the k nearest neighbors of each example x_i and take their mean μ_k. Then b is computed as mean absolute error (MAE):

$$b = \frac{1}{n} \sum_{i}^{n} |x_i - \mu_k| \tag{10}$$

(Discrete attributes use mode and accuracy, not mean and MAE.)

For loops $l > 1$, the LK metric (Equation 9) is used instead of the traditional difference metric with b computed in loop $l - 1$. The iteration continues until b stabilizes. This process is repeated for each feature.

Once b has stabilized for each feature, the following procedure is applied m times to generate m new examples. Howso Engine iteratively selects one random feature to synthesize and generates a synthetic value for that feature based on all of the features previously synthesized until the entire case is synthesized. In the first iteration, values are selected on the basis of the global histogram (for nominal features) or the global Laplace distribution (for continuous features). For all subsequent iterations, values are selected based on the distribution of those features' values in the k nearest neighbors to the partially synthesized case. This process is repeated until every feature of the requested m cases has been synthesized.

3.7 Recursive Random Projections

Recursive random projection (RRP hereafter) is a valuable technique for effectively clustering high dimensional data (Bingham and Mannila, 2001; Fern and Brodley, 2003; Dasgupta and Freund, 2008). It has previously proven beneficial for various software engineering tasks (Ling and Menzies, 2023; Lustosa and Menzies, 2023;

[4] Minkowski is a generalization of many other distance measures; e.g. Euclidean distance is Minkowski with $\Delta(x, y) = |x - y|$ and $p = 2$.

Chen et al, 2018). In the context of data synthesis, RRP first recursively splits data into multiple leaf clusters of size $N = 12$ (a number inspired by the success of the Howso algorithm). Subsequently, data synthesis is applied to each leaf using the mutation and crossover operators from differential evolution algorithm (described below).

RRP is a Nyström algorithm, which involves the approximation of the eigenvectors and eigenvalues of a matrix (Platt, 2005). A full Nyström is like a principle components analysis, while faster versions such as FASTMAP (Faloutsos and Lin, 1995) are more approximate.

RRP recursively bi-clusters the data as follows. Let W be any point chosen at random, RRP identifies A as the furthest point from W, and finds B as the furthest point from A. All other points X have distance a, b to A, B correspondingly. X falls at distance x along the line $\overline{AB}$ at $x = (a^2 + c^2 - b^2)/(2c)$ (by the cosine rule). Items can now be clustered into two bins based on the median x value. RRP will recursively apply this step to the clusters until the size of all clusters less than the certain threshold.

After we get clusters from the above algorithms, we then designed some scenarios to generate synthetic samples for each of the leaf cluster found by RRP. The logic is shown as follow. All the mutation operators are utilized from differential evolution algorithm (Price, 2013; Mallipeddi et al, 2011; Wang et al, 2018).

If a leaf cluster has no minority point, then we ignore it.

If a leaf cluster contains only 1 minority point, then we loop through the majority points and mutate them towards to that minority point. i.e. if we notate the x_i as a randomly picked majority point, and x_b is the minority point in that cluster, then we mutate the data point as follow:

$$p_{\text{new}_k} = \begin{cases} x_{i_k} + F_m * (x_{b_k} - x_{i_k}), & \text{if } r_i < CR \text{ or } k == R \\ x_{i_k}, & \text{O.W.} \end{cases} \tag{11}$$

where F_m is the mutation scalar and we set as 0.9 which mutate close to the minority point; CR is the crossover scalar which we set as 1 since we want to mutate every element in the data point. r_i is a uniformly random number between 0 to 1, and R is a random index which can make sure at least one element is mutated if CR is small.

If a leaf cluster contains more than 1 minority point, but the number of minority point still less than half to the total data points in that cluster, then we add a support term after the major mutation in previous step, which mutate a little bit more towards to any other minority point. The mutation strategy of previous step will then be improved as follow:

$$p_{\text{new}_k} = x_{i_k} + F_m * (x_{b_k} - x_{i_k}) + F_n * (x_{r1_k} - x_{r2_k}) \tag{12}$$

Finally, if a cluster contains half more minority points, then we pick any three minority data points x_{b1}, x_{b2}, and x_{b3} with no replacement and perform the mutation.

$$p_{\text{new}_k} = x_{b1_k} + F_m * (x_{b2_k} - x_{b3_k}) \tag{13}$$

4 Experimental Setup

4.1 Experimental Workflow

In this subsection, we will introduce our experimental workflow. The workflow can be described in the following steps.

Firstly, excluding the "Ambari Vulnerable" case study (where the data has already been split into training and test set), we perform the train-test split which divides the data into 80% training and 20% testing.

Next, we generate synthetic minority data using different algorithms that can only obtain information from the training dataset.

After synthetic samples are generated, we use the standard scalar to scale the new training data. This step does not apply to Shu et al.'s DAZZLE algorithm since DAZZLE has a build-in minmax scalar pipeline, and the generated samples are already scaled.

Subsequently, we train various machine learning models with default parameters on the **over-sampled training data**, and evaluate the models with the **original test data**.

We then collect recall, false alarm, and G-score from 10 repeats and use statistical analysis introduced latter to empirically evaluate the performance of different over-sampling algorithms.

Note the ubiquity of the GAN algorithm: it appears in WGAN, DAZZLE, and SDV-GAN.

4.2 Hyper-parameters Tuning

Two algorithms (i.e. SMOTUNED and DAZZLE) use the hyper-parameter tuning to find the optimal parameters for the generative model. More specifically, SMOTUNED uses differential evolution algorithm (Feoktistov, 2006) and DAZZLE the hyperopt Python package (Bergstra et al, 2013). We illustrate the parameters in Table 1.

Table 1 Parameters in hyper-parameter tuning in SMOTUNED and DAZZLE.

Parameters	data type	Tuning Range	Description
SMOTUNED tuning parameters			
m	int	[50, 500]	Number of synthetic data points to generate.
r	int	[1, 6]	The power used in Minkowski distance.
neighbors	int	[5, 21]	Number of neighbors used in SMOTE.
DAZZLE tuning parameters			
d_updates_per_g	int	[2, 10]	Update rate of generator in each epoch.
gp_weight	int	[5, 20]	Gradient penalty weight in loss function.
epochs	int	[100, 400]	Number of training epochs.
batch_size	int	{16, 32, 164, 128}	Size of batch.
G_activation	str	{relu, leaky_relu, tanh, sigmoid}	Activation function of generator.
D_activation	str	{relu, leaky_relu, tanh, sigmoid}	Activation function of discriminator.
noisy_num_cols	bool	{True, False}	Apply noise to training data.
G_layer_norm	bool	{True, False}	Apply normalization layer to generator.
D_layer_norm	bool	{True, False}	Apply normalization layer to discriminator.

The parameters, and their ranges, were selected via reference to the core algorithms (e.g. SMOTE only has three tuning parameters) or via reference to prior work (e.g. GAN has many parameters but our reading of the literature is that the values shown in Table 1 are the ones most explored (Shu et al, 2022)).

As to our choice of hyper-parameter optimizers Hyperopt was used for consistency with Shu et al. Also, we added differential evolution since DE has the interesting property that mutants are generated via interpolating between known good solutions. This means that, unlike standard genetic algorithms, if there are associations between attributes, DE tends to preserve those associations. Hence we have observed much success with this algorithm in recent papers Agrawal and Menzies (2018); Ling et al (2023); Xia et al (2018).

