1-2023

How to find actionable static analysis warnings: A case study with FindBugs

Rahul YEDIDA

Hong Jin KANG

Huy TU

Xueqi YANG

David LO
Singapore Management University, davidlo@smu.edu.sg

See next page for additional authors

Citation

YEDIDA, Rahul; KANG, Hong Jin; TU, Huy; YANG, Xueqi; LO, David; and MENZIES, Tim. How to find actionable static analysis warnings: A case study with FindBugs. (2023). *IEEE Transactions on Software Engineering*. 1-17.
Available at: https://ink.library.smu.edu.sg/sis_research/7768

Author
Rahul YEDIDA, Hong Jin KANG, Huy TU, Xueqi YANG, David LO, and Tim MENZIES

2023. DOI: 10.1109/TSE.2023.3234206

How to Find Actionable Static Analysis Warnings: A Case Study with FindBugs

Rahul Yedida, Hong Jin Kang, Huy Tu, Xueqi Yang, David Lo *Fellow, IEEE*, Tim Menzies, *Fellow, IEEE*

Abstract—Automatically generated static code warnings suffer from a large number of false alarms. Hence, developers only take action on a small percent of those warnings. To better predict which static code warnings should *not* be ignored, we suggest that analysts need to look deeper into their algorithms to find choices that better improve the particulars of their specific problem. Specifically, we show here that effective predictors of such warnings can be created by methods that *locally adjust* the decision boundary (between actionable warnings and others). These methods yield a new high water-mark for recognizing actionable static code warnings. For eight open-source Java projects (cassandra, jmeter, commons, lucene-solr, maven, ant, tomcat, derby) we achieve perfect test results on 4/8 datasets and, overall, a median AUC (area under the true negatives, true positives curve) of 92%.

Index Terms—software analytics, static analysis; false alarms; locality, hyperparameter optimization

1 Introduction

Static analysis (SA) tools report errors in source code, without needing to execute that code. This makes them very popular in industry. For example, the FindBugs tool [6] has been downloaded over a million times. Unfortunately, due to the imprecision of static analysis and the different contexts where bugs appear, SA tools often suffer from a large number of false alarms that are deemed to be not actionable [57]. Hence, developers never act on most of their warnings [23, 24, 33]. Previous research work shows that 35% to 91% of SA warnings reported as bugs by SA tools are routinely ignored by developers [24, 23, 32].

Those false alarms produced by SA tools are a significant barrier to the wide-scale adoption of these SA tools [27, 15]. Accordingly, in 2018 [64], 2020 [68] and 2021 [69], Wang et al. and Yang et al. proposed data miners that found the subset of static code warnings that developers found "actionable" (i.e. those that motivate developers to change the code). But in 2022, Kang et al. [29] showed that of the 31,000+ records used by Wang et al. and Yang et al., they could only generate 768 error-free records– which meant all the prior Wang and Yang et al. results need to be revisited.

When Kang et al. tried to build predictors from the 768 good records, they found that their best-performing predictors were not effective (e.g., very low median AUCs of 41%), for details see Table 2. Hence the following remains an open research question:

> **RQ1:** *For detecting actionable static code warnings, what data mining methods should we recommend?*

This paper conjectures that prior work failed to find good predictors because of a *locality problem*. In the learners used in that prior work, the decision boundary between actionable warnings and other was determined by a single *global policy*. In detail, changes to the values of the hyper-parameters of (e.g.) an SVM learner (used by Kang et al. [29]) make *global* changes to the decision boundary (i.e., the global shape of the decision boundary is modified); instead, we argue for *local* changes to the decision boundary. This allows us to make different local adjustments at different regions of the decision boundary to adapt it to the local data.

More specifically, we conjecture that:

> *For complex data,* **global** *treatments perform worse than* **localized** *treatments which adjust different parts of the landscape in different ways.*

To test this, we use *local* treatments to adjust the decision boundary in different ways in different parts of the data.

1) *Boundary engineering*: adjust the decision boundary near our data points;
2) *Label engineering*: control outliers in a local region by using just a small fraction of those local labels;
3) *Instance engineering*: addressing class imbalance in local regions of the data;
4) These treatments are combined with *parameter engineering* to control how we build models.

We call this combination of treatments GHOST2 (GHOST2 extends GHOST [70] which just used one of these treatments). When researchers propose an intricate combination of ideas, it is prudent to ask several questions:

> **RQ2:** *Does GHOST2's combination of* instance, label, boundary *and* parameter *engineering, reduce the complexity of the decision boundary?*

Later in this paper, we will show evidence that our proposed methods simplifies the "error landscape" of a data set (a concept which we will discuss, in detail in §4).

- *R. Yedida, X. Yang and T. Menzies are with the Department of Computer Science, North Carolina State University, Raleigh, USA. E-mail: ryedida@ncsu.edu, xyang37@ncsu.edu, timm@ieee.org*
- *H.J. Kang and D. Lo are with the School of Computing and Information Systems, Singapore Management University, Singapore. E-mail: hjkang.2018@smu.edu.sg, davidlo@smu.edu.sg*
- *H. Tu is with Meta Platforms, Inc. E-mail: huyqtu7@gmail.com*

> **RQ3:** *Does GHOST2's use of* instance, label, boundary *and* parameter *treatments improve predictions?*

Using data from Kang et al. (768 records from eight open-source Java projects), we show that GHOST2 was able to generate excellent predictors for actionable static code warnings.

> **RQ4:** *Are all parts of GHOST2 necessary; i.e. would something simpler also achieve the overall goal?*

To answer **RQ4**, this paper reports an *ablation study* that removes one treatment at a time from our four recommended treatments. For the purposes of recognizing and avoiding static code analysis false alarms, it will be shown that, ignoring any part of our proposed solution leads to worse predictors. Hence, while we do not know if changes to our design might lead to *better* predictors, the ablations study does show that removing anything from that design makes matters *worse*.

This work has six key contributions:

1) As a way to address, in part, the methodological problems raised by Kang et al. GHOST2 makes its conclusions using a small percentage of the raw data (10%). That is, to address the issues of corrupt data found by Kang et al., we say "use less data" and, for the data that is used, "reflect more on that data".
2) A case study of successful open collaboration by software analytics researchers. This paper is joint work between the Yang et al. and Kang et al. teams from the United States and Singapore. By recognizing a shared problem, then sharing data and tools, in eight weeks these two teams produced a new state-of-the-art result that improves on all of the past papers by these two teams (within this research arena). This illustrates the value of open and collaborative science, where groups with different initial findings come together to help each other in improving the state-of-the-art for the benefit of science and industry.
3) Motivation for changing the way we train software analytics newcomers. It is insufficient to just reflect on the different properties of off-the-shelf learners. Analysts may need to be skilled in boundary, label, parameter and instance engineering.
4) GHOST2's design, implementation, and evaluation.
5) A new high-water mark in software analytics for learning actionable static code warnings.
6) A reproduction package that other researchers can use to repeat/refute/improve on our results[1].

The rest of this paper is structured as follows. The next section offers some background notes. §3 discusses the locality problem for complex data sets and §4 offers details on our treatments. §5 describes our experimental methods after which, in §6, we show that GHOST2 outperforms (by a large margin) prior results from Kang et al. We discuss threats to validity for our study in §7, before a discussion in §8 and concluding in §9.

Before all that, we digress to make two important points.

- A learned model must be tested on the kinds of data expected in practice. Hence, any treatments to the data (e.g. via instance, label, boundary engineering) *are restricted to the training data, and do* **not** *affect the test data.*
- There are many ways we could define "actionable" warnings. We use the definition that is consistent with past work [68, 69, 64, 29]. However, it can be argued that "actionable" refers to someone taking action (as demonstrated by an open issue being closed in a future commit, for example). Our definition then is a subset of the actionable ones according to this definition (since something could be actionable and some programmers choose to take no action). Another way to say that is that, by our definition, something is *definitely* actionable in the sense that in the historical record there is a clear record that some action was taken on it. Perhaps it might be more precise to call these warnings "the one that programmers choose to actually notice"– which is what we take to be the essence of the prior work in this area [68, 69, 64, 29].

2 Background

2.1 Static Code Analysis

Automatic static analysis (SA) tools, such as Findbugs are tools for detecting bugs in source code, without having to execute that code. As they can find real bugs at low cost [55, 21], they have been adopted in open source projects and in industry [6, 52, 9, 72, 47, 62]. For example, Zheng et al. [74] discuss how a majority of the static analysis warnings were generated from a few patterns which lead to security vulnerabilities. Their study was based on defect data from over 3 million lines of C/C++ code.

However, as they do not guarantee that all warnings are real bugs, these tools produce false alarms. The large number of false alarms produced is a barrier to adoption [27, 15, 56, 45]; it is easy to imagine how developers will be frustrated by using tools that require them to inspect numerous false alarms before finding a real bug. While this problem was raised by Johnson et al. [27] nearly ten years ago, a recent paper assessing the usability of static analysis tools [45] confirmed that the challenge of too many false alarms remains an open problem. While false alarms include spurious warnings caused by the over-approximation of possible program behaviors during program analysis, false alarms also refer to warnings that developers do not act on. For example, developers may not think that the warning represents a bug (e.g. due to "style" warnings that developers perceive to be of little benefit) or may not wish to modify obsolete code.

The problem of addressing false alarms from static analysis tools has been widely studied [30, 44, 28, 31, 30, 46, 18]. There have been many recent attempts to address the problem. Some researchers have proposed new SA tools that use more sophisticated, but costly, static analysis techniques (e.g. Infer [12], NullAway [7]). Despite their improvements, these tools still produce many false alarms [57]. Another approach [46] involves the refinement of bug detection rules to avoid false positives, but requires manual analysis and design of the rules. Other attempts to prune false alarms include the use of test case generation to validate

[1] https://github.com/yrahul3910/static-code-warnings/

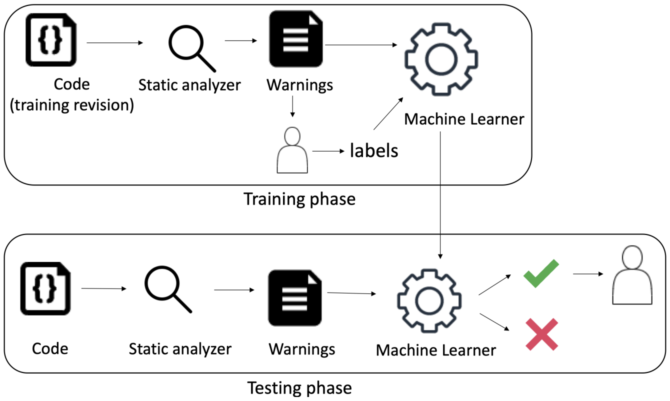


Fig. 1: To detect actionable warnings, a learner is trained on warnings from a training revision. Each warning is annotated with a label. When deployed on the latest revision, only warnings classified as actionable warnings by the machine learner are presented to the developers.

the presence of a bug at the source code location indicated by the warning [28]. As generating test cases is expensive, these techniques may face issues when scaling up to larger projects, limiting their practicality.

