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

Improved Recognition of Security Bugs via Dual Hyperparameter Optimization

Rui Shu · Tianpei Xia · Jianfeng Chen · Laurie Williams · Tim Menzies

Received: date / Accepted: date

Abstract **Background:** Security bugs need to be handled by small groups of engineers before being widely discussed (otherwise the general public becomes vulnerable to hackers that exploit those bugs). But learning how to separate the security bugs from other bugs is challenging since they may occur very rarely. Data mining that can find such scarce targets required extensive tuning effort.

Goal: The goal of this research is to aid practitioners as they struggle to tune methods that try to distinguish security-related bug reports in a product's bug database, through the use of a dual hyperparameter optimizer that learns good settings for both learners and for data pre-processing methods.

Method: The proposed method, named SWIFT, combines learner hyperparameter optimization and pre-processor hyperparameter optimization. SWIFT uses a technique called *ϵ-dominance*, the main idea of which is to ignore operations that do not significantly improve the performance. As a result, the optimization effort can be efficiently reduced.

Result: When compared to recent state-of-the-art results (from FARSEC which is published in TSE'18), we find that SWIFT's dual optimization of both pre-processor and learner is more useful than optimizing each of them individually. For example, in a 10-way cross-validation study looking for security bugs from the Chromium web-browser, the FARSEC and SWIFT recalls were 20.4% and 77.1%, respectively, with false alarm rates under 20%. For another example, in experiments with data from the Ambari software project, recalls improved from 30.4 to 83.9% while false alarms remained under 22%.

Conclusion: Overall, our approach shows advantages in achieving better performance in a fast way than existing stat-of-the-art method. Therefore, this encourages us in solving similar problems with dual optimization in the future work.

Keywords Hyperparameter Optimization · Data Imbalance · Data Pre-processing · Security Bug Report Classification

Rui Shu, Tianpei Xia, Jianfeng Chen, Laurie Williams, Tim Menzies
Department of Computer Science, North Carolina State University, Raleigh, NC, USA
Email: rshu@ncsu.edu, txia4@ncsu.edu, jchen37@ncsu.edu, lawilli3@ncsu.edu, timm@ieee.org

1 Introduction

Security bug detection is a pressing current concern. Daily, news reports reveal increasingly sophisticated security breaches. As seen in those reports, a single vulnerability can have devastating effects. For example, a data breach of Equifax caused the personal information of as many as 143 million Americans – or nearly half the country – to be compromised [2]. The WannaCry ransomware attack [1] crippled British medical emergency rooms, delaying medical procedures for many patients.

Developers capture and document software bugs and issues in bug reports which are submitted to bug tracking systems. For example, the Mozilla bug database maintains more than 670,000 bug reports with 135 new bug reports added each day [13]. Submitted bug reports are explicitly labeled as a security bug report (SBR) or non-security bug report (NSBR). Peters et al. [53] warn that it is important to correctly identify security bug reports and distinguish them from other non-security bugs reports. They note that software vendors ask that bug reporters should be reported directly and privately to their own engineers. These engineers then assess the bug reports and, when necessary, offer a security patch. The security bug, and its associated patch, can then be documented and disclosed via public bug tracking systems. This approach maximizes the probability that a patch is widely available before hackers exploit a vulnerability. Sometimes, bug reporters lack the security knowledge [20] to know when their bug is a normal bug (which can be safely disclosed) or when that bug is a security matter (that needs to be handled more discretely). Hence, lamentably, security bug are often publicly disclosed before they can be patched [66].

To address this problem of proper identification of security bugs, researchers have adopted various techniques [20, 22, 67, 68]. These approaches mainly identified relevant keywords in bug reports as well as features such as word frequency, which were then used in learning classification models. But learning such models is a challenging problem. Such bugs may be a very rare occurrence in the data. For example, data sets from [53] show among the 45,940 bug reports, only 0.8% are security bug reports. Various methods exist for mining such rarefied data– but those methods require extensive tuning efforts before they work well on a particular data set. Peters et al. proposed FARSEC [53], a text mining method that used irrelevancy pruning (i.e., filtering). In their approach, developers first identified security related words. Next, they pruned away the irrelevant bug reports (where "irrelevant" meant "does not have these security-related keywords"). FARSEC was evaluated using bug reports from Chromium and four Apaches projects.

The conjecture of this paper is that text mining-based bug report classification approach (e.g., FARSEC) can be further enhanced. For example, FARSEC applied its data miners using their default "off-the-shelf" configurations. Software engineering research results show that *hyperparameter optimization* (to automatically learn the "magic" control parameters of an algorithm) can outperform "off-the-shelf" configurations [3, 4, 19, 28, 44, 58, 62]. To the best of our knowledge, this paper is the first to attempt to apply hyperparameter optimization to the problem of distinguishing security bug reports. To that end, we distinguish and apply three kinds of hyperparameter optimization:

1. *Learner* hyperparameter optimization to adjust the parameters of the data miner; e.g. how many trees to use in random forest, or what values to use in the kernel of a Support Vector Machine (SVM).
2. *Pre-processor* hyperparameter optimization to adjust any adjustments to the training data, prior to learning; e.g. to learn how to control outlier removal or, how to handle the class imbalance problem.
3. *Dual* hyperparameter optimization that combines 1 and 2.

Standard practice in the search-based SE literature is just to explore just learner or pre-processor options [3, 4, 7, 19, 59], but not both. This paper shows that for distinguishing security bug reports, dual optimization performs much better than just optimizing learner and pre-processor options in isolation.

To make that demonstration, we apply dual hyperparameter optimization to the options of Table 1. We make no claim that this is the entire set of possible optimizations. Rather, we just say that (a) any reader of the recent SE data mining literature might have seen many of these; (b) that reader might be tempted to try tuning the Table 1 options; (c) when the the methods of this method automatically tuned these options, we found that our models were dramatically better than the prior state of the art [53].

This study is structured around the following research questions:

RQ1. Can hyperparameter optimization technique distinguish which bug are actually security problems?

We find that, hyperparameter optimization technique help better distinguish security bug reports from non-security bugs reports, i.e., improving bug reports classification performance. Further, the best results for recall/pd come at the cost of a relatively modest increase in the false alarm rate.

RQ2. Is it better to dual optimize the learners or the data pre-processors in security bug report classification?

The results show that dual optimizing both data mining learners and data pre-processors work better than each individual treatment with regards to recall, as well as as with a high rank in the IFA (Initial False Alarms) metric. In addition, the SWIFT algorithm used here is much faster (and scales better to more complex problems) that other methods.

RQ3. What are the relative merits of FARSEC's irrelevancy pruning (e.g., filtering) vs SWIFT's hyperparameter optimization in security bug report classification?

Without hyperparameter optimization, FARSEC's irrelevancy pruning does indeed improve the performance of vulnerability classification. However, with SWIFT's hyperparameter optimization, there is no added benefits to FARSEC's irrelevancy filtering technique.

Table 1 List of pre-processors and learners explored in this study. Standard practice in the literature is to optimize none or just one of these two groups [3, 4, 7, 19, 59]. Note that a dual optimizer simultaneously explores both pre-processing and learner options.

Type	Name	Description
Pre-processor	Normalizer	Normalize samples individually to unit norm.
	StandardScalar	Standardize features by removing the mean and scaling to unit variance.
	MinMaxScaler	Transforms features by scaling each feature to a given range.
	MaxAbsScaler	Scale each feature by its maximum absolute value.
	RobustScalar	Scale features using statistics that are robust to outliers.
	KernelCenterer	Center a kernel matrix.
	QuantileTransformer	Transform features using quantiles information.
	PowerTransformer	Apply a power transform featurewise to make data more Gaussian-like.
	Binarizer	Binarize data (set feature values to 0 or 1) according to a threshold.
	PolynominalFeatures	Generate polynomial and interaction features.
	SMOTE	Synthetic Minority Over-sampling Technique.
Learner	Random Forest (RF)	Generate conclusions using multiple entropy-based decision trees.
	K Nearest Neighbors (KNN)	Classify a new instance by finding "K" examples of similar instances.
	Naive Bayes (NB)	Classify a new instance by (a) collecting mean and standard deviations of attributes in old instances of different classes; (b) returning the class whose attributes are statistically most similar to the new instance.
	Logistic Regression (LR)	Map the output of a regression into $0 \leq n \leq 1$; thus enabling using regression for classification.
	Multilayer Perceptron (MLP)	A deep artificial neural network which is composed of more than one perceptron.

Note: The listed pre-processors and learners are based on scikit-learn 0.21.2.