4.3 Machine Learning Models

The synthesized data is used to train learners. We employ a diverse set of machine learning classifiers, including Support Vector Machine (SVM), Logistic Regression Classifier (LR), Decision Tree (DT), Random Forest (RF), K-nearest Neighbors (KNN), Gradient-boosted Decision Tree (GBDT), LightGBM, and Adaptive Boosting (Adaboost). All these learners have been widely applied in the software engineering domain, such as vulnerability detection (Shu et al, 2022; Bilgin et al, 2020) and defect prediction (Ren et al, 2014; Challagulla et al, 2008; Alsaeedi and Khan, 2019). In this work, we apply these highly used machine learning classification algorithms to empirically evaluate the quality of synthetic samples. All learners are implemented using the Python package scikit-learn with default parameters, as suggested by Shu et al. in their experiments (Shu et al, 2022).

4.4 Data

We explore vulnerability detection and efect prediction datasets:

- **JavaScript Vulnerability** is extracted from Node Security Platform and the Snyk Vulnerability Database (Ferenc et al, 2019).
- **Moodle Vulnerability** dataset comes from the National Vulnerabilities Database, which includes a variety of vulnerabilities (Walden et al, 2014).
- **Ambari Vulnerability** is the Apache open source project to monitor the Apache Hadoop cluster (Peters et al, 2017).

Table 2 Data in this study. The "minority" class is the positive class which represents vulnerable files in vulnerability detection and represents defect files in defect prediction.

Dataset	Minority Class	Majority Class	Imbalance Ratio	Features
JavaScript	904	5367	1:6	35
Moodle	24	2032	1:85	12
Ambari	29	917	1:32	100
JDT_Core	206	791	1:4	81
PDE_UI	376	2318	1:6	81
Mylyn	245	1616	1:7	81

– **Eclipse_JDT_Core**, **Eclipse_PDE_UI**, and **Mylyn** are Eclipse projects that highly used for defect prediction.

All datasets were checked for duplicate rows. As mentioned in the introduction, removing duplicates from the examples used by Shu et al. resulted in the removal of 6271 examples. The statistics of datasets, post-removal, are presented in Table 2.

4.5 Evaluation Metrics

In our study, we utilize the confusion matrix to analyze the prediction, and report recall, false alarm, and G-score. Menzies et al. (Menzies et al, 2007a) reported that when the minority class is less than 10%, the precision will less informative. Thus, in our study, we choose recall, false alarm, and G-score, which is the harmonic mean of recall and the complement of false alarm. More specifically, if we notate *true positive*, *false positive*, *true negative*, *false negative* as TP, FP, TN, and FN correspondingly,

– **recall** = $\frac{TP}{TP+FN}$, which is the ratio of the number of correct prediction of positive classes and the number of actual positive classes.
– **false alarm** = $\frac{FP}{FP+TN}$, which is the ratio of the number of wrong prediction of negative classes and the number of actual negative classes.
– **G-score** is the harmonic mean of recall and the complement of false alarm and is calculated using Equation 7.

In our experiment, we expect the prediction result has **higher** recall, AUC, and G-score, and has **lower** false alarm.

4.6 Statistical Analysis

Since most of the synthetic sample generation algorithms are stochastic, and the different train test split may result different samples generated by those algorithms, we run each synthetic data generation algorithm 10 times, and use statistical analysis to compare results through 10 repeats. Scott-Knott is the statistical analysis we used in our study.

More specifically, Scott-Knott recursively partitions the list of candidates l into two sub-lists l_1 and l_2. The split should maximize the expected mean value before and after the division (Tu et al, 2020; Xia et al, 2018; Tu et al, 2021):

$$E(\Delta) = \frac{|l_1| * |l_1.\mu - l.\mu| + |l_2| * |l_2.\mu - l.\mu|}{|l|} \tag{14}$$

We check if two sub-lists differ significantly by using the Cliff's Delta procedure. The delta value is

$$Delta = (\#(x > y) - \#(x < y))/(|l_1| * |l_2|) \tag{15}$$

for $\forall x \in l_1$, and for $\forall y \in l_2$. Cliff's delta estimates the probability that a value in the sub-list l_1 is greater than a value in the sub-list l_2, minus the reverse probability (Macbeth et al, 2011). Two sub-lists differ significantly if the delta is not a "small" effect ($Delta >= 0.147$) (Hess and Kromrey, 2004).

Table 3 Median performance results on *Recall* from 10 repeats. *Larger* numbers are *better*. "Best" over-sampling technique(s) are marked in gray (where "best" is defined in §4.6). "W" column (wins) counts the number of top performance for each over-sampling technique.

