ABSTRACT

YEDIDA, RAHUL. Guidelines for the Application of Neural Technologies in Software Analytics (or: How to Do More with Less in SE). (Under the direction of Tim Menzies).

Deep learning is now prevalent in software engineering (SE) research, and its applications are well-understood. However, can we improve these models through tuning? This dissertation argues that hyper-parameter optimization (HPO) on computationally cheaper neural network architectures outperforms the state-of-the-art. We demonstrate that, for five SE tasks (defect prediction, issue lifetime prediction, code smell detection, automated microservice partitioning, and false alarm detection in FindBugs warnings), a combination of operators, each devised based on the idiosyncrasies of the datasets under study, enables a feedforward model to outperform other deep learning technologies that require orders of magnitude more time to run.

To encourage the use of HPO, we offer two novel strategies based on theoretical links between the flatness of minima and their generalization ability. One strategy is purely derived from theoretical machine learning literature, while the other additionally leverages our observations of SE data complexity across tasks. These methods outperform prior HPO methods at a fraction of the runtime.

In summary, we caution against using technologies as black-box instruments without reflecting on the nature of their internal search space. By reflecting on that space, we can offer a new high-water mark for faster HPO based on the flatness of minima.

Guidelines for the Application of Neural Technologies in Software Analytics (or: How to Do More with Less in SE)

by
Rahul Yedida

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Science

Raleigh, North Carolina
2024

APPROVED BY:

Rada Chirkova	Noboru Matsuda
David Lo	John-Paul Ore

Tim Menzies
Chair of Advisory Committee

DEDICATION

To my parents, Rajasekhar and Sridevi Yedida, and to my aunt and uncle, Madhavi Kolachala and Naveen Vedula.

BIOGRAPHY

Rahul Yedida was born in Raipur, Chhattisgarh, India in 1997. In 2019, he received his Bachelor of Engineering degree in Computer Science from PES Institute of Technology, Bangalore, where he worked on theory-driven ML with Dr. Snehanshu Saha. He then worked under the supervision of Dr. Tim Menzies on better, faster deep learning in software analytics. His research interests lie in ML for SE and theory-driven deep learning.

ACKNOWLEDGEMENTS

I am grateful to my family for their unwavering support throughout my Ph.D. journey.

I also wish to extend my gratitude to my fellow graduate students at NC State, who have provided additional support and camaraderie.

This dissertation would not have been possible without the invaluable advice of my advisor over the past five years.

Various professors with whom I have interacted have also been instrumental in shaping my approach to problem-solving. In particular, I would like to acknowledge Dr. Snehanshu Saha and Dr. Aritra Mitra for their guidance.

Finally, the relatable narratives in Taylor Swift's songs have provided a comforting soundtrack to many writing sessions. Her music has been a source of motivation and resilience during this endeavor.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER

1

INTRODUCTION

1.1 Statement of Thesis

Can deep learning (DL) be applied to SE data without first adjusting those learners to the particulars of SE? Many researchers believe so. A common claim is that DL supports a kind of automated feature engineering [152, 163, 198, 231, 246] that lets data scientists avoid tedious manual feature engineering, prior to running their learners.

It is timely to assess such claims. DL is now widely applied to many SE tasks[1]. Despite its widespread use, few SE researchers are critically examining the utility of deep learning for SE. For example, in their literature review, Li et al. [126] explored over 40 SE papers facilitated by DL models and found that 33 of them used DL without comparison to other (non-DL) techniques. In our own literature review on DL in SE (reported below in Section 2), we find experimental comparisons of DL-vs-non-DL in less than 10% of papers.

In Figure 1.1, we show the distribution of the "smoothness" of SE and non-SE datasets. We will formalize the notion of smoothness in Chapter 2, but for now, we will

[1]Bug localization ([100]), sentiment analysis ([81, 128]), API mining ([41, 79, 156]), effort estimation for agile development ([43]), code similarity detection ([251]), code clone detection ([223]), and more.

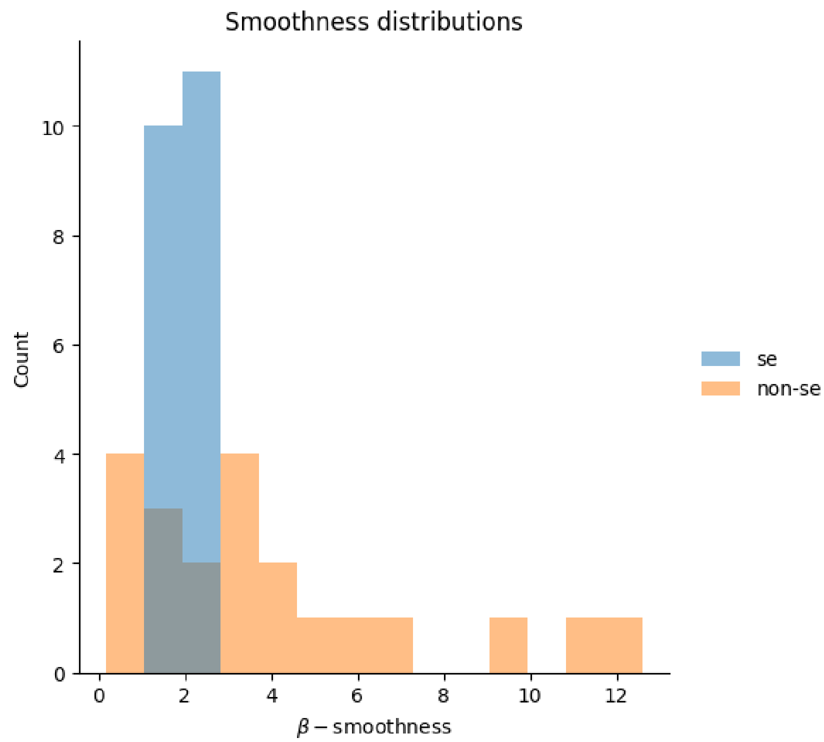


Figure 1.1: Distribution of smoothness for SE and non-SE datasets. SE datasets are a subset of the datasets explored in this dissertation. ML datasets come from the UCI repository and the OpenML tasks used in HPOBench.

merely state that it is one measure of the difficulty of learning from a dataset. Clearly, there is a significant difference between SE and non-SE datasets, and the former are simpler. We argue that this should lead researchers to try simpler machine learning methods before more complex ones.

One way to address the above, which will **not** be used here, is massively expensive runs over multiple data sets and methods, repeated multiple times, to cover a vast space of possible settings. Rather, we seek some alternate, less compute-intensive approach. We seek a more fundamental description of what it means to commission a learner for a particular task. Ideally, that description should be *formal*, *fast*, and *effective*:

- *Formal:* Be strongly grounded in mathematical theory;
- *Fast:* Guide the design of very fast algorithms for the tuning of data miners to particular data sets;
- *Effective:* After tuning, those data miners should perform better than the prior

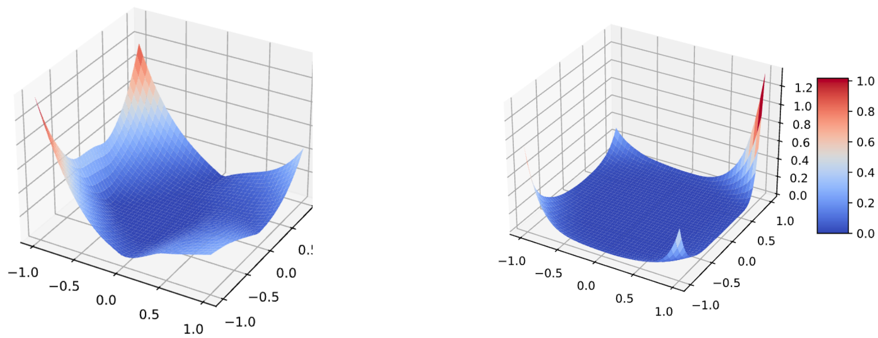


Figure 1.2: Left & right shows loss landscape before & after HPO. The loss function is visualized as the z dimension of domain data mapped into two PCA components x, y. In this figure, red indicates regions with highest mean squared error.

state-of-the-art.

To find such methods, we turn to the *hyperspace boundary* thata divides examples of different classes. We say that:

> **Statement of thesis**: Learning algorithms need to be guided toward "better" decision boundaries through a combination of engineering methods that preprocess data, with hyper-parameter optimization to find "good" minima during optimization.

We will elaborate on what constitutes a "better" decision boundary and a "good" minimum later in this dissertation. This method, which we call BOUNDED, is a set of methods, only some of which are needed for some tasks. We strongly advocate for experimenting with simpler methods before using more complex ones, and in accordance with that tenet, we show that for some tasks, not all the engineering methods of BOUNDED are necessary.

Figure 1.2 shows a preview of the motivation behind our approach. The right side shows a "good" minima; Chapter 2 will discuss the literature on why this is the case. We will also show that the techniques proposed in this dissertation change the loss landscape from looking like the one on the left, to the one on the right.

Other researchers have argued that for something like BOUNDED. However, prior work has been limited in its scope of decision boundary manipulation. For example, support vector machines (SVMs) use convex optimization to produce a maximum-margin classifier in a kernel space.But the boundary manipulation methods are hard-wired into the learner. Here, we explore learner-independent methods for boundary manipulation.

Recently, there has been some work in optimizing for "good" minima [72], by modifying gradient descent. That work, however, only looks at the loss function to change the decision boundary, rather than attempting to directly manipulate it. Moreover, their work is limited to a fixed set of hyper-parameters. In this work, on the other hand, boundary manipulation is a primary design feature of our algorithms.

Our boundary manipulation methods will divide into four areas:

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

To confirm the effectiveness of this approach, the experiments reported later in this paper explore 20 repeated trials on 38 software engineering data sets exploring issues such as defect prediction, detecting false alarms in static code warnings, issue lifetime prediction, code smell detection, and automated microservice partitioning.

1.2 Contributions

The major contributions of this dissertation are summarized below:

- We develop formal theory to help make informed decisions about the engineering problems discussed above.
- For the five software analytics tasks listed above, we achieve state-of-the-art results. Moreover, our methods are significantly (and often orders of magnitude) faster than the prior state-of-the-art.
- We show that learning problems can be divided into a set of engineering decisions, each of which needs careful consideration. We further show that some problems are simpler than others, and advocate for matching the complexity of the solution to that of the problem.

[2] We use this term synonymously with hyper-parameter optimization.

- We demonstrate that SE datasets are often simpler than their non-SE counterparts because the smoothness of the loss landscapes generated by them is much lower.
- We demonstrate that a distinct minority of software analytics papers use hyper-parameter optimization despite repeated results showing its efficacy.
- To motivate researchers to adopt hyper-parameter optimization, we derive the theory for, and implement, two novel algorithms for hyper-parameter optimization that outperforms prior work, while also being significantly faster.

1.3 Papers from this Dissertation

This thesis is a culmination of several research works [2, 232, 234, 235, 236, 238, 239, 241, 242, 243], discussed below. They, along with other research by the author during his Ph.D., are listed in Table 1.1

In Agrawal et al. [2] we explored standard HPO methods for SE and was one of the papers advocating for hyper-parameter optimization. While successful, this paper encouraged us to seek a more fundamental approach. In Yedida & Menzies [234], we presented an early version of BOUNDED. Yedida et al. [241] showed that a simpler version of BOUNDED could outperform a much more complex model, orders of magnitude faster. Yang et al. [232] marked an initial exploration into the problem of detecting false alarms in static code analysis tools. Yedida et al. [239] was the first collaborative work with IBM Research, showing that hyper-parameter optimization can significantly improve automated microservice partitioning algorithms and change the conclusions from research. We continued this work in Yedida et al. [243], which proposed a novel approach based on recommendations from ML and SE literature. In Yedida & Menzies [235], we showed that our early version of BOUNDED could be applied successfully to another SE domain. Yedida et al. [242] matured BOUNDED and showed that it could be applied to yet another SE domain. We introduced `SMOOTHIE` in Yedida & Menzies [236], a novel hyper-parameter optimization method tailored to SE data, based on the observation that the various parts of BOUNDED increased the smoothness of the loss and led to "good" minima. Finally, Yedida & Saha [238] was a second step, proposing another novel hyper-parameter optimization method that directly optimized for "good minima".

Table 1.1: List of papers written and published during Ph.D. program.

Authors	**Title**	**Venue**	**Year**
	Directly related to dissertation		
Agrawal, A., Yang, X., Agrawal, R., **Yedida, R.**, Shen, X., & Menzies, T.	Simpler Hyperparameter Optimization for Software Analytics: Why, How, When?	IEEE Trans. SE	2021
Yedida, R., & Menzies, T.	On the Value of Oversampling for Deep Learning in Software Defect Prediction	IEEE Trans. SE	2021
Yang, X., Chen, J., **Yedida, R.**, Yu, Z., & Menzies, T.	Learning to recognize actionable static code warnings (is intrinsically easy)	Empirical SE	2021
Yedida, R., Krishna, R., Kalia, A., Menzies, T., Xiao, J., & Vukovic, M.	Lessons learned from hyper-parameter tuning for microservice candidate identification	Automated SE (Conf.)	2021
Yedida, R., Krishna, R., Kalia, A., Menzies, T., Xiao, J., & Vukovic, M.	An Expert System for Redesigning Software for Cloud Applications	Exp. Sys. with App.	2023
Yedida, R., Menzies, T.	How to Improve Deep Learning for Software Analytics (a case study with code smell detection)	Mining Software Repositories	2022
Yedida, R., Kang, H. J., Tu, K., Lo, D., & Menzies, T.	How to Find Actionable Static Analysis Warnings: A Case Study with FindBugs	IEEE Trans. SE	2023
Yedida, R., & Menzies, T.	Is Hyper-Parameter Optimization Different for Software Analytics?	Under review	2024
Yedida, R., & Saha, S.	Flatness-guided hyper-parameter optimization	Under review	2024
Yedida, R., Yang, X., & Menzies, T.	Old but Gold: Reconsidering the value of feedforward learners for software analytics	Unpublished	2021
	Unrelated to dissertation		
Saha, S., Nagaraj, N., Mathur, A., **Yedida, R.**, & HR, S.	Evolution of novel activation functions in neural network training for astronomy data: habitability classification of exoplanets	Eur. Phy. J. Spec. Top.	2020
Yedida, R., Saha, S., & Prashanth, T.	LipschitzLR: Using theoretically computed adaptive learning rates for fast convergence	Applied Intelligence	2021
Sridhar, S., Saha, S., Shaikh, A., **Yedida, R.**, & Saha, S.	Parsimonious computing: A minority training regime for effective prediction in large microarray expression data sets	IJCNN	2020
Yedida, R., & Saha, S.	Beginning with machine learning: a comprehensive primer.	Eur. Phy. J. Spec. Top.	2021
Baldassarre, M. T., Ernst, N., Hermann, B., Menzies, T., & **Yedida, R.**	(Re) Use of Research Results (Is Rampant)	Comm. ACM	2023

CHAPTER

2

BACKGROUND

This chapter will cover background concepts in deep learning and will briefly touch on important results in convex optimization.

2.1 Deep learning

2.1.1 Feedforward networks

Feedforward neural networks are a slight extension of multi-layer perceptrons, and are a technology from the 1980s [176]. Briefly, a feedforward network is a directed acyclic graph of nodes, where the edges are weighted–these weights form the parameters of the model. At each node, a weighted sum of the inputs is performed, followed by an "activation function" to form an output, and weights are updated using the backpropagation algorithm [176]. A typical activation function used is the ReLU function, $f(x) = \max(0, x)$, which was introduced by Nair & Hinton [152].

Despite their simplicity, feedforward networks have been shown theoretically to have large representative capacity; for example, Hornik et al. [98] shows that these networks can represent any arbitrary decision boundary. This interest is not merely theoretical: Galke & Scherp [74] show that for image classification, feedforward networks can perform competitively with the state-of-the-art algorithms. Nevertheless, modern approaches continue to be more popular in both deep learning and software engineering literature, partly because optimization with modern deep learning layers is easier [181], and partly because over-parameterized neural networks can still be optimized by simple algorithms such as gradient descent [59, 254].

2.1.2 Deep learning

Deep learning refers to an extension of feedforward networks (or "artificial neural networks"). Those had a limited number of hidden layers and used the sigmoid or threshold function at each node (this function is called the "activation function" in deep learning literature). They differ from "deep learners" in the following respects:

- **Hidden layers:** Modern deep learners typically have many hidden layers, that allow for hierarchical feature selection [246].
- **Activation functions:** Deep learners now use a variety of activation functions, most notably, the ReLU ($f(x) = \max(0, x)$) [152] function.
- **Architectures:** The "architecture" of a deep learner refers to the arrangement of the nodes and the connections between them. For example, convolutional neural networks and recurrent neural networks [96] are two classes of architectures. Crucially, the "feedforward" network architecture refers to the standard multi-layer perceptron setup, but is typically implemented with more layers and the ReLU activation function.

Importantly for the context of this paper, however, it is noteworthy that the more recent learners are more complex in that they have orders of magnitude more parameters than prior, simple, feedforward networks (which have been around for decades). For example, a modern deep learner for language modeling can have billions of parameters [169]. Consequently, these are significantly slower to optimize, although they typically achieve better results. As we will see in the next section, the more complex networks dominate the SE field as well, and the feedforward network is rather abandoned.

Autoencoders

An autoencoder is an encoder-decoder architecture that is used to compress an input into fewer dimensions. While several variations exist, such as the variational autoencoder, in this paper, we only discuss the standard model (called a *stacked autoencoder*).

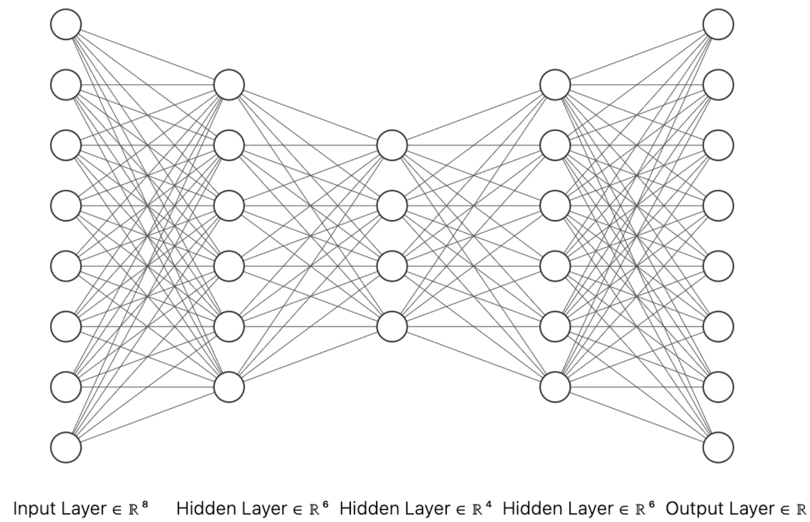


Figure 2.1: A basic autoencoder. Here, the input of size 8 is reduced to the bottleneck layer of size 4, and the network then increases back to size 8.

In this model, the network is designed with hidden layers that decrease in size to a certain point (called the *bottleneck* layer), and then increase back to the original input length (see Figure 2.1). The network is then trained to recreate the original inputs, by minimizing the mean squared error (MSE) loss function. This forces the network to learn a mapping from the original input length to the bottleneck length (this part of the network forms the *encoder*), and another mapping from the bottleneck layer to the original input length (called the *decoder*). These can then be used to reduce dimensionality of some data, or recreate the original data from the lower-dimensional embeddings.

2.2 Deep learning in SE

Using a literature review, this section argues that the issue raised in this dissertation (that researchers seen rush to use the latest methods from deep learning literature, without baselining them against simpler) is widespread in the software analytics literature.

To understand how deep learning are used in SE, we performed the following steps.

- **Seed:** Our approach started with collecting relevant papers. As a seed, we collected papers from the recent literature review conducted by [218].

- **Search:** To this list, we added papers added by our own searches on Google Scholar. Our search keywords included "deep learning AND software", "deep learning AND

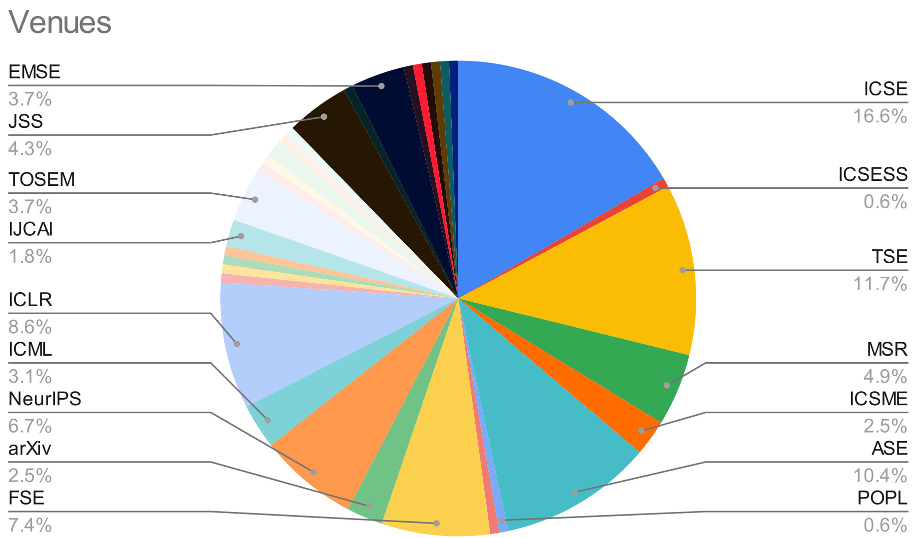


Figure 2.2: Summary of venues in our literature review.

defect prediction", and "deep learning AND bug fix" (this last criteria was added since we found that some recent papers, such as Lee et al. [121], used the term "bug fix time" rather than "issue close time").

- **Filter:** Next, we filtered papers using the following criteria: (a) published in top venues as listed in Google Scholar metrics for Software Systems, Artificial Intelligence, and Computational Linguistics; or, released on arXiv in the last 3 years or widely cited (> 100 cites) (b) has at least 10 cites per year, unless it was published in or after 2017 (the last three years). The distribution of papers across different venues is shown in Figure 2.2.

- **Backward Snowballing:** As recommended by Wohlin [225], we performed "snowballing" on our paper (i.e. we added papers cited by the papers in our list that also satisfy the criteria above). Our snowballing stopped when either (a) the list of papers cited by the current generation is a subset of the papers already in the list, or (b) there were no further papers found.

This led to a list of 155 papers, which we summarize in Figure 2.3. Some engineering judgement was used in assigning papers to the categories of that figure. For example, a paper on learning a latent embedding of an API [156] for various purposes, such as

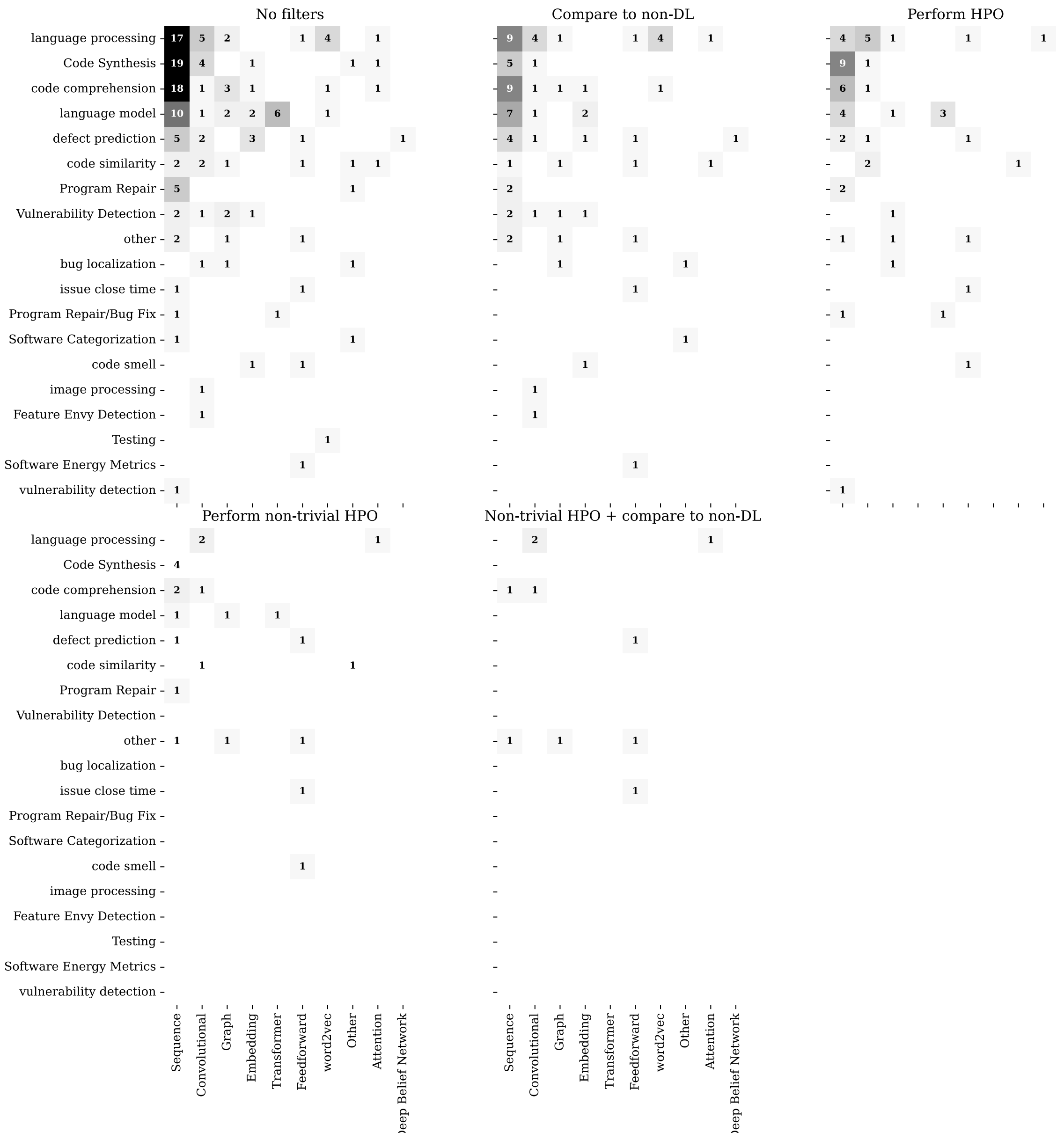


Figure 2.3: A summary of our literature review of deep learning methods in SE. For a list of the papers shown in the right-hand-side column, see Table 2.2

discovering analogous APIs among third-parties [41], was categorized as "code comprehension". Similarly, most papers performing some variant of code translation, including API translation as in [80], were categorized into "language processing"–a bin that contains programming language processing and natural language processing. Tasks that we could not justifiably merge into an existing bin (e.g. on image processing [160, 199] were given their own special category.