In summary, the contributions of this paper are:

- *A new high-water mark result.* Specifically, for the purpose of distinguishing security bugs from non-security bugs, our methods are better than those reported in the FARSEC paper from TSE'18.
- *A comment on the value of tuning (a) data pre-processor or (b) data mining algorithms.* Specifically, for the purposes of identifying rare events, we show that dual tuning of (a) and (b) does much better than tuning either, separately.
- *A demonstration of the practicality of dual tuning.* As shown below, the overall runtimes for SWIFT are not impractically slow. This is an important result since our pre-experimental concern was that the cross-product of the option space between the (a) data pre-processors and (b) data mining algorithms would be so large as to preclude dual optimization.
- *A reproduction package that other researchers can use to reproduce/improve our results.* While this is more a system contribution than a research result, such reproduction packages are a significant contribution since they enable the faster replication and improvement of prior results.

The remainder of this paper is organized as follows. We introduce research background in Section 2, then present related works in Section 3. We describe the details of our approach in Section 4. In Section 5, we present our experiment details, including hyperparameter tuning ranges, data sets, evaluation metrics, etc. We answer proposed research questions in section 6. We discuss the threats to validity in Section 7 and conclude in Section 8.

2 Background

2.1 On the Need for More Secure Software

As noted by the US National Institute of Standards and Technology (NIST), "Current systems perform increasingly vital tasks and are widely known to possess vulnerabilities" [11]. Here, by a "vulnerability", they mean a weakness in the computational logic (e.g., code) found in software and some hardware components (e.g., firmware) that, when exploited, results in a negative impact on confidentiality, integrity, or availability [47].

In their report to the White House Office of Science and Technology Policy, "Dramatically Reducing Software Vulnerabilities" [11], NIST officials urge the research community to explore more technical approaches to reducing security vulnerabilities. The need to reduce vulnerabilities is also emphasized in the 2016 US Federal Cybersecurity Research and Development Strategic Plan [54].

Experience shows that predicting for security vulnerabilities is difficult (see §3.1). Accordingly, this paper explores text-mining related techniques on bug reports, augmented by hyperparameter optimization and oversampling methods.

2.2 Hyperparameter Optimization

In machine learning, model parameters are the properties of training data that will learn on its own during training by the classifiers, e.g., split point in CART (Classification And Regression Trees). Model *hyperparameters* are values in machine learning models that can require different constraints, weights or learning rates to generate different data patterns, e.g., the number of neighbors in K Nearest Neighbors (KNN) [35].

Hyperparameter are important because they directly control the behaviors of the training algorithms and impact the performance of the models being trained. Choosing appropriate hyperparameters plays a critical role in the performance of machine learning models. *Hyperparameter optimization* is the process of searching the most optimal hyperparameters in machine learning learners [10]. There are several common types of hyperparameter optimization algorithms, including grid search, random search, and Bayesian optimization.

Grid search [9] [58] is a "brute force" hyperparameter optimizer that wraps a learner in a for-loops that walk through a wide range of all a learner's control parameters. Simple to implement, it suffers from the "curse of dimensionality". That is, after just a handful of options, grid search can miss important optimizations. Worse still, much CPU can be wasted during grid search since experience has shown that only a few ranges within a few tuning parameters really matter [8].

Random search [8] stochastically samples the search space and evaluates sets from a specified probability distribution. Evolutionary algorithms are a variant of random search that runs in "generations" where each new generation is seeded from the best examples selected from the last generation [21]. Simulated annealing is a special form of evolutionary algorithms where the population size is one [37] [46]. The drawback of using random search algorithm is that it does not use information from prior experiment to select the next set and also it is very difficult to predict the next set of experiments.

Bayesian optimization [52] works by assuming the unknown function was sampled from a Gaussian Process and maintains a posterior distribution for this function as observation is made. Bayesian optimizers reflect on their distributions to propose the most informative guess about where to sample next. Such optimizers might be best-suited for optimization over continuous domains with a small number of dimensions.

All the above algorithms have proven useful in their home domains. But for software engineering, certain algorithms such as differential evolution (DE) [55] are known to run very quickly and deliver useful results [3, 4]. Also, The differential evolution has been proven useful in prior software engineering optimization studies [19]. Further, other evolutionary algorithms (e.g., genetic algorithms [21], simulated annealing [37]) mutate each attribute in isolation. When two attributes are correlated, those algorithms can mutate variables inappropriately in different directions. The differential evolution algorithm, on the other hand, mutates attributes in tandem along with known data trends. Hence, the differential evolution algorithm's tandem search can outperform other optimizers such as (a) particle swarm optimization [63]; (b) the grid search used by Tantithamthavorn et al. [58] to tune their defect predictors; or (c) the genetic algorithm used by Panichella et al. [49] to optimize a text miner.

The premise of the *differential evolution algorithm* is that the best way to mutate the existing tunnings is to extrapolate between current solutions. Three solutions a, b, c are selected at random. For each tuning parameter k, at some probability cr, we replace the old tuning x_k with y_k. For booleans $y_k = \neg x_k$ and for numerics, $y_k = a_k + f \times (b_k - c_k)$ where f is a parameter controlling differential weight. The differential evolution algorithm loops g times over the population of size np, replacing old items with new candidates (if new candidate is better). This means that, as the loop progresses, the population is full of increasingly more valuable solutions (which, in turn, helps extrapolation). As to the control parameters of the differential evolution algorithm, using advice from a differential evolution algorithm user group (see `http://bit.ly/2MdDba6`), we set $\{np, f, cr\} = \{10n, 0.8, 0.9\}$, where n are the number of parameters to optimize. Note that we set the number of iteration $\{g\}$ to 3, 10, which are denoted as DE3 and DE10 respectively. A small number (i.e., 3) is used to test the effects of a very CPU-light effort estimator. A larger number (i.e., 10) is selected to check if anything is lost by restricting the inference to small iterations.

2.3 Data Balancing

Many data sets exhibit highly imbalanced class frequencies (e.g., fake reviews in Amazon, fraudulent credit card charges, security bug reports). In these cases, only a small fraction of observations are actually positive because of the scarce occurrence of those events. In the field of data mining, such a phenomenon makes classification models difficult to detect rare events [57]. Both misclassification and failing to identify rare events may result in poor performance by predictors.

One technique to tackle the imbalanced class issue is based on the sampling approach [26]. There are three common ways to resample imbalanced data sets [12, 43, 59, 64, 65]:

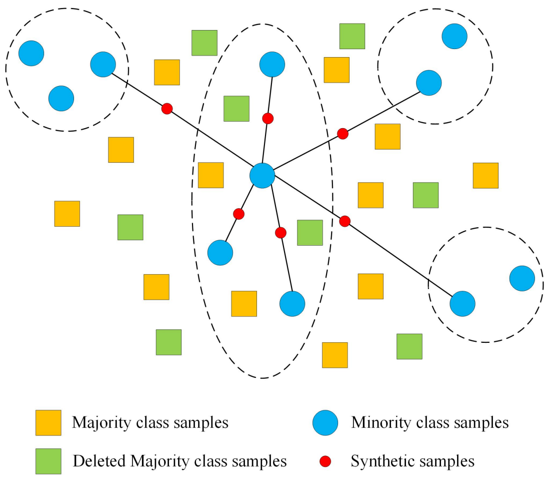

Fig. 1 An illustration on how the SMOTE technique works.

- Oversampling to make more of the minority class;
- Undersampling to remove majority class items;
- Some hybrid of the first two.

Machine learning researchers [26] advise that undersampling can work better than oversampling if there are hundreds of minority observations in the datasets. When there are only a few dozen minority instances, oversampling approaches are superior to undersampling. In the case of large size of training samples, the hybrid methods would be a better choice.

The Synthetic Minority Oversampling TEchnique, also known as SMOTE [12], is a hybrid algorithm that performs both over- and under-sampling. In oversampling, SMOTE calculates the k nearest neighbors for each minority class samples. Depending on the amount of oversampling required, one or more of the k-nearest neighbors are picked to create the synthetic samples. This amount is usually denoted by oversampling percentage (e.g., 50% by default). The next step is to randomly create a

Algorithm 1 Pseudocode of SMOTE. From [3].

```
def SMOTE(k=2, m=50%, r=2): # default configurations
    while Majority > m:
        delete any majority sample
    while Minority < m:
        add synthetic samples

def synthetic_samples(X0):
    relevant = []
    k1 = 0
    while(k1++ < 20) and size(relevant) < k:
        some = k1 nearest neighbors of X0 #. 'near' calculated via minkowski_distance
        relevant += item in some of X0 class
    Z = any of relevant
    Y = interpolate(X0, Z)
    return Y

def minkowski_distance(a, b, r):
    return $(\Sigma_i abs(a_i - b_i)^r)^{1/r}$
```

synthetic sample along the line connecting two minority samples. This can be illustrated in Figure 1 where the blue dots represent the minority class sample, and the red dots represent the synthetic samples. In undersampling, SMOTE just removes majority samples randomly.

Algorithm 1 shows the SMOTE algorithm and Table 2 shows the "magic" parameters that control SMOTE. The best settings for these magic parameters are often domain-specific. SMOTUNED [3] is a tuned version of SMOTE that uses the differential evolution algorithm to learn good settings for the Table 2 parameters. SMOTUNED will serve as our optimizer for data pre-processing to handle data imbalance for security bug classification.