Over-sampling	Moodle Vulnerability Prediction									Ambari Vulnerable Bug Report								
Technique	SVM	RF	LR	LGBM	KNN	GBDT	DT	AdaB	**W**	SVM	RF	LR	LGBM	KNN	GBDT	DT	AdaB	**W**
No	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0	0.00	0.00	0.14	0.57	0.00	0.14	0.14	0.29	1
Random	0.20	0.00	0.60	0.00	0.00	0.00	0.00	0.00	1	0.00	0.00	0.14	0.14	0.00	0.00	0.57	0.00	1
SMOTE	0.20	0.00	0.40	0.00	0.20	0.00	0.00	0.00	2	0.43	0.43	0.57	0.29	0.57	0.29	0.57	0.29	2
SVMSMOTE	0.00	0.00	0.20	0.00	0.00	0.00	0.00	0.00	0	0.71	0.00	0.71	0.14	0.29	0.14	0.00	0.14	1
BdlineSMOTE	0.00	0.00	0.20	0.00	0.00	0.00	0.00	0.00	0	0.57	0.14	0.71	0.43	0.14	0.00	0.00	0.14	1
ADASYN	0.20	0.00	0.40	0.00	0.20	0.00	0.00	0.00	2	0.43	0.43	0.43	0.29	0.57	0.14	0.43	0.14	1
SMOTUNED	0.60	0.40	0.60	0.40	0.40	0.20	0.40	0.20	7	0.43	0.14	0.29	0.57	0.14	0.29	0.57	0.57	3
WGAN	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0	0.00	0.00	1.00	0.14	0.00	0.29	0.00	1.00	3
DAZZLE	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0	0.00	0.00	0.14	0.14	0.00	0.00	0.14	0.29	0
DS	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0	0.00	0.00	0.14	0.00	0.00	0.00	0.14	0.00	0
SDV-GAN	0.20	0.00	0.20	0.00	0.00	0.00	0.00	0.00	0	0.29	0.57	0.43	0.57	0.14	0.43	0.57	0.29	4
SDV-GC	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0	0.00	0.00	0.00	0.14	0.00	0.00	0.14	0.00	0
RRP	0.20	0.00	0.60	0.00	0.00	0.00	0.00	0.00	2	0.00	0.00	0.14	0.14	0.00	0.43	0.57	0.29	2
Howso-Engine	0.40	0.20	0.60	0.20	0.20	0.20	0.20	0.20	6	0.43	0.43	0.43	0.57	0.00	0.29	0.57	0.29	2
Over-sampling	JavaScript Vulnerable Function Code									Eclipse JDT Core Defect Prediction								
Technique	SVM	RF	LR	LGBM	KNN	GBDT	DT	AdaB	**W**	SVM	RF	LR	LGBM	KNN	GBDT	DT	AdaB	**W**
No	0.35	0.47	0.17	0.50	0.47	0.53	0.51	0.49	0	0.32	0.44	0.41	0.46	0.41	0.49	0.49	0.46	0
Random	0.60	0.50	0.59	0.57	0.61	0.6	0.52	0.66	1	0.63	0.54	0.63	0.56	0.61	0.54	0.49	0.58	1
SMOTE	0.62	0.56	0.59	0.59	0.66	0.59	0.57	0.62	2	0.58	0.58	0.61	0.54	0.63	0.54	0.56	0.56	2
SVMSMOTE	0.61	0.56	0.62	0.60	0.63	0.58	0.57	0.61	1	0.58	0.63	0.58	0.58	0.63	0.58	0.54	0.54	1
BdlineSMOTE	0.67	0.57	0.44	0.62	0.66	0.59	0.56	0.65	1	0.58	0.61	0.66	0.56	0.63	0.56	0.56	0.54	2
ADASYN	0.75	0.58	0.59	0.62	0.62	0.59	0.57	0.67	3	0.63	0.61	0.68	0.58	0.61	0.56	0.58	0.58	2
SMOTUNED	0.58	0.68	0.59	0.70	0.66	0.68	0.69	0.67	6	0.63	0.68	0.71	0.71	0.63	0.63	0.61	0.61	2
WGAN	0.00	0.43	0.25	0.14	0.07	0.45	0.47	0.10	0	0.00	1.00	0.95	1.00	1.00	1.00	0.15	1.00	6
DAZZLE	0.38	0.49	0.34	0.50	0.46	0.51	0.51	0.47	0	0.34	0.49	0.46	0.51	0.39	0.51	0.54	0.49	0
DS	0.24	0.48	0.16	0.50	0.47	0.50	0.50	0.48	0	0.24	0.44	0.49	0.49	0.37	0.51	0.54	0.46	0
SDV-GAN	0.40	0.49	0.27	0.51	0.47	0.51	0.51	0.47	0	0.39	0.49	0.41	0.51	0.44	0.51	0.49	0.49	0
SDV-GC	0.38	0.49	0.20	0.50	0.47	0.51	0.51	0.49	0	0.39	0.51	0.41	0.49	0.44	0.51	0.54	0.46	0
RRP	0.46	0.51	0.48	0.51	0.59	0.54	0.54	0.51	0	0.66	0.68	0.66	0.56	0.58	0.56	0.56	0.54	2
Howso-Engine	0.58	0.68	0.58	0.68	0.62	0.64	0.69	0.70	6	0.73	0.76	0.68	0.71	0.71	0.71	0.68	0.73	3
Over-sampling	Eclipse PDE UI Defect Prediction									Mylyn Defect Prediction								
Technique	SVM	RF	LR	LGBM	KNN	GBDT	DT	AdaB	**W**	SVM	RF	LR	LGBM	KNN	GBDT	DT	AdaB	**W**
No	0.02	0.12	0.17	0.21	0.17	0.26	0.31	0.24	0	0.04	0.22	0.12	0.24	0.2	0.33	0.35	0.16	0
Random	0.57	0.31	0.62	0.26	0.48	0.31	0.29	0.52	0	0.51	0.33	0.59	0.33	0.51	0.41	0.31	0.47	0
SMOTE	0.48	0.38	0.59	0.33	0.62	0.38	0.38	0.41	1	0.47	0.43	0.59	0.33	0.57	0.39	0.39	0.39	1
SVMSMOTE	0.29	0.33	0.43	0.26	0.43	0.36	0.36	0.36	0	0.27	0.39	0.41	0.35	0.47	0.39	0.37	0.35	0
BdlineSMOTE	0.52	0.38	0.57	0.26	0.59	0.33	0.41	0.41	2	0.49	0.41	0.53	0.37	0.57	0.41	0.37	0.41	1
ADASYN	0.52	0.36	0.64	0.31	0.57	0.36	0.38	0.43	1	0.49	0.43	0.59	0.37	0.57	0.39	0.33	0.37	1
SMOTUNED	0.59	0.62	0.62	0.57	0.62	0.57	0.48	0.57	3	0.57	0.57	0.57	0.61	0.59	0.61	0.57	0.53	4
WGAN	0.00	1.00	0.95	0.88	0.95	1.00	0.52	0.21	6	0.00	0.96	0.39	0.65	0.61	0.92	0.53	0.22	4
DAZZLE	0.17	0.17	0.31	0.19	0.19	0.31	0.29	0.26	0	0.06	0.27	0.18	0.27	0.18	0.33	0.31	0.18	0
DS	0.02	0.10	0.19	0.21	0.17	0.29	0.29	0.24	0	0.02	0.22	0.14	0.24	0.18	0.31	0.33	0.18	0
SDV-GAN	0.05	0.14	0.21	0.21	0.19	0.26	0.29	0.26	0	0.12	0.24	0.14	0.27	0.18	0.33	0.35	0.20	0
SDV-GC	0.01	0.14	0.19	0.24	0.19	0.26	0.29	0.29	0	0.10	0.22	0.14	0.24	0.20	0.37	0.27	0.18	0
RRP	0.55	0.36	0.64	0.29	0.57	0.41	0.41	0.38	2	0.57	0.45	0.57	0.35	0.55	0.39	0.45	0.37	2
Howso-Engine	0.69	0.67	0.69	0.64	0.62	0.62	0.55	0.67	5	0.63	0.61	0.63	0.61	0.57	0.57	0.65	0.65	6

The reason we choose Scott-Knott because (a) it is fully non-parametric and (b) it reduces the number of potential errors in the statistical analysis since it only requires at most $O(log2(N))$ statistical tests for the $O(N^2)$ analysis. Other researchers also advocate for the use of this test (Gates and Bilbro, 1978) since it overcomes a common limitation of alternative multiple-comparisons statistical tests (e.g., the Friedman test (Friedman, 1937)) where treatments are assigned to multiple groups (making it hard for an experimenter to distinguish the real groups where the means should belong (Carmer and Walker, 1985).

5 Results

In this section, we present our experimental results. We design our experimental procedures based on the research question we raised in Section §1. Specifically, in the first research question, we explore the best over-sampling algorithm(s) in terms of performance, while in the second research question, we consider the runtime and empirically recommend the Howso engine as the best algorithm in this study. We will explain how we make our conclusion in the rest of this section.

Regarding the performance, Table 3 shows the results of recall, Table 4 shows the results of false alarm, and Table 5 shows the results of G-score. In those tables:

– A cell is colored in gray if, across its column, it ranks statistically best (via the methods of §4.6).

Table 4 Median performance results on *False Alarm* from 10 repeats. *Smaller* numbers are *better*. "Best" over-sampling technique(s) is marked in gray (where "best" are defined in §4.6).