2.2 Early Results: Wang et al., 2018

By framing the problem as a binary classification problem, machine learning techniques can identify actionable warnings (allowing us to prune false alarms) [22, 23, 40, 51, 64, 68, 69]. These techniques use features extracted from code analysis and metrics computed over the code and warning's history in the project.

Figure 1 illustrates this process. A static analyzer is ran on a training revision and the warnings produced are labelled. When applied to the latest revision, only warnings classified as actionable warnings by the machine learner are presented to the developers.

To assess proposed machine learners, datasets of warnings produced by Findbugs have been created. As the ground-truth label of each warning is not known, a heuristic was applied to infer them. This heuristic compares the warnings reported at a particular revision of the project against a revision set in the future. If a warning is no longer present, but the file is still present, then the heuristic determines that the warning was fixed. As such, the warning is actionable. Otherwise, if the warning is still present, then the warning is a false alarm.

Wang et al. [64] ran a systematic literature review to collect and analyze 100+ features proposed in the literature, categorizing them into 8 categories. To remove ineffective features, they performed a greedy backward selection algorithm. From the features, they identified a set of features that offered effective performance.

2.3 Further Result: Yang et al., 2021

Yang et al. [68] further analyzed the features using the data collected by Wang et al. [64]. They found that all machine learning techniques were effective and performed similarly to one another. Their analysis revealed that the intrinsic dimensionality of the problem was low; the features used in the experiments were more verbose than the actual attributes required for classifying actionable warnings. This motivates the use of simpler machine learners over more complex learners. From their analysis, SVMs were recommended for use in this problem, as they were both effective and can be trained at a low cost. In contrast, deep learners were effective but more costly to train.

TABLE 1: Evaluation metrics based on TP (true positives); TN (true negatives); TP (true positives) and FP (false positives)

Evaluation Metric	Description
Precision	$\frac{TP}{TP+FP}$
AUC	area under the receiver operating characteristics curve (the true positive rate against the false positive rate)
False alarm rate	$\frac{FP}{FP+TN}$
Recall	$\frac{TP}{TP+FN}$

For each project in their experiments, one revision (training revision) was selected for extracting warnings for training the learner, and another revision (testing revision) set chronologically in the future of the training revision is selected for extracting warnings for evaluating the learner. This simulates a realistic usage scenario of the tool, where the learner is trained using past data before developers apply it to another revision of the source code.

2.4 Issues in Prior Results: Kang et al., 2022

Subsequently, Kang et al. [29] replicated the Yang et al. [68] study to find subtle methodological issues in the Wang et al. data [64] which led to overoptimistic results.

Firstly, Kang et al. found data leakage where the information regarding the warning in the future, used to determine the ground-truth labels, leaked into several features. Five features (warning context in method, file, for warning type, defect likelihood, discretization of defect likelihood) measure the ratio of actionable warnings within a subset of warnings (e.g. warnings in a method, file, of a warning type). To determine if a warning is actionable, the ground-truth label was used to compute these features, leading to data leakage. Kang et al. reimplemented the features such that they are computed using only historical information, without reference to the ground truth determined from the future state of the projects. As only the features were reimplemented, the total number of training and testing instances remained unchanged.

Secondly, they found many warnings appearing in both the training and testing dataset. As some warnings remain in the project at the time of both the training and testing dataset, the model has access to the ground-truth label for the warning at training time. Kang et al. addressed this issue by removing warnings that were already present during the training revision from the testing dataset, ensuring that the learner does not see the same warning in both datasets. After removing these warnings, the number of warnings in the testing revision decreased from 15,695 to 2,615.

Next, Kang et al. analyzed the warning oracle, based on the heuristic comparing warnings at one revision to another

TABLE 2: The predictors reported by Kang et al. did not perform well on the repaired data. In this table, *lower* false alarms are better while *higher* precisions, AUC, and recall are *better*.

Dataset	Precision	AUC	False alarm rate	Recall
cassandra	0.67	0.33	0.25	0.67
commons	0.67	0.52	0.57	0.62
lucene-solr	0.56	0.70	0.36	0.71
maven	0.52	0.41	0.19	0.32
jmeter	0.50	0.36	0.14	0.17
tomcat	0.52	0.41	0.19	0.32
derby	0.20	0.64	0.12	0.08
ant	0.00	0.00	0.00	0.00

revision in the future, used to automatically produce labels for the warnings in the dataset. After manual labelling of the actionable warnings, Kang et al. found that only 47% of warnings automatically labelled actionable were considered by the human annotators to be actionable. This indicates that the heuristic employed as the warning oracle is not sufficiently reliable for automatically labelling the dataset.

Kang et al. manually labelled 1,357 warnings. After filtering out duplicates and uncertain labels, a dataset of 768 warnings remained. On this dataset, Kang et al. again applied off-the-shelf SVM models, assessing them with the evaluation metrics listed in Table 1.

For their reasoning, Kang et al. used the learners recommended by prior work; i.e. radial bias SVMs. The results of the SVM are shown in Table 2. Those results are hardly impressive:

- Median precisions barely more than 50%;
- Very low median AUCs of 41%;
- Extremely low median recalls of 32%.

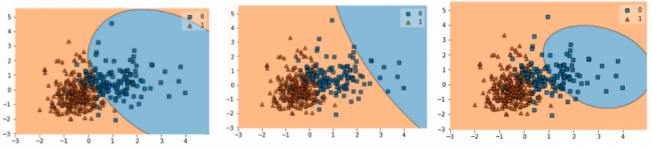
That is to say, while Kang et al. were certainly correct in their criticisms of the data used in prior work, based on their paper, it is still an open issue about how to generate good predictors for static code false alarms.

3 Rethinking the Problem

This section suggests that detecting actionable static code warnings is a "bumpy" problem (defined below) and that such problems can not be understood by learners that use simplistic boundaries between classes.

The core task of any classification problem is the creation of a hyperspace boundary that let us isolate what is most desired or most interesting. Different learners build their boundaries in different ways:

- Simple decision tree learners can only build straight-line boundaries.
- Neural networks can produce very complex and convoluted boundaries.
- And internal to Kang et al.'s support vector machine was a "radial basis function" that allowed those algorithms to build circular hyperspace boundaries.



$C = 0.1$
$\gamma = 0.1$
$acc = 90\%$

$C = 0.1$
$\gamma = 0.08$
$acc = 64\%$

$C = 0.02$
$\gamma = 0.1$
$acc = 81\%$

Fig. 2: The C parameter of a radial basis function alters the shape of the hyperspace boundary. *Acc* is accuracy which is the ratio of true positives plus true negatives divided by a SVM making predictions across that boundary. Example from [36].

Boundaries can be changed by adjusting the parameters that control the learner. For example, in Kang et al.'s radial basis functions, the C regularization parameter is used to set the tolerance of the model to (some) classifications. By adjusting C, an analyst can change the generalization error; i.e. the error when the model is applied to as-yet-unseen test data.

Figure 2 shows how changes to C can alter the decision boundary between some red examples and blue examples. Note that each setting to C changes the accuracy of the predictor; i.e. for good predictions, it is important to fit the shape of the decision boundary to the shape of the data.

(Technical aside: while this example was based on SVM technology, the same line of argument applies to any other classifier; i.e. changing the control parameters of the learner also changes the hyperspace boundary found by that learner and, hence, the predictive prowess of that learner.)

We have tried applying hyperparameter optimization to C in a failed attempt to improve that performance (see the C1 results of Table 8). From that failed experiment, we conclude that however C works for radial bias functions, they do not work well enough to fix the unimpressive predictive performances – see Table 2.

Why do radial bias SVMs fail in this domain? Our conjecture is that the hyperspace boundary dividing the static code examples (into false positives and others) is so "bumpy"[2] that the kinds of shape changes seen in Figure 2 can never adequately model those examples.

To test that conjecture, we first checked for "bumpiness" using a technique from Li et al. [38]. That technique visualizes the "error landspace" (i.e. how fast small changes in the independent variables altered the error estimation). For our TOMCAT data, Li et al.'s methods resulted in Figure 3. There, we see a "bumpy" landscape with several multiple local minima.

Having confirmed that our data is "bumpy", our second step was to look for ways to reduce that bumpiness. Initially, we attempted to use neural nets since that kind of learner is meant to be able to handle complex hyperspace boundaries [67]. As discussed in §6, that attempt failed even after trying several different architectures such as feedforward networks, CNN, and CodeBERT [50, 21, 63] (with and without tuning learner control parameters).

Since standard neural net technology failed, we tried several manipulation techniques for the training process, described in the next section.

4 Treatments

This section discusses a framework that holds operators for treating the data in order to adjust the decision boundary

[2] "Bumpy" data contain complexities such as many local minima, saddle points, very flat regions, and/or widely varying curvatures. For example, see Figure 3.

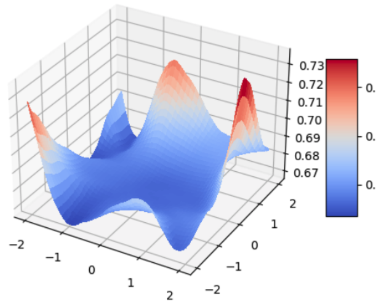


Fig. 3: Error landscape in the TOMCAT data before applying the methods of this paper. In the plot, the larger the vertical axes, the greater the loss value. Later in this paper, we will show this plot again, after it has been smoothed via the methods of §4 (see Figure 4 and Table 11).

(in different ways for different parts of the data). For the purposes of illustration and experimentation, we offer operational examples for each part of the framework:

- SMOTE for instance engineering;
- SMOOTH for label engineering;
- GHOST for boundary engineering;
- DODGE for parameter engineering.

Before presenting those parts we note here that the framework is more than just those four treatments. As SE research matures, we foresee that our framework will become a workbench within which researchers replace some/all of these treatments with more advanced options.