Note that data mining algorithms should be assessed on the kinds of data they might see in the future. That is, data miners should be tested on data that has *naturally occurring class distributions*. Hence, while applying SMOTE (or SMOTUNED) to the *training data* is good practice, it is a bad practice to apply SMOTE to the *testing data*.

2.4 Other pre-processing techniques

Pre-processing techniques are often involved in practical machine learning pipelines. Beside oversampling technique like SMOTE as we mentioned before, there are many other available pre-processors. Table 1 lists the pre-processors that we explore in this study and Table 2 shows the exact ranges which we explore. They are chosen based on the characteristics of the datasets, for example, we do not select those pre-processing techniques that are not applicable to our study, e.g., handle missing values, converting categorical features to a numerical representation or dividing a continuous feature into a pre-specified number of categories.

Usually, different algorithms make different assumptions about the data, and they may require different transformation. Machine learning algorithm sometimes can deliver better results with appropriate pre-processing techniques. For example, for the

Table 2 List of hyperparameters optimized in different learners and pre-processors.

Type	Name	Parameters	Default	Tuning Range
Learner	Random Forest	n_estimators	10	[10, 150]
		min_samples_leaf	1	[1, 20]
		min_samples_split	2	[2, 20]
		max_leaf_nodes	None	[2, 50]
		max_features	auto	[0.01, 1]
		max_depth	None	[1, 10]
	Logistic Regression	C	1.0	[1.0, 10.0]
		max_iter	100	[50, 200]
		verbose	0	[0, 10]
	Multilayer Perceptron	alpha	0.0001	[0.0001, 0.001]
		learning_rate_init	0.001	[0.001, 0.01]
		power_t	0.5	[0.1, 1]
		max_iter	200	[50, 300]
		momentum	0.9	[0.1, 1]
		n_iter_no_change	10	[1, 100]
	K Nearest Neighbor	leaf_size	30	[10, 100]
		n_neighbors	5	[1, 10]
	Naive Bayes	var_smoothing	1e-9	[0.0, 1.0]
Pre-processor	SMOTE	k	5	[1, 20]
		m	50%	[50, 400]
		r	2	[1, 6]
	Normalizer	norm	l2	[l1, l2, max]
		copy	True	[True, False]
	StandardScaler	copy	True	[True, False]
		with_mean	True	[True, False]
		with_std	True	[True, False]
	MinMaxScaler	copy	True	[True, False]
		min	0	[-5, 0]
		max	1	[1, 5]
	MaxAbsScaler	copy	True	[True, False]
	RobustScaler	with_centering	True	[True, False]
		with_scaling	True	[True, False]
		q_min	25.0	[10, 40]
		q_max	75.0	[60, 90]
		copy	True	[True, False]
	QuantileTransformer	n_quantiles	1000	[10, 2000]
		output_distribution	uniform	[uniform, normal]
		ignore_implicit_zeros	False	[True, False]
		subsample	1e5	[100, 150000]
		copy	True	[True, False]
	PowerTransformer	method	yeo-johnson	[yeo-johnson, box-cox]
		standardize	True	[True, False]
		copy	True	[True, False]
	Binarization	threshold	0.0	[0, 10]
		copy	True	[True, False]
	PolynomialFeatures	degree	2	[2, 4]
		interaction_only	False	[True, False]
		include_bias	True	[True, False]
		order	C	[C, F]

data that is comprised of attributes with varying scales, many machine learning algorithms can benefit from rescaling the attributes so that all attributes have the same scale. Often this is referred to as normalization and attributes are often rescaled into the range between 0 and 1. This is useful for optimization algorithms in used in the core of machine learning algorithms like gradient descent, and is also useful for algorithms that weight inputs like regression and neural networks and algorithms that use distance measures such as K Nearest Neighbors.

2.5 Machine Learning Algorithms

Once pre-processed, the resulting data must be analyzed by a data miner. In this work, we use the same five machine learning learners as seen in the FARSEC study, i.e., Random Forest (RF), Naive Bayes (NB), Logistic Regression (LR), Multilayer Perceptron (MLP) and K Nearest Neighbor (KNN). They are widely used for software engineering classification problems [42]. Table 1 also gives a brief description of each learning algorithm and Table 2 shows the exact ranges which we explore.

3 Related Work

3.1 Mining Bug Reports

Text mining (also known as Text Analytics) is the process of exploring and analyzing massive sets of unstructured (e.g., word documents) or semi-structured (e.g., XML and JSON) text data, in order to identify concepts, patterns, topics and other attributes in the data. To pre-process the textual content, several steps such as tokenization, filtering, lemmatization, stop-words removal and stemming are usually leveraged [30].

Text mining has recently been widely applied in bug report analysis, such as identification of duplicated bug reports [17, 29, 41, 56], prediction of the severity or impact of a reported bug [40, 60, 69, 70, 72], extraction of execution commands and input parameters from performance bug reports [27], assignment of the priority labels to bug reports [61], bug report field reassignment and refinement prediction [68].

In particular, a few studies of bug report classification are more relevant to our work. Some of those approaches focus on building bug classification models based on analyzing bug reports with text mining. For example, Zhou et al. [73] leveraged text mining techniques, analyzed the summary parts of bug reports and fed into machine learners. Xia et al. [67] developed a framework that applied text mining technology on bug reports, and trained a model on bug reports with known labels (i.e., configuration or non-configuration). The trained model was used to predict the new bug reports. Popstojanova et al. [22] used different types of textual feature vectors and focused on applying both supervised and unsupervised algorithms in classifying security and non-security related bug reports. Wijayasekara et al. [66] extracted textual information by utilizing the textual description of the bug reports. A feature vector was generated through the textual information, and then presented to a machine learning classifier.

Some other approaches use a more heuristic way to identify bug reports. For example, Zaman et al. [71] combined keyword searching and statistical sampling to distinguish between performance bugs and security bugs in Firefox bug reports. Gegick et al. [20] proposed a technique to identify security bug reports based on keyword mining, and performed an empirical study based on an industry bug repository.

3.2 FARSEC: Extending Text Mining for Bug Reports

Table 3 Different Filters used in FARSEC.

Filter	Description
farsecsq	Apply the Jalali et al. [33] support function to the frequency of words found in SBRs
farsectwo	Apply the Graham version [23] of multiplying the frequency by two.
farsec	Apply no support function.
clni	Apply CLNI filter to non-filtered data.
clnifarsec	Apply CLNI filter to farsec filtered data.
clnifarsecsq	Apply CLNI filter to farsecsq filtered data.
clnifarsectwo	Apply CLNI filter to farsectwo filtered data.

FARSEC [53] is a technique that adds an irrelevancy pruning step to text mining in building security bug prediction models. Table 3 lists the pruners explored in the FARSEC research. The purpose of filtering in FARSEC is to remove non-security bug reports with security related keywords. To achieve this goal, FARSEC applied an algorithm that firstly calculated the probability of the keyword appearing in security bug report and non-security bug report, and then calculated the score of the keyword.

Inspired by previous works [23] [33], several tricks were also introduced in FARSEC to reduce false positives. For example, FARSEC built the *farsectwo* pruner by multiplying the frequency of non-security bug reports by two, aiming to achieve a good bias. The *farsecsq* filter was created by squaring the numerator of the support function to improve heuristic ranking of low frequency evidence.

In addition, FARSEC also tested a noise detection algorithm called CLNI (Closet List Noise Identification) [36]. Specifically, CLNI works as follows: During each iteration, for each instance i, a list of closest instances are calculated and sorted according to Euclidean Distance to instance i. The percentage of top N instances with different class values is recorded. If percentage value is larger or equal to a threshold, then instance i is highly probable to be a noisy instance and thus included to noise set S. This process is repeated until two noise sets S_i and S_{i-1} have the similarity over ϵ. A threshold score (e.g., 0.75) is set to remove any non-buggy reports above the score.

3.3 Pre-processor Optimization

Class imbalance is a common problem in bug reports, which could lead to poor performance in classifiers. To handle such problem, previous work employed several strategies, such as Random over-sampling (ROS), random under-sampling (RUS), SMOTE, and cost-matrix adjuster (CMA) [70]. In addition, some work also proposed enhanced approaches over SMOTE, such as CR-SMOTE [25] or RSMOTE [24] [14]. The general idea of these is to generate more efficient synthetic instances in the minority category.

Different from previous techniques, Agrawal et al. [3] improve SMOTE in a more efficient way from the perspective of pre-processor hyperparameter optimization, and in their results they summarize as "better data" does better than "better learners". In their work, Argawal et al. studied defect classification and usually methods that work for defect classification might not work for vulnerability classification. This is because the kinds of data explored by defect classification and vulnerability classification are very different. The vulnerability frequencies reported in the Introduction (1-4% of files) are very common. Yet defect classification usually is performed on data with much larger target classes. For example, the defect data in a recent defect classification modeling (DPM) paper used 11 data sets with median number of files with defects of 34% [39].