Over-sampling	Moodle Vulnerability Prediction									Ambari Vulnerable Bug Report								
Technique	SVM	RF	LR	LGBM	KNN	GBDT	DT	AdaB	**W**	SVM	RF	LR	LGBM	KNN	GBDT	DT	AdaB	**W**
No	0.00	0.00	0.00	0.00	0.00	0.03	0.01	0.01	8	0.00	0.00	0.04	0.01	0.00	0.04	0.02	0.01	8
Random	0.14	0.00	0.20	0.01	0.02	0.00	0.01	0.03	5	0.01	0.00	0.07	0.01	0.02	0.02	0.03	0.02	6
SMOTE	0.16	0.02	0.20	0.01	0.08	0.01	0.04	0.04	4	0.39	0.02	0.35	0.01	0.23	0.01	0.05	0.02	4
SVMSMOTE	0.02	0.01	0.06	0.01	0.02	0.02	0.02	0.01	7	0.41	0.01	0.49	0.00	0.07	0.01	0.02	0.02	5
BdlineSMOTE	0.03	0.01	0.07	0.01	0.02	0.05	0.01	0.01	6	0.42	0.00	0.49	0.02	0.07	0.00	0.01	0.01	5
ADASYN	0.16	0.02	0.20	0.01	0.07	0.01	0.03	0.04	4	0.38	0.01	0.35	0.01	0.23	0.01	0.02	0.01	5
SMOTUNED	0.16	0.18	0.21	0.15	0.18	0.17	0.15	0.16	0	0.38	0.00	0.17	0.02	0.07	0.03	0.10	0.07	3
WGAN	0.00	0.00	0.88	0.01	0.00	0.05	0.15	0.06	4	0.00	0.00	1.00	0.00	0.00	0.03	0.00	0.98	6
DAZZLE	0.00	0.00	0.01	0.00	0.00	0.01	0.01	0.00	8	0.00	0.00	0.01	0.00	0.00	0.02	0.02	0.02	8
DS	0.00	0.00	0.01	0.00	0.00	0.01	0.01	0.01	7	0.00	0.00	0.02	0.00	0.00	0.00	0.02	0.01	8
SDV-GAN	0.02	0.01	0.09	0.01	0.01	0.01	0.03	0.01	7	0.04	0.06	0.09	0.04	0.10	0.03	0.05	0.04	2
SDV-GC	0.02	0.01	0.09	0.01	0.01	0.01	0.03	0.01	7	0.00	0.00	0.03	0.00	0.00	0.01	0.01	0.01	8
RRP	0.15	0.01	0.21	0.01	0.06	0.01	0.03	0.04	4	0.01	0.00	0.14	0.01	0.03	0.03	0.03	0.01	6
Howso-Engine	0.19	0.09	0.21	0.06	0.10	0.06	0.11	0.16	0	0.42	0.24	0.36	0.19	0.12	0.18	0.26	0.18	0
Over-sampling	JavaScript Vulnerable Function Code									Eclipse JDT Core Defect Prediction								
Technique	SVM	RF	LR	LGBM	KNN	GBDT	DT	AdaB	**W**	SVM	RF	LR	LGBM	KNN	GBDT	DT	AdaB	**W**
No	0.00	0.02	0.01	0.02	0.03	0.03	0.08	0.01	8	0.03	0.06	0.05	0.06	0.07	0.07	0.14	0.08	7
Random	0.07	0.03	0.17	0.06	0.13	0.09	0.09	0.13	3	0.17	0.07	0.21	0.07	0.20	0.09	0.13	0.13	5
SMOTE	0.11	0.06	0.19	0.06	0.15	0.08	0.13	0.10	1	0.20	0.10	0.21	0.08	0.22	0.09	0.18	0.12	4
SVMSMOTE	0.09	0.06	0.17	0.06	0.14	0.08	0.12	0.19	1	0.13	0.09	0.15	0.09	0.18	0.10	0.17	0.11	4
BdlineSMOTE	0.26	0.07	0.24	0.08	0.17	0.09	0.12	0.18	0	0.23	0.09	0.23	0.08	0.22	0.11	0.18	0.13	4
ADASYN	0.33	0.08	0.31	0.08	0.18	0.10	0.13	0.18	0	0.25	0.11	0.27	0.08	0.23	0.09	0.18	0.14	3
SMOTUNED	0.08	0.15	0.18	0.15	0.18	0.19	0.23	0.14	1	0.21	0.15	0.21	0.16	0.21	0.16	0.21	0.19	1
WGAN	0.00	0.35	0.04	0.04	0.01	0.33	0.43	0.00	4	0.00	1.00	0.83	0.99	0.97	0.96	0.38	1.00	1
DAZZLE	0.01	0.02	0.09	0.01	0.03	0.04	0.08	0.02	8	0.04	0.06	0.07	0.07	0.07	0.09	0.16	0.09	6
DS	0.00	0.02	0.01	0.02	0.03	0.04	0.09	0.02	8	0.03	0.06	0.06	0.06	0.07	0.07	0.13	0.07	8
SDV-GAN	0.01	0.02	0.04	0.02	0.03	0.04	0.09	0.02	8	0.06	0.07	0.06	0.06	0.07	0.09	0.16	0.08	7
SDV-GC	0.01	0.02	0.02	0.02	0.03	0.05	0.09	0.02	8	0.04	0.05	0.06	0.07	0.08	0.09	0.14	0.09	7
RRP	0.02	0.03	0.07	0.02	0.08	0.05	0.09	0.03	7	0.22	0.11	0.23	0.07	0.19	0.10	0.18	0.11	3
Howso-Engine	0.09	0.12	0.18	0.13	0.15	0.13	0.20	0.19	1	0.26	0.21	0.23	0.20	0.26	0.25	0.30	0.22	1
Over-sampling	Eclipse PDE UI Defect Prediction									Mylyn Defect Prediction								
Technique	SVM	RF	LR	LGBM	KNN	GBDT	DT	AdaB	**W**	SVM	RF	LR	LGBM	KNN	GBDT	DT	AdaB	**W**
No	0.00	0.02	0.03	0.03	0.04	0.06	0.13	0.04	7	0.00	0.01	0.02	0.03	0.05	0.08	0.11	0.04	7
Random	0.20	0.04	0.23	0.05	0.15	0.08	0.10	0.15	3	0.22	0.03	0.24	0.05	0.17	0.08	0.10	0.17	2
SMOTE	0.17	0.06	0.23	0.05	0.21	0.09	0.15	0.11	2	0.17	0.07	0.25	0.05	0.19	0.07	0.14	0.10	1
SVMSMOTE	0.04	0.05	0.11	0.05	0.13	0.09	0.14	0.10	2	0.06	0.05	0.12	0.04	0.14	0.08	0.13	0.09	2
BdlineSMOTE	0.16	0.07	0.21	0.05	0.19	0.08	0.14	0.11	2	0.17	0.06	0.24	0.05	0.18	0.09	0.14	0.11	1
ADASYN	0.19	0.07	0.24	0.05	0.22	0.09	0.17	0.11	2	0.17	0.06	0.27	0.05	0.18	0.08	0.14	0.10	1
SMOTUNED	0.24	0.21	0.25	0.21	0.26	0.24	0.27	0.23	0	0.24	0.19	0.25	0.23	0.21	0.22	0.28	0.23	0
WGAN	0.00	0.98	0.98	0.78	0.87	0.99	0.59	0.31	1	0.02	0.92	0.34	0.41	0.60	0.78	0.54	0.18	1
DAZZLE	0.01	0.02	0.07	0.04	0.04	0.07	0.13	0.05	6	0.01	0.02	0.04	0.03	0.06	0.07	0.10	0.03	7
DS	0.00	0.02	0.02	0.03	0.04	0.07	0.12	0.05	7	0.00	0.01	0.01	0.03	0.05	0.07	0.11	0.04	7
SDV-GAN	0.02	0.02	0.03	0.04	0.04	0.07	0.12	0.05	7	0.02	0.02	0.02	0.03	0.05	0.06	0.12	0.05	8
SDV-GC	0.02	0.02	0.03	0.04	0.04	0.07	0.12	0.06	7	0.01	0.02	0.03	0.03	0.05	0.07	0.11	0.04	7
RRP	0.20	0.07	0.24	0.06	0.20	0.08	0.17	0.10	2	0.23	0.08	0.27	0.04	0.19	0.08	0.13	0.09	1
Howso-Engine	0.31	0.28	0.33	0.27	0.28	0.25	0.35	0.29	0	0.27	0.24	0.32	0.25	0.28	0.27	0.32	0.30	0

– The "**W**" column counts how often a data synthesizer wins when its data is used by eight different learners.