From our analysis, we observe the following:

- 80 (51.9%) of those papers compare their results to non-DL methods. We suggest that number should be higher–it is important to benchmark new methods against prior state-of-the-art.

- Only a minority of papers (50 = 32.5%) performed any sort of hyper-parameter optimization (HPO), i.e., used methods that tune the various "hyper-parameters", such as the number of layers of the deep learner, to eke out the best performance of deep learning (39.4%).

- Even fewer papers (12 = 7.8%) applied hyper-parameter optimization in a non-trivial manner; i.e., not using deprecated grid search [25] and using a hold-out set to assess the tuning before going to a separate test set).

  These "best of breed" papers are listed in Table 2.2.

Table 2.1: Static code analysis 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

Table 2.2: Papers satisfying all criteria of Figure 2.3.

Paper	**Reference**
Suggesting Accurate Method and Class Names	[5]
An expert system for redesigning software for cloud applications	[243]
How to Find Actionable Static Analysis Warnings: A Case Study With FindBugs	[242]
How to improve deep learning for software analytics: (a case study with code smell detection)	[235]
A convolutional attention network for extreme summarization of source code	[6]
Automating intention mining	[99]
500+ times faster than deep learning: A case study exploring faster methods for text mining stackoverflow	[145]
On the Value of Oversampling for Deep Learning in Software Defect Prediction	[234]
Why and what happened? Aiding bug comprehension with automated category and causal link identification	[252]
Old but Gold: Reconsidering the value of feedforward learners for software analytics	[241]

In summary, we find that the general pattern in the literature is that while there is much new work on deep learning, there is not so much work on comparing these new methods to older, simpler approaches. This is a concern since, as shown in this paper, those older simpler methods, being faster, are more amenable to hyper-parameter optimization, and can yield better results when tuned. As we stated above, 48% of papers do *not* compare against simpler, non-deep learning methods, and only 9% of papers apply hyper-parameter optimization in a non-trivial manner to their approach, possibly due to the computational infeasible nature of doing so with more complex methods.

2.3 Hyper-parameter optimization

Hyper-parameters dictate the specifics of how a learning algorithm operates. Classically, the example of k in $k-$nearest neighbors is used to demonstrate the difference between hyper-parameters and parameters (i.e., model parameters that are learned). Within the neural networks realm, there are more hyper-parameters to work with: the batch size, the learning rate and other hyper-parameters of the optimizer used, the method of initialization, etc. Hyper-parameter optimization is a systematic technique of finding a set of hyper-parameters that yield a high performance for a given machine learning model. Precisely, we seek to perform the following optimization:

$$\begin{aligned} &\min_{x \in \mathbb{R}^n} f(x; \theta, D_{val}) \\ &\text{s.t.} \theta = \arg\max_{\theta} f(x; \theta, D_{train}) \end{aligned} \tag{2.1}$$

where D_{val} and D_{train} are the validation and training datasets respectively and f.

Within deep learning, the community has taken to theoretical and empirical methods of setting hyper-parameters. For example, [88] derives a theoretical reason to initialize the weights using a scaled Gaussian distribution. Similarly, [191] discusses empirical methods to manually tune hyper-parameters based on the performance of the current system. However, more automated methods of hyper-parameter optimization have now been developed and are also used often.

(2.1) is challenging to solve because f is often a black-box and computationally expensive function. There is significant prior work in hyper-parameter optimization [1, 9, 25, 26, 48, 66, 67, 125]. Indeed, as learning systems become more intricate, it is crucial that we eke out the most performance. However, this is a non-trivial problem, as evidenced by the long line of research in this direction.

The simplest form of hyper-parameter search is random search, which tries n randomly chosen hyper-parameter configurations. Opentuner [8] is a multi-armed bandit meta-technique with a sliding window that incorporates an exploration/exploitation trade-off based on the number of times a specific technique is used. It combines DE, a greedy bandit mutation technique, and hill-climbing methods.

Bayesian Optimization (BO) has emerged as the most popular technique for HPO. Bergstra et al. [26] propose the Tree of Parzen Estimators (TPE) algorithm. Rather than model $p(y|x)$, TPE models $p(x|y)$ as

$$p(x|y) = \begin{cases} l(x) & y < y^* \\ g(x) & y \geq y^* \end{cases}$$

where y^* is chosen so that $p(y < y^*) = \gamma$ for some quantile γ. The functions $l(x)$ and $g(x)$ are kernel density estimates. TPE optimizes the EI, which they show is equivalent to maximizing $l(x)/g(x)$. Snoek et al. [193] use Gaussian Process (GP) models as the surrogate function in Bayesian optimization. They use Expected Improvement (EI) as the acquisition function. Similarly, Hernández-Lobato et al. [91] use predictive entropy search (PES) as the acquisition function. Swersky et al. [201] exploit iterative training procedures in their Bayesian optimization framework, which they call freeze-thaw Bayesian optimization. Snoek et al. [194] use neural networks for modeling distributions over functions that yields an approach that scales linearly over data size (rather than cubically as in GP-based Bayesian optimization). BOHB [67] combines the BO-based TPE with HyperBand [125], replacing the initial random configurations with a model-based search. Notably, BOHB uses a single multi-dimensional KDE instead of hierarchical single-dimensional KDEs used by TPE. The authors of HEBO [48] note that (i) even simple HPO problems can be non-stationary and heteroscedastic (ii) different acquisition functions can conflict. To tackle the former, they use the Box-Cox [30] and Yeo-Johnson [244] output transformations and the Kumaraswamy [118] input transformation. It also uses NSGA-II to optimize a multi-objective acquisition function. TuRBO [58] assumes the hyper-parameter to performance mapping is Lipschitz, and generates pseudo-points to improve convergence of vanilla BO. It also uses an ensemble of learners to predict performance, and if the prediction is poor, uses it to instead update the GP model. Finally, we mention PriorBand [136], which incorporates an expert's prior beliefs about good configurations, but maintains good performance even if that prior is bad.

We defer to [69] and [29] for recent reviews on hyper-parameter optimization techniques. There is also a long line of work studying neural architecture search, which aims to find optimal architectures for a dataset. We refer the reader to White et al. [222] for a comprehensive review of the field.

There is also a long (and growing) line of work studying *neural architecture search* [16, 64, 129, 130, 171, 197, 229, 253]. This field studies systematic approaches to find optimal neural network architectures for a dataset. For example, [129] start with small building blocks and work up to a hierarchical, more complex architecture. [130] frame the problem in a way that allows them to use gradient descent to find an optimal architecture. We do not use neural architecture search in this work, but mention it due to its relevance to hyper-parameter optimization; indeed, neural architecture search focuses on one specific subset of hyper-parameters–the network architecture–without focusing on others such as batch size, optimizer-specific hyper-parameters such as learning rate, preprocessing algorithms used, etc. Moreover, neural architecture search approaches can be slow and take days on a modern GPU to find an optimal architecture, for one dataset and for one run. Our methods use statistical analysis to check that our approach is better by a statistically significant margin, and therefore neural architecture search could not be incorporated in our overall framework. Nevertheless, we direct the reader to [65] for a recent literature review of neural architecture search approaches.

Several benchmarks have been proposed for hyper-parameter optimization: notable ones include YAHPO Gym [166], HPO-B [10], and HPOBench [63].

2.4 Flat minima and generalization

The idea of flat minima was first studied by Hochreiter & Schmidhuber [94]. In particular, they define "flat minima" as large connected regions where the weights are $\epsilon-$optimal. Hochreiter & Schmidhuber [95] intuit that because sharper minima require higher precision, flatter minima require less bits to describe. They use this intuition to show that flat minima correspond to minimizing the number of bits required to describe the weights of a neural network. This notion of minimum description length (MDL) [78, 174] was also exploited early on by Hinton & Van Camp [92]. Flat minima were revisited by Chaudhari et al. [39], who noted that minima with low generalization error have a large proportion of their eigenvalues close to zero. They then construct a modified Gibbs distribution corresponding to an energy landscape f, and minimize the negative local entropy of this modified distribution, and approximate the gradient via stochastic gradient Langevin dynamics (SGLD) [220]. However, their assumptions were, admittedly unrealistic. Keskar et al. [112] show that when using large batch sizes, optimizers converge to sharp minima, which are characterized by many large positive eigenvalues of the Hessian. Further, they define the notion of sharpness as a generalization measure as the robustness to adversarial perturbations in the parameter space:

$$\zeta(\boldsymbol{w};\epsilon) = \frac{\max\limits_{|\boldsymbol{v}|\leq\epsilon(|\boldsymbol{w}|+1)} f(\boldsymbol{w}+\boldsymbol{v}) - f(\boldsymbol{w})}{1+f(\boldsymbol{w})} \tag{2.2}$$

and compute this using 10 iterations of L-BFGS-B [36] with $\epsilon = \{10^{-3}, 5\cdot 10^{-4}\}$. In particular, the above is closely related to the largest eigenvalue of $\nabla^2 f(\boldsymbol{w})$. This notion of sharpness was also endorsed by Jiang et al. [103], who performed a large-scale study of many complexity measures on two datasets, with 2,187 convolutional networks. In particular, they endorse the following metrics for generalization: (i) variance of gradients (ii) squared ratio of magnitude of parameters to magnitude of perturbation, à la Keskar et al. [112] (iii) path norm [154] (iv) VC-dimension (inversely correlated).

A line of work in physics [17, 18] showed that in the discrete weight scenario (a much more difficult problem), isolated minima were rare, but there existed accessible, dense regions of subdominant minima, and that these were robust to perturbations and generalized better. These authors devised algorithms explicitly designed to search for nonisolated minima. In the continuous weight space, nonisolated minima correspond to flat minima. Dziugaite & Roy [61] obtain nonvacuous generalization bounds for deep overparameterized neural networks using the PAC-Bayes framework [141]. Neyshabur et al. [153] showed that increasing the number of hidden units (which in turn, increases the number of trainable parameters) can lead to a decrease in generalization error with the same training error. Neyshabur et al. [154] showed that sharpness as computed by (2.2) is not sufficient to capture the generalization behavior (but noting that "combined with the norm, sharpness does seem to provide a capacity measure"), and advocate for expected sharpness in the PAC-Bayesian framework, similar to Dziugaite & Roy [61]. They show that plots of expected sharpness versus KL divergence in PAC-Bayes bounds for varying dataset sizes capture generalization well. Li et al. [124] showed that the sharpness of the loss surface correlates well with the generalization error. In their seminal paper, Jastrzebski et al. [102] showed that SGD is a Euler-Maruyama discretization of a stochastic differential equation whose dynamics are influenced by the ratio of learning rate to batch size (which they call "stochastic noise"), and that SGD finds wider minima with higher stochastic noise levels than sharper minima. Wu & Su [228] study the flat minima hypodissertation through the lens of dynamical stability, and show that SGD will escape from overly sharp (measured by the Frobenius norm of the Hessian), low-loss areas exponentially fast. We note that they use the associate empirical Fisher matrix (AEFM) as an approximation for the Hessian, which holds for low empirical risk (and converges to the Hessian, see Kunstner et al. [119]).

In search of flatter loss surfaces, Seong et al. [185] propose the use of non-monotonic learning rate schedules. They advocate for large learning rates, which enable the optimization algorithm to escape sharp minima, and descend into flatter minima. This paper proves a significant theoretical result that provides intuition on why flat minima generalize better. Without stating the proof, we will reiterate that theorem below:

Theorem 2.1. *Suppose* $\left|\mathcal{L}(W+\delta^W;x)-\mathcal{L}(W;x)\right| < \epsilon$ *holds for any* δ^W *such that* $\left\|\delta^W\right\|_F < \delta$ *for some* $\delta > 0, \epsilon > 0$*. Then* $|\mathcal{L}(W;x+\delta^x)-\mathcal{L}(W;x)| < \epsilon$ *holds for any* δ^x *such that* $\frac{\|\delta^x\|}{\|x\|} < \frac{\delta}{\sigma_{max}(W)}$.

This result shows that a translation in weight space is equivalent to a translation in data space. Critically, this explains the generalization of flat minima: if the test data differs from the training data by some δ^x, then under some conditions, that deviation can be translated into weight space. Since the minima is flat in weight space, this means the loss value does not change much, and so the network still performs well on the test set.

The seminal works of Dauphin et al. [51] and Choromanska et al. [44] showed both theoretically and empirically that local minima are more likely to be located close to the global minimum. In particular, Dauphin et al. [51] showed using the perspectives of random matrix theory (via the eigenvalue distribution of Gaussian random matrices [224]), statistical physics (via the analysis of critical points in Gaussian fields by Bray & Dean [32]), and neural network theory [182] that saddle points are exponentially less likely than local minima.

Most recently, Wen et al. [221] showed that sharpness is neither necessary, nor sufficient for generalization, by studying some simple architectures, showing that generalization depends on the data distribution as well as the architecture; for example, merely adding a bias to a 2-layer MLP makes generalization impossible for the XOR dataset.

Of course, it is not possible to discuss generalization in deep learning without discussing the results of Zhang et al. [249], who showed that deep learners can fit with zero training error on random labels using an architecture that generalizes well when fit to the correct labels. Bartlett et al. [22] and Harvey et al. [87] found that the VC-dimension for deep ReLU networks is $\mathcal{O}(WL\log W)$ and $\Theta(WU)$ respectively, where W is the number of parameters, L the number of layers, and U the number of units. Hardt et al. [86] show that stochastic gradient methods are uniformly stable[1], which implies generalization *in expectation*.

2.5 Convex optimization

In this section, we summarize some results from convex optimization, which will be useful in our discussion of hyper-parameter optimization.

Lemma 1. *Let $f : \mathbb{R}^d \to \mathbb{R}$ be a differentiable function. Then, $\mu-$strong convexity implies:*

(i) (Polyak-Łojasiewicz (PL) inequality)

$$\frac{1}{2}\|\nabla f(x)\|^2 \geq \mu(f(x) - f(x^*))$$

[1] An algorithm A is $\epsilon-$uniformly stable if $\forall S, S' \in Z$ for some space Z, such that the datasets S and S' differ by at most one example, $\sup_z \mathbb{E}_A[f(A(S); z) - f(A(S'); z)] \leq \epsilon$

(ii)

$$\|\nabla f(x) - \nabla f(y)\| \geq \mu \|x - y\|$$

(iii)

$$\left(\nabla f(x) - \nabla f(y)\right)^T (x - y) \leq \frac{1}{\mu}\|\nabla f(x) - \nabla f(y)\|^2$$

Proof. (i) Strong convexity implies

$$f(y) \geq f(x) + \nabla f(x)^T (y - x) + \frac{\mu}{2}\|y - x\|^2$$

Minimizing with respect to y yields the result.

(ii) Strong convexity gives us:

$$\left(\nabla f(x) - \nabla f(y)\right)^T (x - y) \geq \mu \|x - y\|^2 \tag{2.3}$$

Applying Cauchy-Schwarz inequality gives us:

$$\begin{aligned}\|\nabla f(x) - \nabla f(y)\|\|x - y\| &\geq \left(\nabla f(x) - \nabla f(y)\right)^T (x - y) \\ &\geq \mu \|x - y\|^2\end{aligned}$$

where the last step comes from (2.3).

(iii) Set $\phi_x(z) = f(z) - \nabla f(x)^T z$. It is easy to see that ϕ_x is also $\mu-$strongly convex. Applying the Polyak-Łojasiewicz inequality to $\phi_x(z)$ with $z^* = x$,

$$\begin{aligned}\left(f(y) - \nabla f(x)^T y\right) - \left(f(x) - \nabla f(x)^T x\right) &= \phi_x(y) - \phi_x(z^*) \\ &\leq \frac{1}{2\mu}\|\nabla \phi_x(y)\|^2 \\ &\leq \frac{1}{2\mu}\|\nabla f(y) - \nabla f(x)\|^2\end{aligned}$$

Swapping x and y in the above,

$$\left(f(x) - \nabla f(y)^T x\right) - \left(f(y) - \nabla f(y)^T y\right) \leq \frac{1}{2\mu}\|\nabla f(x) - \nabla f(y)\|^2 \tag{2.4}$$

Adding (2.4) and (2.4) yields the result.

□

Lemma 2. *If $f : \mathbb{R}^n \to \mathbb{R}$ is smooth and $\mu-$strongly convex, then*

$$\frac{1}{2\mu}\|\nabla f(x)\|^2 \geq f(x) - f(x^*) \geq \frac{\mu}{2}\|x - x^*\|^2$$

Proof. The first part is the Polyak-Łojasiewicz inequality. The second part follows from the definition of strong convexity and setting $y = x, x = x^*$ and using $f(x^*) \geq \min_y f(y)$. $\square$

Theorem 2.2 (Smooth and strongly convex gradient descent)**.** *Suppose $f : \mathbb{R}^n \to \mathbb{R}$ be $\beta-$smooth and $\mu-$strongly convex. Then with the gradient descent update rule*

$$x_{k+1} = x_k - \frac{1}{\beta}\nabla f(x_k)$$

where $\frac{1}{\beta}$ is the learning rate, we have

$$f(x_k) - f(x^*) \leq \left(1 - \frac{\mu}{\beta}\right)^k (f(x_0) - f(x^*))$$

Consequently, we require $\frac{\beta}{\mu}\log\frac{f(x_0)-f(x^*)}{\epsilon}$ iterations to find an $\epsilon-$optimal point.

Proof. From the above results,

$$f(x_{k+1}) \leq f(x_k) - \frac{1}{2\beta}\|\nabla f(x_k)\|^2$$

and Lemma 2 gives us

$$\|\nabla f(x_k)\|^2 \geq 2\mu(f(x_k) - f(x^*))$$

Therefore,

$$\begin{aligned}
f(x_{k+1}) - f(x^*) &\leq f(x_k) - f(x^*) - \frac{1}{2\beta}\|\nabla f(x_k)\|^2 \\
&\leq f(x_k) - f(x^*) - \frac{\mu}{\beta}(f(x_k) - f(x^*)) \\
&= \left(1 - \frac{\mu}{\beta}\right)(f(x_k) - f(x^*)) \\
f(x_k) - f(x^*) &\leq \left(1 - \frac{\mu}{\beta}\right)^k (f(x_k) - f(x^*)) \\
&\leq \exp\left(-\frac{\mu k}{\beta}\right)(f(x_0) - f(x^*))
\end{aligned}$$

Therefore we need $k = \frac{\beta}{\mu}\log\frac{f(x_0)-f(x^*)}{\epsilon}$ iterations for $\epsilon-$optimality. $\square$

Note that strong convexity guarantees optimality. $\beta-$smoothness can only assure $\epsilon-$criticality. This implies the existence of global minima.

Strong convexity is a necessary condition for learnability, as along with $\beta-$smoothness, it can be shown that such problems are learnable [186]. Additionally, strong convexity provides a quadratic lower bound on the growth of the loss function, which implies that the convexity condition will never be violated in the local domain of the function in the context of deep regression with regularization.

CHAPTER

3

CASE STUDIES

To understand the significance of BOUNDED, we need to look at how hyper-parameter optimization (HPO) is currently conducted in software engineering. This chapter reviews a sample of that work, and provides a background on the tasks that we study and prior work exploring those tasks. Each part of this chapter is divided into:

- A problem statement
- The prior state of the art in handling that problem with HPO
- Open issues with that approach. These open issues will motivate the need for BOUNDED.

3.1 Defect Prediction

Every quality assurance decision is associated with a human and resource cost to the developer team. It is impractical and inefficient to distribute equal effort to every component in a software system [33]. Hence, it is common to match the quality assurance (QA) effort to the perceived criticality and bugginess of the code for managing resources efficiently.

Table 3.1: Software projects data used in the defect prediction case study [1]. For an explanation of attributes used in this data, see Table 3.2.

Project	Train versions	Test versions	Training Buggy %	Test Buggy %
ivy	1.1, 1.4	2.0	22	11
lucene	2.0, 2.2	2.4	53	60
poi	1.5, 2.0, 2.5	3.0	46	65
synapse	1.0, 1.1	1.2	20	34
velocity	1.4, 1.5	1.6	71	34
camel	1.0, 1.2, 1.4	1.6	21	19
jEdit	3.2, 4,0, 4.1, 4.2	4.3	23	2
log4j	1.0, 1.1	1.2	29	92
xalan	2.4, 2.5, 2.6	2.7	38	99
xerces	1.2, 1.3	1.4	16	74

Table 3.2: Static attributes for the classes seen in the Table 3.1 data.

Attribute	Description (with respect to a class)
wmc	Weighted methods per class [42]
dit	Depth of inheritance tree [42]
noc	Number of children [42]
cbo	Coupling between objects [42]
rfc	Response for a class [42]
lcom	Lack of cohesion in methods [42]
lcom3	Another lcom metric proposed by Henderson-Sellers [90]
npm	Number of public methods [20]
loc	Number of lines of code [20]
dam	Fraction of private or protected attributes [20]
moa	Number of fields that are user-defined types [20]
mfa	Fraction of accessible methods that are inherited [20]
cam	Cohesion among methods of a class based on parameter list [20]
ic	Inheritance coupling [202]
cbm	Coupling between methods [202]
amc	Average method complexity [202]
ca	Number of classes depending on a class [202]
ce	Number of classes a class depends on [202]
max_cc	Maximum McCabe's cyclomatic complexity score of methods [142]
avg_cc	Mean of McCabe's cyclomatic complexity score of methods [142]

Defect prediction models [73] are one way to take a look at the incoming changes and focus inspection effort on specific modules (ignoring others). There is much industrial interest in such predictors: a survey of 395 practitioners from 33 countries and five continents [213] found that over 90% of the respondents were willing to adopt defect prediction techniques. Such predictors can save labor compared with traditional manual methods. For a telecommunications company, Misirli et al. [148] built a defect prediction model that predicted 87% of code defects and decreased inspection efforts by 72% (and reduced post-release defects by 44%). Kim et al. [114] applied a defect prediction model to API development process at Samsung. Their models could predict the bug-prone APIs with reasonable accuracy (0.68 F1 scores) and reduce the resources required for executing test cases. These defect predictors require very simple code features and hence are very cheap to build (compared to using, say, a static code analysis tool with an expensive license) And while these models use simple attributes, they are still remarkably effective. Rahman et al. [170] compared (a) static code analysis tools FindBugs, Jlint, and PMD with (b) defect predictors (which they called "statistical defect prediction") built using logistic regression. No significant differences in cost-effectiveness were observed.

However, prior work has the key issue of *solely* using deep learning as automated feature extractors, such as Wang et al. [217]. Numerous prominent papers [152, 163, 198, 231, 246] say that deep learning offers "automated feature engineering" that removes the need for intricate preprocessing. In that view, the first few layers of a deep learning network automatically re-weight/ignore important/irrelevant inputs [152]. While it is certainly true that deep neural networks automatically learn hierarchical representations of data, which can often correlate well with human-observed hierarchies of features [247], we argue that this does not excuse practitioners from performing *no* feature engineering *a priori*. In our method (see Chapter 4), we will show that a principled approach to designing a deep learning system outperforms the prior work, orders of magnitude faster.

3.2 Bugzilla issue lifetime prediction

When programmers work on repositories, predicting issue close time has multiple benefits for the developers, managers, and stakeholders since it helps:

- Developers prioritize work;
- Managers allocate resources and improve consistency of release cycles;
- Stakeholders understand changes in project timelines and budgets.
- It is also useful to predict issue close time when an issue is created; e.g. to send a notification if it is predicted that the current issue is an easy fix.

We explore issue close time prediction for two reasons. Firstly, it is a well-studied problem [3, 76, 82, 83, 113, 121, 139, 173, 211]. Secondly, recent work has proposed a state-of-the-art deep learning approach to issue close time prediction (see the DeepTriage deep learning systems from COMAD'19, described later in this paper [137]).

3.2.1 Traditional Algorithms for Predicting Issue Close Time

Most large software systems have a system to track bugs, or issues, in the product. These issues typically go through the same lifecycle, in which they transition across various states, including UNCONFIRMED and CLOSED, while also being assigned final states such as WONTFIX [219].

To find prior work on predicting issue close time, we searched for papers in the last ten years (since 2010) in Google Scholar using keywords "bug fix time", "issue close time", and "issue lifetime". Then, we filtered them according to the criterion that they must be published in a top venue according to Google Scholar metrics for Software Systems[1]. Finally, using engineering judgement, we added in systems that were recommended by reviewers of a prior draft of this paper. That search found several noteworthy systems:

- Guo et al. [82] use logistic regression on a large closed-source project (Microsoft Windows), to predict whether or not a bug will be fixed. Using regression analysis, they identified the factors that led to bugs being fixed or not fixed.

- Giger et al. [76] use decision trees to predict the bug-fix time for Mozilla, Eclipse, and GNOME projects. They divided their target class into two labels: fast and slow, to get a binary classification problem, and used the area under the ROC curve (AUC) metric as their evaluation criteria.

- Marks et al. [139] also used decision trees, but instead, use an ensemble method, i.e., random forests, on Eclipse and Mozilla data. Their motivation for using random forests, apart from the better performance as compared to standard decision trees, is the ability to extract the relative importance of features in the input data. They report accuracy scores of 63.8% and 67.2% on the Mozilla and Eclipse repositories respectively.

- At MSR'16, Kikas, Dumas, and Pfahl [113] built time-dependent models for issue close time prediction using Random Forests with a combination of static code features, and non-code features to predict issue close time with high performance

- More recently, Habayeb et al. [83] reported in IEEE TSE'17 a prediction system based on hidden Markov chains. Like Giger et al. [76], they divided their target

[1] `https://scholar.google.com/citations?view_op=top_venues&hl=en&vq=eng_softwaresystems`

labels into fast and slow fix-times and experimented with different values of the number of hidden states of the hidden Markov model.

Based on the above, we assert that the two prior state-of-the-art non-neural methods in area used random forests and logistic regression. Hence we will we use these two systems as part of the study.

3.2.2 Deep Learning and Issue Close Time

As to deep learning and issue close time prediction, two contenders for "state-of-the-art" are DASENet [121] and DeepTriage[137]. The DASENet paper asserts that their algorithm defeats DeepTriage but after much effort, we could not reproduce that result[2]. Hence, for this study, we use DeepTriage since:

- It is a state-of-the-art deep learner that performs for lifetime prediction.
- It has been very recently published (2019);
- Its reproduction package allowed us to run that code on our machines.
- It uses datasets commonly used in the literature (Aside: we were tempted to use the dataset provided by Vieira et al. [211] for our deep learning baseline. However, their lack of prior benchmarks meant we could not provide a comparison to demonstrate the efficacy of our approach.)