That said, we have some evidence that SMOTE, SMOOTH, GHOST, DODGE are useful:

- The ablation study of §5.4 shows that removing any one of these treatments leads to worse performance.
- All these treatments are very fast: sub-linear time for SMOTE and SMOOTH, linear time for GHOST, and DODGE is known to be orders of magnitude faster than other hyperparameter optimizers [3].

4.1 Instance Engineering (via SMOTEing)

To remove the "bumpiness" in data like Figure 3, we need to pull and push the decision boundaries between different classes into a smoother shape. But also, unlike simplistic C tuning available in radial SVMs, we want that process to perform differently in different parts of the data.

One way to adjust the decision boundary in different parts of the data is to add (or delete) artificial examples around each example X. This builds a little "hill" (or valley) in the local region. As a result, in that local region, it becomes more (or less) certain that all predictions which reach the same conclusion as X. In effect, adding/deleting examples pushes the decision boundary away (or, in the case of deletions, pulls it closer). SMOTE [14] is one instance engineering technique that:

- Finds five nearest neighbors to X with the same label;
- Selects one at random;
- Creates a new example R, with the same label as X at some random point between X and R.

4.2 Label Engineering (via SMOOTHing)

SMOTE has seen much success in recent SE papers as a way to improve predication efficacy [2]. But this technique makes a *linearity* assumption that all the data around X is correctly labelled (in our case, as examples of actionable or unactionable static code warnings). This may not be true. Cordeiro and Carneiro [17] and recent SE researchers [58, 59, 61, 71] note that noisy labels can occur when human annotators are present [42] or those humans have divergent opinions about the labels [8, 41]. Although our labels were re-checked by the authors of Kang et al. [29], our ablation study (below) reports that it is best to apply some mitigation method for poorly labelled examples. For example, in this work we applied the following SMOOTHing operator where data is assigned labels using multiple near neighbors. This has the effect of removing outliers in the data. At least within the domain of static analysis warning data, SMOOTHing, by reducing noise in the training labels, SMOOTH compensates for misclassification accuracy.

Our SMOOTH operator works as follows:

- Given n training samples (and therefore, n labels), we keep $\sqrt{n}$ at random and discard the rest.
- Next, we use a KD-tree to recursively sub-divide the remaining data into leaf clusters of $\sqrt[4]{n}$ nearest neighbors. Within each leaf, all examples are assigned a label that is the mode of the labels in that leaf.

One interesting and beneficial side-effect of SMOOTHing is that we make conclusions on our test data using just 10% of the training data. By reducing the labelling required to make conclusions, SMOOTHing offers a way to help future studies avoid the problems reported by Kang et al. [29]:

- One of the major finding of the Kang et al. study was that earlier work [68] had mislabelled much of its data. From that study, we assert that it is important for analysts to spend more time checking their labels. We note that there are many other ways to reduce the labels required for supervised learning.
- SMOOTHing reduces the effort required for that checking process (by a factor of ten).

As an aside, we note that SMOOTHing belongs to a class of algorithms called *semi-supervised learning* [58, 59] that try to make conclusions using as few labels as possible. The literature on semi-supervised learning is voluminous [11, 13, 34, 73, 75] and so, in the theory, there could be many other better ways to perform label engineering. This would be a productive area for future research. But for now, the ablation study (reported below) shows that SMOOTHing is useful (since removing it degrades predictive performance).

4.3 Boundary Engineering (via GHOSTing)

As defined above, instance and label engineering do not reflect on the quality of data in the local region.

To counter that, this study employs a boundary method called "GHOSTing", recently developed and applied to software defect prediction by Yedida and Menzies [70]. Boundary engineering is different to label and instance engineering since it adjusts the frequency of different classes in the local region (while the above typically end up repeating the same label for a particular locality). Hence, in that region, it changes the decision boundary.

GHOSTing addresses class imbalance issues in the data. When an example with one label is surrounded by too many examples of another label, then the signal associated with example can be drowned out by its neighbors To fix this, for a two-class dataset D with class c_0 being the minority, GHOSTing oversamples the class by adding concentric boxes of points around each minority sample. The number of concentric boxes is directly related to the class imbalance: higher the imbalance, more the number of boxes. Specifically, if n is the fraction of samples in the minority class, then $\lfloor \log_2(1/n) \rfloor$ boxes are added. While the trivial effect of this is to oversample the class (indeed, as pointed out by Yedida and Menzies [70], this *reverses* the class imbalance), we note that the algorithm effectively builds a wall of points around minority samples. This pushes the decision boundary away from the training samples, which is preferred since a test sample that is close to a training sample has a lesser chance of being misclassified due to the decision boundary being in between them.

Our pre-experimental intuition was that boundary engineering would replace the need to use instance engineering. However, as shown by our ablation study, for recognizing actionable static code warnings, we needed both tools. On reflection, we realized both may be necessary since while (a) boundary engineering can help make local adjustments to the decision boundary, it can (b) only work in regions where samples *exist*; instance engineering can help fill in gaps in sparser regions of the dataset.

4.4 Parameter Engineering (via DODGEing)

We noted above that different learners generate different hyperspace boundaries (e.g. decision learners generate straight-line borders while SVMs with radial bias functions generate circular borders). Further, once a learner is selected, then as seen in Figure 3, it is possible to further adjust a border by altering the control parameters of that learner (e.g. see Figure 2). We call this adjustment *parameter engineering*.

Parameter engineering is like a scientist probing some phenomenon. After the data is divided into training and some separate test cases, parameter engineering algorithms conduct experiments on the training data looking for parameter settings that improve the performance of a model learned and assessed on the training data. Once some conclusions are reached about what parameters are best, then these are applied to the test data. Importantly, the parameter engineering should only use the training data for its investigations (since otherwise, that would be a threat to the external validity of the conclusions).

TABLE 3: List of hyper-parameters tuned in our study. CodeBERT is not shown in that table since, as mentioned in the text, this analysis lacked the resources required to tune such a large model.

Learner	Hyper-parameter	Range
Feedforward network	#layers	$[2, 6]$
	#units per layer	$[3, 20]$
Logistic regression	Penalty	$\{l_1, l_2\}$
	C	$\{0.1, 1, 10, 100\}$
Random forest	Criterion	{ gini, entropy }
	n_estimators	$[10, 100]$
Decision Tree	Criterion	{ gini, entropy }
	Splitter	{ best, random }
SVM	C	$\{0.1, 1, 10, 100\}$
	Kernel	{sigmoid, rbf, polynomial }
CNN	#convolutional blocks	[1, 4]
	#convolutional filters	{4, 8, 16, 32, 64}
	Dropout probability	(0.05, 0.5)
	Kernel size	{16, 32, 64}

TABLE 4: Neural net architectures used in this study.

Feedforward networks These are artificial neural networks, comprising an acyclic graph of nodes that process input and produce an output. These dates back to the 1980s, and the parameters of these models are learned via backpropagation [50]. These networks have $\mathcal{O}(10^3) - \mathcal{O}(10^4)$ parameters. For these networks, we used the ReLU (rectified linear activation) function ($f(x) = \max(0, x)$). This is a piecewise linear function that will output the input directly if it is positive, otherwise, it will output zero.

A *convolutional neural net* (CNN) is a structured neural net where the first several layers are sparsely connected in order to process information (usually visual). CNN is an example of an *deep learner* and are much larger than feedforward networks (these may span $\mathcal{O}(10^5) - \mathcal{O}(10^7)$ parameters). Optimizing an CNN is a very complex task (so many parameters) so following advice from the literature [26, 54], we used the following architecture. Our CNNs had multiple "convolutional blocks" defined as follows:

1) ReLU activation
2) Conv (with "same" padding)
3) Batch norm [26]
4) Dropout [54]

We note that this style of building convolutional networks, by building multiple "convolutional blocks" is very popular in the CNN literature [35, 37]. Our specific design of the convolutional blocks was based on a highly voted answer on Stack Overflow [3].

Note that with that architecture there is still room to adjust the ordering of the blocks– which is what we adjust when we tune our CNNs.

CodeBERT [20] is a *transformer-based* model that been pre-trained model using millions of examples from contemporary programming languages such as Python, Java, JavaScript, PHP, Ruby, and Go. Such transformer models are those based on the "self-attention" mechanism proposed by Vaswani et al. [63]. CodeBERT is even large than CNN and can contain $\mathcal{O}(10^8) - \mathcal{O}(10^9)$ parameters. One advantage of such large models is that can learn intricacies that are missed by smaller models.

Parameter engineering executes within the space of control parameters of selected learners. These learners have the internal parameter space shown in Table 3. We selected this range of learners using the following rationale:

- In order to compare our new results to prior work by Kang et al. [29], we use the Kang et al. *SVMs* with the radial basis kernel and balanced class weights.
- In order to compare our work to Kang et al. [68], we used a range of *traditional learners* (logistic regression, random forests, and single decision tree learners);
- Also, we explored the various *neural net algorithms* shown in Table 4 since these algorithms have a reputation of being able to handle complex decision boundaries [67]. In this textbook on *Empirical Methods for AI*, Cohen [16] advises that supposedly more

complex solutions should be compared to a range of alternatives, including very simple methods. Accordingly, for neural nets, we used (a) feedforward networks from the 1980s; (b) the CNN deep learner used in much of contemporary SE analytics; and (c) the state-of-the-art CodeBERT model.

There are many algorithms available for automatically tuning these learning control parameters. As recommended by a prior study [4], we use Agrawal et al.'s DODGE algorithm [3]. DODGE is based on early work by Deb et al. in 2005 that proposed a "$\mathcal{E}$-domination rule" [19]; i.e.

> *If one setting to an optimizer yield results within $\mathcal{E}$ or another, then declare the region $\pm$ $\mathcal{E}$ as "tabu" and search elsewhere.*

A surprising result from Agrawal et al.'s research was that $\mathcal{E}$ can be very large. Agrawal et al. noted that if learners were run 10 times, each time using 90% of the training data (selected at random), then they often exhibited a standard deviation of 0.05 (or more) in their performance scores. Assuming that performance differences less than $\pm 2\mu$, are statistically insignificantly different, then Agrawal reasoned that $\mathcal{E}$ could be as large as $4 * .05 = 0.2$. This is an important point. Suppose we are trying to optimize for two goals (e.g. recall and false alarm). Since those measures have the range zero to one, then $\mathcal{E} = 0.2$ divides the output space of those two goals divides into just a $5 \times 5 = 25$ regions. Hence, in theory, DODGE could find good optimizations after just a few dozen random samples to the space of possible configurations.