Using those tables, we can answer the RQs raised in Section 1.

RQ1: Which synthetic oversampling technique generates most informative minority samples, leading to the higher performance for learners trained on the original dataset combined with synthetic samples?

In the result tables, false alarm is considered the least informative. It is noteworthy that in Table 4, many cells are colored in gray since many methods demonstrate the best (i.e. statistically least) false alarms. However, these winning false alarms are often associated with low recalls, meaning that achieving high false alarms is contingent on rarely identifying the target.

The G-score in Table 5 offers a more balanced view as it considers both recall and false alarm. From the "W" column, two methods stand out:

– SMOTUNED and Howso Engine often rank in the highest in 6 times (or more);
– While other methods often win 2 times (or less).

Surprisingly, some of the more recent technologies, such as WGAN and Shu et al's tuning variant called DAZZLE, are not showing promising results. For example, WGAN's high recall comes at the cost of very high false alarm. This implies that models trained with datasets synthesized by WGAN predict most of the negative cases as positive. Also, surprisingly, we find that state-of-the-art synthetic data generation algorithms DataSyntheizer and SDV do not perform well in this task.

Overall, we say:

Table 5 Median performance results on *G-score* seen in 10 repeats. *Larger* numbers are *better*. "Best" over-sampling technique(s) are marked in gray (where "best" is defined in §4.6).

Over-sampling	Moodle Vulnerability Prediction									Ambari Vulnerable Bug Report								
Technique	SVM	RF	LR	LGBM	KNN	GBDT	DT	AdaB	**W**	SVM	RF	LR	LGBM	KNN	GBDT	DT	AdaB	**W**
No	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0	0.00	0.00	0.25	0.72	0.00	0.25	0.25	0.44	1
Random	0.32	0.00	0.67	0.00	0.00	0.00	0.00	0.00	1	0.00	0.00	0.25	0.25	0.00	0.00	0.72	0.00	1
SMOTE	0.33	0.00	0.53	0.00	0.33	0.00	0.00	0.00	2	0.50	0.60	0.61	0.44	0.66	0.44	0.71	0.44	4
SVMSMOTE	0.00	0.00	0.33	0.00	0.00	0.00	0.00	0.00	0	0.65	0.00	0.60	0.25	0.44	0.25	0.00	0.25	3
BdlineSMOTE	0.00	0.00	0.33	0.00	0.00	0.00	0.00	0.00	0	0.57	0.25	0.59	0.60	0.25	0.00	0.00	0.25	2
ADASYN	0.33	0.00	0.53	0.00	0.33	0.00	0.00	0.00	2	0.51	0.60	0.52	0.44	0.66	0.25	0.60	0.25	3
SMOTUNED	0.67	0.54	0.65	0.53	0.54	0.32	0.52	0.32	8	0.51	0.25	0.43	0.72	0.25	0.44	0.70	0.71	3
WGAN	0.00	0.00	0.14	0.00	0.00	0.00	0.00	0.00	0	0.00	0.00	0.00	0.25	0.00	0.44	0.00	0.03	1
DAZZLE	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0	0.00	0.00	0.25	0.25	0.00	0.00	0.25	0.44	0
DS	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0	0.00	0.00	0.25	0.00	0.00	0.00	0.25	0.00	0
SDV-GAN	0.33	0.00	0.33	0.00	0.00	0.00	0.00	0.00	0	0.44	0.71	0.45	0.71	0.24	0.59	0.70	0.44	4
SDV-GC	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0	0.00	0.00	0.00	0.25	0.00	0.00	0.25	0.00	0
RRP	0.32	0.00	0.66	0.00	0.00	0.00	0.00	0.00	1	0.00	0.00	0.24	0.25	0.00	0.59	0.72	0.44	2
Howso-Engine	0.53	0.33	0.68	0.33	0.33	0.33	0.33	0.33	6	0.47	0.56	0.50	0.56	0.00	0.56	0.41	0.41	3
Over-sampling	JavaScript Vulnerable Function Code									Eclipse JDT Core Defect Prediction								
Technique	SVM	RF	LR	LGBM	KNN	GBDT	DT	AdaB	**W**	SVM	RF	LR	LGBM	KNN	GBDT	DT	AdaB	**W**
No	0.52	0.64	0.29	0.67	0.64	0.68	0.65	0.65	0	0.47	0.60	0.57	0.62	0.57	0.63	0.62	0.62	0
Random	0.72	0.66	0.68	0.71	0.71	0.72	0.66	0.75	3	0.73	0.69	0.73	0.69	0.70	0.68	0.63	0.70	5
SMOTE	0.73	0.70	0.68	0.72	0.73	0.72	0.69	0.73	2	0.68	0.72	0.71	0.69	0.71	0.67	0.67	0.70	6
SVMSMOTE	0.73	0.70	0.70	0.73	0.72	0.71	0.69	0.73	3	0.71	0.73	0.69	0.71	0.70	0.71	0.64	0.67	5
BdlineSMOTE	0.70	0.71	0.56	0.74	0.73	0.71	0.68	0.73	1	0.66	0.73	0.69	0.68	0.70	0.69	0.67	0.67	5
ADASYN	0.71	0.72	0.63	0.74	0.71	0.71	0.68	0.73	1	0.69	0.72	0.71	0.70	0.68	0.69	0.69	0.69	5
SMOTUNED	0.71	0.75	0.69	0.76	0.72	0.74	0.72	0.74	5	0.71	0.76	0.71	0.75	0.68	0.73	0.69	0.72	8
WGAN	0.00	0.43	0.34	0.24	0.13	0.43	0.43	0.19	0	0.00	0.00	0.07	0.01	0.03	0.03	0.06	0.00	0
DAZZLE	0.54	0.65	0.49	0.66	0.63	0.67	0.66	0.64	0	0.51	0.64	0.62	0.65	0.55	0.66	0.65	0.63	1
DS	0.39	0.64	0.28	0.67	0.63	0.66	0.65	0.65	0	0.39	0.60	0.55	0.64	0.52	0.66	0.65	0.61	1
SDV-GAN	0.56	0.65	0.42	0.67	0.63	0.67	0.65	0.64	0	0.56	0.65	0.57	0.67	0.59	0.66	0.63	0.64	1
SDV-GC	0.55	0.65	0.34	0.66	0.64	0.67	0.65	0.65	0	0.55	0.67	0.57	0.64	0.57	0.65	0.66	0.61	1
RRP	0.63	0.67	0.63	0.67	0.71	0.69	0.68	0.67	1	0.72	0.75	0.72	0.70	0.68	0.69	0.68	0.67	5
Howso-Engine	0.70	0.77	0.67	0.77	0.72	0.74	0.75	0.75	6	0.71	0.76	0.71	0.75	0.73	0.74	0.69	0.75	8
Over-sampling	Eclipse PDE UI Defect Prediction									Mylyn Defect Prediction								
Technique	SVM	RF	LR	LGBM	KNN	GBDT	DT	AdaB	**W**	SVM	RF	LR	LGBM	KNN	GBDT	DT	AdaB	**W**
No	0.05	0.21	0.28	0.35	0.28	0.41	0.46	0.38	0	0.08	0.37	0.22	0.39	0.34	0.48	0.50	0.28	0
Random	0.67	0.47	0.67	0.41	0.61	0.46	0.43	0.63	4	0.62	0.49	0.67	0.49	0.63	0.56	0.46	0.60	1
SMOTE	0.60	0.54	0.67	0.49	0.69	0.54	0.53	0.56	2	0.59	0.59	0.67	0.49	0.66	0.55	0.54	0.55	2
SVMSMOTE	0.44	0.49	0.56	0.41	0.58	0.51	0.51	0.52	0	0.42	0.55	0.56	0.51	0.60	0.54	0.51	0.50	0
BdlineSMOTE	0.65	0.54	0.67	0.41	0.67	0.49	0.55	0.56	4	0.61	0.57	0.63	0.53	0.68	0.56	0.53	0.56	2
ADASYN	0.63	0.52	0.70	0.46	0.66	0.51	0.52	0.58	2	0.61	0.59	0.66	0.53	0.67	0.54	0.47	0.52	2
SMOTUNED	0.66	0.68	0.68	0.66	0.67	0.65	0.56	0.66	8	0.64	0.68	0.66	0.69	0.67	0.67	0.62	0.63	7
WGAN	0.00	0.01	0.02	0.29	0.23	0.01	0.00	0.28	1	0.00	0.11	0.48	0.53	0.45	0.29	0.39	0.33	0
DAZZLE	0.28	0.28	0.46	0.32	0.32	0.46	0.43	0.41	0	0.12	0.42	0.30	0.42	0.31	0.48	0.46	0.31	0
DS	0.05	0.17	0.32	0.35	0.28	0.44	0.43	0.38	0	0.04	0.37	0.25	0.39	0.31	0.46	0.47	0.31	0
SDV-GAN	0.09	0.25	0.35	0.35	0.32	0.41	0.43	0.41	0	0.22	0.39	0.25	0.42	0.31	0.48	0.50	0.34	0
SDV-GC	0.17	0.25	0.32	0.38	0.32	0.41	0.43	0.44	0	0.18	0.36	0.25	0.39	0.34	0.50	0.41	0.31	0
RRP	0.64	0.52	0.68	0.44	0.67	0.56	0.55	0.54	3	0.67	0.60	0.65	0.51	0.66	0.55	0.59	0.52	4
Howso-Engine	0.68	0.69	0.67	0.67	0.67	0.66	0.60	0.68	8	0.64	0.68	0.65	0.67	0.63	0.63	0.65	0.69	8