Hence we say that prior to this paper, it was an open question if methods (like those of Agrawal et al.) which are certified for one kind of software engineering problem (defect classification) work for a very different kind of problem (vulnerability classification).

4 SWIFT: the Dual Optimization Approach

As we see from previous discussions, even for each optimization objectives, the tuning work requires evaluations of hundreds to thousands of tuning options. If we optimize the learners as well as other pre-processors (list in Table 1) at the same time, the cost of running a data miner through all those options would be very expensive.

To better address the dual optimization problem, we use a technique called ϵ-*dominance*. In 2005, Deb et al. [16] proposed partitioning the output space of an optimizer into ϵ-sized grids. In multi-objective optimization (MOO), a solution is said to *dominate* the other solution if and only if it is better in at least one objective, and no worse in other objectives. A set of optimal solutions that are not dominated by any other feasible solutions form the *Pareto frontier*. Fig 2 is an example of the output space based on ϵ-dominance. The yellow dots in the figure form the Pareto frontier.

Deb's principle of ϵ-dominance is that if there exists some ϵ value below which is useless or impossible to distinguish results, then it is superfluous to explore anything less than ϵ [16]. Specifically consider the security bug recognition discussed in this paper, if the performances of two learners (or a learner with various parameters) diff in less than some ϵ, then we cannot statistically distinguish them. For the learners

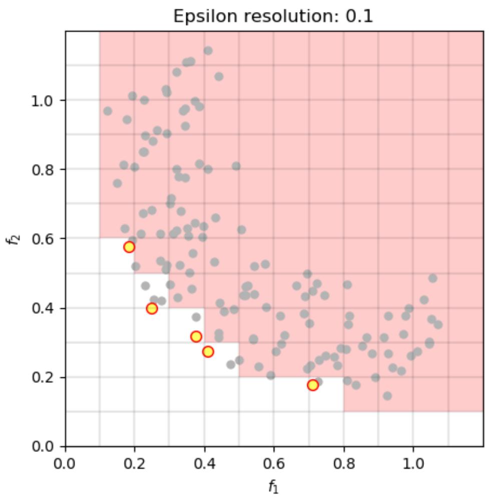

Fig. 2 An example of Pareto frontier. Objectives f1 and f2 are both to be minimized. ϵ value is set to 0.1. The gray dots represent the feasible solutions. The yellow dots represent the Pareto optimal solutions.

do not significantly improve the performance, we can further reduce the attention on them.

Agrawal et al. [5] successfully applied ϵ-dominance to software defect prediction and SE text mining. Their approach, called DODGE, was a tabu search; i.e. if some setting resulted in some performance within ϵ of any older result, then DODGE marked that option as "to be avoided". That algorithm returned the best setting seen during the following three stage process:

– *Initialization*: all option items i are assigned equal weightings;
– The *item ranking* stage reweights items i in column 2 of Table 2; e.g. terms like "Random Forest" or "RobustScaler";
– The *numeric refinement* stage adjusts the tuning ranges of the last column in Table 2.

That is to say, item selection handles "big picture" decisions about what pre-processor or learner to use while numeric refinement focuses on smaller details about numeric ranges.

More specifically, the algorithm runs as follows:

– *Initialization*: assign weights $w_i = 0$ to all items i in column 2 of Table 2.
– *Item ranking*: N_1 times, make a random selection of a learner and pre-processor from column 2, favoring those items with higher weights. For the selected items, select a value at random from the "Tuning Range"s of the last column of Table 2. Using that selection, build a model and evaluated it on test data. If we obtain a model whose performance is is more/less than ϵ of any prior results, then we add/subtract (respectively) 1.0 from w_i.

– *Numeric refinement*: N_2 times, we refine the numeric tuning ranges (lo, hi) seen in the last column of Table 2. In this step, the item ranking continues. But now, if ever some numeric tuning value $lo \leq b \leq hi$ produces a better model, then DODGE adjust that range, as follows. Whichever of $x \in$*lo,hi* that is the furthest from b is moved to $(b + x)/2$.

For the text mining and defect prediction case studies studied by DODGE, that approach was able to explore a large space of hyperparameter options, while at the same time generate models that performed as well or better than the prior state of the art in defect prediction and SE text mining [5].

When we applied DODGE to our security data, it did not work very well. On investigation, we found that the distribution of the w_i weights were highly skewed; i.e. there was usually only one very good learner and one very good data pre-processor (we conjecture that this is so since we require very specific biases to find the target concept of something so particular as a security bug report). But for the original version of DODGE, this is a problem since, as mentioned above, item ranking continues during the numeric refinement stage. This meant, in turn, that DODGE was wasting much time considering less-than-useful items during numeric refinement.

Accordingly, we introduce an improved approach called SWIFT, which changes DODGE as follows. After item ranking, only the best learner and data pre-processor are carried forward into numeric refinement. While that was a small coding change to the original DODGE, its effects were profound. As shown later in this paper, SWIFT significantly out-performs other methods studied here.

5 Experiment

5.1 Hyper-parameter Tuning Ranges

This paper compares SWIFT against the differential evolution algorithm (described above) since recent papers at ICSE [3] the IST journal [4] reported that this optimizer can find large improvements in learner performance for SE data. See Table 4 for the control settings for the differential evolution algorithm control parameters used in this paper (that table was generated by combining the advice at the end of §2.2 with Table 2). For SWIFT, we used the settings recommended by Agrawal et al. [5].

Note that SWIFT and differential evolution algorithm were applied to learners from the Python scikit-learn toolkit [51]. Table 2 lists all the hyperparameters we select for both data mining learners and data pre-processors based on scikit-learn.

We choose not to explore other hyperparameter optimizers, for pragmatic reasons. In our experience, the differential evolution algorithm runs much faster than many other standard optimizers such as NSGA-II [15]. But as shown below, even this "fast" optimizer takes a full working day to terminate on the dual optimization problem for our more complex data sets. When repeated multiple times (within a cross-validation), we needed 40+ hours of CPU to get a result. Meanwhile, SWIFT was accomplishing the same tasks in just 2 hours while producing results that dramatically out-performed the prior state-of-the-art. For that reason, we elected to report the current results and leave the comparison to other, much more slower, optimizers for future work.

Table 4 List of parameters in differential evolution algorithm for different learners and pre-processor.

Learner & Pre-processor	DE Parameter			
	NP	F	CR	ITER
Random Forest	60	0.8	0.9	3, 10
Logistic Regression	30			
Multilayer Perceptron	60			
K Nearest Neighbor	20			
Naive Bayes	10			
SMOTE	30	0.8	0.9	10

5.2 Data

For this work, we compared the differential evolution algorithm (DE) and SWIFT to FARSEC using the same data as used in the FARSEC study; i.e., one Chromium project and four Apache projects (i.e., Wicket, Ambari, Camel and Derby). According to FARSEC, these bug reports from the Apache projects were randomly selected with a *BUG* or *IMPROVEMENT* label, and then manually labeled via human inspection. For the Chromium project, security bugs are labeled as *Bug-Security* when submitted to bug tracking systems. All other types of bug reports are treated as non-security bug reports.

Table 5 shows the FARSEC datasets. As mentioned in the introduction, one unique feature of that data is the rariety of the target class. As shown in the “Percent Buggy” column, security reports make up a very small percentage of the total number of bug reports in projects like Chromium.

5.3 Experimental Rig

The results of this paper come from a 10-way cross-validation. After dividing each *training data* into $B = 10$ bins, we tested our models using bin B_i after training them on *data* $-B_i$. The 10-way cross-validation is used to pick the best candidate learner with the highest performance for that data set in tuning learners or pre-processors. Note that selecting the best candidate learner in SWIFT is based on weight calculation as we discuss in Section 4. We then train the candidate learner with the whole training data set, and test on the separate testing data set as FARSEC.

This 10-way process was applied to five projects (described in the next section); the five learners listed above; and the following eight data pruning operators:

- *train*; i.e. no data pruning;
- The seven variations of *farsec* and *clni* listed in Table 3.