From a technical perspective, DeepTriage is Mani et al. [137]'s extension of bidirectional LSTMs with an "attention mechanism". A Long Short-Term Memory (LSTM) [96] is a form of recurrent neural network that has additional "gate" mechanisms to allow the network to model connections between long-distance tokens in the input. Bidirectional variants of recurrent models, such as LSTMs, consider the token stream in both forward and backward directions; this allows for the network to model both the previous and the following context for each input token.

Attention mechanisms[15] use learned weights to help the network "pay attention" to tokens that are more important than others in a context. Formally, an attention mechanism maps a query and key-value pairs to an output, where the queries, keys, and values are all vectors. These are obtained as projections of the inputs into different vector

[2]We found that the reproduction package published with DASENet has missing files. We tried contacting the authors of that paper, without success.

spaces. The output is computed as a weighted sum of the values, where the weights are learned during training. This allows the network to handle long-range dependencies and context efficiently. The most popular form of attention today is the scaled dot-product attention:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

where d_k is the dimension of the queries and keys.

Prior to running DeepTriage, its authors recommend using a standard set of pre-processing techniques: pattern matching to remove special characters and stack traces, tokenization, and and pruning the corpus to a fixed length. Beyond these steps, they rely on the deep learner to perform automated feature engineering.

Similar to defect prediction, we argue that the reliance on automated representation learning leads to over-engineered systems that are needlessly computationally expensive. This also has the (major) side-effect of accelerating climate change [54, 56, 106]. We will show that an extremely simple method outperforms the prior state-of-the-art, and strongly advise that SE researchers try out simpler methods before using computationally expensive ones.

3.3 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 [84, 205], they have been adopted in open source projects and in industry [13, 24, 164, 179, 209, 245]. 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 [45, 105]; 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 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. There have been many recent attempts to address the problem. Some researchers have

Table 3.3: The features studied in prior work [214, 232, 233]. The "Golden Features" are in bold.

Feature type	Feature
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**
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
Warning characteristics	**warning pattern** **warning type** **warning priority** warning rank, warnings in method, file **package**
File history	latest file, package modification file, package staleness **file age, creation** deletion revision **developers**
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
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)
Warning history	**warning lifetime by revision**, by time warning modifications, open revision
File Characteristics	file type, name package name

proposed new SA tools that use more sophisticated, but costly, static analysis techniques (e.g. Infer [37], NullAway [19]). Regardless, the large volume of false alarms remains unsolved; despite their improvements, these tools still produce many false alarms [206]. Other attempts to prune false alarms include the use of test case generation to validate the presence of a bug at the source code location indicated by the warning [109]. As generating test cases is expensive, these techniques may face issues when scaling up to larger projects, limiting their practicality.

Table 3.4: Summary of dimensionality of eight projects. The projects are sorted in the ascending order of their intrinsic dimensionality.

Project	Original Dimensionality	Intrinsic Dimensionality	Instance number
commons	127	3.45	22
derby	292	3.45	458
tomcat	284	3.64	184
lucene-solr	278	3.83	30
ant	329	4.62	35
cassandra	265	5.47	17
maven	22	5.47	4
jmeter	287	5.79	18

3.3.1 Early results (Golden features)

A promising line of research is the use of machine learning techniques to identify and prune false alarms [85, 89, 127, 178, 214, 232, 233]. By framing the problem as a binary classification problem, these tools use a machine learner that classifies warnings as actionable warnings or as false alarms. These techniques use features extracted from code analysis and metrics computed over the code and warning's history in the project. Figure 3.1 demonstrates the use of a machine learner for pruning false alarms. 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.

Recently, Wang et al. [214] performed a systematic literature review to collect and analyze over 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 23 Golden Features that are most important

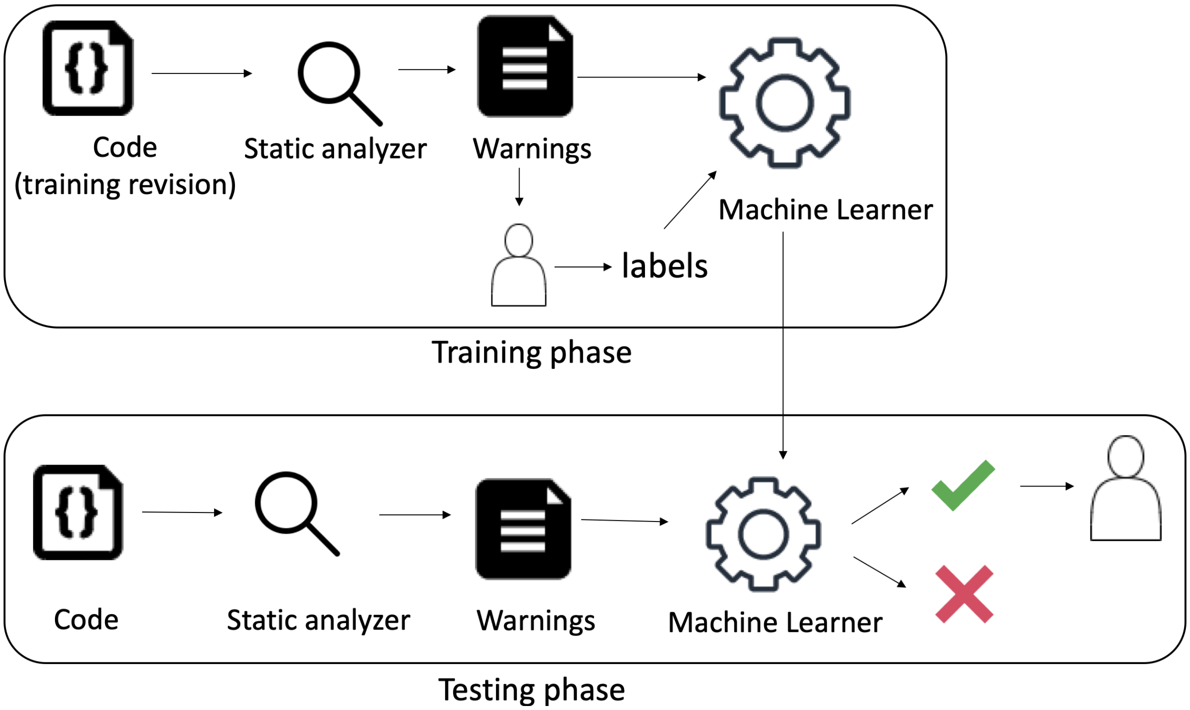


Figure 3.1: To detect actionable warnings, a machine learner is trained based on warnings from a training revision. Each warning is annotated with a label. When deployed on the latest revision, only warnings classified as actionable warnings by the machine learner are presented to the developers.

Table 3.5: Initial results with SVMs on static analysis data

Dataset	Precision	AUC	False alarm rate	Recall
cassandra	0.67	0.33	0.25	0.67
jmeter	0.5	0.36	0.14	0.17
commons	0.67	0.52	0.57	0.62
lucene-solr	0.56	0.70	0.36	0.71
ant	0	0	0	0
tomcat	0.52	0.41	0.19	0.32
derby	0.2	0.64	0.12	0.08
median	0.52	0.41	0.19	0.32

for achieving effective performance. Table 3.3 lists these features.

3.3.2 Further Results

Yang et al. [232] further analyzed the Golden Features using the data collected by Wang et al. [214]. 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 low cost. In contrast, deep learners were effective but were relatively more costly to train.

For each project in their experiments, one revision (training revision) is 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.

3.3.3 Issues in Prior Results

Subsequently, Kang et al. [110] replicated the Yang et al. [232] study but found subtle methodological issues in the data collected by Wang et al. [214] that led to overoptimistic results.

Firstly, they 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. addressed this issue by reimplementing 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 that a substantial number of warnings appear 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 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 and after filtering out duplicate warnings and warnings with 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 4.8. The results of the SVM are shown in Table 3.5.

From the results, we observe that the performance of the SVMs using the Golden Features declined after addressing the methodological issues. These issues do not show that machine learning techniques do not work in filtering false alarms; in their analysis, Kang et al. found that the Golden Features still had predictive power. Moreover, Kang et al.'s analysis was only done using the best-performing predictor pointed out by prior works. Specifically, only the off-the-shelf SVMs were used. Here, we build upon the features identified in the prior studies [85, 89, 127, 178, 214, 232, 233]; but instead of only using off-the-shelf machine learners, we tap on the power of more advanced and specialized learners, data engineering methods and hyperparameter optimization steps.

3.4 Code smell detection

[23] proposed the idea of *code smells*, which are "certain structures in the code that suggest (sometimes they scream for) the possibility of refactoring". They introduce 22 different code smells, of which we study four:

- **Feature envy** refers to a condition when a method accesses the data of another object more than its own. This is a sign that the method should be a part of the other class instead.
- **Large class** refers to when a class tries to do too much. This can be identified by having too many instance variables, which may lead to duplicated code.
- **Long method** refers to methods that are too long and cause difficulty in comprehending their functionality and scope. Beck et al. [23] state that shorter methods

facilitate "explanation, sharing, and choosing", and suggest being aggressive about decomposing methods.

- **Misplaced class** occurs when classes are improperly distributed, and should be moved to the correct package.

We use the same four code smells as prior work [131] to make a fair comparison. Detecting code smells using machine learning is widely studied in the SE literature. Azeem et al. [14] present a systematic literature review of the field, but we discuss some here. Schumacher et al. [184] study the god class smell in a commercial environment, and the efficacy of automated metric-based systems to detect it. Contrary to other researchers' beliefs, Yamashita & Moonen [230] conclude that code smells have a minor impact on software maintainability. Fontana et al. [70] study four code smells: god class, data class, long method, and feature envy using twelve subjects from Qualitas Corpus of [204]. Sahin et al. [180] use bilevel optimization to study seven code smells over nine open-source projects. Palomba [162] study long method detection on three software systems. Fontana et al. [71] conclude that machine learning can effectively be applied for code smell detection, achieving high accuracy. They show through an extensive comparison, that J48 and random forests were the best at detecting code smells. Pecorelli et al. [165] study the role of different class imbalance mitigating solutions for 11 code smells on 13 software systems. They conclude that balancing classes does not significantly improve performance.

Liu et al. [131] use deep learning to detect code smells. Studying the same four code smells as above, they use different deep learning architectures for each code smell. For all code smells, they preprocess text features using word2vec [146, 147]. For feature envy, they use convolutional neural networks; for large class detection they use LSTMs; for long method, they use feedforward networks; and for misplaced classes, they use only the word2vec model. Based on our literature review, we assert that Liu et al. [131] is the prior state-of-the-art in code smell detection. Further, it is published in a top venue as listed by Google Scholar metrics (TSE) and is recent (the latest revision is from September 2021).

We believe it should be possible to detect all code smells using a single model type. As such, our method will use the same feedforward network structure as the above case studies for code smell detection. We will show that except in the case where Liu et al. [131] also use feedforward networks, we outperform their methods on all types of code smells.

3.5 Automated microservice partitioning

To fully exploit the cloud, systems have to be rewritten as "microservices" comprising multiple independent, loosely coupled pieces that can scale independently. Microservice architectures have many advantages [4, 227] such as technology heterogeneity, resilience (i.e., if one service fails, it does not bring down the entire application), and ease of deployment. Therefore, it is of significant business interest to port applications to the cloud under the microservice architecture. For example, Netflix states that the elasticity of the cloud and increased service availability are two of the primary reasons it switched to a microservice architecture[3]. They use the cloud for distributed databases, big data analytics, and business logic. Further, they mention that moving to the cloud reduced their averaged cost.

Daya et al. [52] report on the long queue of applications waiting to be ported to the cloud. Once ported to cloud-based microservices, code under each microservice can be independently enhanced and scaled, providing agility and improved speed of delivery. But converting traditional enterprise software to microservices is problematic. Such traditional code is *monolithic*, i.e., all the components are tightly coupled, and the system is designed as one large entity. Thus, some *application refactoring* process is required to convert the monolithic code into to a set of small code clusters.

When done manually, such refactoring is expensive, time-consuming, and error-prone. Hence, there is much current interest in automatically refactoring traditional systems into cloud services.

But when we try to use AI tools for this task, we often encounter similar problems. Specifically, there are so many algorithms and partitioning goals:

- Table 3.6 lists the various coupling and cohesion goals used by prior work on partition generation. Note that in column one, any goal marked with "[-]" should be *minimized* and all other goals should be *maximized*.

- See also Table 3.7 which lists some of the partitioning tools, as well as their control parameters. In the table, the core idea is to understand what parts of the code call each other (e.g., by reflecting on test cases or use cases), then cluster those parts of the code into separate microservices. These algorithms assess the value of different partitions using scores generated from the Table 3.6 metrics.

[3] https://about.netflix.com/en/news/completing-the-netflix-cloud-migration

Table 3.6: Yedida et al. [239] report five widely used metrics to assess the quality of microservice partitions. In this table, [-] denotes metrics where *less* is better and [+] denotes metrics where *more* is better. For details on how these metrics are defined, see §4.11.5.

Metric	Name	Description	Goal
BCS [-] ([107]	Business context sensitivity	Mean entropy of business use cases per partition	If **minimized** then more business cases handled locally
ICP [-] ([107]	Inter-partition call percentage	Percentage of runtime calls across partitions	If **minimized** then less traffic between clusters
SM [+] ([104]	Structural modularity	A combination (see §4.11.5) of cohesion and coupling defined by Jin et al. [104]	If **maximized** then more self-contained clusters with fewer connections between them
MQ [+] ([149]	Modular quality	A ratio involving cohesion and coupling, defined by Mitchell & Mancoridis [149]	If **maximized** then more processing is local to a cluster
IFN [-] ([149]	Interface number	Number of interfaces needed in the microservice architecture	If **minimized** then fewer calls between clusters
NED [-] ([107]	Non-extreme distribution	Number of partitions with non-extreme values	If **minimized**, fewer "boulders" (large monolithic partitions) or "dust" (partitions with only one class)

In such a rich space or options, different state-of-the-art AI approaches may generate different recommendations. Yedida et al. [239] concluded that looking at prior work, we seem to have a situation where analysts might run several partitioning algorithms to find an algorithm that performs the best based on their business requirements. For example, analysts might prefer Mono2Micro [107, 108] over others considering the clean separation of business use cases where as they might prefer CoGCN [55] and FoSCI [104] for low coupling. As stated, this is hardly ideal since these partitioning algorithms can be slow to execute. Accordingly, this paper seeks optimization methods that support partitioning for a wide range of data sets and goals.

Table 3.7: Hyper-parameter choices studied for automated microservice partitioning All the methods take runtime traces (or use cases) as their input. These methods return suggestions on how to build partitions from all the classes seen in the traces. For further details on these partitioning algorithms, see §3.5.

Algorithm	**Hyper-parameter**	**(min, default, max)**
Mono2Micro [107, 108]	Number of clusters	(2, 5, 10)
FoSCI [104]	Number of clusters	(5, 5, 13)
	Number of iterations to run NSGA-II	(1, 3, 6)
	Population size for NSGA-II	(30, 100, 200)
	Parent size to use in NSGA-II	(10, 30, 50)
MEM [140]	Maximum partition size	(17, 17, 17)
	Number of partitions	(2, 5, 13)
Bunch [149]	Number of partitions	(3, 5, 13)
	Initial population size for hill climbing	(2, 10, 50)
	Number of neighbors to consider in hill-climbing iterations	(2, 5, 10)
CoGCN [55]	Number of clusters	(2, 5, 13)
	Loss function coefficients $\alpha_1, \alpha_2, \alpha_3$	(0, 0.1, 1)
	Number of hidden units in each layer, h_1, h_2	(4, 32, 64)
	Dropout rate	(0, 0.2, 1)

We formalize the problem as follows. Consider classes[4] in an application A as $\mathcal{C}^A$ such that $\mathcal{C}^A = \{c_1^A, c_2^A, \ldots, c_k^A\}$, where c_i^A represents an individual class. With this, we define a partition as follows:

Definition 3.1. *A partition $\mathcal{P}^A$ on $\mathcal{C}^A$ is a set of subsets $\{P_1^A, P_2^A, \ldots, P_n^A\}$ such that*

- $\bigcup_{i=1}^{n} P_i^A = \mathcal{C}^A$*, i.e., all classes are assigned to a partition.*
- $P_i^A \neq \phi \ \forall i = 1, \ldots, n$*, i.e., there are no empty partitions.*
- $P_i^A \cap P_j^A = \phi \ \forall i \neq j$*, i.e., each partition is unique.*

Definition 3.2. *A partitioning algorithm f is a function that induces a partition $\mathcal{P}_A$ on an application with classes $\mathcal{C}^A$.*

The goal of a microservice candidate identification algorithm is to identify a function f that jointly maximizes a set of metrics $m_1, m_2, \ldots, m_p$ that quantify the quality of the

[4]This definition trivially extends to languages without classes; translation units such as functions could be used instead, for example.

partitions, i.e., given an application characterized by its class set $\mathcal{C}_A$, we aim to find:

$$\mathcal{P}_A^* = \underset{\mathcal{P} \in B_{\mathcal{C}_A}}{\arg\max} \sum_{i=1}^{p} \alpha_i m_i(\mathcal{P}; f) \tag{3.1}$$

where B_{C_A} denotes the set of all partitions of C_A and p is the number of partitions.

FoSCI [104] uses runtime traces as a data source. They prune traces that are subsets of others, and use hierarchical clustering with the Jaccard distance to create "functional atoms". These are merged using NSGA-II [53], a multi-objective optimization algorithm to optimize for the various goals. Since FoSCI relies on a somewhat older optimization algorithm (from 2002 [53]), and all the goals are given equal priority, this can lead to suboptimal results if the different goals have, for example, different scales.

Bunch [149] is based on search techniques such as hill-climbing[5] to find a partition that maximizes two specific metrics. It is well established that such greedy search algorithms can get stuck in local optima easily [177].

Mono2Micro [107, 108] collects runtime traces for different business use cases. Then, they use hierarchical clustering with the Jaccard distance to partition the monolith into a set of microservices. However, as noted by Yedida et al. [239], this approach takes as input the number of partitions, which different architects may disagree on, or may not know the value of.

CO-GCN uses the call graph (built from the code) as input to a graph convolutional neural network [116]. They develop a custom loss function that relates to the metrics being optimized for, and thus, the neural network can be seen as a black-box system that optimizes for Equation (3.1). However, while their use of a custom loss function tailored to the goals is novel, their approach has several hyper-parameters that a non-expert may not know how to set, and their study did not explore hyper-parameter optimization.

CO-GCN uses an exponential learning rate scheduling policy, where the learning decays exponentially. In our approach, we instead use the 1cycle [192] policy, which has been shown to exhibit the *super-convergence* phenomenon [192] and perform better.

CO-GCN also uses *dropout* [196], which involves removing some nodes with a probability p during training, and adjusting the weights of the model during testing. Srivastava et al. [196] argues that this enforces sparsity in the model weights, which improves the model performance and makes it more robust to noise.

CO-GCN is controlled by the settings of Table 3.7. Later in this paper, we show that

[5]Their tool offers other heuristic-based approaches as well.

there are several benefits in using hyper-parameter optimization to automatically select good subsets of these hyper-parameters. Note that (a) all our partitioning methods (CO-GCN and all the others shown above) utilize hyper-parameter optimization; (b) even better results can be obtained via augmenting HPO with a novel loss function. We will refer to that combination of HPO and loss function as `DEEPLY`.

CHAPTER

4

METHODS

This chapter describes the BOUNDED method. Recall that BOUNDED has many parts, not all of which are needed for simpler data sets. Those parts include:

- *Learner choice*: appropriate choices of learning algorithms to approximate the desired decision function;
- *Boundary engineering*: adjust the decision boundary near our data points;
- *Label engineering*: control outliers in a local region by using just a small fraction of those local labels;
- *Instance engineering*: addressing class imbalance in local regions of the data;
- These treatments are combined with *parameter engineering* to control how we build models.

But first, several parts of the chapter involve mathematical derivations. Hence, we offer the notation in the next section.

4.1 Notation

We denote a dataset in a supervised setting as (X, y), where $X \in \mathcal{X} \subseteq \mathbb{R}^{m \times n}$ and $y \in \mathbb{R}^m$. Therefore, we assume the number of examples is m and the number of features is n. It will often be necessary to refer to a specific instance in the training set. We will use a superscript with parentheses to do this, so that $X^{(i)}, y^{(i)}$ denote the ith training example.

This dissertation is primarily concerned with feedforward networks. In that context, we use L to denote the number of layers[1]. For some integer n, we use $[n] = \{1, 2, \ldots, n\}$. For a layer $l \in [L]$, we will use $g^{[l]}(\cdot)$ to denote the activation used there. If unspecified, it should be assumed that this activation is the ReLU function $g^{[l]}(x) = \max(0, x)$, except in the last layer (where either a sigmoid or softmax activation is used). The computation performed at that layer is

$$\begin{aligned}
z^{[l]} &= W^{[l]} a^{[l-1]} + b^{[l]} \\
a^{[l]} &= g^{[l]}(z^{[l]} \\
a^{[0]} &= X
\end{aligned}$$

We will use $n^{[l]}$ to denote the number of units in layer l. Therefore, $W^{[l]} \in \mathbb{R}^{n^{[l]} \times n^{[l-1]}}$ and $a^{[l]}, z^{[l]}, b^{[l]} \in \mathbb{R}^{n^{[l]}}$.

Generally speaking, the proofs in this dissertation involve the *gradients*, and not the total derivative. As such, both the notations $\nabla_x f(x)$ and $\frac{\partial f}{\partial x}$ should be understood to denote the gradient.

It will be convenient to involve the *Kronecker delta*,

$$\delta_{ij} = \begin{cases} 1 & i = j \\ 0 & i \neq j \end{cases}$$

However, sometimes, the Kronecker delta notation becomes cumbersome, and we will resort to the Iverson notation $[i = j]$, which has the equivalent meaning.

As usual, $\sup f$ denotes the least upper bound (supremum) of f, and $\inf f$ denotes the greatest lower bound (infimum) of f. We do not deal with the cases where it is necessary to use the essential supremum/essential infimum.

Unless deemed necessary, we do not distinguish between scalars and vectors in notation through, for example, the use of bold symbols. Unless specified, it should be assumed that for a vector x, $\|x\|$ denotes the Euclidean norm, and for a matrix X, $\|X\|$ denotes the

[1]The input "layer" is usually not counted.

Frobenius norm. However, it sometimes becomes convenient to instead use the spectral norm. When there is potential for confusion, we will denote the choice of norm explicitly. In the context of the matrices, we use λ_i to denote the ith eigenvalue, when the eigenvalues are ordered in a nonincreasing fashion. Therefore, $\lambda_1 = \lambda_{max}$ is the maximum eigenvalue. Similarly, we will use σ_i to denote the ith singular value, and $\sigma_{min}(X)$ denotes the smallest nonzero singular value of X.

Finally, this dissertation will make use of tensors[2]. From a notation perspective, tensors will not be distinguished from matrices (or vectors); instead, tensors are treated as higher-dimensional generalizations of vectors and matrices. However, it will be important to be precise about the way multiplication occurs with tensors. For tensors X and Y, XY will denote a tensor contraction, which proceeds similar to matrix multiplication. That is, the last dimension of X will need to equal the first dimension of Y, and in the product XY, this common dimension will vanish. We will use $\otimes$ to denote the tensor outer product, sometimes called the dyadic product. It is crucial to note that this is *not* the Kronecker product; the latter is a vectorization (flattening) of the former. This dissertation does not use the Kronecker product, so there will be no notational confusion there. If X has dimension $(m_1, m_2, \ldots, m_k)$ and Y has dimension $(n_1, n_2, \ldots n_p)$, then the tensor outer product $X \otimes Y$ will have dimension $(m_1, m_2, \ldots, m_k, n_1, n_2, \ldots, n_p)$. In most cases, we will deal with cases where X is a higher-order identity tensor and Y will be a vector or matrix. Intuitively, $X \otimes Y$ should be thought of as "embedding" Y in a higher-order tensor, similar to how diagonalization proceeds. For clarity, we will refrain from merely using I to denote an identity tensor, and will explicitly write out the dimensions of the tensor we are denoting. For example $I_{m \times m}$ denotes an identity tensor with dimensions (m, m).

4.2 Learner Choice

This section discusses appropriate choices of learning algorithms to approximate the desired decision function. We will motivate the use of feedforward networks, since they can be studied mathematically in order to motivate why we want to change boundaries.

Chapter 1 introduced the problem with using the SVM for static analysis. That problem, to recap, was that the decision boundary cannot be changed globally. Suppose we were to design the requirements for an optimal learner. One might intuitively propose

[2]This being a computer science dissertation, it is most convenient to simply treat tensors as multidimensional arrays.

a human-like learner where the decision boundary appears to be freely drawn by hand, but does not overfit to the dataset. Is such a learner possible?

It turns out that it is. To understand how, we must understand the **Universal Approximation Theorems** that follow. Broadly, a universal approximation theorem lays down conditions under which a class of functions is **dense** in L^p. We will define these terms below.

Definition 4.1 (Dense set)**.** *A subset A of a topological space X is said to be dense in X if every item in X is present in A, or is arbitrarily "close" to some point in A.*

Definition 4.2. *(Lebesgue integral) The Lebesgue integral is a generalization of the Riemann integral. It is defined in terms of the Lebesgue sum*

$$\sum_i \alpha_i \mu(S_i)$$

where α_i is the value of the function in the subinterval i, and $\mu(S_i)$ is the Lebesgue measure of the interval S_i of points for which the value of the function is approximately α_i. A function is Lebesgue integrable if its Lebesgue integral is finite.

An intuition for the Lebesgue integral is as follows. The Riemann integral of $f(x)$ is a sum over the input space, and can be approximated via Riemann sums, where each term is a product of the length of an interval and the value of the function at that interval. The Lebesgue integral, on the other hand, can be thought of as the sum over the *output space*, and is approximated by products of the Lebesgue measure of the (possibly discontinuous) intervals where the function has a certain value, times the value itself.

Definition 4.3 (L^p space)**.** *An L^p space is a space of measurable functions for which the pth power of the absolute value is Lebesgue integrable.*

We have now established sufficient theory to introduce the first Universal Approximation Theorem, by [49]. For this theorem, let I_n denote the $n-$dimensional unit cube.

Theorem 4.1. *Let σ be a continuous sigmoidal function. Let f be the decision function for any finite measurable partition of I_n. For any $\epsilon > 0$, there is a finite sum of the form*

$$G(x) = \sum_{j=1}^{N} \alpha_j \sigma(y_j^T x + \theta_j)$$

and a set $D \subset I_n$, so that $m(D) \geq 1 - \epsilon$ and

$$|G(x) - f(x)| < \epsilon$$

for $x \in D$.