Answer1: SMOTUNED and Howso Engine are two promising synthetic over-sampling algorithms for vulnerability prediction and defect prediction.

RQ2: When generating the synthetic samples, which approach has the highest efficiency (measured in terms of runtimes)?

To measure runtimes, we conduct all experiments on a machine with 11th Gen Intel Core i7-11800H with 16 cores. All algorithms are executed without GPU acceleration. Table 6 presents the runtime for each algorithm. Among the two best methods identified by RQ1, we see that Howso engine runs around five times faster than SMOTUNED.

The contrast is more vividly illustrated in Figure 1. We select five representative algorithms (either have promising performance or suggested by previous literature) and plot their runtime in the bar chart. The y-axis of the figure presents the actual runtime for a single execution. From the figure, we can clearly see that the DAZZLE from Shu et al. is needlessly slow. Hence we do not recommend DAZZLE since (a) their significantly larger runtime comparing to anything else observed in this study and (b) as demonstrated in RQ1, longer runtime is not associated with better performance.

Therefore, we say

Answer2: Between two "best" algorithms with highest performance in RQ1, we recommend Howso engine than SMOTUNED since it runs much faster.

Table 6 Median runtime report for each synthetic over-sampling algorithm (in seconds, across 10 repeats).

Over-sampling Technique	Moodle Vuln	Ambari Bug	JavaScript Code	Eclipse JDT Core	Eclipse PDE UI	Mylyn
No	0.00	0.00	0.00	0.00	0.00	0.00
Random	0.00	0.01	0.01	0.01	0.01	0.01
SMOTE	0.00	0.01	0.02	0.04	0.01	0.02
SVMSMOTE	0.01	0.01	0.31	0.23	0.07	0.06
BorderlineSMOTE	0.01	0.01	0.04	0.10	0.03	0.03
ADASYN	0.01	0.01	0.06	0.04	0.03	0.02
SMOTUNED	113.53	95.53	287.52	241.26	316.20	261.83
WGAN	6.68	6.21	171.73	45.93	45.78	52.53
DAZZLE	373.32	132.23	1064.78	362.97	479.74	667.50
DataSyntheizer	0.07	0.27	0.23	0.26	0.29	0.32
SDV-GAN	7.06	29.87	19.19	30.89	31.12	28.32
SDV-GC	0.59	4.42	2.29	4.33	4.67	4.44
RandomProjection	0.19	0.23	1.86	0.49	0.84	0.95
Howso-Engine	17.01	56.14	40.56	47.57	54.50	54.06

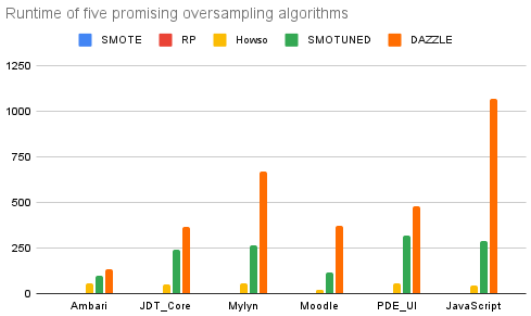

Fig. 1 Runtime comparison of five of our oversampling techniques. Note that SMOTE and Random Projection framework (RP) are so fast, that they are invisible on this chart.

6 Threat to Validity

Sample validity threatens any study that reports conclusions from learners L, synthesis methods S across data sets D for task T. It is quite possible that changing any of L, S, D, T would result in a different conclusion. That said, at the very least, this study can say that the results from Shu et al. are questionable.

Construct validity is an issue with our parameter setting. In this study, there are many parameter choices. In our replication package, we carefully separated these parameter choices to allow other researchers to explore their own design choices.

Conclusion validity concerns the evaluation metrics (i.e. if we check for X, are we missing the impact of Y), which directly indicate the conclusion. In this work, we choose the commonly used metrics (e.g. recall, false alarm, and G-score) to reduce the effect of this validity. It should be noted that the our G-score gives equal weights to recall and false alarm and in certain domains, that weighting would have to be re-examined. Future researchers can choose their own metrics based on their own needs.

Internal validity focuses on the correctness of the treatment caused the outcome. To reduce the effect of this threat, we running all synthetic over-sampling algorithms on same case studies. Moreover, we control the over-sampling rate that for all algorithms, we only generate n minority samples which n is the difference between majority samples and the minority samples in the original datasets.

External validity concerns the application of this study in other problems. To reduce the threat caused by this validity, we run our experiments on both software defect prediction datasets and security vulnerability detection datasets. Future researchers can use our replication package with their own datasets.

7 Conclusion

Class imbalance is a critical problem in data mining. In this study we explore the class imbalance problem in software analytics such as vulnerability detection and defect prediction.