That is, we ran our learners 2000 times:

$$5\ projects * 8\ pruners * 5\ learners * 10\ ways$$

Table 5 Imbalanced characteristics of bug report data sets from FARSEC [53]

Project	Filter	#Security Bugs	#Bug reports	Percent buggy
Chromium	train	77	20970	0.4
	farsecsq		14219	0.5
	farsectwo		20968	0.4
	farsec		20969	0.4
	clni		20154	0.4
	clnifarsecsq		13705	0.6
	clnifarsectwo		20152	0.4
	clnifarsec		20153	0.4
Wicket	train	4	500	0.8
	farsecsq		136	2.9
	farsectwo		143	2.8
	farsec		302	1.3
	clni		392	1.0
	clnifarsecsq		46	8.7
	clnifarsectwo		49	8.2
	clnifarsec		196	2.0
Ambari	train	22	500	4.4
	farsecsq		149	14.8
	farsectwo		260	8.5
	farsec		462	4.8
	clni		409	5.4
	clnifarsecsq		76	28.9
	clnifarsectwo		181	12.2
	clnifarsec		376	5.9
Camel	train	14	500	2.8
	farsecsq		116	12.1
	farsectwo		203	6.9
	farsec		470	3.0
	clni		440	3.2
	clnifarsecsq		71	19.7
	clnifarsectwo		151	9.3
	clnifarsec		410	3.4
Derby	train	46	500	9.2
	farsecsq		57	80.7
	farsectwo		185	24.9
	farsec		489	9.4
	clni		446	10.3
	clnifarsecsq		48	95.8
	clnifarsectwo		168	27.4
	clnifarsec		435	10.6

5.4 Evaluation Metrics

To understand the open issues with vulnerability classification, firstly we must define how they are **assessed**. If (TN, FN, FP, TP) are the true negatives, false negatives, false positives, and true positives, respectively, found by a detector, then:

– *pd* = Recall = TP/(TP+FN), the percentage of the actual vulnerabilities that are predicted to be vulnerabilities;
– *pf* = False Alarms = FP/(FP+TN), the percentage of the non-vulnerable artifacts that are reported as vulnerable;

– *prec* = Precision = TP/(TP+FP), the percentage of the predicted vulnerabilities that are actual vulnerabilities;

This paper adopts the same evaluation criteria of the original FARSEC paper; i.e. the recall (*pd*) and false alarm (*pf*) measures. Also, to control the optimization of differential evolution algorithm, we instructed it to minimize false alarms while maximizing recall. To achieve those goals, in differential evolution algorithm, we maximize the *g-measure* which is the harmonic mean of recall and the compliment of false alarms.

$$g = \frac{2 \times pd \times (1 - pf)}{pd + (1 - pf)} \tag{1}$$

g is maximal when *both* recall (*pd*) is high and false alarm (*pf*) is low. In the case of classifying unbalanced data, *g-measure* is more suitable for evaluation purpose than other metrics such as precision.

Note that recall evaluates a classifiers performance with respect to false negatives, while precision evaluates its performance with respect to false positives. For imbalanced dataset such as bug reports where security bugs are in a very small portion, we want to focus more on minimising False Negatives. Therefore, g-measure is a better choice.

Besides, we also adopt another evaluation measure called IFA (Initial False Alarm) to evaluate the performance. IFA is the number of initial false alarm encountered before we make the first correct prediction [31] [32]. IFA is widely used in defect prediction, and previous works [38] [50] have shown that developers are not willing to use a prediction model if the first few recommendations are all false alarms.

5.5 Statistics

This study ranks treatments using the Scott-Knott procedure recommended by Mittas & Angelis in their 2013 IEEE TSE paper [48]. This method sorts results from different treatments, then splits them in order to maximize the expected value of differences in the observed performances before and after divisions. For lists l, m, n of size ls, ms, ns where $l = m \cup n$, the "best" division maximizes $E(\Delta)$; i.e. the difference in the expected mean value before and after the spit:

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

Scott-Knott then checks if that "best" division is actually useful. To implement that check, Scott-Knott would apply some statistical hypothesis test H to check if m, n are significantly different (and if so, Scott-Knott then recurses on each half of the "best" division). For this study, our hypothesis test H was a conjunction of the A12 effect size test of and non-parametric bootstrap sampling; i.e. our Scott-Knott divided the data if *both* bootstrapping and an effect size test agreed that the division was statistically significant (95% confidence) and not a "small" effect ($A12 \geq 0.6$).

For a justification of the use of non-parametric bootstrapping, see Efron & Tibshirani [18, p220-223]. For a justification of the use of effect size tests see Kampenes [34] who warn that even if a hypothesis test declares two populations to be "significantly" different, then that result is misleading if the "effect size" is very small.

Hence, to assess the performance differences we first must rule out small effects. Vargha and Delaney's non-parametric A12 effect size test explores two lists M and N of size m and n:

$$A12 = \left(\sum_{x \in M, y \in N} \begin{cases} 1 & if\ x > y \\ 0.5 & if\ x == y \end{cases} \right) / (mn)$$

This expression computes the probability that numbers in one sample are bigger than in another. This test was endorsed by Arcuri and Briand [6]. Table 6, Table 7 present the reports of our Scott-Knott procedure for each project data set. These results are discussed, extensively, over the next two pages.

6 Results

We now answer our proposed research questions:

6.1 RQ1

RQ1. Can hyperparameter optimization technique distinguish which bug are actually security problems?

Table 6, Table 7 and Table 8 report results with and without hyperparameter optimization of the pre-processor or learners or both. In the table, in each row, the gray highlights show all results ranked "best" by a statistical Scott-Knot test.

6.1.1 Recall Results

In the recall results of Table 6 , FARSEC very rarely achieved best results while SWIFT achieved results that were dramatically better than FARSEC. For example:

- In Chromium, mean recall changed from 20.4% to 77.1% from FARSEC to SWIFT
- In Ambri, mean recall changed from 30.4 to 83.9% from FARSEC to SWIFT.

While SWIFT's recall reults inn Chromium are good, the gray cells show us that other methods did even better. In Table 5, we saw that this data had the smallest target class (often less than half a percent). For this data set, it seems more important to fix the class imbalance (with SMOTE) than to adjust the learners (with SWIFT). But even here SWIFT performed much better than FARSEC. Also, in 6 of 8 cases, SWIFT's Chromium recalls were either best or within 5% of the best.

Just for completeness, we note that for all methods that did any data preprocessing (SMOTE, SMOTUNED, SWIFT) performed very well for Wicket. Clearly, for this data set, repairing the class imbalance issue is essential.

Table 6 **RQ1** results: recall. In these results, *higher* recalls (a.k.a. pd) are *better*. For each row, best results are highlighted in gray (these are the cells that are statistically the same as the best median result– as judged by our Scott-Knot test). Across all rows, SWIFT has the most number of best results.

		Prior state of the art [53]	Tuning learner (only)	Imbalance class repair (no tuning)	Imbalance class repair (tuned via DE)	Tune everything (dual)
Project	**Filter**	**FARSEC**	**DE+ Learner**	**SMOTE**	**SMOTUNED**	**SWIFT**
Chromium	train	15.7	46.9	68.7	73.9	86.1
	farsecsq	14.8	64.3	80.0	84.3	72.2
	farsectwo	15.7	40.9	78.3	77.4	77.4
	farsec	15.7	46.1	80.8	72.2	77.4
	clni	15.7	30.4	74.8	72.2	80.9
	clnifarsecsq	49.6	72.2	82.6	86.1	72.2
	clnifarsectwo	15.7	50.4	79.1	74.8	78.3
	clnifarsec	15.7	47.8	78.3	74.7	72.2
Wicket	train	16.7	0.0	66.7	66.7	50.0
	farsecsq	66.7	50.0	83.3	83.3	83.3
	farsectwo	66.7	50.0	66.7	66.7	66.7
	farsec	33.3	66.7	66.7	66.7	66.7
	clni	0.0	16.7	50.0	50.0	50.0
	clnifarsecsq	33.3	83.3	83.3	83.3	83.3
	clnifarsectwo	33.3	50.0	66.7	66.7	66.7
	clnifarsec	50.0	66.7	66.7	66.7	66.7
Ambari	train	14.3	28.6	57.1	57.1	85.7
	farsecsq	42.9	57.1	57.1	57.1	85.7
	farsectwo	57.1	57.1	57.1	57.1	85.7
	farsec	14.3	57.1	57.1	57.1	85.7
	clni	14.3	28.6	57.1	57.1	85.7
	clnifarsecsq	57.1	57.1	57.1	57.1	71.4
	clnifarsectwo	28.6	57.1	57.1	57.1	85.7
	clnifarsec	14.3	57.1	57.1	57.1	85.7
Camel	train	11.1	16.7	33.3	44.4	55.6
	farsecsq	16.7	44.4	44.4	55.6	66.7
	farsectwo	50.0	44.4	61.1	61.1	61.1
	farsec	16.7	22.2	33.3	33.3	55.6
	clni	16.7	16.7	33.3	38.9	50.0
	clnifarsecsq	16.7	38.9	27.8	33.3	61.1
	clnifarsectwo	11.1	61.1	72.2	61.1	61.1
	clnifarsec	16.7	22.2	33.3	38.9	55.6
Derby	train	38.1	47.6	54.7	59.5	69.0
	farsecsq	54.8	59.5	54.7	66.7	66.7
	farsectwo	47.6	59.5	47.6	66.7	78.6
	farsec	38.1	47.6	57.1	59.5	64.3
	clni	23.8	45.2	57.7	61.9	69.0
	clnifarsecsq	54.8	59.5	76.2	69.0	66.7
	clnifarsectwo	35.7	59.5	54.8	61.9	66.7
	clnifarsec	38.1	47.6	61.9	57.1	66.7

Table 7 RQ1 results: false positive rate, the lower values are better. Same as Table 6; i.e. best results are shown in grey. While FARSEC has the the most best results, these low false alarms are only achieved by settling for very low recalls (see Table 6).