When this theoretical prediction was checked experimentally of SE data, Agrawal [4] found that DODGE with $\mathcal{E} = 0.2$ defeated traditional single-point cross-over genetic algorithms as well as state-of-the-art optimizers (e.g. Bergstra and Bengio's HYPEROPT algorithm [10][4]). Accordingly, this study used DODGE [4, 60] for its parameter engineering.

Our pre-experimental intuition was that DODGEing would be fast enough to tune even the largest neural net model. This turned out not to be the case. The resources required to adjust the CodeBERT model are so large that, for this study, we had to use the "off-the-shelf" CodeBERT.

In this study, whenever we say we perform hyper-parameter optimization (or equivalently in this paper, parameter engineering), we mean we run DODGE for 30 iterations to maximize the difference between recall and false alarm rate. This simultaneously aims to maximize recall while minimizing false alarm rate, prioritizing both goals equally.

5 Experimental Methods

5.1 Data

This paper tested the efficacy of instance, label, boundary and parameter engineering using the revised and repaired data from Kang et al. paper [29].

Recall that Kang et al. manually labelled warnings from the same projects studied by Yang et al. [68] to assess the level of agreement between human annotators and the heuristic. The manual labelling was performed by two human annotators. One annotator is an undergraduate student, while the other is a graduate student with two years of industrial experience. When the annotators disagreed on the label of a warning, they discussed the disagreement to reach a consensus. While they achieved a high level of agreement, achieving a Cohen's Kappa of above 0.8, manual labelling is costly, requiring human analysis of both the source code and the commit history of the code. That said, considering the subsequent evolution of the source code allows the annotators to analyze each warning with a greater amount of context. These labels are essential since it removed closed warnings which are not actionable (e.g., the warnings may have been removed for reasons unrelated to the Findbugs warning).

Two other filters employed by Kang et al. where:

- Unconfirmed actionable warnings were removed;
- False alarms were randomly sampled to ensure a balance of labels (40% of the data were actionable) consistent with the rest of the experiments.

One of the complaints of the Kang et al. paper [29] against earlier work [68] was that, for data that comes with some time stamp, it is inappropriate to use future data to predict past labels. To avoid that problem, in this study, we sorted the Kang et al. data by time stamps, then used 80% of the past data to predict the remaining 20% future labels.

The Kang et al. data comes from eight projects and we analyzed each project's data separately. The 80:20 train:test splits resulted in the train:test sets shown in Table 6 (exception: for MAVEN, we split 50:50, since there are only 4 samples in total).

In this data, the dependent variables provide information about the warning, file, and source code that the warning is reported on. They are further categorized based on how they are obtained, e.g. through the history of the code. Finally, the variables fall into eight broad categories. They are summarized in Table 5. Prior studies on static analysis warnings have worked with datasets with a wide range of actionable warnings. For example, Heckman and Williams [23] experimented on 2 datasets, one of which had a percentage of actionable warnings of 89%. Imtiaz et al. [25] experimented on several datasets, one of which had a percentage of 49.5% of actionable warnings. Therefore, the percentage of actionable warnings in our experiments is consistent with some prior studies.

Pre-experimentally, we were concerned that learning from the smaller data sets of Table 6 would complicate our ability to make any conclusions from this data. That is, we needed to know:

> **RQ5:** *Are larger training sets necessary (for the task of recognizing actionable static code warnings)?*

This turned out not to be a critical issue. As shown below, the performance patterns in our experiments were stable across all the six smaller data sets used in this study.

Technical aside: In other papers, we have run repeated trials with multiple 80:20 splits for training:test data. This was not here since some of our data sets are too small (see the first few rows of Table 6) that any reduction in the

[4] At the time of this writing (April 2022), the paper proposing HYPEROPT has 7,557 citations in Google Scholar.

TABLE 5: The dependent variables used in this study. These features were identified in prior work [64, 68, 69]. The "Golden Features" are in bold.

Feature type	Feature	Description
Warning combination	size context for warning type, method size context in file, package **warning context in method, file**, package **warning context for warning type** fix, non-fix change removal rate. **defect likelihood for warning pattern** variance of likelihood defect likelihood for warning type **discretization of defect likelihood** **average lifetime for warning type**.	Features related to the warnings and other information (e.g. total number of warnings of each type, percentage of actionable warnings)
Code characteristics	method, file, package size. comment length **comment-code ratio** **method depth** **file depth** method callers, callees **# methods in file**, package # classes in file **# classes in package** indentation complexity	Features related to the source file where the warning is reported (e.g., the number of methods in the file)
Warning characteristics	**warning pattern** **warning type** **warning priority** warning rank, warnings in method, file **package**	Features related to the warning (e.g., its priority)
File history	latest file, package modification file, package staleness **file age, creation** deletion revision **developers**	Features related to the file where where the warning was reported (e.g., creation date of the file)
Code analysis	call name, class, **parameter signature** **method visibility** return type new type, new concrete type operator field access class, field catch field name, type, visibility, is static/final return type is static/ final/ abstract/ protected class visibility,is interface	Features obtained through program analysis related to the source code where the warning was reported (e.g., if the method is public, protected, or private)
Code history	added, changed, deleted, growth, total, percentage of LOC in file (past 3 months) LOC **added** , changed, deleted, growth total, percentage in file (last 25 revisions) **added**, changed, deleted, growth, total, percentage of LOC in package (past 3 months) added, changed, deleted, growth, total, percentage of LOC in package (last 25 revisions)	features related to the revision history of the source code where the warning was reported (e.g., number of lines of code added in past 3 months)
Warning history	**warning lifetime by revision**, by time warning modifications, open revision	features related to the history of the warning in the project
File Characteristics	file type, name package name	features related to the metadata of the file

training set size might disadvantage the learning process. Hence, the external validity claims of this paper come from patterns seen in eight different software projects.

5.2 Models

In this section, we discuss the models used by our approach. Briefly, we use feedforward networks, which are neural networks where each layer is fully connected. We do not use more modern approaches such as batch normalization [26] or dropout [54]; as pointed out in the original GHOST paper [70], the work of Montufar et al. [43] shows that by setting the number of hidden layers to at least the number of inputs, the lower bound of the number of piecewise linear regions of the decision boundary is non-zero, and therefore (more likely to be) non-trivial.

Although there are practical challenges when optimizing such basic models that are overcome by recent advances [53, 38], we use the approach of GHOST, instead relying on hyper-parameter optimization. The defect prediction study of the original paper showed that hyper-parameter optimization along with weighted loss functions and "fuzzy sampling" (in this paper, we refer to the combination of

TABLE 6: Summary of the data distribution

Project	# train	# labels	imbalance%	# test
maven	2	1	33	1
cassandra	9	4	38	4
jmeter	10	4	43	4
commons	12	5	59	5
lucene-solr	19	5	38	6
ant	22	6	36	7
tomcat	134	13	41	37
derby	346	20	37	92
total	554	58		156

these techniques as "GHOST" for simplicity) suffices to achieve state-of-the-art results–this paper shows that result extends to the task of demarcating actionable static code warnings as well.

5.3 LIME

It is worth looking inside the models at this point instead of treating them as black boxes. In an effort to understand the most important variables, we used LIME [49], a standard explanation algorithm. However, this pursuit was unsuccessful for several reasons:

- Some of our data sets are very small and, for such small data sets, LIME reports that all features are unimportant.
- All our data sets have different attributes (e,g, who made a comment, what part of the code they commenting on, etc). So even though we could generate results with high performance values, we could not find common patterns across the different data sets.

We conjecture that static code warning classification is a hard problem with a bumpy decision boundary that cannot be characterized by (e.g.) LIME's simple linear models. Rather, we may need some hyper-dimensional inferred description, which is why we see low prediction scores in Table 2 and much higher scores when we apply neural technology.

5.4 Experimental Rig

This study explores:

- $N = 4$ pre-processors (boundary, label, parameter, instance) that could be mixed in $2^4 = 16$ ways.
- Six traditional learners: logistic regression, decision trees, random forests, SVMs (with 3 basis functions);
- Three neural net architectures: CNN, CodeBERT, feedforward networks;

To clarify the reporting of these $16\times(6+3) = 144$ treatments, we made the following decisions. Firstly, when reporting the results of the traditional learner, just show the results of the one that beat the other traditional learners (which, in our case, was typically random forest or logistic regression).

Secondly, we do not apply pre-processing or parameter engineering on CodeBERT. This decision was required, for pragmatic reasons. Due to the computational cost of training that model, we could only run off-the-shelf CodeBERT.

Thirdly, rather than explore all 16 combinations of use/avoid different pre-processing, we ran the *ablation study* recommended in Cohen's *Empirical Methods for AI* textbook [16]. Ablation studies let us explore some combination of N parts can be assessed in time $O(N)$, not $O(2^N)$. Such ablation studies work as follows:

- Commit to a preferred approach, with N parts;
- If removing any part $n_i \in N$ degrades performance, then conclude that all N parts are useful.

With these decisions, instead of having to report on 144 treatments, we need only show the 13 treatments in the ablation study of Table 7. In that table, for treatments that use any of boundary or label or parameter or instance engineering, we apply those treatments in the order recommended by the original GHOST paper [70]. That paper found that it could improve recall by 30% (or more) by multiple rounds of SMOTE + GHOST. As per that advice, A1 executes our pre-processors in the order:

$$smooth \rightarrow smote \rightarrow ghost \rightarrow ghost \rightarrow smote \rightarrow dodge$$

The rationale for this approach is as follows. Learning can be divided into three stages: preprocessing, "in-processing", and post-processing. This paper does not explore post-processing (and that might be a useful direction for future work). The crux of this paper (and the original GHOST paper) is that we should focus more than we currently do on pre- and in-processing. As such, we start the pre-processing with SMOOTH, which has the effect of eliminating outliers in the data. We then balance classes using SMOTE. After that, we applied a recommendation from the GHOST paper, i.e. it is best to run GHOST twice on the data (hence our call to GHOST -> GHOST). It turns out that we should not pass those outputs directly to the learner, since we want a robust decision boundary. Hence, we use another recommendation from the GHOST paper which is

$$ghost \rightarrow ghost \rightarrow smote$$

All that said, it is an open issue if *other* ordering might be *more* useful. In this paper, we mote that our ablation study reports no obvious problem with this ordering. But that is not to say that other orderings might improve our results even further. We leave this matter for future work.

All the treatments labelled "A" (A1,A2,A3,A4,A5) in Table 7, use the order shown above, perhaps (as part of the ablation study) skipping over one or more the steps. We acknowledge that there are many possible ways to order the applications of our treatments, which is a matter we will for future work. For the moment,the ordering shown above seems useful (evidence: see next section).