At first glance, deep learning models (e.g. GAN) seem to be a natural choice for solving the problem by generating synthetic minority instances. Hence it may seem hardly surprising that prior studies, such as Shu et al. at MSR'22, endorsed GAN for that task.

That Shu et al. study was somewhat limited in that it only explored an limited number of learners, data sets, and synthesis methods. Also, we found what looked like a pre-processing error in their reproduction package: nearly 40% of rows in their training data had duplicated in some of their case studies.

In this paper, we repeated Shu et al. with cleaner data (all the duplicates removed) and we ran a large number of learners, data sets, and synthesis methods. For example, while Shu et al. only ran datasets and five learners (15 trials), we ran eight learners and six datasets (48 trials). Also, we ran a much large sample of data synthesis methods (e.g Shu et al. did not use the Synthetic Data Vault or the Data Synthesizer).

Our extended results **do not** recommend GAN on the class imbalance problem, as GAN is found to be unnecessarily slow (as seen in Table 6, Shu et al's DAZZLE method is the slowest of everything seen here) and performs worse than alternative approaches. Further, we observed that better data synthesis methods focus on adjusting the meaning of ''distance between examples" before synthesis. Among the two methods identified as the best in our results, we recommend Howso engine since its five times faster than SMOTUNED.

It is important to stress that our results should be tested, on other data. Here, we have only offered evidence for the value of particular data synthesis methods for the tasks of defect prediction and vulnerability detection. Future work should check if our conclusions hold in other domains.

It is also important to say that there are many other applications of data synthesis than just repairing class imbalance (e.g. from §2.4: building larger training sets, forecasting, building stress tests, removing bias, privacy, and data security, just to name a few). Hence, studies like this one will become increasingly important, since data synthesis will become a widely used technique for the MSR community in the very near future.

Acknowledgements In this work, Howso funded NCState to comparatively assess numerous data synthesis methods. All the conclusions here are the product of NCState and were not altered by our Howso collaborators. To enable other researchers to check our conclusions, all the code used in this analysis is available open source at our reproduction repo `https://github.com/Anonymity941212/SyntheticOversampling`.

References

Abd Elrahman SM, Abraham A (2013) A review of class imbalance problem. Journal of Network and Innovative Computing 1(2013):332–340

for Adolescent Depression Trials Study Team including: NCDS, Perrino T, Howe G, Sperling A, Beardslee W, Sandler I, Shern D, Pantin H, Kaupert S, Cano N, et al (2013) Advancing science through collaborative data sharing and synthesis. Perspectives on Psychological Science 8(4):433–444

Agrawal A, Menzies T (2018) Is" better data" better than" better data miners"? on the benefits of tuning smote for defect prediction. In: Proceedings of the 40th International Conference on Software engineering, pp 1050–1061

Alsaeedi A, Khan MZ (2019) Software defect prediction using supervised machine learning and ensemble techniques: a comparative study. Journal of Software Engineering and Applications 12(5):85–100

Arjovsky M, Chintala S, Bottou L (2017) Wasserstein generative adversarial networks. In: International conference on machine learning, PMLR, pp 214–223

Bergstra J, Yamins D, Cox DD, et al (2013) Hyperopt: A python library for optimizing the hyperparameters of machine learning algorithms. In: Proceedings of the 12th Python in science conference, Citeseer, vol 13, p 20

Bilgin Z, Ersoy MA, Soykan EU, Tomur E, Çomak P, Karaçay L (2020) Vulnerability prediction from source code using machine learning. IEEE Access 8:150672–150684

Bingham E, Mannila H (2001) Random projection in dimensionality reduction: applications to image and text data. In: Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining, pp 245–250

Carmer SG, Walker WM (1985) Pairwise multiple comparisons of treatment means in agronomic research. Pairwise multiple comparisons of treatment means in agronomic research (Spring)

Chakraborty J, Majumder S, Menzies T (2021) Bias in machine learning software: Why? how? what to do? In: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp 429–440

Challagulla VUB, Bastani FB, Yen IL, Paul RA (2008) Empirical assessment of machine learning based software defect prediction techniques. International Journal on Artificial Intelligence Tools 17(02):389–400

Chawla NV, Bowyer KW, Hall LO, Kegelmeyer WP (2002) Smote: synthetic minority over-sampling technique. Journal of artificial intelligence research 16:321–357

Chen J, Nair V, Krishna R, Menzies T (2018) "sampling" as a baseline optimizer for search-based software engineering. IEEE Transactions on Software Engineering 45(6):597–614

Chidamber SR, Kemerer CF (1994) A metrics suite for object oriented design. IEEE Transactions on software engineering 20(6):476–493

Dasgupta S, Freund Y (2008) Random projection trees and low dimensional manifolds. In: Proceedings of the fortieth annual ACM symposium on Theory of computing, pp 537–546

Devi D, Biswas SK, Purkayastha B (2020) A review on solution to class imbalance problem: Undersampling approaches. In: 2020 international conference on computational performance evaluation (ComPE), IEEE, pp 626–631

Faloutsos C, Lin KI (1995) Fastmap: A fast algorithm for indexing, data-mining and visualization of traditional and multimedia datasets. In: Proceedings of the 1995 ACM SIGMOD international conference on Management of data, pp 163–174

Fedus W, Rosca M, Lakshminarayanan B, Dai AM, Mohamed S, Goodfellow I (2017) Many paths to equilibrium: Gans do not need to decrease a divergence at every step. arXiv preprint arXiv:171008446

Feoktistov V (2006) Differential evolution. Springer

Ferenc R, Hegedűs P, Gyimesi P, Antal G, Bán D, Gyimóthy T (2019) Challenging machine learning algorithms in predicting vulnerable javascript functions. In: 2019 IEEE/ACM 7th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering (RAISE), IEEE, pp 8–14

Fern XZ, Brodley CE (2003) Random projection for high dimensional data clustering: A cluster ensemble approach. In: Proceedings of the 20th international conference on machine learning (ICML-03), pp 186–193

Frazier PI (2018) A tutorial on bayesian optimization. arXiv preprint arXiv:180702811

Friedman M (1937) The use of ranks to avoid the assumption of normality implicit in the analysis of variance. Journal of the american statistical association 32(200):675–701

Gao X, Saha RK, Prasad MR, Roychoudhury A (2020) Fuzz testing based data augmentation to improve robustness of deep neural networks. In: Proceedings of the acm/ieee 42nd international conference on software engineering, pp 1147–1158

Gates C, Bilbro J (1978) Illustration of a cluster analysis method for mean separation 1. Agronomy Journal 70(3):462–465

Goodfellow I, Pouget-Abadie J, Mirza M, Xu B, Warde-Farley D, Ozair S, Courville A, Bengio Y (2014) Generative adversarial nets. Advances in neural information processing systems 27

Han H, Wang WY, Mao BH (2005) Borderline-smote: a new over-sampling method in imbalanced data sets learning. In: International conference on intelligent computing, Springer, pp 878–887

Hazard CJ, Fusting C, Resnick M, Auerbach M, Meehan M, Korobov V (2019) Natively interpretable machine learning and artificial intelligence: preliminary results and future directions. arXiv preprint arXiv:190100246

He H, Bai Y, Garcia EA, Li S (2008) Adasyn: Adaptive synthetic sampling approach for imbalanced learning. In: 2008 IEEE international joint conference on neural networks (IEEE world congress on computational intelligence), Ieee, pp 1322–1328