		Prior state of the art [53]	Tuning learner (only)	Imbalance class repair (no tuning)	Imbalance class repair (tuned via DE)	Tune everything (dual)
Project	**Filter**	**FARSEC**	**DE+ Learner**	**SMOTE**	**SMOTUNED**	**SWIFT**
Chromium	train	0.2	6.8	24.1	17.8	24.0
	farsecsq	0.3	10.3	31.5	25.1	14.3
	farsectwo	0.2	6.5	27.6	23.1	26.1
	farsec	0.2	6.9	36.1	14.9	14.7
	clni	0.2	4.1	24.8	13.6	26.2
	clnifarsecsq	3.8	14.2	30.4	25.6	14.0
	clnifarsectwo	0.2	7.0	29.9	12.8	18.9
	clnifarsec	0.2	10.4	29.0	17.1	20.2
Wicket	train	7.1	5.1	32.0	12.1	27.5
	farsecsq	38.3	44.5	71.3	66.8	66.7
	farsectwo	36.6	42.3	68.2	62.9	61.5
	farsec	8.1	23.1	43.9	26.1	23.3
	clni	5.5	2.4	21.1	12.5	14.4
	clnifarsecsq	25.5	66.8	66.8	66.8	57.5
	clnifarsectwo	27.7	39.9	61.3	61.3	52.8
	clnifarsec	10.5	23.1	38.9	22.9	22.1
Ambari	train	1.6	0.8	20.1	10.8	17.8
	farsecsq	14.4	2.8	30.4	17.2	23.7
	farsectwo	3.0	2.8	22.1	17.8	19.7
	farsec	4.9	2.0	19.9	7.1	20.3
	clni	2.6	0.8	12.4	8.9	18.1
	clnifarsecsq	7.7	2.4	13.4	7.1	29.0
	clnifarsectwo	4.5	2.8	13.0	5.1	22.7
	clnifarsec	0.0	2.4	7.9	3.9	18.9
Camel	train	3.5	1.5	27.4	35.9	15.8
	farsecsq	11.4	24.7	20.5	23.4	27.8
	farsectwo	41.8	17.6	71.0	53.1	45.2
	farsec	6.9	12.4	39.4	28.0	35.7
	clni	12.3	7.9	33.6	35.3	24.7
	clnifarsecsq	13.9	14.9	12.4	15.6	27.2
	clnifarsectwo	7.7	50.0	64.9	51.9	38.8
	clnifarsec	5.0	11.6	24.9	34.4	37.1
Derby	train	6.8	39.3	22.2	20.7	19.7
	farsecsq	29.9	40.6	51.7	51.5	22.5
	farsectwo	12.4	24.2	27.9	33.6	40.0
	farsec	6.3	4.1	21.0	19.0	13.8
	clni	0.4	3.5	16.8	24.5	25.5
	clnifarsecsq	29.9	42.4	74.7	65.1	42.3
	clnifarsectwo	9.2	24.2	36.5	30.3	52.2
	clnifarsec	6.8	3.9	28.8	10.9	19.6

6.1.2 False Alarm Results

As to the false alarm results, Table 7 shows that FARSEC has the lowest false alarm. However, it is clear that these low false alarms are only achieved by settling for very low recalls (as seen in Table 6). SWIFT's false alarm rare are highest but with one exception, not dramatically so. For example, in Chromium, SWIFT's mean false alarm rate is less than 20%.

The one exception is that SWIFT had high false alarm rates in Wicket– but then so did every other method that achieved recalls over 50%. From Table 5, we know that Wicket only has too few known security bugs. For such an extremely rarefied data set, we would argue that high false alarm rates are the inevitable result of achieving medium to high recalls.

6.1.3 Initial False Alarms

IFA is the number of false alarms a programmer must suffer through before they find a real security bug. Table 8 show our IFA results. There are three things to note from that table:

– FARSEC has no results in this table because FARSEC was implemented in Weka which does not report IFA.
– For IFA, methods that only with/tune the data pre-processor (SMOTE and SMOTUNED) perform worse than methods that tune the learner (DE+Learner and SWIFT).
– In terms of absolute numbers, the IFA results are very low for Ambrai and Derby. From Table 5, we can conjecture some reason for this– of the data sets with more than 1% (on average) security bugs, these data sets have the most known security bugs.
– At the other end of the spectrum, IFA is much larger for Chromium (mean values for DE+learner or SWIFT of 50 or 60). This result highlights the high cost of building highly secure software. When the target class is rare, even with our best-of-breed methods, some non-trivial amount of manual effort may be required.

6.1.4 RQ1, Summary

In summary, for **RQ1** we can answer, hyperparameter optimization technique help better distinguish security bug reports from non-security bugs reports, i.e., improving bug reports classification performance. Further, the best results for recall/pd come at the cost of a relatively minor increase in the false alarm rate.

6.2 RQ2

RQ2. Is it better to dual optimize the learners or the data pre-processors in security bug report classification?

Table 8 **RQ1** results: initial false alarm (IFA). IFA is the number of false alarms developers must suffer through before finding their first error. Lower values are better. Same as format as Table 6; i.e. best results are shown in grey.

		Prior state of the art [53]	Tuning learner (only)	Imbalance class repair (no tuning)	Imbalance class repair (tuned via DE)	Tune everything (dual)
Project	**Filter**	**FARSEC**	**DE+ Learner**	**SMOTE**	**SMOTUNED**	**SWIFT**
Chromium	train	N/A	62	75	61	58
	farsecsq	N/A	20	72	54	36
	farsectwo	N/A	37	91	78	87
	farsec	N/A	62	112	62	56
	clni	N/A	41	86	48	74
	clnifarsecsq	N/A	41	57	62	37
	clnifarsectwo	N/A	37	89	47	58
	clnifarsec	N/A	62	113	63	54
Wicket	train	N/A	25	60	34	46
	farsecsq	N/A	29	37	33	39
	farsectwo	N/A	32	35	34	31
	farsec	N/A	23	44	30	22
	clni	N/A	12	44	21	27
	clnifarsecsq	N/A	9	8	9	6
	clnifarsectwo	N/A	8	11	12	8
	clnifarsec	N/A	17	33	15	18
Ambari	train	N/A	7	8	9	4
	farsecsq	N/A	8	21	14	7
	farsectwo	N/A	1	19	12	3
	farsec	N/A	1	35	24	17
	clni	N/A	1	32	19	13
	clnifarsecsq	N/A	8	18	10	8
	clnifarsectwo	N/A	7	28	8	11
	clnifarsec	N/A	5	10	4	17
Camel	train	N/A	6	19	23	15
	farsecsq	N/A	23	29	32	14
	farsectwo	N/A	4	13	8	25
	farsec	N/A	17	21	20	8
	clni	N/A	16	37	33	30
	clnifarsecsq	N/A	5	3	3	4
	clnifarsectwo	N/A	19	22	15	12
	clnifarsec	N/A	14	23	29	22
Derby	train	N/A	4	6	3	2
	farsecsq	N/A	4	4	4	4
	farsectwo	N/A	4	3	5	3
	farsec	N/A	1	8	7	4
	clni	N/A	1	8	5	3
	clnifarsecsq	N/A	1	2	2	1
	clnifarsectwo	N/A	2	9	8	4
	clnifarsec	N/A	1	3	3	2

Table 9 How often is each treatment seen to be best in Table 6, Table 7 and Table 8.

Metric	Rank	Method	Win Times
Recall	1	SWIFT	31/40
	2	SMOTE	14/40
	3	SMOTUNED	13/40
	4	DE+Learner	3/40
False Alarm	1	DE+Learner	14/40
	2	SMOTE	1/40
	3	SWIFT	1/40
	5	SMOTUNED	0/40
IFA	1	DE+Learner	22/40
	2	SWIFT	18/40
	3	SMOTUNED	4/40
	4	SMOTE	3/40

This question explores the merits of dual optimization of learner plus pre-processor versus just optimizing one or the other. To answer that question, we count how often each method achieved top-rank (and had gray-colored results) across all three metrics of the rows on Table 6, Table 7 and Table 8.

Those counts are shown in Table 9. From that table, we can say, in terms of recall:

– SWIFT's dual optimization is clearly best.
– Tuning just the data pre-processing (with SMOTE or SMOTUNED) comes a distant second;
– And tuning just the learners (with DE+Learner) is even worse.

Hence we say that, when distinguishing security bugs, it is not enough to just tune the learners.

In terms of false alarms, we see that:

– Tuning just the learner is a comparatively better method than anything else.
– Tuning anything else does not do well on the false alarm scale.