As to the specifics of the other treatments:

- Treatment A5 is the treatments from the TSE'21 paper that proposed GHOSTing [70].
- Treatment D1 contains the treatments applied in prior papers by Yang et al. [68] and Kang et al. [29].
- Anytime we applied *parameter engineering*, this meant that some automatic algorithm (DODGE) selected the control parameters for the learners (otherwise, we just used the default off-the-shelf settings).
- Anytime we apply *label engineering*, we are only used 10% of the labels in the training data.
- The last line, showing CodeBERT, has no pre-processing or tuning. As said above, CodeBERT is so complex that we must run it "off-the-shelf".

TABLE 7: Design of our ablation study. In the learner choice column, F = feedforward networks, T = traditional learners, C = CNN, B = CodeBERT.

	Engineering decisions						
Treatment	*Boundary*	*Label*	*Learner*	*Parameter*	*Instance*	% Labels	Description
A1	✓	✓	F	✓	✓	10	Our recommended method
A2	✓	✓	F	✓		10	A1 without instance engineering (no SMOTE)
A3	✓	✓	F		✓	10	A1 without hyper-parameter engineering (no DODGE)
A4		✓	F	✓	✓	10	A1 without boundary engineering (no GHOST)
A5	✓		F	✓	✓	100	A1 without label engineering (no SMOOTH). From TSE'21 [70]
A6	✓		T	✓	✓	100	A1 without label engineering, replacing feedforward with traditional learners
A7	✓	✓	T	✓	✓	10	A1 replacing feedforward with traditional learners
B1		✓	T	✓	✓	10	A1 without boundary engineering, replacing feedforward with traditional learners
B2		✓	C	✓	✓	10	A1 without boundary engineering, replacing feedforward with CNN
C1			T	✓	✓	100	A1 without boundary engineering or label engineering, replacing feedforward with traditional learners
C2			C	✓	✓	100	A1 without boundary engineering or label engineering, replacing feedforward with CNN
D1			T		✓	100	Setup used by the Yang et al. [68] and Kang et al. [29] studies.
CodeBERT			B			100	CodeBERT without modifications

6 Results

The results of the Table 7 treatments are shown in Table 8 (and another brief summary is offered in Table 10). These results are somewhat extensive so, by way of an overview, we offer the following summary tool. The cells shown in pink are those that are worse than the A1 results (and A1 is our recommended GHOST2 method). Looking over those pink cells we can see that across our data sets and across our different measures, our recommend method (A1) is rarely outperformed by anything else.

(Technical aside: looking at the pink cells, it could be said that A5 comes close to A1, but A5 loses a little on recalls). Nevertheless, we have strong reasons for recommending A1 over A5 since, recalling Table 7, A5 requires a labelling for 100% of the data. On the other hand A1, that uses label engineering, achieves its results using 10% of the labels. This is important since, as said in our introduction, one way to address, in part, the methodological problems raised by Kang et al. GHOST2 makes its conclusions using a small percentage of the raw data (10%). That is, to address issues of corrupt data found by Kang et al., we say "use less data" and, for the data that is used, "reflect more on that data".

These answer our research questions as follows.

RQ1: For detecting actionable static code warnings, what data mining methods should we recommend?

Regarding feedforward networks versus, say, traditional learners (decision trees, random forests, logistic regression and SVMs), the traditional learners all performed worse than the feedforward networks used in treatment A1 (evidence: compare treatments A1 with A7 which use feedforward or traditional learners, respectively; there are four perfect AUCs for feedforward networks in A1, i.e AUC=100%, but only two for the A7 results).

As to why the 1980s style feedforward networks worked better than newer neural net technology, we note that feedforward networks run so fast than it is easier to extensively tune them. Perhaps (a) faster learning plus (b) more tuning might lead to better results that then non-linear modeling of an off-the-shelf learner. This could be an interesting avenue for future work.

As to the value of *boundary, label, instance* and *parameter* engineering, in the ablation study, removing any of these increased the number of times A1 had larger performance scores. For example, with *boundary engineering*, A1 (that uses boundary engineering) generates more perfect scores (e.g. AUC=100%) than A4 (that does not use it). Also, for recall, A1 always had larger or same scores than n A4 in 6/8 data sets. Similarly, A4 always suffers from a drop in AUC score.

As for *label engineering*, from A1 to A5, specializing our data to just 10% of the labels (in A1) yields nearly the same precisions which using 100% of the data (in A5) in nearly all the AUC results. Moreover, the AUC score for A1 is perfect in 4/8 cases, while for A5, it is rarely the case.

As to *instance engineering*, without it the precision can crash to zero (compare A1 to A2, particularly the smaller data sets) while often leading to lower recalls. The smaller datasets also see a decrease in AUC for A2.

Measured in terms of false alarm, these results strongly recommend *parameter engineering*. Without parameter engineering, some of those treatments could find too many static code warnings and hence suffer from excessive false alarms (evidence: see the A3 false alarm results in nearly every data set). A1 (which used all the treatments of §4) had lower false alarm rates than anything else (evidence: we rarely see the dark blue A1 spike in the false alarm results). The only exception to the observation that "parameter engineering leads to lower false alarm results" are seen in the DERBY data set. That data set turns out to be particularly tricky in that, nearly always, modeling methods that achieved low false alarm rates on that data set also had to be satisfied with much lower recalls.

One final point is that these results do not recommend the use of certain widely used neural network technologies such as CNN or CodeBERT for finding actionable static code

TABLE 8: Results (8 datasets, 4 metrics. Pink shows performance worse than A1 (our recommend method).

Treatment	maven	cassandra	jmeter	commons	lucene-solr	ant	tomcat	derby	median	#cells better than A1
					PRECISION (*better* results are *larger*)					
A1	1	1	1	1	0.8	1	0.79	0.72	1	-
A2	0	0.25	1	0.67	1	1	0.68	0.73	0.71	2
A3	0.5	0.25	0.33	0.2	0.25	0.33	0.33	0.4	0.33	0
A4	1	0.75	1	1	0.75	1	1	0.75	1	2
A5	1	1	1	1	0.8	1	0.72	0.84	1	1
A6	1	1	0.5	0.33	0.67	1	0.85	0.89	0.87	2
A7	1	0.5	0.5	1	1	0	0.55	0.42	0.53	1
B1 (DODGE)	1	1	0	0.5	0	0	0.47	0.59	0.49	0
B2 (CNN)	0.5	0	0	0.6	0	0.29	0.51	0.61	0.4	0
C1 (DODGE)	1	1	1	0.33	0.67	0.67	0.67	0.81	0.74	1
C2 (CNN)	0.5	0.5	0.5	0.6	0.83	0.43	0.4	0.73	0.5	2
D1	0	0.5	0	0.6	0	0	0.39	0.39	0.2	0
CodeBERT	0.5	1	0.8	0.63	0.6	0	0.41	0.25	0.55	0
					AUC: TP vs. TN (*better* results are *larger*)					
A1	1	1	0.83	1	0.75	1	0.68	0.57	0.92	-
A2	0	0.5	0.75	0.83	0.63	0.75	0.6	0.7	0.67	1
A3	0.5	0.5	0.67	0.5	0.38	0.55	0.51	0.51	0.51	0
A4	1	0.5	1	0.75	0.63	0.8	0.54	0.59	0.69	2
A5	1	0.67	1	0.88	0.75	0.9	0.67	0.78	0.83	2
A6	1	1	0.83	0.75	0.88	1	0.85	0.76	0.87	3
A7	1	0.83	0.83	1	0.75	0.5	0.59	0.62	0.79	1
B1 (DODGE)	1	1	0.5	0.88	0.5	0.5	0.58	0.62	0.6	1
B2 (CNN)	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.62	0.5	1
C1 (DODGE)	1	1	1	0.75	0.88	0.9	0.8	0.76	0.89	4
C2 (CNN)	0.5	0.5	0.17	0.5	0.5	0.5	0.63	0.82	0.5	1
D1	0.5	0.17	0.5	0.5	0	0.38	0.48	0.47	0.48	0
CodeBERT	0.5	0.56	0.68	0.53	0.63	0.48	0.44	0.63	0.54	1
					FALSE ALARM RATE (*better* results are *smaller*)					
A1	0	0	0	0	0.5	0	0.29	0.79	0	-
A2	0	1	0	0.33	0	0	0.4	0.38	0.17	2
A3	1	1	0.67	1	0.75	0.4	0.71	0.05	0.73	1
A4	0	1	0	0	0.5	0	0	0.48	0	2
A5	0	0	0	0	0.5	0	0.57	0.41	0	1
A6	0	0	0.33	0.5	0.25	0	0.09	0.03	0.06	3
A7	0	0.33	0.33	0	0	0	0.17	0.44	0.09	3
B1 (DODGE)	0	0	0	0.25	0	0	0.35	0.11	0	2
B2 (CNN)	1	0	0	1	0	1	1	0.25	0.63	2
C1 (DODGE)	0	0	0	0.5	0.25	0.2	0.26	0.06	0.13	3
C2 (CNN)	1	1	1	1	1	1	0.46	0.17	1	1
D1	0	1	0	1	1	0.25	0.77	0.67	0.72	0
CodeBERT	1	0	0.2	1	0.25	0	0.28	0.17	0.23	2
					RECALL (*better* results are *larger*)					
A1	1	1	0.67	1	1	1	0.65	0.94	1	-
A2	0.5	1	0.5	1	0.25	0.5	0.59	0.77	0.54	0
A3	1	1	1	1	0.5	0.5	0.75	0.07	0.88	2
A4	1	1	1	0.5	0.75	0.6	0.09	0.67	0.71	1
A5	1	0.33	1	0.75	1	0.8	0.91	0.97	0.94	3
A6	1	1	1	1	1	1	0.79	0.55	1	2
A7	1	1	1	1	0.5	0	0.36	0.69	0.85	1
B1 (DODGE)	1	1	0	1	0	0	0.5	0.34	0.42	0
B2 (CNN)	1	0	0	1	0	1	1	0.49	0.75	1
C1 (DODGE)	1	1	1	1	1	1	0.86	0.59	1	2
C2 (CNN)	1	1	0.33	1	1	1	0.73	0.82	1	1
D1	0	0.33	0	1	0	0	0.73	0.61	0.17	0
CodeBERT	1	0.33	0.67	1	0.5	0	0.26	0.25	0.42	0

warnings. CNN-based treatments (B2 and C2) suffer from low precision and AUC scores (see Table 8). Similarly, as shown Table 8, CodeBERT often suffers from low precision and poor false alarms and (in the case of CodeBERT) some very low recalls indeed.