Hess MR, Kromrey JD (2004) Robust confidence intervals for effect sizes: A comparative study of cohen'sd and cliff's delta under non-normality and heterogeneous variances. In: annual meeting of the American Educational Research Association, Citeseer, pp 1–30

Hovsepyan A, Scandariato R, Joosen W, Walden J (2012) Software vulnerability prediction using text analysis techniques. In: Proceedings of the 4th international workshop on Security measurements and metrics, pp 7–10

Iqbal A, Aftab S, Ali U, Nawaz Z, Sana L, Ahmad M, Husen A (2019) Performance analysis of machine learning techniques on software defect prediction using nasa datasets. International Journal of Advanced Computer Science and Applications

10(5)
Japkowicz N (2000) The class imbalance problem: Significance and strategies. In: Proc. of the Int'l Conf. on artificial intelligence, vol 56, pp 111–117
Japkowicz N, Stephen S (2002) The class imbalance problem: A systematic study. Intelligent data analysis 6(5):429–449
Japkowicz N, et al (2000) Learning from imbalanced data sets: a comparison of various strategies. In: AAAI workshop on learning from imbalanced data sets, Citeseer, vol 68, pp 10–15
Jiang Y, Lu P, Su X, Wang T (2020) Ltrwes: A new framework for security bug report detection. Information and Software Technology 124:106314
Khan B, Naseem R, Shah MA, Wakil K, Khan A, Uddin MI, Mahmoud M, et al (2021) Software defect prediction for healthcare big data: an empirical evaluation of machine learning techniques. Journal of Healthcare Engineering 2021
Li J, Ji S, Du T, Li B, Wang T (2018a) Textbugger: Generating adversarial text against real-world applications. arXiv preprint arXiv:181205271
Li Z, Jing XY, Zhu X (2018b) Progress on approaches to software defect prediction. Iet Software 12(3):161–175
Li Z, Zou D, Xu S, Ou X, Jin H, Wang S, Deng Z, Zhong Y (2018c) Vuldeepecker: A deep learning-based system for vulnerability detection. arXiv preprint arXiv:180101681
Li Z, Zou D, Xu S, Jin H, Zhu Y, Chen Z (2021) Sysevr: A framework for using deep learning to detect software vulnerabilities. IEEE Transactions on Dependable and Secure Computing 19(4):2244–2258
Ling X, Menzies T (2023) On the benefits of semi-supervised test case generation for cyber-physical systems. arXiv preprint arXiv:230503714
Ling X, Menzies T, Hazard C, Shu J, Beel J (2023) Trading off scalability, privacy, and performance in data synthesis. arXiv preprint arXiv:231205436
Longadge R, Dongre S (2013) Class imbalance problem in data mining review. arXiv preprint arXiv:13051707
Lustosa A, Menzies T (2023) Optimizing predictions for very small data sets: a case study on open-source project health prediction. arXiv preprint arXiv:230106577
Macbeth G, Razumiejczyk E, Ledesma RD (2011) Cliff's delta calculator: A non-parametric effect size program for two groups of observations. Universitas Psychologica 10(2):545–555
Mallipeddi R, Suganthan PN, Pan QK, Tasgetiren MF (2011) Differential evolution algorithm with ensemble of parameters and mutation strategies. Applied soft computing 11(2):1679–1696
Mazuera-Rozo A, Mojica-Hanke A, Linares-Vásquez M, Bavota G (2021) Shallow or deep? an empirical study on detecting vulnerabilities using deep learning. In: 2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC), IEEE, pp 276–287
Menzies T, Hazard C (2023) "the best data are fake data?": An interview with chris hazard. IEEE Software 40(5):121–124
Menzies T, Dekhtyar A, Distefano J, Greenwald J (2007a) Problems with precision: A response to" comments on'data mining static code attributes to learn defect predictors'". IEEE Transactions on Software Engineering 33(9):637–640
Menzies T, Elrawas O, Hihn J, Feather M, Madachy R, Boehm B (2007b) The business case for automated software engineering. In: Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering, pp

303–312
Moser R, Pedrycz W, Succi G (2008) A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In: Proceedings of the 30th international conference on Software engineering, pp 181–190
Nguyen HM, Cooper EW, Kamei K (2011) Borderline over-sampling for imbalanced data classification. International Journal of Knowledge Engineering and Soft Data Paradigms 3(1):4–21
Nowok B, Raab GM, Dibben C (2016) synthpop: Bespoke creation of synthetic data in r. Journal of statistical software 74:1–26
Patki N, Wedge R, Veeramachaneni K (2016) The synthetic data vault. In: 2016 IEEE International Conference on Data Science and Advanced Analytics (DSAA), IEEE, pp 399–410
Peters F, Tun TT, Yu Y, Nuseibeh B (2017) Text filtering and ranking for security bug report prediction. IEEE Transactions on Software Engineering 45(6):615–631
Ping H, Stoyanovich J, Howe B (2017) Datasynthesizer: Privacy-preserving synthetic datasets. In: Proceedings of the 29th International Conference on Scientific and Statistical Database Management, pp 1–5
Platt J (2005) Fastmap, metricmap, and landmark mds are all nyström algorithms. In: International Workshop on Artificial Intelligence and Statistics, PMLR, pp 261–268
Price KV (2013) Differential evolution. In: Handbook of optimization: From classical to modern approach, Springer, pp 187–214
Ren J, Qin K, Ma Y, Luo G, et al (2014) On software defect prediction using machine learning. Journal of Applied Mathematics 2014
Russell R, Kim L, Hamilton L, Lazovich T, Harer J, Ozdemir O, Ellingwood P, McConley M (2018) Automated vulnerability detection in source code using deep representation learning. In: 2018 17th IEEE international conference on machine learning and applications (ICMLA), IEEE, pp 757–762
Saatci Y, Wilson AG (2017) Bayesian gan. Advances in neural information processing systems 30
Scandariato R, Walden J, Hovsepyan A, Joosen W (2014) Predicting vulnerable software components via text mining. IEEE Transactions on Software Engineering 40(10):993–1006
Shu R, Xia T, Williams L, Menzies T (2022) Dazzle: using optimized generative adversarial networks to address security data class imbalance issue. In: Proceedings of the 19th International Conference on Mining Software Repositories, pp 144–155
Tu H, Yu Z, Menzies T (2020) Better data labelling with emblem (and how that impacts defect prediction). IEEE Transactions on Software Engineering 48(1):278–294
Tu H, Papadimitriou G, Kiran M, Wang C, Mandal A, Deelman E, Menzies T (2021) Mining workflows for anomalous data transfers. In: 2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR), IEEE, pp 1–12
Walden J, Stuckman J, Scandariato R (2014) Predicting vulnerable components: Software metrics vs text mining. In: 2014 IEEE 25th international symposium on software reliability engineering, IEEE, pp 23–33

Wang S, Li Y, Yang H, Liu H (2018) Self-adaptive differential evolution algorithm with improved mutation strategy. Soft Computing 22:3433–3447

Xia T, Krishna R, Chen J, Mathew G, Shen X, Menzies T (2018) Hyperparameter optimization for effort estimation. arXiv preprint arXiv:180500336

Zheng M, Li T, Zhu R, Tang Y, Tang M, Lin L, Ma Z (2020) Conditional wasserstein generative adversarial network-gradient penalty-based approach to alleviating imbalanced data classification. Information Sciences 512:1009–1023

Zimmermann T, Nagappan N (2008) Predicting defects using network analysis on dependency graphs. In: Proceedings of the 30th international conference on Software engineering, pp 531–540