That said, tuning just the learner achieves a score of 14/40– which is not even half the results. Hence, based on false alarm rates, we cannot comment on what works best for improving false alarms.

In terms of IFA (initial false alarms), we see that:

– Methods that do not tune a learner (SMOTE and SMOTUNED) perform very badly.
– As to the other methods, there is is no clear winner for best method. DE+Learner or SWIFT perform nearly the same as each other.

6.3 RQ2: Summary

Based on the above, we say that:

– We have unequivocal results that dual tuning works very well for recall.

Table 10 Average runtime (in minutes) of tuning all learners' hyperparameters, SMOTE's hyperparameters and running SWIFT.

Project	DE3	DE10	SMOTUNED	SWIFT
Chromium	455	876	20	12
Wicket	8	11	8	5
Ambari	8	11	8	5
Camel	8	11	8	5
Derby	8	11	8	5

- Also, not tuning the learner performs very badly for IFA.
- There is no clear pattern in Table 9 regarding false alarm rates.

That said, the absolutely numbers for false alarm see in Table 7 are somewhat lower than the false alarm rates seen in other analytics papers [45]. Hence, on a more positive note, we can still recommend dual optimization since:

- It has much benefit (dramatically higher recalls);
- With no excessive cost (no large increase in false alarms; IFA rates nearly as good as anything else).

Further to this comment of "no excessive cost", Table 10 shows the average runtime for each treatment. From the table, differential evolution algorithm on learners consume much more CPU time than others, while dual optimization as SWIFT shows slight advantages even better than SMOTUNED.

6.4 RQ3

RQ3. What are the relative merits of irrelevancy pruning (e.g., filtering) vs hyperparameter optimization in security bug report classification?

In this question, we explore the core assumptions of the FARSEC study and this study:

- FARSEC assumed that it was best to remove non-security bug reports by checking for any security related keywords.
- We assume that it is best to perform hyperparamter optimziation on both the data miner and data pre-processing.

Our results show that with hyperparameter optimization, there is no added benefit to FARCECs irrelevancy pruning. For example, for each project, the *train* pd results in the *FARSEC* column of Table 6 are usually smaller than the other values seen after applying some of the filters proposed by Peters et al. This result confirms that, without hyperparameter optimization, irrelevancy pruning does indeed improve the performance of vulnerability classification.

Hence, while we recommend hyperparameter optimization, if data analysts wish to avoid that added complexity, they could apply FARSECs irrelevancy pruning.

7 Threats to Validity

As to any empirical study, biases can affect the final results. Therefore, conclusions drawn from this work must be considered with threats to validity in mind.

Sampling Bias. Sampling bias threatens any classification experiments. For example, the data sets used here come from FARSEC, i.e., one Chromium project and four Apache projects in different application domains. In addition, the bug reports from Apache projects are randomly selected with a BUG or IMPROVEMENT label for each project. On way to justify these decisions is to say that they are exactly the same decisions as those adopted by Peters et al. in their TSE paper.

Learner Bias. Research into automatic classifiers is a large and active field. While different machine learning algorithms have been developed to solve different classification problem tasks. Any data mining study, such as this paper, can only use a small subset of the known classification algorithms. For this week, we selected our learners such that we can compare our results to prior work. According, we used the same learners as Peters et al. in their FARSEC research.

Input Bias. Our results come from the space of hyperparameter tunings explored in this paper. In theory, other ranges might lead to other results. That said, our goal here is not to offer the *best* tuning but to argue that *dual* tuning of data pre-processors and algorithms is preferable to tuning either, just by itself. For those purposes, we would argue that our current results suffice.

8 Conclusion

Distinguishing security bugs from other kinds of bugs is a pressing problem that threatens not only the viability of software services, but also consumer confidence in those services. Prior results on how to distinquish such bugs have had issues with the scarcity of vulnerability data (specifically, such incidents occur very rarely). In a recent TSE'18 paper, Peters et al. proposed some novel filtering algorithms to help improve security bug report classification. Results from FARSEC show that such filtering techniques can improve bug report classification.

But more than that, our experiments show that we can do better than FARSEC using hyperparameter optimization of learners and data pre-processors. Of those two approaches, our results show that it is more advantageous to apply *dual* optimization of *both* the data-processor *and* the learner, which we will recommend in solving similar problems in the future work.

Acknowledgements This work was partially funded via an NSF-CISE grant #1909516.

References

1. (2017) WannaCry ransomware attack. `https://en.wikipedia.org/wiki/WannaCry_ransomware_attack`

2. (2019) The Equifax Data Breach. `https://www.ftc.gov/equifax-data-breach`
3. 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, ACM, pp 1050–1061
4. Agrawal A, Fu W, Menzies T (2018) What is wrong with topic modeling? and how to fix it using search-based software engineering. Information and Software Technology 98:74–88
5. Agrawal A, Fu W, Chen D, Shen X, Menzies T (2019) How to" dodge" complex software analytics? arXiv preprint arXiv:190201838
6. Arcuri A, Briand L (2011) A practical guide for using statistical tests to assess randomized algorithms in software engineering. In: Proceedings of the 33rd International Conference on Software Engineering, ACM, New York, NY, USA, ICSE '11, pp 1–10, DOI 10.1145/1985793.1985795, URL `http://doi.acm.org/10.1145/1985793.1985795`
7. Bennin KE, Keung JW, Monden A (2019) On the relative value of data resampling approaches for software defect prediction. Empirical Software Engineering 24(2):602–636
8. Bergstra J, Bengio Y (2012) Random search for hyper-parameter optimization. Journal of Machine Learning Research 13(Feb):281–305
9. Bergstra JS, Bardenet R, Bengio Y, Kégl B (2011) Algorithms for hyper-parameter optimization. In: Advances in neural information processing systems, pp 2546–2554
10. Biedenkapp A, Eggensperger K, Elsken T, Falkner S, Feurer M, Gargiani M, Hutter F, Klein A, Lindauer M, Loshchilov I, et al. (2018) Hyperparameter optimization. Artificial Intelligence 1:35
11. Black PE, Badger L, Guttman B, Fong E (2016) Nistir 8151: Dramatically reducing software vulnerabilities
12. Chawla NV, Bowyer KW, Hall LO, Kegelmeyer WP (2002) Smote: synthetic minority over-sampling technique. Journal of artificial intelligence research 16:321–357
13. Chen L, et al. (2013) R2fix: automatically generating bug fixes from bug reports. Proceedings of the 2013 IEEE 6th ICST
14. Chen R, Guo S, Wang X, Zhang T (2019) Fusion of multi-rsmote with fuzzy integral to classify bug reports with an imbalanced distribution. IEEE Transactions on Fuzzy Systems
15. Deb K, Pratap A, Agarwal S, Meyarivan T (2002) A fast and elitist multiobjective genetic algorithm: Nsga-ii. IEEE Transactions on Evolutionary Computation 6(2):182–197, DOI 10.1109/4235.996017
16. Deb K, Mohan M, Mishra S (2005) Evaluating the ε-domination based multi-objective evolutionary algorithm for a quick computation of pareto-optimal solutions. Evolutionary computation 13(4):501–525
17. Deshmukh J, Podder S, Sengupta S, Dubash N, et al. (2017) Towards accurate duplicate bug retrieval using deep learning techniques. In: 2017 IEEE International conference on software maintenance and evolution (ICSME), IEEE, pp 115–124

18. Efron B, Tibshirani RJ (1994) An introduction to the bootstrap. CRC press
19. Fu W, Menzies T, Shen X (2016) Tuning for software analytics: Is it really necessary? Information and Software Technology 76:135–146
20. Gegick M, Rotella P, Xie T (2010) Identifying security bug reports via text mining: An industrial case study. In: Mining software repositories (MSR), 2010 7th IEEE working conference on, IEEE, pp 11–20
21. Goldberg DE (2006) Genetic algorithms. Pearson Education India
22. Goseva-Popstojanova K, Tyo J (2018) Identification of security related bug reports via text mining using supervised and unsupervised classification. In: 2018 IEEE International Conference on Software Quality, Reliability and Security (QRS), IEEE, pp 344–355
23. Graham P (2004) Hackers & painters: big ideas from the computer age. " O'Reilly Media, Inc."
24. Guo S, Chen R, Wei M, Li H, Liu Y (2018) Ensemble data reduction techniques and multi-rsmote via fuzzy integral for bug report classification. IEEE Access 6:45934–45950
25. Guo S, Chen R, Li H, Zhang T, Liu Y (2019) Identify severity bug report with distribution imbalance by cr-smote and elm. International Journal of Software Engineering and Knowledge Engineering 29(02):139–175
26. Haixiang G, Yijing L, Shang J, Mingyun G, Yuanyue H, Bing G (2017) Learning from class-imbalanced data: Review of methods and applications. Expert Systems with Applications 73:220–239
27. Han X, Yu T, Lo D (2018) Perflearner: learning from bug reports to understand and generate performance test frames. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ACM, pp 17–28
28. Herodotou H, Lim H, Luo G, Borisov N, Dong L, Cetin FB, Babu S (2011) Starfish: a self-tuning system for big data analytics. In: Cidr, vol 11, pp 261–272
29. Hindle A, Alipour A, Stroulia E (2016) A contextual approach towards more accurate duplicate bug report detection and ranking. Empirical Software Engineering 21(2):368–410
30. Hotho A, Nürnberger A, Paaß G (2005) A brief survey of text mining. In: Ldv Forum, Citeseer, vol 20, pp 19–62
31. Huang Q, Xia X, Lo D (2017) Supervised vs unsupervised models: A holistic look at effort-aware just-in-time defect prediction. In: 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, pp 159–170
32. Huang Q, Xia X, Lo D (2018) Revisiting supervised and unsupervised models for effort-aware just-in-time defect prediction. Empirical Software Engineering pp 1–40
33. Jalali O, Menzies T, Feather M (2008) Optimizing requirements decisions with keys. In: Proceedings of the 4th international workshop on Predictor models in software engineering, ACM, pp 79–86
34. Kampenes VB, Dybå T, Hannay JE, Sjøberg DIK (2007) A systematic review of effect size in software engineering experiments. Information and Software Technology 49(11-12):1073–1086