This theorem establishes the fact that a neural network with sigmoidal activation functions can, to any arbitrary precision ϵ, approximate an arbitrary function f. DasGupta & Schnitger [50] further establishes a set of activation functions that have the same approximating power as the sigmoid activation. However, in practice, the sigmoid activation function is known to perform sub-optimally. The seminal work of Nair & Hinton [152] introduced the ReLU (Rectified Linear Unit) activation function, $f(x) = \max(0, x)$. Indeed, there are universal approximation theorems that apply to this function. Leshno et al. [122] proves a universal approximation theorem for non-polynomial activation functions. While this is certainly applicable, we instead use the theorem provided by Hornik [97], for some unbounded and non-constant activation function ψ.

Theorem 4.2. *If ψ is unbounded and nonconstant, then $\mathfrak{R}(\psi)$ is dense in $L^p(\mu)$ for all finite measures μ on $\mathbb{R}^k$.*

Here, $\mathfrak{R}(\psi)$ denotes the set of functions representable by neural networks with activation function ψ.

Thus far, we have established an important fact: that ReLU feedforward networks are universal approximators. That is, they can produce decision boundaries of arbitrary shape. This is the human-like learner that we discussed at the beginning of this chapter.

However, the universal approximation theorems themselves are insufficient. Recall the wording of Theorem 2.2: $\mathfrak{R}(\psi)$ is dense in $L^p(\mu)$; in other words, the set of functions representable by all ReLU networks is dense in $L^p(\mu)$. This does not say anything about the *architecture* necessary to produce an optimal decision boundary. It is here that [151] provides a useful result:

Theorem 4.3. *The maximal number of linear regions of the functions computed by a neural network with n_0 input units and L hidden layers, with $n_i \geq n_0$ rectifiers at the i-th layer, is lower bounded by*

$$\left(\prod_{i=1}^{L-1} \left\lfloor \frac{n_i}{n_0} \right\rfloor^{n_0}\right) \sum_{j=0}^{n_0} \binom{n_L}{j}$$

We are interested in the product: note that if $n_i < n_0$ for any i, then this lower bound collapses. That is, in such a case, the neural network is not *forced* to try producing a complex decision boundary (which is preferable with real-world, noisy datasets). [151] gives us a hint that perhaps we should set the number of units per layer to be at least the number of inputs. This gives us a ballpark of what the hyper-parameter space of the number of units per layer should be.

There is one problem to tackle: computational cost. Because datasets can be high-dimensional, setting the number of units per layer to around the number of inputs can be computationally taxing. This leads to slow runtimes, which is exacerbated by any hyper-parameter optimization we perform (since it would require multiple runs of the network). Therefore, in an attempt to speed up computation, we use a stacked autoencoder to reduce the input dimensionality. In our experiments, we found that the exact details of the autoencoder architecture are less important than the other design choices; in our experiments, we used an autoencoder with architecture D - 10 - 7 - 5 - 7 - 10 - D where D is the input dimensionality. Rather than fret about the architecture, we instead looked at the fit. Specifically, we check if the mean squared error (MSE) loss is below 1000. If so, we proceed and use the encoded inputs during training. Otherwise, we attempt this process two more times to account for variability in performance caused by initialization ([200]). If in none of the three attempts, the MSE is below 1000, then we do not encode the input.

There are other theoretical considerations that we mention, but do not expand upon, here. The standard manner of quantifying the representation capability of a machine learning system is the VC-dimension.

Definition 4.4 (Shattering)**.** *A learning algorithm is said to shatter a set of points if it can perfectly classify all of the points in the set.*

Definition 4.5 (VC-dimension)**.** *The VC dimension of a learning algorithm is the size of the largest set of points that the algorithm can shatter.*

The VC-dimension of feedforward networks with ReLU activations has been shown to be $\mathcal{O}(WL \log W)$ or $\Theta(WU)$ ([22, 87], where W is the number of weights, L is the number of layers, and U the number of units. In theory, this result allows us to tighten bounds on the hyper-parameters of feedforward networks further; however, as we will see later, we are not interested in the *shattering* properties of the learning algorithm, but its *generalization* performance.

4.3 Parameter engineering

Parameter engineering guides the selection of control values when building models. The more common name for parameter engineering is hyper-parameter optimization.

Much of the work of this dissertation establishes the value of hyper-parameter optimization (HPO). There are many HPO methods used in SE including:

- The reinforcement learning methods of HyperBand [125];
- The Bayesian Optimization methods of TPE [26].
- The tabu search of DODGE, which relies on a *contraction map*[3] assumption. If two configurations lead to similar options, DODGE marks that region as tabu, nudging the search to other regions. Because this is a stronger assumption than the Lipschitz assumption used by Dou et al. [58], the authors use a tree-based method to partition the space, iteratively updating weights of leaves based on the performance deltas observed. [1].

However, like many of the hyper-parameter optimization methods of its time, DODGE has a few drawbacks: (i) it has a rather strong assumption, and as such, can fail if this does not hold; (ii) it has no theoretical guarantees beyond an assumption derived from a relatively small-scale empirical study; (iii) it requires 30 evaluations to declare a good hyper-parameter configuration. In the modern SE literature, where larger models are the norm, such drawbacks discourage researchers from using HPO at all; indeed, we observed this in our literature review.

HPO is well-studied in the ML literature as well, with the most popular approaches being based on Bayesian optimization [27, 91, 193, 201], while other work suggests alternative approaches such as random search [25]. Smith [191] discusses empirical methods to manually tune hyper-parameters based on the performance of the current system. However, although HPO has repeatedly been shown to improve learner performance [135, 203], much applied machine learning research either does not use HPO, or uses computationally expensive methods such as grid search. Some of this reluctance to use HPO stems from the general view that it is computationally expensive. For example, Tran et al. [207] comment, "*Regardless of which hyper-parameter optimization method is used, this task is generally very expensive in terms of computational costs.*" Moreover, there is a growing concern to reduce the carbon emissions from ML experiments [120].

A fortiori, there is a need for a better hyper-parameter optimization method, with computational efficiency being a primary concern. Based on the above discussion, a "good" HPO method has the following desiderata:

- **Theoretical backing:** Practitioners may be more willing to adopt methods that are based on established theory.

[3]A contraction map is an $L-$Lipschitz function over a metric space, where $0 \leq L \leq 1$.

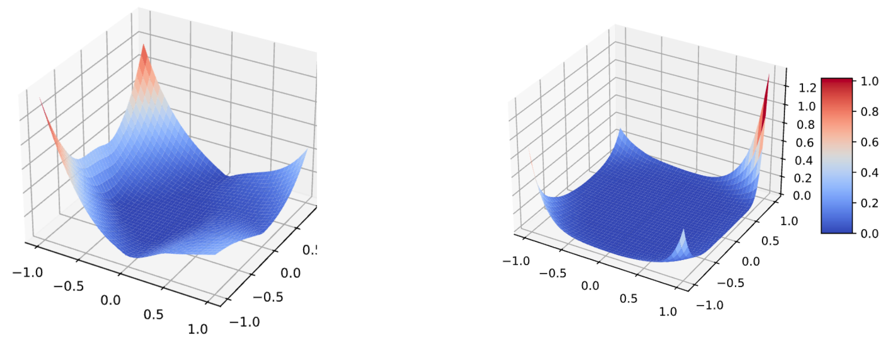


Figure 4.1: Left & right shows loss landscape before & after HPO. The loss function is visualized as the z dimension of domain data mapped into two PCA components x, y. In this figure, red indicates regions with highest mean squared error.

- **Weaker assumptions:** Weaker assumptions improve the odds of success when an HPO method is deployed "off-the-shelf", as our literature review reveals that most algorithms are employed this way. This, coupled with the theoretical backing, gives confidence to practitioners about the generalizability of the method.

- **Runtime:** An ideal HPO method would run fast, as measured by wall time. The simplest way to achieve this is to reduce the number of evaluations needed to return a good configuration.

Although Bayesian Optimization (BO) methods have strong theoretical backing, and do not have particularly strong assumptions, the necessity of building a surrogate model for the performance of some configuration means several evaluations are necessary for it to converge. Therefore, we do not adopt the BO framework. Similarly, although the Lipschitz assumption (or the stronger contraction map assumption) may be intuitive, these cannot be shown to hold in general.

To achieve the above desiderata, we first develop a method tailored for SE data; then, we develop a more general version.

To understand the idea behind the first method, note that the *smoothness* of the loss function provides a measure of the difficulty of learning the problem (a function with a high smoothness is necessarily sharper, which impedes generalization, see Section 2.4). Our work on detection false alarms in FindBugs demonstrated that the operators developed above led to flat minima, but contrary to expectations, the smoothness *increased* (see Figure 4.1). This was a repeated result across datasets and random seeds. A closer examination reveals why these are simultaneously true: although there exists a flat minima, there also exists a sharp corner, which contributes to a higher smoothness value. These two phenomena co-occurring is very desirable: at random initialization, the optimizer (SGD) may either be on the sharp wall, or in the flat basin. If it starts on the sharp wall, theoretical guarantees on SGD mean that it quickly descends to the flat minima, at which point we achieve good generalization. On the other hand, if it starts close to the flat basin, it is close to convergence. Of course, if the loss is particularly ill-behaved and there exists another flat basin at the top of the sharp corner, it is possible for this to fail. However, we did not empirically observe this.

In both methods, we will rely on the following lemma:

Lemma 3. *Let $A, B \in \mathbb{S}^n$ and suppose $A \succeq B$. Then $\|A\|_2 \geq \|B\|_2$*

Proof. We have $\|A\|_2 = \sqrt{\lambda_{max}(A^T A)} = |\lambda_{max}(A)|$ since A is symmetric. Since $A \succeq B$, we have by definition $x^T A x \geq x^T B x$ for an arbitrary $x \in \mathbb{R}^n$. From the min-max theorem, we have $\forall k \in \{1, \ldots, n\}$,

$$\lambda_k(A) = \min_{\substack{\mathcal{M} \subset \mathbb{S}^n \\ \dim \mathcal{M} = k}} \max_{x \in \mathcal{M} \setminus \{0\}} \tfrac{x^T A x}{x^T x} \geq \min_{\substack{\mathcal{M} \subset \mathbb{S}^n \\ \dim \mathcal{M} = k}} \max_{x \in \mathcal{M} \setminus \{0\}} \tfrac{x^T B x}{x^T x} = \lambda_k(B)$$

where λ_k is the kth eigenvalue in the spectrum ordered in non-increasing order. Using the above with $k = 1$ completes the proof. □

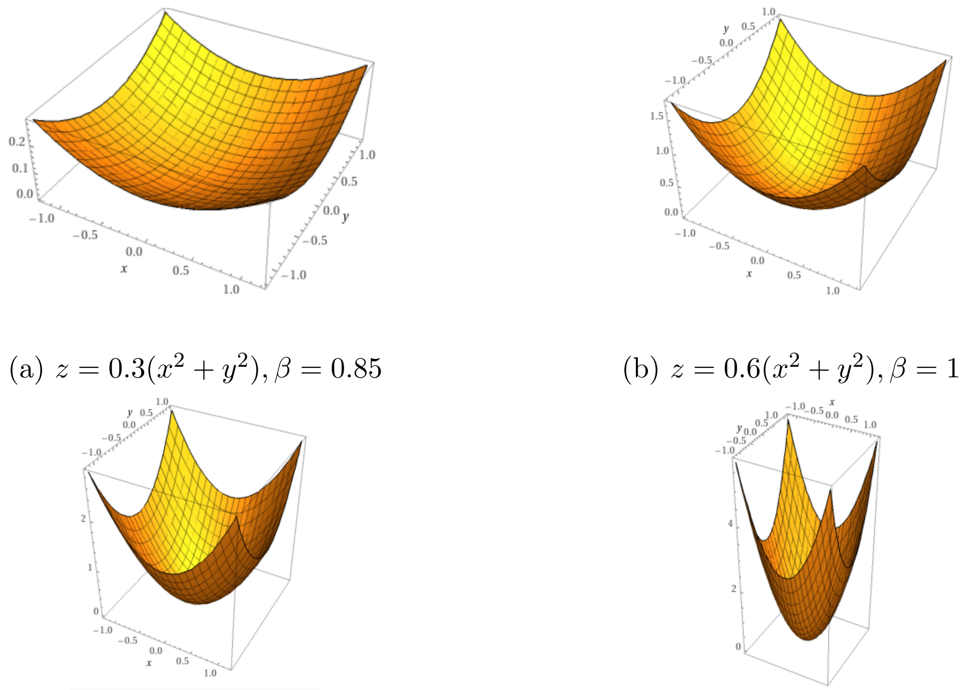
Our first method solves the following optimization problem in a randomized fashion:

$$\sup_{h \in \mathcal{H}} \sup_{\substack{x_i \subset X \\ \cup x_i = X}} \left\| \nabla^2 \mathcal{L}(x; \theta, h) \right\|$$

This leverages the intuition that searching for the loss with the highest peak sharpness will yield a surface with a sharp corner and a flat basin. Note that this is in direct contradiction to what would be suggested from convex optimization literature, which would instead recommend minimizing the sharpness. Indeed, the idea of minimizing the sharpness directly has been studied [72], albeit in the context of directly optimizing a fixed network as opposed to the general hyper-parameter optimization setting we discuss here.

4.3.1 A smoothness-based HPO method

At this point, it is important to disambiguate the *concept* of smoothness (that of a smooth, flat landscape) from the *mathematical notion* of smoothness: a lower mathematical smoothness value corresponds to lower fluctuations in the gradient, and as such, we would expect those to be intuitively smoother and flatter.



(a) $z = 0.3(x^2 + y^2), \beta = 0.85$

(b) $z = 0.6(x^2 + y^2), \beta = 1.7$

(c) $z = x^2 + y^2, \beta = 2.8$

(d) $z = 2(x^2 + y^2), \beta = 5.6$

Figure 4.2: An illustration of sharp minima forming with higher smoothness. All figures generated using Wolfram Alpha.

We provide the reader with the following intuition, and encourage them to follow along in Figure 4.2: suppose one was provided a single knob to modify a class of functions, which controlled the smoothness. This knob starts at 0 and can be dialed to arbitrarily high values. At 0, the user would see a completely flat landscape (since all constant and linear functions have zero smoothness); as the user slowly increases the value, they might see either all corners to rise slowly, or a single corner rise sharply–but at this stage, the value is not high enough to raise multiple corners high and form a sharp valley (see Fig. 4.2b). As the user further increases the smoothness to even higher values, there is flexibility to further raise the corners of this hypothetical function (Fig. 4.2c), until a sharp valley is formed (Fig. 4.2d).

We empirically demonstrate this phenomenon. In Figure 4.4, we show the distribution of smoothness values for 21 SE and 21 non-SE datasets. It is evident that the smoothness values for tabular SE data, such as the ones used in this study, are bounded and generally very small, while non-SE datasets have a wide range of smoothness. As such, the

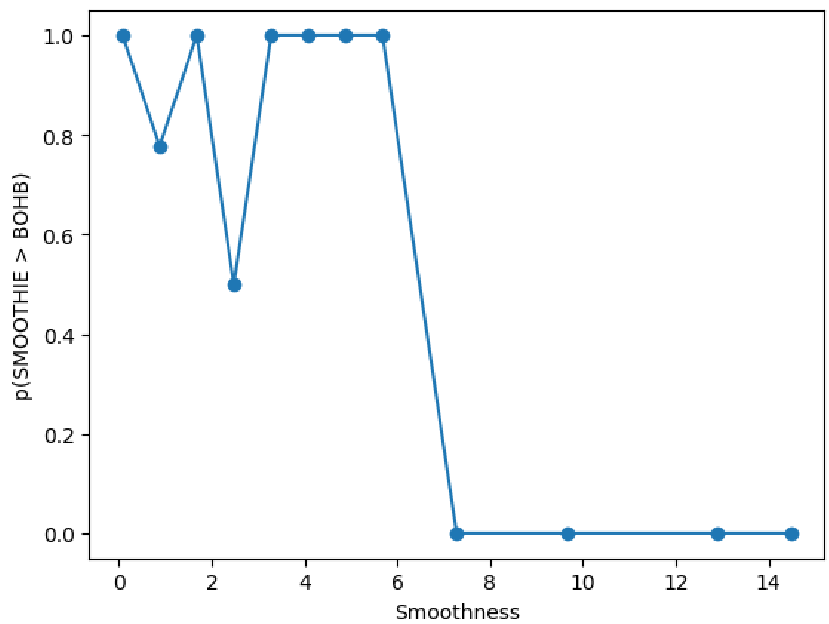


Figure 4.3: Probability of SMOOTHIE outperforming BOHB as a function of smoothness

SE landscapes cannot form sharp valleys, only sharp corners[4] This translates well to performance: in Figure 4.3, we plot the probability that `SMOOTHIE` outperforms BOHB (the SOTA) as a function of the smoothness of the dataset. The plot shows a strong negative correlation (indeed, the Pearson correlation coefficient is -0.79). This figure discourages the use of `SMOOTHIE` for datasets with a "high enough" smoothness (which is around 7).

The intuition developed above yields a simple HPO algorithm: randomly sample N_1 configurations, compute their smoothness, and run the configurations with the top N_2 smoothness values. This raises the issue of how to compute the smoothness: because the smoothness is a second-order metric, it would require computing the Hessian, which is computationally expensive. We circumvent this issue by designing a *white-box* solution: because at each iteration, the architecture of the neural network is known (because it is materialized from the specific hyper-parameter configuration), we can develop a semi-empirical form of the smoothness. However, when the data (and hyper-parameters) is kept constant, the loss function is a function of the weights–as such, as SGD progresses, the projection of the overall loss onto the weight space changes. Therefore, we need to

[4]There is an exception–in our case studies, we found the 2-class issue lifetime prediction datasets to have extremely high smoothness values; consequently, `SMOOTHIE` also performed poorly on those.

run SGD (or any algorithm) for a few iterations to ensure the loss we compute the smoothness for is close to the loss at convergence. We use one epoch in our experiments. We will later bound the deviation using a covering argument, and also show empirical results demonstrating that this one-epoch cycle yields a balanced exploration-exploitation trade-off. We now develop the semi-empirical approximations of the smoothness.

Smoothness for Feedforward Classifiers

This section shows the smoothness computation for feedforward learners.

Lemma 4. *Suppose the activations at the last layer, $a^{[L]}$ are computed as the softmax of the pre-activations. Then,*

$$\nabla_{z^{[L]}} a^{[L]} = diag(a^{[L]}) - a^{[L]} a^{[L]T}$$

Proof. The proof is a straightforward application of the chain rule:

$$\begin{aligned}
\frac{\partial a_j^{[L]}}{\partial z_p^{[L]}} &= \frac{\partial}{\partial z_p^{[L]}} \left(\frac{e^{z_j^{[L]}}}{\sum_{l=1}^{k} e^{z_l^{[L]}}} \right) \\
&= \frac{\delta_{pj} e^{z_j^{[L]}} \sum_{l=1}^{k} e^{z_l^{[L]}} - e^{z_j^{[L]}} \cdot e^{z_p^{[L]}}}{\left(\sum_{l=1}^{k} e^{z_l^{[L]}} \right)^2} \\
&= \frac{\delta_{pj} e^{z_j^{[L]}}}{\sum_{l=1}^{k} e^{z_l^{[L]}}} - \frac{e^{z_j^{[L]}}}{\sum_{l=1}^{k} e^{z_l^{[L]}}} \cdot \frac{e^{z_p^{[L]}}}{\sum_{l=1}^{k} e^{z_l^{[L]}}} \\
&= \left(\delta_{pj} a_j^{[L]} - a_j^{[L]} a_p^{[L]} \right) \\
&= a_j^{[L]} (\delta_{pj} - a_p^{[L]}) \qquad (4.1)
\end{aligned}$$

Writing this in vectorized notation yields the result. □

Lemma 5. *For a deep learner with ReLU activations in the hidden layers, the last two layers being fully-connected, and a softmax activation at the last layer,*

$$\frac{\partial E}{\partial z_j^{[L]}} \leq \frac{k-1}{km} \left(\sum_{i=1}^{m} [y^{(i)} = j] \right)$$

under the cross-entropy loss.

Proof. We will use the chain rule, as follows:

$$\frac{\partial E}{\partial z_j^{[L]}} = \frac{\partial E}{\partial a_j^{[L]}} \cdot \frac{\partial a_j^{[L]}}{\partial z_j^{[L]}} \qquad (4.2)$$

Consider the Iverson notation version of the general cross-entropy loss:

$$E(\mathbf{a}^{[L]}) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{h=1}^{k}[y^{(i)} = h]\log a_h^{[L]}$$

Then the first part of (4.2) is trivial to compute:

$$\frac{\partial E}{\partial a_j^{[L]}} = -\frac{1}{m}\sum_{i=1}^{m}\frac{[y^{(i)} = j]}{a_j^{[L]}} \tag{4.3}$$

The second part is computed as follows.

$$\begin{aligned}
\frac{\partial a_j^{[L]}}{\partial z_p^{[L]}} &= \frac{\partial}{\partial z_p^{[L]}}\left(\frac{e^{z_j^{[L]}}}{\sum_{l=1}^{k} e^{z_l^{[L]}}}\right) \\
&= \frac{\delta_{pj} e^{z_j^{[L]}} \sum_{l=1}^{k} e^{z_l^{[L]}} - e^{z_j^{[L]}} \cdot e^{z_p^{[L]}}}{\left(\sum_{l=1}^{k} e^{z_l^{[L]}}\right)^2} \\
&= \frac{\delta_{pj} e^{z_j^{[L]}}}{\sum_{l=1}^{k} e^{z_l^{[L]}}} - \frac{e^{z_j^{[L]}}}{\sum_{l=1}^{k} e^{z_l^{[L]}}} \cdot \frac{e^{z_p^{[L]}}}{\sum_{l=1}^{k} e^{z_l^{[L]}}} \\
&= \left(\delta_{pj} a_j^{[L]} - a_j^{[L]} a_p^{[L]}\right) \\
&= a_j^{[L]}(\delta_{pj} - a_p^{[L]}) \qquad (4.4)
\end{aligned}$$

Combining (4.3) and (4.4) in (4.2) gives

$$\frac{\partial E}{\partial z_j^{[L]}} = \frac{1}{m}\left(\sum_{i=1}^{m}[y^{(i)} = j]\right)\left(\sum_{h=1}^{k} a_j^{[L]} - [h = j]\right) \tag{4.5}$$

As a prudence check, note that E is a scalar and $z_j^{[L]}$ is a *vector* (over all the samples); the right-hand side is also a vector (by vectorizing the Iverson term $[h = j]$). It is easy to show that the limiting case of this is when all softmax values are equal and each $y^{(i)} = p$; using this and $a_j^{[L]} = \frac{1}{k}$ in (4.5) and combining with (4.2) gives us our desired result:

$$\sup_j \left\| \frac{\partial E}{\partial z_j^{[L]}} \right\| = \frac{k-1}{km}\left(\sum_{i=1}^{m}[y^{(i)} = j]\right) \tag{4.6}$$

□

Theorem 4.4 (Smoothness for feedforward networks)**.** *For a deep learner with ReLU activations in the hidden layers, the last two layers being fully-connected, and a softmax* *activation at the last layer, the smoothness of the cross-entropy loss is given by*

$$\sup\|\nabla_W^2 E\| \propto \frac{k-1}{km} \sup \frac{\|a^{[L-1]}\|}{\|W^{[L]}\|} \tag{4.7}$$

Proof. Lemma 5 gives us (using the chain rule one step further):

$$\nabla_W E \leq \frac{(k-1)}{km} \left(\sum_{i=1}^{m} [y^{(i)} = j] \right) a_j^{[L-1]}$$

The $\left(\sum_{i=1}^{m} [y^{(i)} = j] \right)$ term is constant, so we drop it as a proportionality constant. In the following, we use $\leq$ and $=$ to mean "less than equal to the proportionality constant times" and "proportional to" respectively. Therefore,

$$\begin{aligned}
\nabla_W^2 E &\leq \frac{k-1}{km} \nabla_W a_j^{[L-1]} \\
&= \frac{k-1}{km} \frac{\partial a_j^{[L-1]}}{\partial E} \frac{\partial E}{\partial w_{ij}} \\
&\leq \frac{(k-1)^2 a_j^{[L-1]}}{k^2 m^2} \frac{\partial a_j^{[L-1]}}{\partial E} \quad \textit{using chain rule on Lemma 1} \\
&= \frac{(k-1)^2 a_j^{[L-1]}}{k^2 m^2} \frac{1}{\frac{\partial E}{\partial a_j^{[L-1]}}}
\end{aligned}$$

Using $z^{[l]} = W^{[l]T} a^{[l-1]} + b^{[l]}$ and Lemma 5 gives us:

$$\frac{\partial E}{\partial a_j^{[L-1]}} = \frac{\partial E}{\partial z_j^{[L]}} \frac{\partial z_j^{[L]}}{\partial a_j^{[L-1]}} \leq \frac{(k-1)}{km} \|W^{[L]}\|$$

from which (4.7) follows. □

Smoothness for Logistic Regression

Corollary 1 (Smoothness of logistic regression)**.**

$$\beta = \frac{k-1}{km} \sup \frac{\|X\|}{\|W\|}$$

Proof. Logistic regression can be viewed as a feedforward network with no hidden layers. Hence, the smoothness for logistic regression follows from Equation (4.7). □

Theorem 4.5 (L_2 regularization). *For a learner using an L_2 regularization term $\frac{\lambda}{2}\|\boldsymbol{w}\|_2^2$, the increase in smoothness is λ. More generally, if a Tikhonov regularization term $\|\boldsymbol{\Gamma w}\|_2^2$ is used, the increase in smoothness is $2\|\boldsymbol{\Gamma}\|_F^2$.*

Proof. For L_2 regularization, the regularization term is $E = \frac{\lambda}{2}\|\boldsymbol{w}\|_2^2$. We therefore have $\nabla_{\boldsymbol{w}} E = \lambda\|\boldsymbol{w}\|_2$ and so $\|\nabla_{\boldsymbol{w}_1} E(\boldsymbol{w}_1) - \nabla_{\boldsymbol{w}_2} E(\boldsymbol{w}_2)\| \leq \lambda\|\boldsymbol{w}_1 - \boldsymbol{w}_2\|$. For the general Tikhonov case,

$$\begin{aligned}
E(\boldsymbol{w}) &= \|\boldsymbol{\Gamma w}\|_2^2 \\
&= \boldsymbol{w}^T\boldsymbol{\Gamma}^T\boldsymbol{\Gamma w} \\
dE &= d\boldsymbol{w}^T\boldsymbol{\Gamma}^T\boldsymbol{\Gamma w} + \boldsymbol{w}^T d\boldsymbol{\Gamma}^T\boldsymbol{\Gamma w} + \boldsymbol{w}^T\boldsymbol{\Gamma}^T d\boldsymbol{\Gamma} + \boldsymbol{w}^T\boldsymbol{\Gamma}^T\boldsymbol{\Gamma} d\boldsymbol{w} \\
&= (d\boldsymbol{w})^T\boldsymbol{\Gamma}^T\boldsymbol{\Gamma w} + \boldsymbol{w}^T\boldsymbol{\Gamma}^T\boldsymbol{\Gamma}(d\boldsymbol{w}) \\
&= \boldsymbol{w}^T\boldsymbol{\Gamma}^T\boldsymbol{\Gamma}(d\boldsymbol{w}) + \boldsymbol{w}^T\boldsymbol{\Gamma}^T\boldsymbol{\Gamma}(d\boldsymbol{w}) \\
\nabla_{\boldsymbol{w}} E &= 2\boldsymbol{w}^T\boldsymbol{\Gamma}^T\boldsymbol{\Gamma} = 2\boldsymbol{w}^T\|\boldsymbol{\Gamma}\|_F^2
\end{aligned}$$

so that

$$\frac{\|\nabla E(\boldsymbol{w}_1) - \nabla E(\boldsymbol{w}_2)\|}{\|\boldsymbol{w}_1 - \boldsymbol{w}_2\|} = 2\|\boldsymbol{\Gamma}\|_F^2$$

□

Smoothness for Naive Bayes

Theorem 4.6 (Smoothness for Naive Bayes). *For Naive Bayes, the smoothness is given by*

$$\beta = \sup\|\boldsymbol{\Sigma}^{-1}\mathcal{A}G - \frac{1}{2}\boldsymbol{\Sigma}^{-1}\mathcal{A}\boldsymbol{\Sigma}^{-1} + G\mathcal{A}\boldsymbol{\Sigma}^{-1}\|$$

where $\boldsymbol{\Sigma}$ is the covariance matrix, $\mathcal{A}$ is the fourth-order identity tensor, and $G = \dfrac{\partial \ell}{\partial \boldsymbol{\Sigma}}$, where ℓ is the log-likelihood.