Finally, we also test how few training samples we can get away with before significant drops in performance. For this, we test our experimental setup by changing the train/test ratio:

- Our original setup used 80% for the training set and 20% for the test set; we test 60/40, 40/60, and 20/80 splits. These results are shown in Table 9.
- While there are within-dataset variations, the medians show the expected result: as the training size reduces, so does performance.

That said, the drop in performance when using 60% of samples instead of 80% is rather small; the major drop in performance comes when we drop down to 40%. At 40% and lower, there are too few samples to train on maven (which has 4 samples in total), so the learners throw exceptions–this is why we cannot compute metrics.

We note that even at 60%, the median number of labels we use is $\sqrt{0.6n}$ where n is the median number of samples; this computes to 4 samples. Across all datasets, the minimum value of this is 2, and the maximum value is 17. However, this is not to say this is true for all datasets–we note that as shown in Figure 4, the loss landscape after applying our methods was very flat, making optimization easier. Our expectation is that for other datasets, if GHOST and DODGE together flatten the loss landscape to this extent, then learning from sub-10 samples should be possible.

In summary:

> **Answer 1:** To recognize actionable static code warnings, apply all the treatments of §4. Also, spend most tuning faster feedforward neural nets rather than trusting (a) traditional learners or (b) more recent "bleeding edge" neural net methods.

TABLE 9: Our results using A1, with varying train/test ratios.

% train	maven	cassandra	jmeter	commons	lucene-solr	ant	tomcat	derby	median
				PRECISION (*better* results are *larger*)					
80%	1	1	1	1	0.8	1	0.79	0.72	1
60%	1	1	1	0.8	0.8	1	0.65	0.67	0.9
40%	-	0	0.86	0.8	0.82	0.67	0.65	0.65	0.67
20%	-	0	0.73	0.57	0.69	0.67	0.76	0.65	0.67
				AUC: TP vs. TN (*better* results are *larger*)					
80%	1	1	0.83	1	0.75	1	0.68	0.57	0.92
60%	1	0.63	0.8	0.9	0.75	0.83	0.55	0.56	0.78
40%	-	0.5	0.8	0.95	0.77	0.54	0.56	0.53	0.56
20%	-	0.5	0.69	0.71	0.59	0.61	0.7	0.54	0.61
				FALSE ALARM RATE (*better* results are *smaller*)					
80%	0	0	0	0	0.5	0	0.29	0.79	0
60%	0	0	0	0.2	0.5	0	0.52	0.81	0.1
40%	-	0	0.25	0.1	0.29	0.38	0.57	0.8	0.29
20%	-	0	0.5	0.25	0.56	0.73	0.36	0.55	0.5
				RECALL (*better* results are *larger*)					
80%	1	1	0.67	1	1	1	0.65	0.94	1
60%	1	0.25	0.6	1	1	0.67	0.62	0.92	0.8
40%	-	0	0.86	1	0.82	0.46	0.69	0.86	0.82
20%	-	0	0.89	0.67	0.73	0.94	0.75	0.63	0.73

TABLE 10: Median performance improvements seen after applying all the treatments A1 (defined in §4); i.e. all of *instance*, *label*, *boundary* and *parameter* engineering.

		From Table 2	From right-hand-side of Table 8	Improvement
higher is *better*	precision	50	100	50
	AUC	41	90	59
	recall	19	100	89
lower is *better*	false alarm	32	0	32

TABLE 11: Percent changes in Li et al. [38]'s smoothness metric, seen after applying the methods of this paper.

Dataset	% change
maven	158.87
cassandra	73.09
jmeter	55.53
tomcat	36.34
derby	31.35
commons	29.61
ant	24.78
lucene-solr	16.46
median	**33.85**

RQ2: Does GHOST2's combination of instance, label, boundary and parameter engineering, reduce the complexity of the decision boundary?

Previously, this paper argued that reason for the poor performance seen in prior was due to the complexity of the data (specifically, the bumpy shape seen in Figure 3). Our treatments of §4 were designed to simplify that landscape. Did we succeed?

Figure 4 shows the landscape in TOMCAT after the treatments of §4 were applied. By comparing this figure with Figure 3, we can see that our treatments achieved the desired goal of removing the "bumps".

As to the other data sets, Li et al. [38] propose a "smoothness" equation to measure a data set's "bumps". Table 11 shows the percentage change in that smoothness measure seen after applying the methods of this paper. All these changes are positive, indicating that the resulting landscapes are much smoother. For an intuition of what these numbers mean, the TOMCAT change of 36.35% results in Figure 3 changing to Figure 4.

Hence we say:

> **Answer 2:** Label, parameter, instance and boundary engineering can simplify the internal structure of training data.

RQ3: Does GHOST2's combination of *instance*, *label*, *boundary* and *parameter* improve predictive performance?

Table 10 shows the performance improvements after smoothing out our training data from (e.g.) Figure 3 to Figure 4. On 4/8 datasets, we achieve perfect scores. Moreover, we showed through an ablation study that each of the components of GHOST2 is necessary. For example, row A3 in Table 8 is another piece of evidence that hyper-parameter optimization is necessary. The feedforward networks of our approach outperformed more complex learners (CNNs and CodeBERT)–we refer the reader to rows B2, C2, and CodeBERT in Table 8. On the other hand, going too simple for traditional learners leads to A7, which suffers from poor precision scores. Given those large improvements, we say:

> **Answer 3:** Detectors of actionable static code warnings work much better when learned from smoothed training data.

RQ4: Are all parts of GHOST2 necessary; i.e. would something simpler also achieve the overall goal?

We presented an ablation study that showed that each part of GHOST2 was necessary. Among the 13 treatments that we tested, GHOST2 was the only one that consistently scored highly in precision, AUC, and recall, while also generally

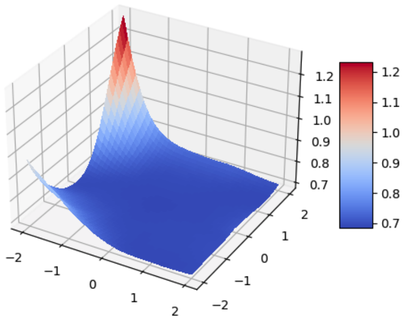


Fig. 4: Error landscape in the TOMCAT after applying the treatments of §4. To understand the simplifications achieved via our methods, the reader might find it insightful to compare this figure against Figure 3.

having low false alarm rates. The crux of our ablation study was that each component of GHOST2 works with the others to produce a strong performer.

Based on the above ablation study results, we say:

> **Answer 4:** Ignoring any of part of instance, label, boundary or parameter engineering leads to worse results than using all parts (at least for the purpose of recognizing actionable static code warnings).

RQ5: Are larger training sets necessary (for the task of recognizing actionable static code warnings)?

In the above discussion, when we presented Table 6, it was noted that several of the train/tests used in this study were very small. At that time, we expressed a concern that, possibly, our data sets explored were too small for effective learning.

This turned out not to be the case. Recall that in Table 8, the data set were sorted left-to-right from smallest to largest training set size. There is no pattern there that smaller data sets perform worse than large ones. In fact– quite the opposite: the smaller data sets were always associated with better performance than those seen on right-left-side. Hence we say:

> **Answer 5:** The methods of this paper are effective, even for very small data sets.

This is a surprising result since one of the truisms of data mining is "the more data, the better": Researchers in linear regression have a rule of thumb that every independent attribute implies needing an additional 10 to 20 training examples [48, 5]. By that reasoning, our data sets with 260 attributes would need 2600 to 5200 examples before a learner could achieve competency.

That said, those rules of thumb for regression models were developed where:

- Inputs are naturally occurring (or handcrafted) attributes (not the more nuanced hyper-dimensions found by neural methods);
- The response variable (output) is a continuous variable that can vary over a large numeric range (and not the two labels we are exploring).

A better conceptual model for our work, that could explain our success when reasoning about small data, might be "pin the tail on the donkey" where there are a handful of green and red pins already in place and we have to add in a ribbon that separates (say) the different colors. Note that since we are using neural, we are free to pull that ribbon across any dimension we like, or even infer a new dimension, if we want. We would argue that in this second conceptualization, it is totally reasonable to expect that such a ribbon can be found *especially* when we are trying to separate only a handful of pins.

7 Threats to Validity

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

1. Sampling bias threatens any classification experiment; i.e., what matters there may not be true here. For example, the data sets used here comes prior work and, possibly, if we explored other data sets we might reach other conclusions. On the other hand, *repeatability* is an important part of science so we argue that our decision to use the Kang et al. data is appropriate and respectful to both that prior work and the scientific method.

2. Learner bias: Machine learning is a large and active field and any single study can only use a small subset of the known algorithms. Our choice of "local learning" tools was explained in §4. That said, it is important to repeat the comments made there that our SMOTEing, SMOOTHing, GHOSTing and DODGEing operators are but one set of choice within a larger framework of possible approaches to instance, label, boundary, and parameter engineering (respectively). As SE research matures, we foresee that our framework will become a workbench within which researchers replace some/all of these treatments with better options. That said, in defence of the current options, we note that our ablation study showed that removing any of them can lead to worse results.

Another threat to validity is the stochastic nature of SMOOTH (since points are selected at random). We tested if SMOOTH leads to conclusion instability as follows. SMOOTH uses a k-d tree to perform hierarchical clustering. We modify it so that it returns the median positions of the leaf clusters. By running this operation 20 times, we have a set of medians. We say that our conclusions are stable if those medians are stable. We conjectured that over each run, because the process is stochastic, that these medians would move slightly; therefore, we ran the following steps:

- From the k-d tree clustering, obtain the number of leaf clusters, call this k
- Run SMOOTH 20 times, and collect a list of the medians. Each set of medians should have k elements, where each element is $d-$dimensional. That is, we have $20 \times k$ points, each $d-$dimensional.
- On these $20k$ points, we run $k-$means clustering, using k from Step 1 as the number of clusters.

- For each cluster, we find the median, and then compute the deviations (via the L1 norm [1]) of each point in that cluster to this median. This gives us a set of deviations, of which we compute the median.
- We compute this median as a percentage of the L1-norm of the dataset

Overall, these percentages are extremely low, with the highest being 0.52%. This suggests that over 20 repeats, while the median does shift, it does so in very small proportions relative to the overall dataset. Therefore, we believe this should not have a meaningful impact on our results.

3. Parameter bias: Learners are controlled by parameters and the resulting performance can change dramatically if those parameters are changed. Accordingly, in this paper, our recommended methods (from Table 4) includes parameter engineering methods to find good parameter settings for our different data sets.