35. Keller JM, Gray MR, Givens JA (1985) A fuzzy k-nearest neighbor algorithm. IEEE transactions on systems, man, and cybernetics (4):580–585
36. Kim S, Zhang H, Wu R, Gong L (2011) Dealing with noise in defect prediction. In: Software Engineering (ICSE), 2011 33rd International Conference on, IEEE, pp 481–490
37. Kirkpatrick S, Gelatt CD, Vecchi MP (1983) Optimization by simulated annealing. science 220(4598):671–680
38. Kochhar PS, Xia X, Lo D, Li S (2016) Practitioners' expectations on automated fault localization. In: Proceedings of the 25th International Symposium on Software Testing and Analysis, ACM, pp 165–176
39. Krishna R, Menzies T, Layman L (2017) Less is more: Minimizing code reorganization using xtree. Information and Software Technology 88:53–66
40. Lamkanfi A, Demeyer S, Giger E, Goethals B (2010) Predicting the severity of a reported bug. In: Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on, IEEE, pp 1–10
41. Lazar A, Ritchey S, Sharif B (2014) Improving the accuracy of duplicate bug report detection using textual similarity measures. In: Proceedings of the 11th Working Conference on Mining Software Repositories, ACM, pp 308–311
42. Lessmann S, Baesens B, Mues C, Pietsch S (2008) Benchmarking classification models for software defect prediction: A proposed framework and novel findings. IEEE Transactions on Software Engineering 34(4):485–496
43. Mani I, Zhang I (2003) knn approach to unbalanced data distributions: a case study involving information extraction. In: Proceedings of workshop on learning from imbalanced datasets, vol 126
44. Menzies T, Shepperd M (2019) bad smells in software analytics papers. Information and Software Technology 112:35–47
45. Menzies T, Greenwald J, Frank A (2006) Data mining static code attributes to learn defect predictors. IEEE transactions on software engineering 33(1):2–13
46. Menzies T, Elrawas O, Hihn J, Feather M, Madachy R, Boehm B (2007) The business case for automated software engineering. In: Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering, ACM, New York, NY, USA, ASE '07, pp 303–312, DOI 10.1145/1321631.1321676, URL `http://doi.acm.org/10.1145/1321631.1321676`
47. Mitre (2017) Common vulnerabilities and exposures (cve). `https://cve.mitre.org/about/terminology.html#vulnerability`
48. Mittas N, Angelis L (2013) Ranking and clustering software cost estimation models through a multiple comparisons algorithm. IEEE Transactions on software engineering 39(4):537–551
49. Panichella A, Dit B, Oliveto R, Di Penta M, Poshyvanyk D, De Lucia A (2013) How to effectively use topic models for software engineering tasks? an approach based on genetic algorithms. In: International Conference on Software Engineering
50. Parnin C, Orso A (2011) Are automated debugging techniques actually helping programmers? In: Proceedings of the 2011 international symposium on software testing and analysis, ACM, pp 199–209

51. Pedregosa F, Varoquaux G, Gramfort A, Michel V, Thirion B, Grisel O, Blondel M, Prettenhofer P, Weiss R, Dubourg V, et al. (2011) Scikit-learn: Machine learning in python. Journal of machine learning research 12(Oct):2825–2830
52. Pelikan M, Goldberg DE, Cantú-Paz E (1999) Boa: The bayesian optimization algorithm. In: Proceedings of the 1st Annual Conference on Genetic and Evolutionary Computation-Volume 1, Morgan Kaufmann Publishers Inc., pp 525–532
53. Peters F, Tun T, Yu Y, Nuseibeh B (2018) Text filtering and ranking for security bug report prediction. IEEE Transactions on Software Engineering pp Early–Access
54. Science N, Council T (2016) US Federal Cybersecurity Research and Development Strategic Plan. `https://www.nitrd.gov/cybersecurity/`
55. Storn R, Price K (1997) Differential evolution–a simple and efficient heuristic for global optimization over continuous spaces. Journal of global optimization 11(4):341–359
56. Sun C, Lo D, Khoo SC, Jiang J (2011) Towards more accurate retrieval of duplicate bug reports. In: Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering, IEEE Computer Society, pp 253–262
57. Sun Y, Wong AK, Kamel MS (2009) Classification of imbalanced data: A review. International Journal of Pattern Recognition and Artificial Intelligence 23(04):687–719
58. Tantithamthavorn C, McIntosh S, Hassan AE, Matsumoto K (2016) Automated parameter optimization of classification techniques for defect prediction models. In: Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on, IEEE, pp 321–332
59. Tantithamthavorn C, Hassan AE, Matsumoto K (2018) The impact of class rebalancing techniques on the performance and interpretation of defect prediction models. IEEE Transactions on Software Engineering
60. Tian Y, Lo D, Sun C (2012) Information retrieval based nearest neighbor classification for fine-grained bug severity prediction. In: 2012 19th Working Conference on Reverse Engineering, IEEE, pp 215–224
61. Tian Y, Lo D, Xia X, Sun C (2015) Automated prediction of bug report priority using multi-factor analysis. Empirical Software Engineering 20(5):1354–1383
62. Van Aken D, Pavlo A, Gordon GJ, Zhang B (2017) Automatic database management system tuning through large-scale machine learning. In: Proceedings of the 2017 ACM International Conference on Management of Data, ACM, pp 1009–1024
63. Vesterstrøm J, Thomsen R (2004) A comparative study of differential evolution, particle swarm optimization, and evolutionary algorithms on numerical benchmark problems. In: Congress on Evolutionary Computation, IEEE
64. Walden J, Stuckman J, Scandariato R (2014) Predicting vulnerable components: Software metrics vs text mining. In: Software Reliability Engineering (ISSRE), 2014 IEEE 25th International Symposium on, IEEE, pp 23–33
65. Wallace BC, Trikalinos TA, Lau J, Brodley C, Schmid CH (2010) Semi-automated screening of biomedical citations for systematic reviews. BMC bioinformatics 11(1):55

66. Wijayasekara D, Manic M, McQueen M (2014) Vulnerability identification and classification via text mining bug databases. In: IECON 2014-40th Annual Conference of the IEEE Industrial Electronics Society, IEEE, pp 3612–3618
67. Xia X, Lo D, Qiu W, Wang X, Zhou B (2014) Automated configuration bug report prediction using text mining. In: 2014 IEEE 38th Annual Computer Software and Applications Conference (COMPSAC), IEEE, pp 107–116
68. Xia X, Lo D, Shihab E, Wang X (2016) Automated bug report field reassignment and refinement prediction. IEEE Transactions on Reliability 65(3):1094–1113
69. Yang X, Lo D, Huang Q, Xia X, Sun J (2016) Automated identification of high impact bug reports leveraging imbalanced learning strategies. In: 2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC), IEEE, vol 1, pp 227–232
70. Yang XL, Lo D, Xia X, Huang Q, Sun JL (2017) High-impact bug report identification with imbalanced learning strategies. Journal of Computer Science and Technology 32(1):181–198
71. Zaman S, Adams B, Hassan AE (2011) Security versus performance bugs: a case study on firefox. In: Proceedings of the 8th working conference on mining software repositories, ACM, pp 93–102
72. Zhang T, Yang G, Lee B, Chan AT (2015) Predicting severity of bug report by mining bug repository with concept profile. In: Proceedings of the 30th Annual ACM Symposium on Applied Computing, ACM, pp 1553–1558
73. Zhou Y, Tong Y, Gu R, Gall H (2016) Combining text mining and data mining for bug report classification. Journal of Software: Evolution and Process 28(3):150–176