Proof. We use the results of Greg [77] here. For Gaussian Discriminant Analysis (which Naive Bayes is a special case of), the log-likelihood is given by:

$$\begin{aligned}
\ell(\phi, \boldsymbol{\mu}_0, \boldsymbol{\mu}_1, \boldsymbol{\Sigma}) \quad &= \textstyle\sum_{i=1}^m \big\{ y^{(i)} \log\phi + (1 - y^{(i)})\log(1-\phi) \\
&\quad -\tfrac{1}{2}\log|\boldsymbol{\Sigma}| + C + (\mathbf{x}^{(i)} - \boldsymbol{\mu})^T\boldsymbol{\Sigma}^{-1}(\mathbf{x}^{(i)} - \boldsymbol{\mu})\big\}
\end{aligned} \tag{4.8}$$

where ϕ is the parameter of the Bernoulli distribution; μ_0 and μ_1 are the mean vectors for the two classes and Σ is the covariance matrix of the independent attributes. Because there are several parameters, we show that the smoothness is *constant* with respect to all except the covariance matrix, and therefore we use the smoothness with respect to the covariance matrix in our experiments.

Taking a gradient of (4.8) with respect to ϕ gives us:

$$\nabla_\phi \ell = \frac{\sum y^{(i)}}{\phi} - \frac{\sum(1-y^{(i)})}{1-\phi}$$
$$\nabla_\phi^2 \ell = -\frac{\sum y^{(i)}}{\phi^2} - \frac{\sum(1-y^{(i)})}{(1-\phi)^2}$$

The second derivative has an extremum when $\nabla_\phi^3 \ell = 0$:

$$\begin{aligned}
\nabla_\phi^3 \ell &= \frac{2\sum y^{(i)}}{\phi^3} - \frac{2\sum(1-y^{(i)})}{(1-\phi)^3} = 0 \\
\frac{\sum y^{(i)}}{\phi^3} &= \frac{\sum(1-y^{(i)})}{(1-\phi)^3} \\
m\phi^3 &= \left(\sum y^{(i)}\right)(1-3\phi+3\phi^2)
\end{aligned}$$

But $\sum y^{(i)} = km$ for some $0 \le k \le 1$. Hence,

$$\phi^3 = k(1-3\phi+3\phi^2)$$

which has constant solutions. For example, if $k = \frac{1}{2}$ then $\phi = \frac{1}{2}$ and $\nabla_\phi^2 E \le -4m$. Similarly, if $k = 1$ then $\phi = 1$ and $\nabla_\phi^2 E \le -m$

From (4.8), computing gradient with respect to μ yields

$$\begin{aligned}
\nabla_{\mu_0} \ell &= \nabla_{\mu_0} \sum_{i=1}^m (\mathbf{x}^{(i)} - \boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1} (\mathbf{x}^{(i)} - \boldsymbol{\mu}) \\
&= \nabla_{\mu_0} \sum_{i=1}^m \text{tr}\ \ (\mathbf{x}^{(i)} - \boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1} (\mathbf{x}^{(i)} - \boldsymbol{\mu}) I \\
&= -\sum_{i=1}^m \left(I(\mathbf{x}^{(i)} - \boldsymbol{\mu}_0)\boldsymbol{\Sigma}^{-1} + I(\mathbf{x}^{(i)} - \boldsymbol{\mu}_0)(\boldsymbol{\Sigma}^{-1})^T \right)[y^{(i)} = 0] \\
&= -\sum_{i=1}^m \left((\mathbf{x}^{(i)} - \boldsymbol{\mu}_0)\boldsymbol{\Sigma}^{-1} +\ (\mathbf{x}^{(i)} - \boldsymbol{\mu}_0)\boldsymbol{\Sigma}^{-1} \right)[y^{(i)} = 0] \\
&= -2\boldsymbol{\Sigma}^{-1} \sum_{i=1}^m (\mathbf{x}^{(i)} - \boldsymbol{\mu}_0)[y^{(i)} = 0]
\end{aligned}$$

where we used the identity $\nabla_A \text{tr}\ ABA^TC = CAB + C^TAB^T$ along with the trace trick. Clearly, the second derivative is constant with respect to $\boldsymbol{\mu}_0$.

Finally, we compute the smoothness with respect to $\boldsymbol{\Sigma}$. For the following, let $\langle x, y \rangle$ denote the Frobenius inner product defined as

$$\langle A, B \rangle = \sum_{i=1}^m \sum_{j=1}^n A_{ij} B_{ij} = \text{tr}\ A^T B$$

Let $\boldsymbol{w} = (\boldsymbol{x} - \boldsymbol{\mu}_i)$ (where μ_i is the mean for class i). Then,

$$\begin{aligned}
d\ell &= \langle \boldsymbol{w}\boldsymbol{w}^T, (d\boldsymbol{\Sigma})^{-1} \rangle - \frac{1}{2}\langle \boldsymbol{\Sigma}^{-1}, d\boldsymbol{\Sigma} \rangle \\
&= \langle \boldsymbol{w}\boldsymbol{w}^T, -\boldsymbol{\Sigma}^{-1} d\boldsymbol{\Sigma}\boldsymbol{\Sigma}^{-1} \rangle - \frac{1}{2}\langle \boldsymbol{\Sigma}^{-1}, d\boldsymbol{\Sigma} \rangle \\
&= -\left\langle \boldsymbol{\Sigma}^{-1}\boldsymbol{w}\boldsymbol{w}^T\boldsymbol{\Sigma}^{-1} + \frac{1}{2}\Sigma^{-1}, d\Sigma \right\rangle \\
\frac{\partial \ell}{\partial \boldsymbol{\Sigma}} &= -\boldsymbol{\Sigma}^{-1}\left(\boldsymbol{w}\boldsymbol{w}^T + \frac{1}{2}\boldsymbol{\Sigma}\right)\boldsymbol{\Sigma}^{-1} = G
\end{aligned}$$

Next, we compute the differential of G, which we use to compute the Hessian.

$$\begin{aligned}
dG &= -d\boldsymbol{\Sigma}^{-1}\left(\boldsymbol{w}\boldsymbol{w}^T + \frac{1}{2}\boldsymbol{\Sigma}\right)\boldsymbol{\Sigma}^{-1} - \boldsymbol{\Sigma}^{-1}\left(\frac{1}{2}d\boldsymbol{\Sigma}\right)\boldsymbol{\Sigma}^{-1} \\
&- \boldsymbol{\Sigma}^{-1}\left(\boldsymbol{w}\boldsymbol{w}^T + \frac{1}{2}\boldsymbol{\Sigma}\right)d\boldsymbol{\Sigma}^{-1} \\
&= \boldsymbol{\Sigma}^{-1}d\boldsymbol{\Sigma}G - \frac{1}{2}\boldsymbol{\Sigma}^{-1}d\boldsymbol{\Sigma}\boldsymbol{\Sigma}^{-1} + Gd\boldsymbol{\Sigma}\boldsymbol{\Sigma}^{-1} \\
&= \langle \boldsymbol{\Sigma}^{-1}\mathcal{A}G - \frac{1}{2}\boldsymbol{\Sigma}^{-1}\mathcal{A}\boldsymbol{\Sigma}^{-1} + G\mathcal{A}\boldsymbol{\Sigma}^{-1}, d\boldsymbol{\Sigma} \rangle \\
\frac{\partial G}{\partial \boldsymbol{\Sigma}} &= \boldsymbol{\Sigma}^{-1}\mathcal{A}G - \frac{1}{2}\boldsymbol{\Sigma}^{-1}\mathcal{A}\boldsymbol{\Sigma}^{-1} + G\mathcal{A}\boldsymbol{\Sigma}^{-1} = \mathcal{H}
\end{aligned}$$

where $\mathcal{H}$ is the Hessian and $\mathcal{A}$ is the fourth-order identity tensor so that $\mathcal{A}_{ijkl} = \delta_{ik}\delta_{jl}$. In the above, the fact that Σ and G are symmetric was used in various steps.

Although the *loss* to minimize would be defined as the negative log-likelihood, the negative sign would be removed by the use of a norm. □

Algorithm 1 shows the pseudo-code for `SMOOTHIE`. We first pick N_1 (= 30) configurations at random. For each of those configurations, we compute the smoothness and store it. Next,we pick the top N_2 (= 5) of those configurations by smoothness values, and run only those, getting the performance for each. The algorithm then return the best-performing configuration.

4.3.2 A general version

To achieve this, we focus on a problem more aligned with convex optimization literature:

$$\inf_{h \in \mathcal{H}} \sup_{\substack{x_i \subset X \\ \cup x_i = X}} \left\| \nabla^2 \mathcal{L}(x; \theta, h) \right\|$$

Algorithm 1: `SMOOTHIE`

Input : Number of configs to sample N_1, number of configs to run N_2.
Defaults: $N_1 = 30, N_2 = 5$.

Output: A near-optimal configuration

$\mathcal{H}_0 \leftarrow \textsc{Random}(\mathcal{H}, N_1)$;
$S \leftarrow \phi$;
$P \leftarrow \phi$;
for *each config h in* $\mathcal{H}_0$ **do**
    S[config] $\leftarrow$ $\textsc{Get-Smoothness}$(h)
end
for *each config in* $\textsc{Top}(S,\ N_2)$ **do**
    P[config] $\leftarrow$ $\textsc{Run}$(config);
end
return $\arg\max P$;

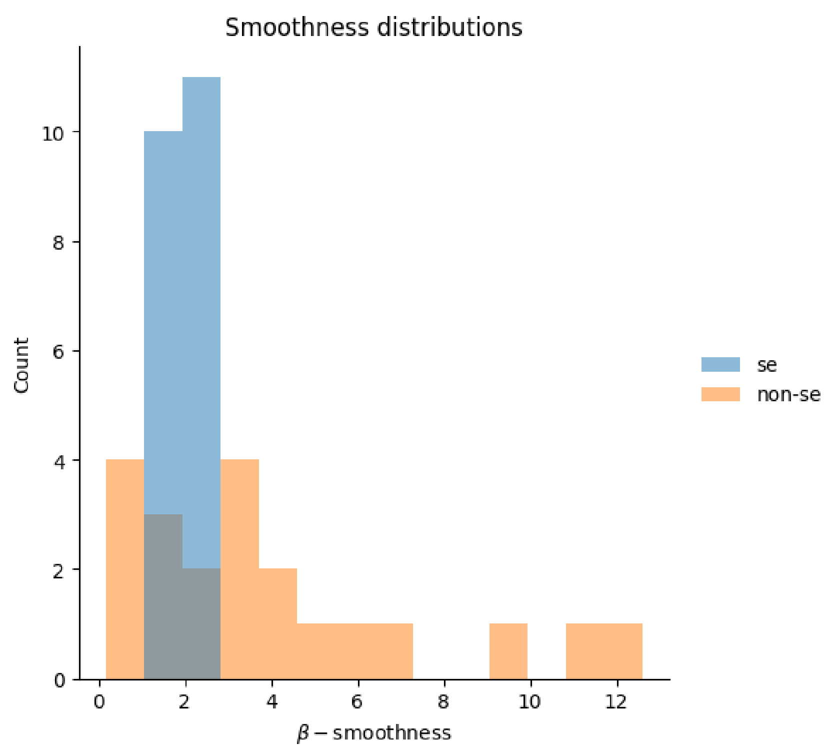


Figure 4.4: Distribution of smoothness for SE and non-SE datasets. SE datasets are a subset of the datasets explored in this dissertation. ML datasets come from the UCI repository and the OpenML tasks used in HPOBench.

This trades off the possibility of discovering flat basins with sharp corners for a less aggressive approach that searches for the least sharp losses.

We start by writing the definition of strong convexity in terms of positive-semidefiniteness as

$$\nabla^2 f(x) \succeq \mu I$$

From Lemma 3, we can rewrite this as

$$\|\nabla^2 f(x)\|_2 \geq \mu \Rightarrow \left\|\nabla^2 f(x)\right\|_F \geq \mu$$

and so we redefine $\mu-$strong convexity as

$$\mu = \inf \|\nabla^2 f(x)\|_F$$

A key motivation for minimizing the strong convexity is to relate it to the flatness of minima. Since $\nabla^2 f$ is symmetric (by Schwarz's theorem), we have

$$\left|\lambda_{max}(\nabla^2 f)\right| = \left\|\nabla^2 f\right\|_2 \leq \left\|\nabla^2 f\right\|_F$$

so that $|\lambda_{max}(\nabla^2 f)| = \inf \|\nabla^2 f\|_F$ which directly relates the sharpness of the landscape with the definition of strong convexity above. We can also relate the above definition of strong convexity to the sharpness measure ζ established by Keskar et al. [112]:

$$\begin{aligned}
\zeta(w) &= \frac{\max_{w' \in \mathcal{B}(\epsilon, w)} f(w') - f(w)}{1 + f(w)} \\
&\overset{(i)}{\approxeq} \max_{w' \in \mathcal{B}(\epsilon, w)} f(w') - f(w) \\
&\overset{(ii)}{\approxeq} \frac{\epsilon^2}{2} \left\|\nabla^2 f(w)\right\|_2 \leq \frac{\epsilon^2}{2} \left\|\nabla^2 f(w)\right\|_F
\end{aligned}$$

where (i) is because the training error is typically small in practice [154] and (ii) follows from a second-order Taylor expansion of f around w, as done by Dinh et al. [57]. Therefore, we wish to minimize the sharpness, which is equivalent to the formulation above. Equivalently, we have (with r denoting the rank of the Hessian)

$$\zeta(w) \approxeq \tfrac{\epsilon^2}{2} \|\nabla^2 f(w)\|_2 \geq \tfrac{\epsilon^2}{2\sqrt{r}} \|\nabla^2 f(w)\|_F \geq \tfrac{\epsilon^2}{2\sqrt{n}} \|\nabla^2 f(w)\|_F$$

so that the strong convexity is a lower bound on the sharpness, and a higher strong convexity implies a higher sharpness, which is correlated with a higher generalization error[5]. Therefore, we have:

$$\frac{\epsilon^2}{2\sqrt{n}}\left\|\nabla^2 f(w)\right\|_F \leq \zeta(w) \leq \frac{\epsilon^2}{2}\left\|\nabla^2 f(w)\right\|_F$$

so that the strong convexity and a scaled version thereof provide bounds on the sharpness, and minimizing the strong convexity implies both the lower and upper bounds on the sharpness are lowered.

We can bound the deviation from the true supremum using a covering argument [60]. Let $f(x) = \sup\|\nabla^2 E(x)\|$ be drawn from some set of functions $\mathcal{F}$, each of whose elements map from $\mathcal{X}$ to $\mathbb{R}$. Define a point-mass empirical distribution on $\{x_i\}_{i=1}^m$ as $P_m = \frac{1}{m}\sum_{i=1}^{m}\delta_{x_i}$ where δ is the Dirac delta. For any function $f : \mathcal{X} \to \mathbb{R} \in \mathcal{F}$, let

$$P_m f \triangleq \mathbb{E}_{P_m}[f(X)] = \sum_{i=1}^{m} f(x_i)$$

be the empirical expectation over a mini-batch and let

$$Pf \triangleq \mathbb{E}_P[f(X)] = \int f(x)dP(x)$$

denote the general expectation under a measure P. Suppose the functions in $\mathcal{F}$ are bounded above by β (trivially, they are bounded below by 0), and define the metric over $\mathcal{F}$ as $\|f-g\|_\infty = \sup_{x\in\mathcal{X}}|f(x)-g(x)|$. Denote by $N(\delta,\Theta,\rho)$, the covering number for a $\delta-$cover of a set Θ with respect to a metric ρ. Then, we use the standard covering number guarantee (cf. Duchi [60] Ch. 4) to get

$$P\left(\sup_{f\in\mathcal{F}}|P_m f - Pf| \geq t\right) \leq \exp\left(-\tfrac{mt^2}{18\beta^2} + \log N(t/3, \mathcal{F}, \|\cdot\|_\infty)\right)$$

Recall that Theorem 2.2 shows that if the loss is smooth and $\mu-$strongly convex, then

$$f(x_t) - f(x^*) \leq (1-\alpha\mu)^t \left(f(x_0) - f(x^*)\right)$$

where α is the learning rate. Therefore,

$$\begin{aligned} f(x_{t+1}) - f(x_t) &\leq (1-\alpha\mu)^{t+1} \\ &\leq \exp(-\alpha\mu(t+1))\left(f(x_0) - f(x^*)\right) \end{aligned}$$

because $f(x_t) \geq f(x^*) \forall t$.

[5]Assuming the network has sufficient capacity.

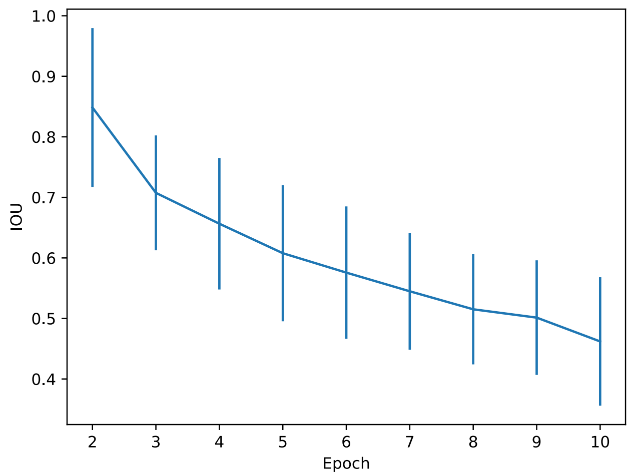


Figure 4.5: IoU of the 10 lowest strong convexity configurations after each epoch for the first 10 epochs.

This implies exponentially decaying benefit as the number of epochs increases. On the other hand, Theorem 2.2 also implies that for vanilla gradient descent, the number of steps required for convergence is inversely proportional to the strong convexity, so that minimizing the latter implies a greater number of steps is required to converge (which increases the runtime). To reduce the impact of this, we use the Adam [115] optimizer. We leave it to future work to explore additional strategies, such as large adaptive learning rates, which can also lead to flatter losses [102].

As an aside, we tested the validity of the one epoch training cycle. Specifically, we sought to answer: *do more epochs provide a better estimate of the strong convexity?* The answer, unsurprisingly, is yes. However, we argue that this is a exploration-exploitation trade-off. To test this, we computed the strong convexity after each epoch, for the first 10 epochs, on the HPOBench datasets. Next, we measured the Intersection over Union (IoU) metric of the 10 lowest strong convexity values after each epoch with the 10 lowest after the first epoch[6], since those are the ones that would be evaluated at the end. Of course, the IoU metric decreases slowly as the epochs go on. After 10 epochs, this IoU had a mean of 47% (see Figure 4.5). This means that after 10 epochs, about half the configurations in the top 10 are the same as after epoch 1. We argue that this is an exploration-exploitation tradeoff: by training for further epochs, we obtain a more

[6]The IoU is a popular metric that computes the ratio implied by its name. Most notably, it is used in object detection [172].

accurate estimate, at additional computational expense. We maintain that the one-epoch training cycle balances the exploration-exploitation trade-off well, since it is not necessary for a landscape to have flat minima to generalize well [57, 221].

Our derivations for the strong convexity proceeds quite similarly to the smoothness-based derivations above:

Lemma 6. *For a neural network with ReLU activations in the hidden layers and a softmax activation at the last layer,*

$$\frac{\partial E}{\partial z_j^{[L]}} = \frac{1}{m}\left(\sum_{i=1}^{m}[y^{(i)} = j]\right)\left(\sum_{h=1}^{k} a_j^{[L]} - \delta_{hj}\right)$$

under the cross-entropy loss.

Proof. We will use the chain rule, as follows:

$$\frac{\partial E}{\partial z_j^{[L]}} = \frac{\partial E}{\partial a_j^{[L]}} \cdot \frac{\partial a_j^{[L]}}{\partial z_j^{[L]}} \tag{4.9}$$

Consider the Iverson notation version of the general cross-entropy loss:

$$E(a^{[L]}) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{h=1}^{k}[y^{(i)} = h]\log a_h^{[L]}$$

Then the first part of (4.9) is trivial to compute:

$$\frac{\partial E}{\partial a_j^{[L]}} = -\frac{1}{m}\sum_{i=1}^{m}\frac{[y^{(i)} = j]}{a_j^{[L]}} \tag{4.10}$$

The second part is the derivative of the softmax and is equal to

$$\frac{\partial a_j^{[L]}}{\partial z_p^{[L]}} = a_j^{[L]}(\delta_{pj} - a_p^{[L]}) \tag{4.11}$$

Combining (4.10) and (4.11) in (4.9) gives

$$\frac{\partial E}{\partial z_j^{[L]}} = \frac{1}{m}\left(\sum_{i=1}^{m}[y^{(i)} = j]\right)\left(\sum_{h=1}^{k} a_j^{[L]} - \delta_{hj}\right) \tag{4.12}$$

$\square$

$$\frac{\partial E}{\partial z_j^{[L]}} = \frac{1}{m}\left(\sum_{i=1}^{m}[y^{(i)} = j]\right)\left(a_j^{[L]} - [y^{(i)} = j]\right) \tag{4.13}$$

Theorem 4.7 (Strong convexity for neural classifiers)**.** *For a neural network with ReLU activations in the hidden layers, a fully connected penultimate layer, and a softmax activation at the last layer, the strong convexity of the cross-entropy loss is given by*

$$\inf \left\|\nabla_W^2 E\right\| \propto \frac{1}{m} \inf \frac{\|a_j^{[L-1]}\|}{\|W^{[L]}\|} \tag{4.14}$$

Proof. For a neural network with ReLU activations, Lemma 6 gives us (using the chain rule one step further):

$$\nabla_{W_j^{[L]}} E = \frac{1}{m}\left(\sum_{i=1}^{m}[y^{(i)} = j]\right)\left(\sum_{h=1}^{k} a_j^{[L]} - \delta_{hj}\right) a_j^{[L-1]}$$

Therefore,

$$\begin{aligned}
\nabla^2_{W_j^{[L]}} E &= \frac{1}{m}\left(\sum_{i=1}^{m}[y^{(i)} = j]\right)\left(\sum_{h=1}^{k} a_j^{[L]} - \delta_{hj}\right) \nabla_{W_j^{[L]}} a_j^{[L-1]} \\
&= \frac{1}{m}\left(\sum_{i=1}^{m}[y^{(i)} = j]\right)\left(\sum_{h=1}^{k} a_j^{[L]} - \delta_{hj}\right) \frac{\partial a_j^{[L-1]}}{\partial E}\frac{\partial E}{\partial W_j^{[L]}} \\
&= \frac{1}{m^2}\left(\sum_{i=1}^{m}[y^{(i)} = j]\right)^2\left(\sum_{h=1}^{k} a_j^{[L]} - [h = j]\right)^2 \frac{\partial a_j^{[L-1]}}{\partial E} a_j^{[L-1]} \\
&= \frac{1}{m^2}\left(\sum_{i=1}^{m}[y^{(i)} = j]\right)^2\left(\sum_{h=1}^{k} a_j^{[L]} - [h = j]\right)^2 \frac{1}{\frac{\partial E}{\partial a_j^{[L-1]}}} a_j^{[L-1]}
\end{aligned}$$

Using $z^{[l]} = W^{[l]T} a^{[l-1]} + b^{[l]}$ and Lemma 6 gives us:

$$\begin{aligned}
\frac{\partial E}{\partial a_j^{[L-1]}} &= \frac{\partial E}{\partial z_j^{[L]}}\frac{\partial z_j^{[L]}}{\partial a_j^{[L-1]}} \\
&= \frac{1}{m}\left(\sum_{i=1}^{m}[y^{(i)} = j]\right)\left(\sum_{h=1}^{k} a_j^{[L]} - [h = j]\right) W_j^{[L]}
\end{aligned}$$

The limiting case of this is when all softmax values are 0, so that the second summation term is -1. The first summation is a positive constant, which we drop as a proportionality constant. Now, (4.14) immediately follows. □

Corollary 2 (Strong convexity for logistic regression)**.** *For a logistic regression model,* *the strong convexity is given by*

$$\inf\left\|\nabla^2 E\right\| \propto \frac{1}{m} \inf \frac{\|X\|}{\|W\|}$$

The proof is trivial and follows directly from the theorem.