4. Evaluation bias: This paper use four evaluation criteria (precision, AUC, false alarm rate, and recall) and it is certainly true that by other measures, our results might not work be seen to work as as well. In defence of our current selection, we note that we use these measures since they let us compare our new results to prior work (who reported their results using the same measures).

Also, to repeat a remark made previously, another evaluation bias was how we separated data into train/test. In other papers, we have run repeated trials with multiple 80:20 splits for training:test data. This was not here since some of our data sets are too small (see the first few rows of Table 5) that any reduction in the training set size might disadvantage the learning process. Hence, our conclusion do **not** come from (say) some 5×5 cross-validation experiments. That experiment would take 20 months of CPU to execute and generate comparison problems since our test sets would be of radically different sizes (since some of our data sets are tiny)

Hence, the external validity claims of this paper come from patterns seen in eight different software projects. Suppose you wanted to refute the hypothesis that our recommended treatment (A1) is better than the rest. To do so, we need to to find examples that satisfied three tests:

- *Test1*: the performance metric collected from the other methods has to be better;
- *Test2*: the size of the difference in the performance metric has to be larger than a small effect (this is the effect size test);
- *Test3*: the populations from which the performance metrics are drawn are distinguishable (this is the statistical significance test).

Note that there is no point doing the significance and effect size tests *unless* we can first find evidence of better performance. Hence, it should be noted that *Test2* and *Test3* are unnecessary if *Test1* fails. Accordingly, we now turn our attention to *Test1* and the results from Table 8.

In Table 8., suppose to define "better" we look at the 13 rows in the four tables (precision, AUC, false alarm and recall tables). Suppose further we define "better" as follows:

- Row X is "better than A1" if *in N cells of row X, the performance metric is better than A1.*

(Aside: recall that "better" is different for different measures; for false alarm, "better" means "less" while for the others "better" means "more".)

At first it might be tempting to use $N > 4$ (since we are dealing with eight data sets and $N > 4$ would mean we are saying "in the majority case"). It turns out that this $N > 4$ definition of "better" is almost unachievable since, reading Table 8 we see that:

- There is only one example of better $N \geq 4$ rows (for C1 (DODGE) AUC);

That said there are six examples where $N = 3$ cells are better in some row X than A1[5]. That is to say, if we define $N >= 3$ as our threshold for row X is "better than A12" then in only 7/48 treatments would it make sense to apply a significance and effect size test.

In summary: at the $N >= 3$ level, in 41/48 of our experiments, we cannot refute the hypothesis that A12 is better than other treatments.

8 Discussion

This discussion section steps back from the above to make some more general points.

We suggest that this paper should lead to a new way of training newcomers in software analytics:

- Our results show that there is much value in decades-old learning technology (feedforward networks). Hence, we say that when we train newcomers to the field of software analytics, we should certainly train them in the latest techniques (deep learning, CodeBERT, etc).
- That said, we should also ensure that they know of prior work since (as shown above), sometimes those older methods still have currency. For example, if some learner is faster to run, then it is easier to tune. Hence, as shown above, it can be possible for old techniques to do better than new ones, just by tuning.

For future work, it would be useful to check what other SE domains simpler, faster, learners (plus some tuning) outperform more complex learning methods.

That said, we offer the following cautionary note about tuning. Hyper-parameter optimization (HPO, which we have call "parameter engineering" in this paper) has received much recent attention in the SE literature [3, 70, 4] We have shown here that reliance on *just* HPO can be foolhardy since better results can be obtained by the judicious use of HPO combined with more nuanced approaches that actually reflect the particulars of the current problem (e.g. our treatments that adjusted different parts of the data in different ways). As to how much to study the internals of a learner, we showed above that there are many choices deep within a learner than can greatly improve predictive performance. Hence we say that it is very important to know the internals of a learner and how to adjust them. In our opinion, all too often, software engineers use AI tools as "black boxes" with little understanding of their internal structure.

[5] One: A6 for AUC; two: C1 for AUC; three: A6 for false alarm rate; four: A7 for false alarm rate; five: C1 for false alarm rate; six: A5 for recall.

Our results also doubt some of the truisms of our field. For example:

- There is much recent work on big data research in SE, the premise being that "the more data, the better". We certainly do not dispute that but our results do show that it is possible to achieve good results with very small data sets.
- There is much work in software analytics suggesting that deep learning is a superior method for analyzing data [70, 65, 39, 66]. Yet when we tried that here, we found that a decades-old neural net architecture (feed-forward networks, discussed in Table 4) significantly out-performed deep learners.

For newcomers to the field of software analytics, truisms might be useful. But better results might be obtained when teams of data scientists combine to suggest multiple techniques – some of which ignore supposedly tried-and-true truisms.

9 Conclusion

Static analysis tools often suffer from a large number of false alarms that are deemed to be unactionable [57]. Hence, developers often ignore many of their warnings. Prior work by Yang et al. [68] attempted to build predictors for actionable warnings but, as shown by Kang et al. [29], that study used poorly labelled data.

This paper extends the Kang et al. result as follows. Table 2 shows that building models for this domain is a challenging task. The discussion section of §3 conjectured that for the purposes of detecting actionable static code warnings, standard data miners can not handle the complexities of the decision boundary. More specifically, we argued that:

> *For complex data,* **global** *treatments perform worse than* **localized** *treatments which adjust different parts of the landscape in different ways.*

§4 proposed four such localized treatments, which we called instance, parameter, label and boundary engineering.

These treatments were tested on the data generated by Kang et al. (which in turn, was generated by fixing the prior missteps of Yang et al.). On experimentation, it was shown that the combination of all our treatments (in the "A1" results of Table 7) performed much better than than the prior results seen in Table 2. As to why these treatments before so well, the analysis of Table 11 showed that instance, parameter, label and boundary engineering did in fact remove complex shapes in our decision boundaries. As to the relative merits of instance versus parameter versus label versus boundary engineering, an ablation study showed that using all these treatments produces better predictions that alternative treatments that ignored any part.

Finally, we comment here on the value of different teams working together. The specific result reported in this paper is about how to recognize and avoid static code analysis false alarms. That said, there is a more general takeaway. Science is meant to be about a community critiquing and improving each other's ideas. Here, we offer a successful example of such a community interaction where teams from Singapore and the US successfully worked together. Initially, in a 2022 paper [29], the Singapore team identified issues with the data that result in substantially lower performance of the previously-reported best predictor of actionable warnings [64, 68, 69]. Subsequently, in this paper, both teams combined to produce new results that clarified and improved the old work. That teamwork leads us to trying methods which, according to the truisms of our field, should not have worked. The teamwork that generated this paper should be routine, and not some rare exceptional case.

Acknowledgments

This work was partially supported by an NSF Grant #1908762.

References

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

[2] Amritanshu Agrawal and Tim Menzies. Is" better data" better than" better data miners"? In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 1050–1061. IEEE, 2018.

[3] Amritanshu Agrawal, Wei Fu, Di Chen, Xipeng Shen, and Tim Menzies. How to "dodge" complex software analytics. *IEEE Transactions on Software Engineering*, 47(10):2182–2194, 2019.

[4] Amritanshu Agrawal, Xueqi Yang, Rishabh Agrawal, Rahul Yedida, Xipeng Shen, and Tim Menzies. Simpler hyperparameter optimization for software analytics: why, how, when. *IEEE Transactions on Software Engineering*, 2021.

[5] Peter C Austin and Ewout W Steyerberg. Events per variable (epv) and the relative performance of different strategies for estimating the out-of-sample validity of logistic regression models. *Statistical methods in medical research*, 26(2):796–808, 2017.

[6] Nathaniel Ayewah and William Pugh. The google findbugs fixit. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, pages 241–252, 2010.

[7] Subarno Banerjee, Lazaro Clapp, and Manu Sridharan. Nullaway: Practical type-based null safety for java. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 740–750, 2019.

[8] Ella Barkan, Alon Hazan, and Vadim Ratner. Reduce discrepancy of human annotators in medical imaging by automatic visual comparison to similar cases, February 9 2021. US Patent 10,916,343.

[9] Moritz Beller, Radjino Bholanath, Shane McIntosh, and Andy Zaidman. Analyzing the state of static analysis: A large-scale evaluation in open source software. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 470–481. IEEE, 2016.

[10] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *J. Mach. Learn. Res.*, 13(null):281–305, feb 2012. ISSN 1532-4435.

[11] David Berthelot, Nicholas Carlini, Ian Goodfellow, Nicolas Papernot, Avital Oliver, and Colin A Raffel. Mixmatch: A holistic approach to semi-supervised learning. *Advances in Neural Information Processing Systems*, 32, 2019.

[12] Cristiano Calcagno, Dino Distefano, Jérémy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. Moving fast with software verification. In *NASA Formal Methods Symposium*, pages 3–11. Springer, 2015.

[13] J. Chakraborty, S. Majumder, and H. Tu. Can we achieve fairness using semi-supervised learning? *Fairware*, 2022.

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

[15] Maria Christakis and Christian Bird. What developers want and need from program analysis: an empirical study. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, pages 332–343. ACM, 2016.

[16] Paul R Cohen. *Empirical methods for artificial intelligence*, volume 139. MIT press Cambridge, 1995.

[17] Filipe R Cordeiro and Gustavo Carneiro. A survey on deep learning with noisy labels: How to train your model when you cannot trust on the annotations? In *2020 33rd SIBGRAPI conference on graphics, patterns and images (SIBGRAPI)*, pages 9–16. IEEE, 2020.

[18] Roland Croft, Dominic Newlands, Ziyu Chen, and M Ali Babar. An empirical study of rule-based and learning-based approaches for static application security testing. In *Proceedings of the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–12, 2021.

[19] Kalyan Deb, Manikanth Mohan, and Shikhar Mishra. Evaluating the ϵ-dominance based multi-objective evolutionary algorithm for a quick computation of pareto-optimal solutions. *Evolutionary computation*, 13:501–25, 02 2005. doi: 10.1162/106365605774666895.

[20] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.

[21] Andrew Habib and Michael Pradel. How many of all bugs do we find? a study of static bug detectors. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 317–328. IEEE, 2018.

[22] Quinn Hanam, Lin Tan, Reid Holmes, and Patrick Lam. Finding patterns in static analysis alerts: improving actionable alert ranking. In *Proceedings of the 11th working conference on mining software repositories*, pages 152–161, 2014.