Theorem 4.8 (Strong convexity for Naive Bayes)**.** *For Naive Bayes, the strong convexity is given by*

$$\mu = \inf\|\boldsymbol{\Sigma}^{-1}\mathcal{A}G - \frac{1}{2}\boldsymbol{\Sigma}^{-1}\mathcal{A}\boldsymbol{\Sigma}^{-1} + G\mathcal{A}\boldsymbol{\Sigma}^{-1}\|$$

where $\boldsymbol{\Sigma}$ is the covariance matrix, $\mathcal{A}$ is the fourth-order identity tensor, and $G = \dfrac{\partial \ell}{\partial \boldsymbol{\Sigma}}$, where ℓ is the log-likelihood.

The proof is exactly the same as for the smoothness case.

Algorithm 2: `AHSC`

1: **Input:** Number of configurations to sample N_1, number of configurations to run N_2. Defaults: $N_1 = 50, N_2 = 10$.
2: $\mathcal{H}_0 \leftarrow \textsc{Random}(\mathcal{H}, N_1)$
3: $S \leftarrow \phi$ {Strong convexity values}
4: $P \leftarrow \phi$ {Performance scores}
5: **for** config h in $\mathcal{H}_0$ **do**
6:     Train for one epoch using h
7:     $\mu_{max} = -\infty$
8:     **for** mini-batch $x \subset X$ **do**
9:         $\mu_{max} = \max(\mu_{max}, \|\nabla^2 f(x; h)\|)$
10:     **end for**
11:     **if** $\mu_{max} > 0$ **then**
12:         S[h] $\leftarrow \mu_{max}$
13:     **end if**
14: **end for**
15: **for** config h in $\textsc{Lowest}$(S, N_2) **do**
16:     P[h] $\leftarrow \textsc{Run}$(h)
17: **end for**
18: **return** $\arg\max P$

Algorithm 2 shows our overall approach. We first sample N_1 random configurations (line 2). For each of these configurations, we first train the model for one epoch to bring the weights closer to their final weights (line 6). Training for a single epoch provides a balance between the cost associated with training fully (which would provide a more accurate estimate for the strong convexity), and not training at all (which provides a very poor estimate). We then compute $\|\nabla^2 E(x)\|$ in a mini-batch fashion (lines 8-10). Since we wish to minimize the strong convexity, it is important that we look at the highest value across mini-batches, and aim to minimize that upper bound. Importantly, if the strong convexity is 0 (implying the function is *not* strongly convex), we discard that configuration (lines 11-13). We pick the configurations corresponding to the N_2 lowest values of strong convexity as computed above, and train those models fully (lines 15-17). Finally, we return the best-performing configuration.

4.4 Boundary engineering

One insight from this dissertation is that the decision boundaries generated by neural networks are *malleable*, i.e., they can be reshaped *locally*. Techniques that perform such local reshaping of the decision boundary, we call boundary engineering techniques. Our literature review returned little work that matches this description of boundary engineering: on searching Google Scholar using the terms "decision boundary", "decision boundary reshape", "decision boundary push", and "decision boundary change", we found few papers seeking explicitly to this, and a passing reference in [212] that Platt scaling ([167]) can change the decision boundary, although this does not seem to be its primary purpose. However, we list the techniques we did find here. [188] use a conditional entropy loss to push the decision boundary away from samples. [226] discuss how the cross-entropy loss and their cosine softmax learner both push the decision boundary away from samples; however, in our datasets, we find that the noise found in real-world data invalidates this. [187] discusses how large margin classifiers generalize better, and [250] propose mixup, which enforces the prediction model to change linearly in between samples; this is done by making the model f predict $\mathrm{Mix}(y, y')$ at location $\mathrm{Mix}(x, x')$ for random pairs of samples (x, y) and (x', y'). Finally, [210] propose "interpolation consistency training", whose effect is to push the decision boundary into low-density regions.

Our approach to boundary engineering (which we call fuzzy sampling) is rather brute-force: we add in samples around existing data points concentrically; this forces the decision boundary to move away in order to reduce the loss function. Specifically, we add in $\lfloor \log_2(1/n) \rfloor$ concentric boxes, where n is the fraction of samples in the minority class. The number of points added in follows an exponential decay rule, with each box having half the samples as the one preceding it. Algorithm 3 describes our boundary engineering

Algorithm 3: Boundary engineering: fuzzy sampling

```
for each sample x in the minority class do
    for i ∈ {0, 1, ...} such that (1/n)/2^i ≥ 1 do
        Add (x ± iΔr, c_0) to the training set (1/n)/2^i times;
    end
end
```

technique.

4.5 Instance engineering

We define instance engineering methods as those that modify the dataset, but without the express intent to reshape the decision boundary. For example, oversampling methods fall under this category.

Our choice of instance engineering is an oversampling technique called SMOTE, proposed by [40]. SMOTE adds samples between two points of the same class. SMOTE has the benefit over random oversampling (which adds in points at the same location as existing data samples) that it fills in regions that might otherwise be empty in the input space. This can work to improve the learner's confidence in regions with few samples.

4.6 Label engineering

Learning algorithms often make assumptions about the data, such as all of it coming from the same distribution. An inherent assumption is also that the labels are correct; but as pointed out by Cordeiro & Carneiro [47], this is often not true. For example, noisy labels can occur when human annotators are present ([143]) and have divergent opinions, as seen in medical images ([21, 134]). Our static code analysis case study is an excellent reminder of this; of 13,128 original labels, [110] could extract only 768 (5.9%) of correct labels. Although our labels were re-checked by the authors of Kang et al. [110], we assume that some labels are noisy. To rectify the situation, we attempt a simple kind of semi-supervised learning, following Occam's razor. First, given n training samples (and therefore, n labels), we keep $\sqrt{n}$ at random and discard the rest. Next, for each "unknown" label, we look at the $\sqrt{\sqrt{n}} = \sqrt[4]{n}$ nearest labels. We take the mode of these labels and assign it as the label for that point. This has the effect of removing outliers in the data. While primitive, our ablation study (of Figure 5.3) confirms that this indeed helps our learning process and improves results. Therefore, we caution researchers to not trust the labels they are given. The details of the number of labels used is given in Table 2.1. From the table, we use 58 / (58+554) = 9.5% labels (which, as we describe later, is $\sqrt{n}$ labels per dataset).

4.7 Overall method

We now have the tools to develop our overall method for each task.

4.7.1 Defect prediction

We can now discuss our overall method for defect prediction, which is a combination of these techniques. Whenever we use label engineering, we call our approach *GHOST2*; otherwise, we call it *GHOST* (from the original paper).

Algorithm 4: GHOST(2)

```
Split data into train and test;
if GHOST2 then
    Perform SSL (label engineering);
    Perform SMOTE (instance engineering);
end
Perform fuzzy sampling (boundary engineering);
Use an autoencoder to reduce dimensionality;
Perform fuzzy sampling;
Use SMOTE;
Perform hyper-parameter optimization of a feedforward network;
```

Algorithm 4 describes our overall approach. Essentially, it is a specific ordering of the different techniques we discussed above. If we are using GHOST2, we first perform label engineering, and fill in sparse regions of the data using instance engineering. Next, we perform boundary engineering; a stacked autoencoder is used to reduce dimensionality. We then perform boundary engineering a second time followed by SMOTE (to rebalance classes); finally, we use DODGE to perform hyper-parameter optimization.

4.7.2 Bugzilla issue lifetime prediction

Our method for Bugzilla issue lifetime prediction is a subset of GHOST2; specifically, it merely uses the DODGE hyper-parameter optimization algorithm over feedforward

Algorithm 5: SIMPLE

```
Set random number seed;
for 20 times do
    Shuffle data;
    Set train, test = 70%,30% splits of the data;
    /* Learning                                                  */
    Apply a feedforward neural network.; On the training data, tune the
      hyper-parameters of Section 4.8.2 using DODGE.;
    Take the best model found from the training data, apply it to the test data;
    Report performance scores on the test data. ;
end
```

networks. We find that this suffices to outperform the prior state-of-the-art, DeepTriage. This method, which we call `SIMPLE`, is shown in Algorithm 5.

4.7.3 Code smell detection

Our overall method for code smell detection is the same as for defect prediction, i.e., we use GHOST2 with fuzzy sampling applied twice. The only change is that we use TPE instead of DODGE for hyper-parameter optimization.

4.7.4 Static code warnings

For detecting false alarms in static code warnings, we studied

- 4 engineering methods (boundary, label, parameter, instance) that could be mixed in $2^4 = 16$ ways.
- Six traditional learners: logistic regression, decision trees, random forests, and SVMs (with 3 kernel 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 learners, we only show the results of the best 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 an ablation study recommended in Cohen's *Empirical Methods for AI* textbook [46]. Ablation studies let us explore some combination of N parts in $O(N)$ time, and 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 4.1.

In that table, for treatments that use any of boundary or label or parameter or instance engineering, we apply those treatments in the order used by the original GHOST method [234]. In Chapter 5, we will show that it can improve recall by 30% (or more) by multiple rounds of SMOTE + GHOST. As per that advice, A1 executes our pre-processors in the order:

$$\textit{smooth} \rightarrow \textit{smote} \rightarrow \textit{ghost} \rightarrow \textit{ghost} \rightarrow \textit{smote} \rightarrow \textit{dodge}$$

All that said, it is an open issue if *other* ordering might be *more* useful. We note 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.

4.7.5 Automated microservice partitioning

Background: CO-GCN

Before discussing our method, it is worth discussing more extensively about **CO-GCN** [55], since our method builds on top of it. CO-GCN is designed to mitigate the effect of "outliers" (which they define as classes with overlapping functionalities), while ensuring high cohesion and low coupling. Specifically, two kinds of outliers are identified: *structural* outliers–which have high interaction with classes from different clusters (microservices)– and *attribute* outliers–which have attributes (such as usage patterns) similar to those from other clusters.

Table 4.1: 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 [234]
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. [232] and Kang et al. [110] studies.
CodeBERT			B			100	CodeBERT without modifications

CO-GCN is based on a graph convolutional autoencoder, which requires a graph Laplacian and an attribute matrix. We first consider the construction of the attribute matrix. Define an *entry point* as an element (such as a UI element or an API endpoint) which acts as the first in any method call chain. Let $G = (V, E)$ denote the call graph of the application, where node i, representing some class i, connected to node j if any method of class i calls any method in class j. Let P be the set of entry points. Define the *entry point matrix*, denoted as $E \in \mathbb{R}^{|V|\times|P|}$, as $E_{ip} = 1$ iff class i occurs in the execution trace of entry point p. Define also, a matrix $C \in \mathbb{R}^{|V|\times|V|}$ where C_{ij} is the number of times classes i and j co-occur in any execution trace. Finally, define a symmetric matrix $I \in \mathbb{R}^{|V|\times|V|}$ as $I_{ij} = 1$ iff classes i and j are related by an inheritance relationship. The attribute matrix $X \in \mathbb{R}^{|V|\times(|P|+2|V|)}$ is a concatenation of E, C, and I, in that order.

The graph Laplacian is constructed in a standard manner, using the Ng et al. [155] Laplacian construction over the adjacency matrix. In detail, let A be the adjacency matrix and let $\tilde{A} = A + I$. Construct the degree diagonal matrix $\tilde{D}_{ii} = \sum_j \tilde{A}_{ij}$. Finally, the Ng et al. [155] Laplacian is $\hat{A} = \tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{\frac{1}{2}}$.

A graph convolutional network (GCN) is a network that uses the graph Laplacian and an attribute matrix as inputs. At each layer of the encoder, the following computation is performed:

$$Z^{[l]} = g^{[l]}(\hat{A}Z^{[l-1]}W_e^{[l]})$$

where $Z^{[0]} = X$ and $g^{[l]} = \max(0, x)\forall l \in [L]$ is the ReLU activation function. The decoder is very similar:

$$\hat{X}^{[l]} = g^{[l]}(\hat{A}Z^{[l-1]}W_d^{[l]})$$

where $Z^{[0]} = Z$. CO-GCN uses a 2-layer GCN for the encoder and a 3-layer network for the decoder. It is also equipped with a novel loss function, which we describe now. The loss itself is a linear combination of three losses: a structure loss, an attribute loss, and a cluster loss. For the first two, scalars O_{si} and O_{ai} are defined for each node i, which denote the structural and attribute "outlier-ness" of that node (class). These are constrained as $\sum_i O_{si} = \sum_i O_{ai} = 1$. The first component, the structure loss, ensures that edges link classes that are similar. We first define the structure loss as

$$\mathcal{L}_{str} = \sum_{i\in[|V|]} \log\left(\frac{1}{O_{si}}\right) \left\|A_i - ZZ_i^T\right\|_2^2$$

where A_i and Z_i are the ith rows of the adjacency matrix A and the embedding matrix Z defined above. As we will see, the other loss terms do not depend on O_{si}, so we can use a Lagrangian method to compute it. Because the squared norm is convex and $\log(\cdot)$ is concave (since $0 < O_{si} \leq 1$ so that $-\log(\cdot)$ is convex), this loss is convex. Taking the Lagrangian:

$$\frac{\partial}{\partial O_{si}} \left[\sum_{j \in [|V|]} \log\left(\frac{1}{O_{sj}}\right) \left\| A_j - ZZ_j^T \right\|_2^2 + \lambda \left(\sum_{j \in [|V|]} O_{sj} - 1 \right) \right] = 0$$

This gives us

$$\lambda = \frac{\left\| A_i - ZZ_i^T \right\|_2^2}{O_{si}} \Rightarrow O_{si} = \frac{\left\| A_i - ZZ_i^T \right\|_2^2}{\lambda}$$

But $\sum_i O_{si} = 1$ so that $\lambda = \sum_{j \in [|V|]} \left\| A_j - ZZ_j^T \right\|_2^T$. Therefore,

$$O_{si} = \frac{\left\| A_i - ZZ_i^T \right\|_2^2}{\sum_{j \in [|V|]} \left\| A_j - ZZ_j^T \right\|_2^T}$$

Note that this needs to be updated at each iteration. The attribute loss ensures that the reconstructed node attributes from the decoder match the original attributes. This loss is also designed to ensure that the effect of attribute outliers is reduced:

$$\mathcal{L}_{attr} = \sum_{i \in [|V|]} \log\left(\frac{1}{O_{ai}}\right) \left\| X_i - \hat{X}_i \right\|_2^2$$

Similar to O_{si}, O_{ai} is updated each iteration based on a Lagrangian derivation. It is easy to show that:

$$O_{ai} = \frac{\left\| X_i - \hat{X}_i \right\|_2^2}{\sum_{j \in [|V|]} \left\| X_j - \hat{X}_j \right\|_2^2}$$

Finally, the clustering loss aims to separate the monolith into microservices. Rather than a separated clustering step, it is desirable to train the cluster assignment along with the GCN in an end-to-end fashion, so that it may be jointly optimized. To that end, CO-GCN employs a binary cluster assignment matrix $M \in \{0,1\}^{|V| \times K}$. Recalling that Z_i is the node embedding for the ith cluster, we define for each of K clusters (where K is assumed to be known), a centroid vector $C_i \in \mathbb{R}^F$. Having set up this notation, we define the clustering loss as

$$\mathcal{L}_{clus} = \sum_{i=1}^{|V|} \sum_{j=1}^{K} M_{ij} \|Z_i - C_j\|_2^2$$

It is trivial to assign the clusters in a fashion similar to k-means:

$$M_{ij} = \begin{cases} 1, & j = \underset{k \in [K]}{\arg\min} \|Z_i - C_k\|_2^2 \\ 0, & \text{otherwise} \end{cases}$$

Having set up the cluster assignments, we can now define the centroids in each iteration as

$$C_i = \frac{1}{|\mathcal{C}_i|} \sum_{j \in \mathcal{C}_i} Z_j$$

where $\mathcal{C}_i = \{i \in V : M_{ik} = 1$ is the ith cluster. Finally, we can set up the overall loss for CO-GCN:

$$\mathcal{L} = \alpha_1 \mathcal{L}_{str} + \alpha_2 \mathcal{L}_{attr} + \alpha_3 \mathcal{L}_{clus}$$

To train CO-GCN, O_{si} and O_{ai} are initialized uniformly at random, and cluster assignments and cluster centroids are initialized using k-means++. Then, $\mathcal{L}_{str}$ and $\mathcal{L}_{attr}$ are trained in a pre-training step. After this pre-training stage, the overall loss is minimized.

`DEEPLY`: Our modifications to CO-GCN

The core of our modifications is a hyper-parameter optimization step using TPE. Because there are several metrics to optimize, we employ a weighted loss function for the hyper-parameter optimization algorithm.

One challenge with designing a weighted loss function is how to obtain the required weights. In `DEEPLY`, we first run the algorithm 1,000 times to generate a set of metrics shown in Table 3.6. Then, we check the correlation among the metrics using the Spearman correlation coefficients. If any two metrics have strong correlations we prefer to keep one to remove the redundant metrics for the optimization. Figure 4.6 indicates the correlations among the metrics. Clearly, some of the metrics are highly correlated. Therefore, we use a reduced set of metrics for evaluating our approaches. We set a threshold of 0.6 to prune the set of metrics. We observe that MQ has higher correlations with other metrics than

SM. Since IFN and ICP are highly correlated across all datasets, we arbitrarily choose ICP. This leads to the final set of metrics: BCS, ICP, and SM. Note that to ensure that we do not generate "dust" (individual classes in a partition) or "boulders" (monolithic partitions), we also include NED in the final metric set (for which we assigned a static weight of 0.2).

Based on these three metrics, we choose a loss function to pick one model from the frontier of best solutions. This loss function is

$$\mathcal{L}_H(P) = \sum_i w_i m_i(P) \tag{4.15}$$

where w_i are *weights* assigned to each of the metrics m_i, and P is the partition. Additionally, we divide the BCS weight by 10 (for an overall weight of 0.1) to normalize its value to the same order of magnitude as the other metrics. For SM, we set the weight to $-1/\text{corr}(SM, MQ)$ (the negative sign is required since we wish to *minimize* this loss function but *maximize* SM). Similarly, the weight for ICP was set to $1/\text{corr}(BCS, ICP)$. For daytrader, this was unnecessary, and we set it to -1. Finally, we add in a term for the NED metric, so that we do not generate "dust" or "boulders" (e.g., a monolith with 20 classes is not partitioned into 19 and 1). For this, we found that a weight of 0.2 was a good balance between all the metrics.

Therefore, the final loss function we use to select from the Pareto frontier is

$$\mathcal{L}_H(P) = \frac{\rho(BCS, IFN)}{10} BCS(P) + \rho(ICP, IFN) ICP(P) - \rho(SM, MQ) SM(P) + 0.2NED$$

where ρ is the Spearman correlation coefficient.

Next, we change the clustering method used in CO-GCN ([55]). Rather than using the k-means clustering, which assumes globular clusters, we use spectral clustering, which can produce more flexible cluster shapes ([237]). We also change the cluster loss function to

$$\mathcal{L}_{clus} = \sum_i \min_j |x_i - x_j| \tag{4.16}$$

where x_i and x_j are data points.

Next, we update the learning rate scheduling policy to the "1cycle" policy ([190, 192]), which has been shown to lead to faster convergence through the "super-convergence" phenomenon. We update the learning rate every 200 steps.

Algorithm 6 shows the overall approach. In the algorithm the lines in red (19-22, 25) are from the CO-GCN algorithm (and all else is our extension). We first collect correlation statistics using a Monte Carlo-style data collection (lines 1-4). Then we set weights (lines 4-7) and run hyperopt for 100 iterations (lines 10-15), while collecting the results of each run. The weighted loss function then picks the “best” candidate in line 16, and returns the optimal hyper-parameters in line 17. The actual model training itself is delegated to `CO-GCN-modified`.

We form the graph Laplacian in lines 18-20, and pretrain (for 1 epoch) the encoder and decoder. We initialize the clusters in line 23, but use spectral clustering instead. Then, we run the actual training for 300 epochs.

4.8 Hyper-parameter space

We describe here, the space of hyper-parameter choices used when performing hyper-parameter optimization for our experiments. Note that in all cases, we also treat the set of preprocessors, along with their mentioned hyper-parameters as part of the overall hyper-parameter space.

4.8.1 Defect prediction

We use the following range of hyper-parameter options for defect prediction.

- **Preprocessor:** Standardization, normalization, or min-max scaling
- **Number of layers:** $\mathcal{U}(2, 5)$
- **Number of units per layer:** $\mathcal{U}(2, 6)$

4.8.2 Bugzilla issue lifetime prediction

Preprocessors:

- StandardScaler : i.e. all input data set numerics are adjusted to $(x - \mu)/\sigma$.
- MinMaxScaler (range = (0, 1)): i.e. scale each feature to $(0, 1)$.
- Normalizer (norm = $\mathcal{U}$([‘l1’, ‘l2’,‘max’])): i.e. normalize to a unit norm.
- MaxAbsScaler (range = $\mathcal{U}(0, 1)$): scale each feature by its maximum absolute value
- Binarizer (threshold = $\mathcal{U}(0, 100)$), i.e., divide variables on some threshold

Algorithm 6: `DEEPLY`. Note *CO-GCN-modified* is a sub-routine containing our changes to the original CO-GCN system. The the lines in red (18-21,24) are from the original CO-GCN algorithm (and all else is our extension).

1 **Algorithm** *DEEPLY*
  Input : graph dataset D
  Output: optimal hyper-parameters θ^*
2 **for** *1,000 iterations* **do**
3   run `CO-GCN-modified`(D) with random settings and collect metrics
4 **end**
5 $w_{SM} \leftarrow -1/corr(SM, MQ)$ `// initialize weights`
6 $w_{ICP} \leftarrow 1/corr(BCS, ICP)$
7 $w_{NED} \leftarrow 0.2, w_{BCS} \leftarrow 0.1$
8 $\theta = \phi$ `// initialize our history to empty set`
9 $M = \phi$ `// initialize results to empty`
10 **for** *i=1..100* **do**
11   $\theta_i \leftarrow$ hyperopt(θ)
12   $M_i \leftarrow$ `CO-GCN-modified`$(D; \theta_i)$ `// run the model`
13   append θ_i to θ `// log hyper-params`
14   append M_i to M `// log results`
15 **end**
16 $\theta^* = \arg\min_{\theta} \sum_{j=1}^{4} w_j M_j$
17 **return** θ^*
18 **Algorithm** *CO-GCN-modified*
  Input : graph dataset $D = (A, X)$
  Output: partitioned microservices
  `// compute the graph Laplacian`
19 $\tilde{A} \leftarrow A + I$
20 define diagonal matrix $\tilde{D}$ such that $\tilde{D}_{ii} = \sum_j \tilde{A}_{ij}$
21 $\hat{A} \leftarrow \tilde{D}^{-\frac{1}{2}} \tilde{A} \tilde{D}^{-\frac{1}{2}}$
22 pretrain the encoder and decoder using $\mathcal{L}_{str}$ and $\mathcal{L}_{attr}$
23 initialize the clusters using spectral clustering on embeddings
24 train the network (E, D) for 300 epochs using Alg. 1 lines 5-9 from ([55]) and 1cycle ([192])
25 get embeddings using encoder
26 cluster the embeddings using spectral clustering
27 **return** *clusters*

Hyper-parameters:

- Number of layers
- Number of units in each layer
- Batch size (i.e., the number of samples processed at a time)

4.8.3 Code smell detection

For code smell detection, we optimized the following set of hyper-parameters using TPE:

- **Number of layers:** $\mathcal{U}(5, 20)$
- **Number of units per layer:** $\mathcal{U}(2, 7)$

4.8.4 Static code warnings

We use the following set of hyper-parameter options for the different learners tested for the static code warnings experiments.

- **Feedforward networks:** Number of layers from $\mathcal{U}(2, 6)$, number of units per layer in $\mathcal{U}(3, 20)$
- **Logistic regression:** l_1 or l_2 penalty and $C \in \{0.1, 1, 10, 100\}$
- **Decision trees:** Gini index or entropy impurity, and best or random feature splitter
- **Random forests:** Gini index or entropy impurity, and $\mathcal{U}(10, 100)$ trees
- **SVM:** $C \in \{0.1, 1, 10, 100\}$ and a sigmoid, RBF, or polynomial kernel
- **CNN:** Number of convolutional blocks from $\mathcal{U}(1, 5)$, number of convolutional filters from $\{4, 8, 16, 32, 64\}$, dropout probability from $\mathcal{U}(0.05, 0.5)$, and kernel size from $\{16, 32, 64\}$.

4.8.5 Automated microservice partitioning

We tune the following set of hyper-parameters:

Table 4.2: Software projects data used in defect prediction. For an explanation of attributes used in this data, see Table 4.3.

Project	Train versions	Test versions	Training Buggy %	Test Buggy %
ivy	1.1, 1.4	2.0	22	11
lucene	2.0, 2.2	2.4	53	60
poi	1.5, 2.0, 2.5	3.0	46	65
synapse	1.0, 1.1	1.2	20	34
velocity	1.4, 1.5	1.6	71	34
camel	1.0, 1.2, 1.4	1.6	21	19
jEdit	3.2, 4,0, 4.1, 4.2	4.3	23	2
log4j	1.0, 1.1	1.2	29	92
xalan	2.4, 2.5, 2.6	2.7	38	99
xerces	1.2, 1.3	1.4	16	74

- **Number of units in layers:** The decoder in set up as a mirrored version of the encoder. As such, only the number of units in each encoder layer needs to be tuned. For the first layer, the number of units is the space $\mathcal{U}(8, 64)$. For the second layer, we use the space $\mathcal{U}(4, 32)$.
- **Training loss weights:** We tune each α_i is the loss function used by CO-GCN, and each is given the range $\mathcal{U}(0, 1)$.
- **Dropout probability:** The dropout probability is from $\mathcal{U}(0.2, 0.5)$.

4.9 Datasets

We now describe the datasets used in this dissertation.

4.9.1 Defect prediction

For this study, we use the same data used in DODGE's prior study [1] on defect prediction: see Table 4.2. These are all open-source Java projects from the PROMISE repository [183]. Each dataset has 20 static attributes for each software project, described in Table 4.3.

4.9.2 Bugzilla issue lifetime prediction

To obtain a fair comparison with the prior state-of-the-art, we use the same data as used in the prior study (DASENet) [121]. One reason to select this baseline is that we were able to obtain the data used in the original study (see our reproduction package) and,

Table 4.3: Static attributes for the classes seen in the Table 4.2 data.