[23] Sarah Heckman and Laurie Williams. On establishing a benchmark for evaluating static analysis alert prioritization and classification techniques. In *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*, pages 41–50, 2008.

[24] Sarah Heckman and Laurie Williams. A model building process for identifying actionable static analysis alerts. In *2009 International conference on software testing verification and validation*, pages 161–170. IEEE, 2009.

[25] Nasif Imtiaz, Brendan Murphy, and Laurie Williams. How do developers act on static analysis alerts? an empirical study of coverity usage. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*, pages 323–333. IEEE, 2019.

[26] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pages 448–456. PMLR, 2015.

[27] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don't software developers use static analysis tools to find bugs? In *2013 35th International Conference on Software Engineering (ICSE)*, pages 672–681. IEEE, 2013.

[28] Ashwin Kallingal Joshy, Xueyuan Chen, Benjamin Steenhoek, and Wei Le. Validating static warnings via testing code fragments. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 540–552, 2021.

[29] Hong Jin Kang, Khai Loong Aw, and David Lo. Detecting false alarms from automatic static analysis tools: How far are we? *arXiv preprint arXiv:2202.05982*, 2022.

[30] Anant Kharkar, Roshanak Zilouchian Moghaddam, Matthew Jin, Xiaoyu Liu, Xin Shi, Colin Clement, and Neel Sundaresan. Learning to reduce false positives in analytic bug detectors. In *Proceedings of the IEEE/ACM International Conference on Software Engineering 2022*, 2022.

[31] Hyunsu Kim, Mukund Raghothaman, and Kihong Heo. Learning probabilistic models for static analysis alarms. In *Proceedings of the IEEE/ACM International Conference on Software Engineering 2022*, 2022.

[32] Sunghun Kim and Michael D. Ernst. Which warnings should i fix first? In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE '07, page 45–54, New York, NY, USA, 2007. Association for Computing Machinery. ISBN 9781595938114. doi: 10.1145/1287624.1287633. URL https://doi.org/10.1145/1287624.1287633.

[33] Sunghun Kim and Michael D Ernst. Which warnings should I fix first? In *Proceedings of the the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, pages 45–54, 2007.

[34] Durk P Kingma, Shakir Mohamed, Danilo Jimenez Rezende, and Max Welling. Semi-supervised learning with deep generative models. *Advances in neural information processing systems*, 27, 2014.

[35] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25, 2012.

[36] Ajitesh Kumar. Svm rbf kernel parameters with code examples. Available on-line at https://dzone.com/articles/using-jsonb-in-postgresql-how-to-effectively-store-1.

[37] Yann LeCun, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551, 1989.

[38] Hao Li, Zheng Xu, Gavin Taylor, Christoph Studer, and Tom Goldstein. Visualizing the loss landscape of neural nets. *Advances in Neural Information Processing Systems*, 31, 2018.

[39] Liuqing Li, He Feng, Wenjie Zhuang, Na Meng, and Barbara Ryder. Cclearner: A deep learning-based clone detection approach. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 249–260. IEEE, 2017.

[40] Guangtai Liang, Ling Wu, Qian Wu, Qianxiang Wang, Tao Xie, and Hong Mei. Automatic construction of an effective training set for prioritizing static analysis warnings. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 93–102, 2010.

[41] Kede Ma, Xuelin Liu, Yuming Fang, and Eero P Simoncelli. Blind image quality assessment by learning from multiple annotators. In *2019 IEEE International Conference on Image Processing (ICIP)*, pages 2344–2348. IEEE, 2019.

[42] Don McNicol. *A primer of signal detection theory*. Psychology Press, 2005.

[43] Guido F Montufar, Razvan Pascanu, Kyunghyun Cho, and Yoshua Bengio. On the number of linear regions of deep neural networks. *Advances in Neural Information Processing Systems*, 27, 2014.

[44] Tukaram Muske and Alexander Serebrenik. Techniques for efficient automated elimination of false positives. In *2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 259–263. IEEE, 2020.

[45] Marcus Nachtigall, Michael Schlichtig, and Eric Bodden. A large-scale study of usability criteria addressed by static analysis tools. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 532–543, 2022.

[46] Jaechang Nam, Song Wang, Yuan Xi, and Lin Tan. A bug finder refined by a large set of open-source projects. *Information and Software Technology*, 112:164–175, 2019.

[47] Sebastiano Panichella, Venera Arnaoudova, Massimiliano Di Penta, and Giuliano Antoniol. Would static analysis tools help developers with code reviews? In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 161–170. IEEE, 2015.

[48] Peter Peduzzi, John Concato, Elizabeth Kemper, Theodore R Holford, and Alvan R Feinstein. A simulation study of the number of events per variable in logistic regression analysis. *Journal of clinical epidemiology*, 49(12):1373–1379, 1996.

[49] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. " why should i trust you?" explaining the predictions of any classifier. In *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*, pages 1135–1144, 2016.

[50] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323 (6088):533–536, 1986.

[51] Joseph Ruthruff, John Penix, J Morgenthaler, Sebastian Elbaum, and Gregg Rothermel. Predicting accurate and actionable static analysis warnings. In *2008 ACM/IEEE 30th International Conference on Software Engineering*, pages 341–350. IEEE, 2008.

[52] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. Lessons from building static analysis tools at google. *Communications of the ACM*, 61(4):58–66, 2018.

[53] Shibani Santurkar, Dimitris Tsipras, Andrew Ilyas, and Aleksander Madry. How does batch normalization help optimization? *Advances in Neural Information Processing Systems*, 31, 2018.

[54] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

[55] Ferdian Thung, David Lo, Lingxiao Jiang, Foyzur Rahman, Premkumar T Devanbu, et al. To what extent could we detect field defects? an empirical study of false negatives in static bug

finding tools. In *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 50–59. IEEE, 2012.
[56] Daniil Tiganov, Lisa Nguyen Quang Do, and Karim Ali. Designing uis for static-analysis tools. *Communications of the ACM*, 65(2):52–58, 2022.
[57] David A Tomassi and Cindy Rubio-González. On the real-world effectiveness of static bug detectors at finding null pointer exceptions. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 292–303. IEEE, 2021.
[58] H. Tu and T. Menzies. FRUGAL: unlocking SSL for software analytics. *ASE*, 2021.
[59] H. Tu and T. Menzies. DebtFree: minimizing labeling cost in self-admitted technical debt identification using semi-supervised learning. *EMSE*, 2022.
[60] H. Tu, G. Papadimitriou, M. Kiran, C. Wang, A. Mandal, E. Deelman, and T. Menzies. Mining workflows for anomalous data transfers. *MSR*, 2021.
[61] H. Tu, Z. Yu, and T. Menzies. Better data labelling with emblem (and how that impacts defect prediction). *IEEE TSE*, 2022.
[62] Carmine Vassallo, Sebastiano Panichella, Fabio Palomba, Sebastian Proksch, Harald C Gall, and Andy Zaidman. How developers engage with static analysis tools in different contexts. *Empirical Software Engineering*, 25(2):1419–1457, 2020.
[63] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.
[64] Junjie Wang, Song Wang, and Qing Wang. Is there a" golden" feature set for static warning identification? an experimental evaluation. In *Proceedings of the 12th ACM/IEEE international symposium on empirical software engineering and measurement*, pages 1–10, 2018.
[65] Song Wang, Taiyue Liu, Jaechang Nam, and Lin Tan. Deep semantic feature learning for software defect prediction. *IEEE Transactions on Software Engineering*, 46(12):1267–1293, 2018.
[66] Martin White. Deep representations for software engineering. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 781–783. IEEE, 2015.
[67] Ian H. Witten, Eibe Frank, and Mark A. Hall. *Data mining: practical machine learning tools and techniques, 3rd Edition*. Morgan Kaufmann, Elsevier, 2011. ISBN 9780123748560. URL https://www.worldcat.org/oclc/262433473.
[68] Xueqi Yang, Jianfeng Chen, Rahul Yedida, Zhe Yu, and Tim Menzies. Learning to recognize actionable static code warnings (is intrinsically easy). *Empirical Software Engineering*, 26(3):1–24, 2021.
[69] Xueqi Yang, Zhe Yu, Junjie Wang, and Tim Menzies. Understanding static code warnings: An incremental ai approach. *Expert Systems with Applications*, 167:114134, 2021.
[70] Rahul Yedida and Tim Menzies. On the value of oversampling for deep learning in software defect prediction. *IEEE Transactions on Software Engineering*, 2021.
[71] Zhe Yu, Fahmid Morshed Fahid, Huy Tu, and Tim Menzies. Identifying self-admitted technical debts with jitterbug: A two-step approach. *IEEE Transactions on Software Engineering*, 2020.
[72] Fiorella Zampetti, Simone Scalabrino, Rocco Oliveto, Gerardo Canfora, and Massimiliano Di Penta. How open source projects use static code analysis tools in continuous integration pipelines. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 334–344. IEEE, 2017.
[73] Xiaohua Zhai, Avital Oliver, Alexander Kolesnikov, and Lucas Beyer. S4l: Self-supervised semi-supervised learning. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 1476–1485, 2019.
[74] Jiang Zheng, Laurie Williams, Nachiappan Nagappan, Will Snipes, John P Hudepohl, and Mladen A Vouk. On the value of static analysis for fault detection in software. *IEEE Transactions on Software Engineering*, 32(4):240–253, 2006.
[75] Xiaojin Jerry Zhu. Semi-supervised learning literature survey. 2005.

Rahul Yedida is a PhD student in Computer Science at NC State University. His research interests include automated software engineering and machine learning for software engineering. https://ryedida.me.

Hong Jin Kang is a Ph.D. student at Singapore Management University. His research interests include machine learning for software engineering, and mining rules and specifications. https://kanghj.github.io/.

Huy Tu holds a Ph.D. in Computer Science from North Carolina State University, Raleigh, NC. They explored frugal labeling processes while improving the data quality for software analytics. Now, they works for Meta Platforms, Inc. https://kentu.us.

Xueqi Yang is a Ph.D. student in Computer Science at North Carolina State University. Her research interests include automatic static analysis and applying human-assisted AI algorithms in software engineering. https://xueqiyang.github.io/.

David Lo is a Professor in Computer Science at Singapore Management University. His research interests include software analytics, empirical software engineering, cybersecurity, and SE4AI. http://www.mysmu.edu/faculty/davidlo/.

Tim Menzies (IEEE Fellow, Ph.D. UNSW, 1995) is a Professor in Computer Science at NC State University, USA. His research interests include software engineering (SE), data mining, artificial intelligence, and search-based SE, open access science. http://menzies.us.