Attribute	Description (with respect to a class)
wmc	Weighted methods per class [42]
dit	Depth of inheritance tree [42]
noc	Number of children [42]
cbo	Coupling between objects [42]
rfc	Response for a class [42]
lcom	Lack of cohesion in methods [42]
lcom3	Another lcom metric proposed by Henderson-Sellers [90]
npm	Number of public methods [20]
loc	Number of lines of code [20]
dam	Fraction of private or protected attributes [20]
moa	Number of fields that are user-defined types [20]
mfa	Fraction of accessible methods that are inherited [20]
cam	Cohesion among methods of a class based on parameter list [20]
ic	Inheritance coupling [202]
cbm	Coupling between methods [202]
amc	Average method complexity [202]
ca	Number of classes depending on a class [202]
ce	Number of classes a class depends on [202]
max_cc	Maximum McCabe's cyclomatic complexity score of methods [142]
avg_cc	Mean of McCabe's cyclomatic complexity score of methods [142]

Table 4.4: Issue close time prediction data. From Lee et al. [121] study. Note that because of the manner of data collection, i.e., using bin-sequences for each day for each report, there are many more data samples generated from the number of reports mined.

Project	**Observation Period**	**# Reports**	**# Train**	**# Test**
Eclipse	Jan 2010–Mar 2016	16,575	44,545	25,459
Chromium	Mar 2014–Aug 2015	15,170	44,801	25,200
Firefox	Apr 2014–May 2016	13,619	44,800	25,201

therefore, were able to obtain results comparable to prior work. For a summary of that data, see Table 4.4.

Table 4.5: Applications used for the code smell detection study. Data is taken from Liu et al. [131].

Application	Version	#classes	# methods	#samples
JUnit	4.10	123	866	80,984
PMD	5.2.0	250	2,097	80,680
JExcelAPI	2.6.12	424	3,118	71,116
Areca	7.4.7	473	5,055	76,364
Freeplane	1.3.12	787	6,938	74,408
jEdit	4.5.0	513	5,964	68,352
Weka	3.9.0	1,348	20,182	59,036
AbdExtractor	20140630	1,695	12,608	75,156
Art of Illusion (AoI)	3.0	492	6,188	67,268
Grinder	3.6	502	3,037	78,588

4.9.3 Code smell detection

We use the datasets provided by Liu et al. [131]. Briefly, they generate the data for different applications by applying *smell-inducing refactoring*, i.e., a set of refactoring steps that induce code smells in an otherwise well-designed application. For example, moving a method from one class (where it should be) to another is expected to induce *feature envy*.

To generate their training data, positive examples are generated as follows: a set of potential smell-inducing refactorings is constructed, and from this set, sampling without replacement is performed to generate positive examples for each type of code smell. For negative examples, *software entities* (i.e., methods for feature envy and long method, and classes for misplaced class and large class) are considered, and sampling without replacement is done.

The subject applications are listed in Table 4.5.

4.9.4 Static code warnings

We use the datasets from Kang et al. [110]. That paper manually labelled warnings from the same projects studied by Yang et al. [232] 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

Table 4.6: 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

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. [110] paper against earlier work [232] was that for data that comes with some timestamp, it is inappropriate to use future data to predict past labels. To avoid that problem, in this study, we sorted the fixed data by timestamps, 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 4.6 (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

Table 4.7: Statistics about the datasets used in this study.

Application	#classes	#methods	#Runtime traces	Class coverage	Method coverage
acmeair	33	163	11	28 (84%)	108 (66%)
daytrader	109	969	83	73 (66%)	428 (44%)
jpetstore	66	350	44	36 (54%)	236 (67%)
plants	37	463	43	25 (67%)	264 (57%)

eight broad categories. They are summarized in Table 3.3.

4.9.5 Automated microservice partitioning

We use four open source projects to evaluate our approach against prior work. These are *acmeair*[7] (an airline booking application), *daytrader*[8] (an online stock trading application), *jpetstore*[9] (a pet store website), and *plants*[10] (an online web store for plants and pets). These applications are built using Java Enterprise Edition (J2EE), and common frameworks such as Spring and Spring Boot.

In Table 4.7, we show statistics about the datasets (number of classes and methods) and the runtime traces that we used (number of traces, and their class and method coverage). While the coverage here may seem low, we note that applications tend to have a significant amount of "dead", or unreachable code. For example, Brown et al. [34] found between 30 to 50% of an industrial software system was dead code. Eder et al. [62] found that for an industrial software system written in .NET, 25% of method genealogies were dead. Therefore, we were not worried about the coverage seeming low.

4.10 Statistics

The statistical methods used in this dissertation were selected according to the particulars of our experiments.

For example, in the *within-project defect prediction experiments*, we are using learners that employ stochastic search. When testing such algorithms, it is standard ([11]) to repeat those runs 20 times with different random number seeds. Such experiments generate

[7] `https://github.com/acmeair/acmeair`
[8] `https://github.com/WASdev/sample.daytrader7`
[9] `https://github.com/mybatis/jpetstore-6`
[10] `https://github.com/WASdev/sample.plantsbywebsphere`

a distribution of 20 results per learner per data set. For those experiments, we use the *distribution statistics* of §4.10.1 to compare the efficacy of different learners.

For the cross-project defect prediction experiments, we are comparing our results against the methods of [216], However, although we do not have access to their implementations, we do have access to the train and test sets they use. For these experiments, we must compare our results to the single performance points mentioned in the [216] study. For those experiments, we use the *point statistics* of §4.10.2 to compare the efficacy of different learners. We also use point statistics in static code analysis results, because of how few data samples we had.

4.10.1 Distribution statistics

Distribution statistics ([12, 75]) are used to distinguish two distributions of data. For example, in our experimental setup for defect prediction, we run GHOST and DODGE 20 times each, and therefore have a distribution of results for each optimizer. This allows us to use distribution statistical methods to compare results.

Our comparison method of choice is the Scott-Knott test, which was endorsed at TSE'13 ([150]) and ICSE'15 ([75]). The Scott-Knott test is a recursive bi-clustering algorithm that terminates when the difference between the two split groups is insignificant. Scott-Knott searches for split points that maximize the expected value of the difference between the means of the two resulting groups. Specifically, if a group l is split into groups m and n, Scott-Knott searches for the split point that maximizes

$$\mathbb{E}[\Delta] = \frac{|m|}{|l|}\left(\mathbb{E}[m] - \mathbb{E}[l]\right)^2 + \frac{|n|}{|l|}\left(\mathbb{E}[n] - \mathbb{E}[l]\right)^2$$

where $|m|$ represents the size of the group m.

The result of the Scott-Knott test is *ranks* assigned to each result set; higher the rank, better the result. Scott-Knott ranks two results the same if the difference between the distributions is insignificant.

4.10.2 Point statistics

For point statistics, we have access to various performance points (e.g. recall) across multiple data sets. To determine if one point is better than another, we have to define a delta Δ below which we declare two points are the same.

To that end, we use recommendations from [175]. Rosenthal comments that for point

Table 4.8: The evaluation metrics

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

statistics, parametric methods have more statistical power (to distinguish groups) than nonparametric methods. Further, within the parametric methods, there are two families of methods: those that use the "r" Pearson product moment correlation; and those that use some "d" normalized difference between the mean.

After making those theoretical points Rosenthal goes on to remark that neither method is intrinsically better than another. Using Rosenthal's advice, we apply the most straightforward method endorsed by that research. Specifically, we compare treatment performance differences using Cohen's delta, which is computed as $\Delta = 0.3 * \sigma$ where σ is the standard deviation of all the values seen across all the data sets. When two methods are different by less than Δ, we say that they *perform equivalently*. Otherwise, we say that one method *out-performs* the other if its performance is larger than Δ.

4.11 Performance Metrics

We use different performance metrics to evaluate our results for each case study. We discuss the performance metrics used for each case study separately below.

4.11.1 Defect prediction

For this study, we use performance metrics widely used in the defect prediction literature. Specifically, we evaluate the performance of our learners using three metrics: Area Under the ROC Curve (AUC) with the false positive rate on the x-axis and true positive rate on the y-axis, recall, false alarm rate, and popt20. We use these measures to compare against the baseline of [1].

- Recall measures the number of positive examples that were correctly classified by a learner.

- The false alarm rate (which we denote by *pf*) is the fraction of negative samples a classifier incorrectly classifies as a positive example.

- The next metric, popt20 comments on the inspection effort required after a defect predictor is run. To calculate popt20, we list the defective modules before the non-defective modules, sorting both groups in ascending order of lines of code. Charting the percentage of defects that would be recalled if we traverse the code sorted in this manner in the y-axis, we report the value at the 20% point.

- Finally, we report the area under the receiver operating characteristics (ROC) curve (AUC), with false positive rate on the x-axis and true positive rate on the y-axis.

A key benefit of using multiple metrics, aside from a more complete view of the model's performance, is that it allows us to more concretely test whether deep learning works for all situations. Since some metrics are more important for certain stakeholders (e.g., popt20 comments on the effort needed after the defect predictor are run; others may prefer a combination of high recalls and low false alarm rates, i.e., d2h), this exploration allows us to point out the cases where deep learning may not be the most suitable option.

4.11.2 Bugzilla issue lifetime prediction

We adopt the same performance scores as that seen in prior work.Lee et al. [121] use the following two metrics in their study:

- **Accuracy** is the percentage of correctly classified samples. If TP, TN, FP, FN are the true positives, true negatives, false positives, and false negatives (respectively), then *accuracy* is $(TP + TN)/(TP + TN + FP + FN)$.

- **Top-2 Accuracy**, for multi-class classification, is defined as the percentage of samples whose class label is among the two classes predicted by the classifier as most likely. Specifically, we predict the probabilities of a sample being in each class, and sort them in descending order. If the true label of the sample is among the top 2 classes ranked by the classifier, it is marked as "correct".

4.11.3 Static code analysis

The static code analysis literature has used four metrics: AUC-ROC (area under the receiver operating characteristics curve), false alarm rate, recall, and precision. We use these same metrics in our comparisons. Table 4.8 lists these metrics along with their descriptions.

4.11.4 Code smell detection

The prior state-of-the-art in code smell detection ([131]) used four metrics: precision, recall, F1 score, and AUC-ROC. All of these metrics have been defined above.

4.11.5 Automated microservice partitioning

We choose a total of five metrics to evaluate our core hypodissertation that hyper-parameter tuning improves microservice extraction algorithms. These are detailed below. These metrics have been used in prior studies, although different papers used different set of metrics in their evaluations. For fairness, we use metrics from all prior papers. These metrics evaluate different aspects of the utility of an algorithm that might be more useful to different sets of users (detailed in the RQ1), e.g., BCS evaluates how well different business use cases are separated across the microservices.

Inter-partition call percentage (ICP) ([107]) is the percentage of runtime calls across different partitions. For lower coupling, *lower* ICP is better.

Business context sensitivity (BCS) ([107]) measures the mean entropy of business use cases per partition. Specifically,

$$BCS = \frac{1}{K}\sum_{i=1}^{K}\frac{BC_i}{\sum_j BC_j}\log_2\left(\frac{BC_i}{\sum_j BC_j}\right)$$

where K is the number of partitions and BC_i is the number of business use cases in partition i. Because BCS is fundamentally based on entropy, *lower* values are better.

Structural modularity (SM), as defined by Jin et al. [104], combines cohesion and coupling, is given by

$$
\begin{aligned}
SM &= \frac{1}{K}\sum_{i=1}^{n}\frac{coh_i}{N_i^2} - \frac{1}{N_i(N_i-1)/2}\sum_{i<j<K} coup_{i,j} \\
coh_i &= \frac{2}{m(m-1)}\sum_{\substack{c_i,c_j\in C\\ i<j}}\sigma(c_i,c_j) \\
coup_{i,j} &= \frac{\sum\limits_{\substack{c_1\in C_i\\ c_2\in C_j}}\sigma(c_1,c_2)}{|C_i||C_j|}
\end{aligned}
$$

where N_i is the number of classes in partition[11] i, K is the number of partitions, coh_i is the cohesion of partition i, and $coup_{i,j}$ is the coupling between partitions i and j, c_i refers to a class in the subset, C_i, C_j are two partitions, σ is a general similarity function bounded in $[0, 1]$, and the $i < j$ condition imposes a general ordering in the classes. Higher values of SM are better.

Modular quality (MQ) ([149]), coined by Mitchell & Mancoridis [149], is defined on a graph $G = (V, E)$ as

$$
\begin{aligned}
MQ &= \frac{2\mu_i}{2\mu_i+\epsilon} \\
\mu_i &= \sum_{(e_1,e_2)\in E}\mathbb{1}(e_1\in C_i \wedge e_2\in C_i) \\
\epsilon &= \sum_{\substack{(e_1,e_2)\in E\\ i\neq j}}\mathbb{1}(e_1\in C_i \wedge e_2\in C_j)
\end{aligned}
$$

where $\mathbb{1}$ is the indicator function, and C_i, C_j are clusters in the partition. Higher values of MQ are better.

The *interface number (IFN)* ([149]) of a partition is the number of interfaces needed in the final microservice architecture. Here, an interface is said to be required if, for an edge between two classes in the runtime call graph, the two classes are in different clusters.

Finally, the *non-extreme distribution (NED)* is the percentage of partitions whose distributions are not "extreme" (in our case, these bounds were set to min=5, max=20).

However, as shown in Figure 4.6, some of these metrics are highly correlated with others. Therefore, to avoid bias in the evaluation and the loss function (i.e., if metrics

[11] Note that we use the term "partition" to refer to both the set of all class subsets, as well as an individual subset, but it is typically unambiguous.

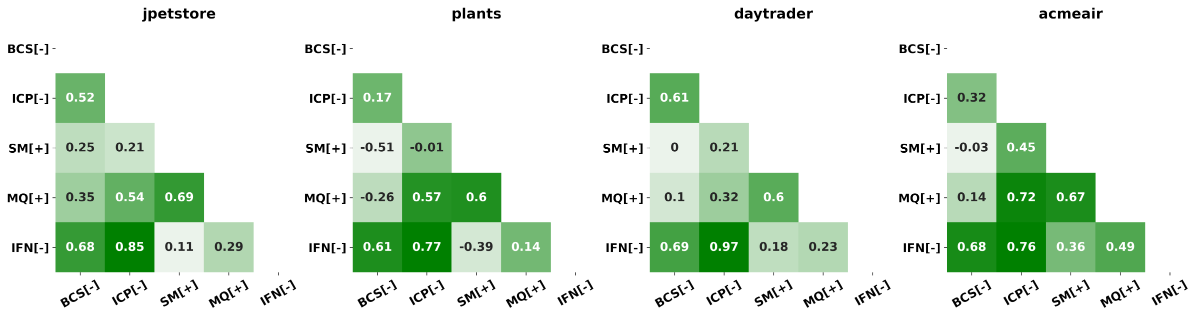


Figure 4.6: Spearman ρ among automated microservice partitioning metrics from 1,000 runs.

M_1 and M_2 are correlated, and optimizer O performs best on M_1, it likely performs best on M_2 as well), we use a subset of these metrics. Specifically, we prune the metric set as discussed in §4.7.5. Note that across all our figures and tables, a "[-]" following a metric means lower values are better, and a "[+]" following a metric means that higher values are better.

CHAPTER

5

RESULTS

This chapter discusses the results of the case studies presented in prior chapters. For each case study, we will introduce research questions with their motivations, and then discuss their answers.

5.1 Defect prediction

5.1.1 Research Questions

We explore the following research questions for defect prediction:

RQ1: *How well does standard deep learning (DL) perform versus the prior state-of-the-art?*

We are critical of SE researchers that merely explore DL without baselining this new method against prior non-DL state-of-the-art. Accordingly, before we do anything else, we make that comparison.

RQ2: *How to fix deep learning for defect prediction?*

The results of RQ1 will show that standard, off-the-shelf deep learning is insufficient for defect prediction. We therefore ask how we can better use deep learning for defect prediction.

RQ3: *Are our methods scalable?*

Table 5.1: RQ1 and RQ2 results. Cells show medians of 20 runs. Cells with an "*" show the few cases where DL worked better than DODGE. Dark gray shows top rank (note: for pf, *less* is *better*), light gray shows rank two; white shows lowest rank (worst performance). Rankings were calculated via §4.10.1. The train and test versions used here are the same as the DODGE paper (see Table 3.1).

		AUC	popt20	recall	pf
ivy	DL	0.58	0.15	0.33	**0.27** *
	DODGE	**0.71**	0.25	0.85	0.36
	GHOST	0.69	**0.31**	**0.95**	0.38
camel	DL	0.55	0.18	0.10	**0.04** *
	DODGE	0.58	**0.54**	0.63	0.36
	GHOST	**0.62**	**0.54**	**0.65**	0.41
jedit	DL	0.61	0.10	0.46	**0.16** *
	DODGE	0.63	0.39	0.64	**0.25**
	GHOST	**0.68**	**0.41**	**0.91**	0.40
log4j	DL	0.55	0.31	0.33	**0.13** *
	DODGE	0.61	**0.99**	0.54	0.22
	GHOST	**0.66**	**0.99**	**0.75**	**0.19**
velocity	DL	0.50	0.57	**0.90** *	0.87
	DODGE	0.61	**0.64**	0.76	**0.47**
	GHOST	**0.68**	0.64	0.82	**0.47**
synapse	DL	0.54	0.23	0.20	**0.05** *
	DODGE	0.65	**0.48**	**0.65**	0.23
	GHOST	**0.67**	**0.48**	0.63	0.33
lucene	DL	0.58	0.51	**0.69** *	0.52
	DODGE	**0.61**	**0.80**	**0.67**	0.36
	GHOST	0.59	**0.80**	**0.7**	**0.34**
xalan	DL	0.55	0.24	0.21	**0.09** *
	DODGE	0.71	**1.0**	0.71	0.14
	GHOST	**0.75**	**1.0**	**0.76**	0.27
xerces	DL	0.52	0.28	0	**0.04** *
	DODGE	0.59	**0.93**	0.54	0.15
	GHOST	**0.62**	**0.94**	**0.57**	0.39
poi	DL	0.61	0.36	0.45	**0.18** *
	DODGE	**0.72**	0.66	**0.78**	0.22
	GHOST	**0.73**	**0.74**	**0.78**	0.38

Having come up with an algorithm in RQ2, we ask if it scales well (so that it will be useful in the real world). We perform a standard scalability analysis, by varying the size of the dataset and checking the change in runtime.

5.1.2 Results

RQ1: *How well does standard deep learning (DL) perform versus the prior state-of-the-art?*

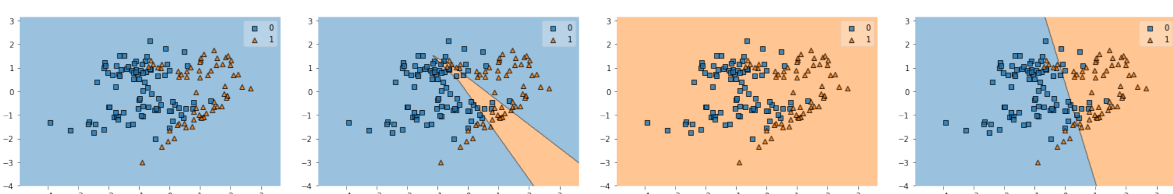
Table 5.1 compares DL versus the prior state-of-the-art (DODGE) versus our preferred

Table 5.2: Quartile charts of F1 scores with various treatments. Results are across 240 data points, i.e., 10 repeats, each over the 24 datasets (combined results of the DODGE data and the Wang et al. [216] data). **Legend: a** = off-shelf deep learning, **b** = oversampling as studied by Buda et al. [35], **c** = SMOTE [40], **d** = fuzzy sampling, **e** = hyper-parameter tuning.

	deep learning a	oversampling b	SMOTE c	fuzzy sampler d	tuning e	**Median** (50th)	**IQR**	•= median and lines shows IQR
#1	✓					0	11	
#2	✓	✓				0	4	
#3	✓	✓	✓	✓		0	15	
#4	✓	✓	✓		✓	0	29	
#5	✓		✓	✓	✓	51	25	
#6	✓	✓		✓	✓	51	25	
#7	✓	✓	✓	✓	✓	51	25	
GHOST = #8	✓	✓	✓	✓✓	✓	80	25	

new method (GHOST). The wins, ties, and losses from that table are summarized on Table 5.3. Note that these wins, tie, losses were computed using the statistical methods of §4.10.

As can be seen in the 40 experiments of Table 5.1, DL only performs better than or the same as DODGE in 10/40 experiments (see all the entries marked with "*"). A repeated pattern is that when DL wins on false alarms, it usually does so at the expense of low recalls. Also, when it does achieve a high recall (e.g. for velocity), this usually comes at the cost of high false alarms. Hence we say:



(a) "Vanilla" network (b) Weighted loss function (c) Weighted loss + fuzzy sampling (d) GHOST

Figure 5.1: An example of how the different components of the GHOST algorithm work. The background color represents the network's predictions, while the points represent the true labels. In each figure is the same dataset artificially generated with some noise (orange is the minority class), and the same feedforward architecture (2 layers, 2 units per layer).

Table 5.3: RQ2 results (our DL method vs the prior state-of-the-art). This table summarizes Table 5.1. The first column indicates the number of wins, ties, and losses for each metric (these are defined using the Δ measure of Section 4.11 and the directions for "better" defined in Section 4.10. Note that for popt20 and pf, there are multiple ties because both DODGE and GHOST achieve the highest possible score.

	AUC	Recall	popt20	pf	Total
win	7	6	3	2	18
tie	1	3	6	1	11
loss	2	1	1	7	11
win + tie	8	9	9	3	29

> For defect prediction, standard deep learning is sub-optimal.

Note that these **RQ1** results merely show that standard deep learning performs badly. Those results do not explain *why* that is so. For this task, we must move on to **RQ2**.

RQ2: *How to fix deep learning for defect prediction?*

The discussion in Chapter 4 showed that (a) widely varying class ratios between train and test sets introduce instabilities into the boundary between classes; and (b) this problem can be alleviated by oversampling around "at risk" members of the minority classes. To check this conjecture, we apply the system designed above. As we applied different stages of GHOST, we found the boundaries of Figure 5.1.

Returning to Table 5.1, GHOST can be seen to have more dark gray cells than anything else (i.e. statistically, it is ranked number one most often). Table 5.3 summarizes those results for comparison with DODGE: 18 times, GHOST defeats DODGE (see top-right, Table 5.3); Tables 5.1 and 5.3 display distribution results where algorithms were run multiple times using different random number seeds. Tables 5.4 and 5.5, on the other hand, display point distribution results where our algorithms are compared to the single set of performance points reported in prior work.

In Table 5.4 and Table 5.5, we show the comparison of GHOST with the results of Wang et al. In these tables, in cases where we used *twoSample* to improve our results, we denote such results with an asterisk. As shown in the summary (Table 5.6), we perform as good or better in 14/16 datasets for within-project defect prediction. In cross-project defect prediction, we also note that we win 14/20 times and tie 2/20 times; this suggests that GHOST may be able to generalize well across different projects.

Table 5.6 summarizes the comparison with the results of Wang et al. [216].

Table 5.4: RQ2 results. Comparison of GHOST (optimized for recall) with the results of Wang et al. [216], using deep learning-generated "Semantic" features to train a Naive Bayes model on within-project defect prediction. Bold indicates better results, as determined by the point statistics of §4.10.2 (and here, $0.3\sigma = 4.91$).

Dataset	Train	Test	Wang et al.			GHOST		
			P	R	F1	P	R	F1
synapse	1	1.1	46	66.7	54.4	74.5	98.5	**84.6***
	1.1	1.2	57.3	59.3	58.3	66.4	100	**79.8***
jEdit	3.2	4	46.7	74.7	**57.4**	42.3	100	**55.9**
	4	4.1	54.4	70.9	61.5	74.7	100	**85.5***
log4j	1	1.1	67.5	73	**70.1**	55.6	100	**65.9**
ivy	1.4	2	21.7	90	35	88.6	100	**83.4***
lucene	2	2.2	75.9	56.9	65.1	61.3	100	**74.5**
	2.2	2.4	66.5	92.1	**77.3**	60.9	100	**75.3**
camel	1.2	1.4	96	66.4	78.5	83.4	100	**90.9***
	1.4	1.6	26.3	64.9	**37.4**	26.7	100	**38.2**
xalan	2.4	2.5	65	54.8	59.5	62.7	100	**66**
xerces	1.2	1.3	40.3	42	41.1	84.8	100	**91.8***
poi	1.5	2.5	76.1	55.2	64	72.2	100	**83.2**
	2.5	3	81.6	79	**80.3**	70.2	100	**79.7**
ant	1.5	1.6	88	95.1	**91.4**	81.3	93.8	87.1*
	1.6	1.7	98.8	90.1	**94.2**	52.6	99.4	57.3

These results recommend GHOST over DODGE and DBN. Also, they deprecate the use of off-the-shelf standard DL for defect prediction (since GHOST clearly is preferred to standard DL). We attribute the super performance of GHOST to its weighted loss function.

The exceptions to the above pattern are the GHOST vs DODGE recall results, which we will discuss in the next RQ. Apart from that, we say that:

> For most evaluation goals, our modified version of deep learning (GHOST) performs better than the prior state-of-the-art.

We also tested whether all the components of GHOST2 were necessary by performing an ablation study, shown in Table 5.2. This table clearly demonstrates that every component, including the use of fuzzy sampling twice, is beneficial and necessary for GHOST2's performance.

RQ3: *Are our methods scalable?*

Any method that tries tuning deep learning (as we do) adds a significant computa-

Table 5.5: RQ2 results. Summary of the Table 5.4 GHOST vs Wang et al. results. Comparison of GHOST with the results of [216] on cross-project defect prediction. All results shown are F-1 scores. GHOST results are medians over 20 runs. Bold and gray indicates better (in case of tie, both are in bold), as determined by point statistics of §4.10.2 (and here, $0.3\sigma = 4.41$).

Source	Target	Wang et al.	GHOST
camel-1.4	jedit-4.1	**61.5**	47.3
lucene-2.2	log4j-1.1	**61.8**	51.7
synapse-1.2	ivy-2.0	**82.4**	30.3
camel-1.4	ant-1.6	**97.9**	84.9*
xalan-2.5	xerces-1.3	**38.6**	**35.7**
jedit-4.1	log4j-1.1	**64.5**	**64.1**
log4j-1.1	jedit-4.1	50.3	**85.5***
xerces-1.3	ivy-2.0	45.3	**94.0***
jedit-4.1	camel-1.4	69.3	**90.9***
ivy-2.0	xerces-1.3	42.6	**91.8***
lucene-2.2	xalan-2.5	55	**65.4**
xerces-1.3	xalan-2.5	57.2	**66**
xalan-2.5	lucene-2.2	59.4	**73.8**
log4j-1.1	lucene-2.2	69.2	**74.6**
ivy-2.0	synapse-1.2	43.3	**60.9**
poi-3.0	synapse-1.2	51.4	**58.1**
synapse-1.2	poi-3.0	66.1	**82.7**
ant-1.6	camel-1.4	31.6	**36**
ant-1.6	poi-3.0	61.9	**77.8**
poi-3.0	ant-1.6	47.8	**62.5**

tional burden to an algorithm that is already very slow. Hence it is appropriate to check the scalability of our methods.

To address this issue, we report the median training time over 20 runs. The measured time is CPU time, when trained on a 4-core Intel Core i5 CPU. These are summarized in Table 5.7. Clearly, our models are very fast (less than 2 seconds to train on a CPU).

Table 5.7 also shows the runtimes for running GHOST on different datasets. Because GHOST runs 20 times to find the median value of a metric, the runtimes we report are for all 20 runs. However, we do not divide this time by 20 as we feel it is scientifically

Table 5.6: RQ2 results. Statistical of the Table 5.4 and Table 5.5 GHOST vs Wang et al. results [216]. Generated using the methods of §4.10.

	WPDP (Tbl. 5.4)	CPDP (Tbl. 5.5)	Total
win	9	14	**23**
tie	5	2	**7**
loss	2	4	**6**
win + tie	14	16	30

Table 5.7: Median training time over 20 repeats.

	Training (secs)	GHOST (secs)
ivy	0.78	9m 17s
lucene	0.81	10m 26s
poi	1.09	13m 44s
synapse	0.81	9m 48s
velocity	0.88	10m 20s
camel	1.41	18m 10s
jEdit	1.11	14m 32s
log4j	0.73	8m 29s
xalan	1.61	20m 16s
xerces	1.08	11m 56s

important to run a stochastic experiment multiple times and report statistical results.

Figure 5.2 shows our scalability results, which we obtain by measuring the training time for different sizes of the datasets. We observe a general trend across all datasets that deep learning scales well with the size of the problem. More specifically, GHOST's runtimes grow sublinearly. For example, in the xalan results, a 400% increase in data (from a fifth to all the data) leads to a runtime increase of only $2.7/1.5 = 180\%$.

Hence we say:

> Tuning deep learners is both practical and tractable for defect prediction.

5.2 Issue lifetime prediction

5.2.1 Research Questions

For issue lifetime prediction, we are interested in the following research question:
RQ1: For Bugzilla issue close time prediction, can simpler models suffice?
RQ2: For Bugzilla issue close time prediction, do classical ML models suffice?

5.2.2 Results

RQ1: For Bugzilla issue close time prediction, can simpler models suffice?

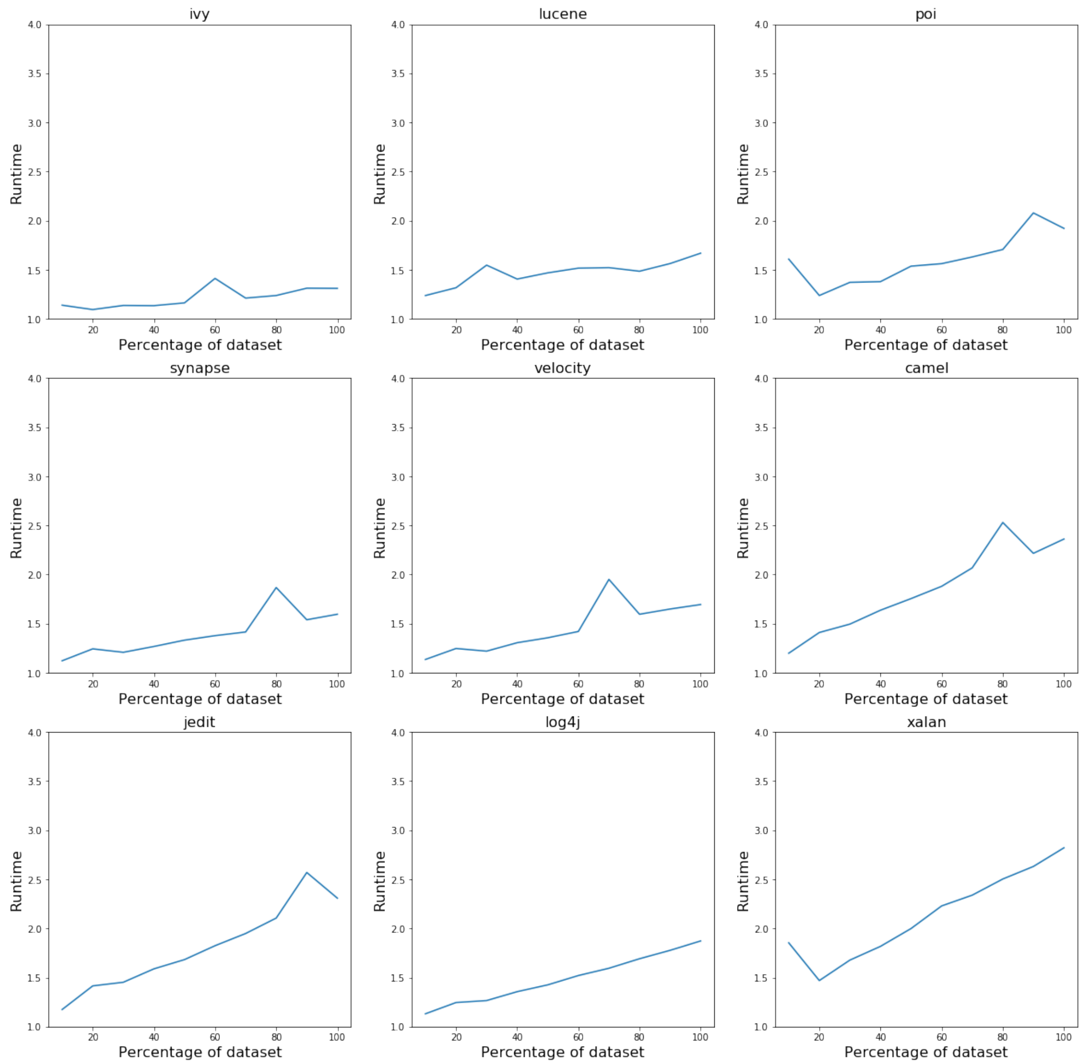


Figure 5.2: RQ3 results: scalability results for 9 datasets. The x-axis is the perentage of the dataset trained on; the y-axis is the runtime in seconds on a CPU.

In Table 5.8, best results are indicated by the **gray cells**. The columns of that table describe how detailed are our time predictions. A column labeled k-class means that the data was discretized into k distinct labels, as done in prior work (see Lee et al. [121] for details).

Recall that cells are in gray if the are statistically significantly better. In all cases, SIMPLE's results were (at least) as good as anything else. Further, once we start exploring more detailed time divisions (in the 3-class, 5-class, etc problems) then SIMPLE is the stand-out best algorithm.

Table 5.8: Results on BugZilla data used in prior deep learning state of the art. The target label is discretized into a different number of classes (columns) as in the prior work. **Dark cells** indicate statistically better performance.
Key: DT = DeepTriage [137]; **NDL-T** = best result of untuned non-neural methods; i.e. best of logistic regression [82] and random forests [139]; **NDL+T** = best of DODGE-tuned non-neural methods; i.e. **NDL-T** plus tuning; **FF** = untuned feedforward network; i.e Algorithm 1, without tuning; **SIMPLE** = SIMPLE i.e. **FF** plus tuning; T_k = Top-k accuracy;

Project	Model	2-class	3-class		5-class		7-class		9-class	
		T_1	T_1	T_2	T_1	T_2	T_1	T_2	T_1	T_2
Firefox	DT	67	44	78	31	58	21	39	19	35
	NDL-T	**70**	43	64	30	42	18	30	18	30
	NDL+T	68	47	79	34	61	25	45	21	39
	FF	**71**	49	82	37	63	26	47	23	41
	SIMPLE	**70**	**53**	**86**	**39**	**67**	**37**	**61**	**25**	**45**
Chromium	DT	63	43	75	27	52	22	38	18	33
	NDL-T	64	35	56	23	36	15	27	15	28
	NDL+T	64	49	79	30	56	26	42	23	40
	FF	65	53	82	35	60	27	45	26	42
	SIMPLE	**68**	**55**	**83**	**36**	**61**	**29**	**48**	**28**	**45**
Eclipse	DT	61	44	73	27	51	20	37	19	34
	NDL-T	66	33	54	23	38	16	29	16	29
	NDL+T	65	52	81	30	56	27	44	27	42
	FF	66	54	81	32	59	**30**	**47**	30	46
	SIMPLE	**69**	**56**	**84**	**35**	**62**	**31**	**48**	**33**	**49**

Another thing we can say about these results is that SIMPLE is much faster than other approaches. The above results took ≈ 90 hours to generate, of which 9 hours was required for SIMPLE (for 20 runs, over all 15 datasets) and 80 hours were required for the deep learner (for 1 run, over all 15 datasets). Recall that if we had also attempted to tune the deep learner, then that runtime would have exploded to six years of CPU.

Therefore, we conclude:

> Simpler models suffice for Bugzilla issue close time prediction.

RQ2: For Bugzilla issue close time prediction, do classical ML models suffice?

Table 5.8 also shows that with or without hyper-parameter optimization, non-neural methods do not perform well. This is because feedforward networks can learn complex representations of the data, and approximate the decision function much better.

> Classical ML models do not suffice for Bugzilla issue close time prediction.

5.3 Code smell detection

5.3.1 Research Questions

We are interested in answering the following questions in the context of code smell detection:

RQ1: Can fuzzy oversampling achieve state-of-the-art results in code smell detection?

This question assess the generalizability of the fuzzy oversampling method proposed for defect prediction. This would have implications for deep learning practitioners going forward, who can use this method in their ML pipelines.

RQ2: Why do feedforward networks work so well for code smell detection?

This is a more general question that assesses why fuzzy sampling works so well across multiple software analytics domains.

5.3.2 Results

RQ1: Can fuzzy oversampling achieve state-of-the-art results in code smell detection?

The results of GHOST on code smell detection are shown in Table 5.9, which are summarized in Table 5.10. For feature envy, GHOST is better *all* the time, reaching near-perfect scores. Because the scores were so high, we took the extra step to ensure that there was no data leakage (i.e., none of the samples from the training set leaked into the test set). To check this, we computed the distance matrix (using the Euclidean distance) between the train and test sets, and checked for values of 0 (which would happen if the points were the same). The total number of values in the distance matrix ranged from $\sim$ 800 million to $\sim$ 1.3 billion (after oversampling, which increases the size of the training set), and of these, between 2,000 to 4,000 were 0. This means that of $\sim \sqrt{10^9} =\sim 30,000$ samples, $\sim \sqrt{10^3} =\sim 30$ were the same. The exact percentages of values that were 0 are shown in Table 5.11. Note that these values are *extremely small*, e.g. for abd-extractor, $33.2 \times 10^{-5}\% = 0.0003\%$ of values were zero. We argue that this had no effect on the performance of the learner on the test set.

On long method detection, we generally lose (11 times), although most of the time, we tie (24 times). We argue that this is because Liu et al. [131] use feedforward networks

Table 5.9: Code smell detection results: performance seen with different code smells. Gray cells indicate better results (comparison is between the second and third sets of columns), using a Scott-Knott test over 20 repeats. In this table "SOTA" = the prior state of the art (and comes from Liu et al. [131]).

(a) Feature envy

Dataset	SOTA (original)				SOTA (reproduced)				Fuzzy sampling (with GHOST)			
	Precision	Recall	F1	AUC	Precision	Recall	F1	AUC	Precision	Recall	F1	AUC
abd-extractor	29.7	76.7	42.8	74.8	79.5	80	79.7	88	**98**	**98**	**98**	**99.3**
aoi30	36.7	97	53.2	93.6	79.2	79.2	79.2	87.3	**97.7**	**97.7**	**97.7**	**99.1**
areca	50	88	63.8	91.9	80.1	79.9	80	88.3	**98**	**98**	**98**	**99.3**
freeplane	36.2	94.1	52.3	83.1	80.6	80.4	80.5	88.8	**98.2**	**98.2**	**98.2**	**99.3**
grinder	31.2	88.6	46.1	85.2	80	80.1	80	88.1	**97.7**	**97.7**	**97.7**	**99.1**
jedit	38.2	91.3	53.9	84.3	78.7	78.5	78.6	87	**97.8**	**97.8**	**97.8**	**99.2**
jexcelapi	34	88.9	49.2	90	78.5	78.2	78.3	86.3	**97.7**	**97.7**	**97.7**	**99.2**
junit	50	82.2	62.1	85.7	80.3	80.2	80.2	88	**97.7**	**97.7**	**97.7**	**99.1**
pmd	37.4	86	52.1	84.9	79.8	79.6	79.7	87.8	**97.8**	**97.8**	**97.8**	**99.2**
weka	38.2	87	53.1	78.7	77.9	77.5	77.7	85.8	**97.8**	**97.8**	**97.8**	**99.1**

(b) Long Method

Dataset	SOTA (original)				SOTA (reproduced)				Fuzzy sampling (with GHOST)			
	Precision	Recall	F1	AUC	Precision	Recall	F1	AUC	Precision	Recall	F1	AUC
areca	42.7	73.8	54.1	78.5	41.7	**77.8**	**52.5**	**72.2**	41.7	70	51.5	71.2
freeplane	46.4	75.6	57.5	78.8	44.6	78.2	55.6	70.3	**45.8**	76.7	**57**	**71.1**
jedit	52.2	83.5	64.2	77.2	53.4	82.5	**63.4**	**71.3**	**54.1**	80.5	62.2	70.4
junit	58.5	52.9	55.6	72.6	51.7	67.1	56	66.1	51.3	64.9	54.1	61.6
pmd	37.1	70.6	48.6	77.4	35.2	84.8	47.6	73.6	35.4	77.8	47.1	72.9
weka	50.2	79.3	61.5	81.8	**49.9**	83.9	**62.3**	**76.7**	47.7	**86.3**	61.2	76.1
abd-extractor	32.3	80.6	46.2	78.8	39.1	73.5	50.7	73.4	38.9	73.6	50.5	73.2
grinder	37.2	71.8	49	74.1	36.1	66.3	47.5	65.9	38	73.9	46.2	64.8
aoi30	37.6	87.6	52.6	80.3	38.4	81.2	51.8	72.6	38.2	81.5	52.2	72.7
jexcelapi	32.6	83.6	46.9	88.5	38.5	**81.5**	**52.1**	**80.3**	37.3	81.1	50.9	79.8

(c) Large Class

Dataset	SOTA (original)				SOTA (reproduced)				Fuzzy sampling (with GHOST)			
	Precision	Recall	F1	AUC	Precision	Recall	F1	AUC	Precision	Recall	F1	AUC
areca	11.4	80	20	68.8	33.7	78.3	47.1	50	**82.9**	77.5	**80.1**	**89.5**
freeplane	12	70	20.4	72.8	6.1	**100**	**11.5**	50	0	0	0	**68.8**
jedit	15	75	25	77.4	61.5	**100**	76.2	50	**76.8**	81.8	79.2	**75**
junit	11.8	40	18.2	71.8	11.9	100	21.3	50	**32.2**	71.4	**44.4**	74.6
pmd	16.7	100	28.6	83.5	11.4	**100**	20.5	50	**23.8**	23.6	23.7	**63.1**
weka	10	94.4	18.2	68.6	0	0	0	50	**42.9**	**71.4**	**53.6**	**70.7**
abd-extractor	16.5	79.4	27.3	79.8	9.1	14.3	11.1	50	**27.9**	28.6	**28.2**	**65.3**
grinder	12.7	70	21.5	79.2	0	0	0	50	0	0	0	50
aoi30	12.3	81.8	21.4	78.5	30	100	46.2	50	**53.8**	80	**64.3**	**75.7**
jexcelapi	22	84.6	34.9	80.9	15.7	**100**	27.1	50	57.7	73.5	64.6	80.4

(d) Misplaced Class

Dataset	SOTA (claimed)				SOTA (reproduced)				GHOST			
	Precision	Recall	F1	AUC	Precision	Recall	F1	AUC	Precision	Recall	F1	AUC
areca	93.6	92.6	93.1	99.8	34.1	100	50.9	98.6	**44.7**	100	**61.8**	**100**
freeplane	94.9	91.5	93.2	99.8	32.7	100	49.3	96.3	**43.3**	99.4	**60.3**	**99.4**
jedit	55	100	71	99.2	32.6	100	49.2	98.7	**36**	100	**53**	**100**
junit	41.4	100	58.5	99.5	35.2	96.1	51.6	93.4	**36.3**	**96.3**	52.7	**97.3**
pmd	56.1	97.9	71.3	99.6	41.2	96.8	57.7	92	**53.8**	**99.3**	**69.8**	**99.6**
weka	96.8	94.2	95.5	99.8	48.5	99.8	65.3	99.7	48.4	**99.9**	65.2	**99.8**
abd-extractor	95.1	92.3	93.7	99.7	**44.5**	99.7	**61.6**	99.5	43.8	**100**	60.9	**99.8**
grinder	92.5	82.2	87.1	98.4	18.2	90.6	30.4	85.6	**24.5**	**100**	**39.4**	**100**
aoi30	76.4	100	86.6	99.4	43.6	100	60.7	92.2	44	100	61.8	**100**
jexcelapi	39.6	90.5	55.1	92.5	21.6	86.4	34.6	74.3	**32.6**	**96.1**	**48.6**	**96.3**

(their code uses the `MLPClassifier` class from `sklearn`) for their long method detection

Table 5.10: Summary of Table 5.9 for each code smell.

	Precision	Recall	F1	AUC	Total
Feature Envy					
win	10	10	10	10	40
tie	0	0	0	0	0
loss	0	0	0	0	0
Long Method					
win	2	1	1	1	5
tie	7	7	5	5	24
loss	1	2	4	4	11
Large Class					
win	8	1	6	7	22
tie	2	5	3	3	13
loss	0	4	1	0	5
Misplaced Class					
win	7	6	6	10	29
tie	2	4	3	0	9
loss	1	0	1	0	2

as well, and therefore there is not much performance gain to be expected.

Our wins continue, however, on large class and misplaced class detection. On the former, we win 22/40 = 55% of the time, and tie 13/40 = 32.5% of the time. Our wins are greater still in the latter case (misplaced class), where we win 29/40 = 72.5% of the time and tie 9/40 = 22.5% of the time.

The above results strongly favor GHOST over prior work. Note that these results are from 20 repeats, with a statistical test to determine "better". Therefore, we have sufficient evidence to say:

> GHOST also achieves state-of-the-art performance on code smell detection.

RQ2: Why do feedforward networks work so well?

There are several theories on the generalizing capabilities of feedforward networks in general:

(a) Hornik et al. [98] show, using prior work [49], that feedforward neural networks

Table 5.11: Percentage of values in the distance matrix (for feature envy) that were 0. Note that the percentages are extremely small (e.g. for abd-extractor, 0.0003% of values were 0).

Dataset	**% of 0s** $\times 10^5$
abd-extractor	33.2
aoi30	47.3
areca	36
freeplane	38.2
grinder	33.4
jedit	16.3
jexcelapi	40.9
junit	32.8
pmd	31.9
weka	58

with as few as one hidden layer, with a sufficient number of units, are universal approximators.

(b) More recently, Jacot et al. [101] showed that a feedforward learner with infinite width (the number of units in the layer)[1] is a linear model under a kernel they called the "neural tangent kernel", and that it can approximate any arbitrary function.

(c) Montufar et al. [151] showed that the decision boundary of neural networks are piecewise linear, and that there is a derivable lower and upper bound on the number of linear pieces constituting the boundary for a given network. This idea was exploited by Yedida & Menzies [234] to design the structure of their networks.

(d) Galke & Scherp [74] show that several tasks for which complex deep learners have been applied recently can also be done with feedforward networks with no loss of performance.

(e) Yedida & Menzies [234] showed that using novel preprocessing methods, one can push the decision boundary away from points, making the classifier more robust to noise.

From the above works, we derive the following important lessons:

[1] In practice, this translates to "sufficient width".

(a) The deep learning literature broadly agrees that feedforward networks can approximate any arbitrary function [49, 98, 151].

(b) The decision boundary is known to be piecewise-linear, but also "malleable" (i.e., its shape can be changed using appropriate preprocessing). It has been shown [98] that the composition of the nonlinearities in feedforward networks allows for arbitrary decision boundary shapes.

Based on these examples we were motivated to see if there is a general pattern or test that lets us recommend using (or avoiding) GHOST. While any such test can only be a heuristic (since it is difficult to predict the performance of a deep learner in general), we offer one such heuristic that seems suitable for future research.

We start by designing a simple autoencoder as follows:

- Let 2^k be the highest power of 2 that is lesser than the input vector length.
- Fix the bottleneck layer with engineering judgement (for SE tasks, which are known to be simpler than general AI tasks [2], we found 32 or 64 to be useful; for general AI tasks, we use 128[2]).
- Then, in the encoder, set the number of units to be $k, k/2, k/4, \ldots$ until the bottleneck layer.
- Design the decoder as the mirrored version of the encoder.

(For clarity, we will give an example of this towards the end of this section.)

As is usual with the standard autoencoder, train this neural network on the input data with the mean squared error (MSE) loss. Our heuristic is then:

> If the MSE loss of the autoencoder designed as discussed above is below 1,000, attempt a feedforward network before trying more complex methods.

It is worth noting that autoencoders, like any deep learner, can get stuck in local optima because of the non-convex nature of loss functions [44]. It is worth attempting this three times to ensure that a loss higher than 1,000 is because it is not possible, rather than poor optimization.

[2]Using 64 did not change the results we show below.

We tested this heuristic on the original defect prediction datasets used by Yedida & Menzies [234], and it was true for *all* of them. Furthermore, this heuristic was also true for the code smell detection datasets of this paper. However, it does not suffice to say the heuristic worked for our datasets; we need to show cases where (a) it is not true, and (b) the simpler network failed to make a valid heuristic. To do this, we pulled 3 common image classification datasets: MNIST (digit classification), CIFAR-10 (10-class image classification), and CIFAR-100 (100-class image classification).

For example, consider the MNIST dataset. This consists of black-and-white images of digits (0-9), sized 28 x 28. We flatten these images to one vector of length 784. Then, the highest power of 2 lower than this is 512, and we choose the bottleneck layer to be 128 units; therefore, our autoencoder architecture is 784 (input) - 512 - 256 - 128 (bottleneck) - 256 - 512 - 784.

To our surprise, for the MNIST dataset, the heuristic was true; on applying a feedforward network (without hyper-parameter optimization) to it, we achieved a classification accuracy of 96.3%–not state-of-the-art, but respectable, and would be improved with the hyper-parameter optimization of GHOST.

For both the CIFAR-10 and CIFAR-100 datasets, the heuristic was false; indeed, the feedforward network fell significantly short of convolutional neural networks that are typically applied to these datasets: on CIFAR-10, we achieved 44% accuracy (modern networks can achieve 90+% accuracy), and on CIFAR-100, we achieved merely 18.3% (while a modern network would achieve 75+% accuracy). Therefore, we have (a) case studies where the heuristic is false and our simpler networks failed (as expected by the result of the heuristic), and also (b) case studies where the heuristic is true, and the feedforward network was sufficient. Our experiments on the defect prediction datasets, code smell detection datasets, and image classification datasets show that this heuristic is reliable.

These experiments seem to reinforce the claim made by Agrawal et al. [2] who said that software analytics may be simpler than standard AI datasets. If a standard autoencoder cannot compress the input data into the 64 dimensions of its bottleneck layer, it may mean that those image datasets (where it failed) are intrinsically much higher dimensional. However, our heuristic's success on both the code smell detection and defect prediction data suggest that it should be applied as a test for whether feedforward networks should be used instead of more complex approaches.

5.4 Static code analysis

5.4.1 Research Questions

We explore our results in the context of the following research questions:

RQ1: *Is recognizing static code warnings (still) intrinsically easy?*

Yang et al. [232] suggested that the low intrinsic dimensionality of the datasets used in their study meant that recognizing actionable static code warnings was an intrinsically easy task. This led to them using SVMs for their work. Given that the dataset was disputed and corrected by Kang et al. [110], we test if that assumption still holds.

RQ2: *Can deep learning help recognize actionable static code warnings?*

The work of Kang et al. [110] showed that the SVM methods of Yang et al. [232] were no longer sufficient for the task. We show that this is indeed true, as part of an ablation study. We further show that in fact, classical learners are insufficient for the task, even with hyper-parameter optimization [1, 145] and SMOTE (see treatment C1 in our ablation study, Table 5.12). Therefore, we turn to deep learning, but begin by exploring the simplest deep learning model, the feedforward network. Motivated by a recent success in defect prediction [234], we try the GHOST algorithm, which combines feedforward networks with hyper-parameter optimization and a novel fuzzy sampling approach. However, this falls short of our expectations (see treatment A5 from our ablation study). Moreover, the use of more complex deep learners such as CNNs (see treatments B2 and C2), and CodeBERT (see the last line of Table 5.12) were also insufficient. These disappointments lead to our next research question:

RQ3: *How can we recognize actionable static code warnings?*

We recognize that the fuzzy sampling technique combined with hyper-parameter optimization yielded excellent results in the original GHOST study. Therefore, this approach is employed with classical learners. Here, we obtain excellent results (median AUC=1.0, see treatment A6). However, the data used in this study, which was provided by Kang et al. [110], was manually labeled, which is an expensive task. We therefore ask if this labeling is inherently necessary.

RQ4: *Can we recognize actionable static code warnings with fewer labels?*

In this RQ, we explore the problem under the semi-supervised setting. Consequently, we develop a novel yet trivial semi-supervised algorithm that provides labels to samples that do not have any. In doing so, we can still achieve excellent results (median AUC=0.88), while using only 9.5% labels. Moreover, looking closely at this result (see

A1 in our ablation study) and A6, we see that the IoU of the median performance (intersection over union, an extension of Jaccard similarity) is 0.67. More interestingly, however, we observe a small data result, where our recommended approach (A1, which we call GHOST2) performs well on datasets with few samples, and worse on larger datasets. Therefore, we ask:

RQ5: *Are neural networks practical on small datasets in SE?*

While we can only provide an answer about our specific problem, this opens up a larger discussion about truisms of deep learning that do not hold up in software engineering. Specifically, our results challenge the notion that deep learning requires large amounts of data to work well; instead, we say that for simpler problems like in SE, neural networks can achieve excellent results using less data. Finally, we ask:

RQ6: *What lessons can practitioners learn from our results?*

We notice that our work provides yet another case study where using AI tools off-the-shelf does not work. Specifically, we characterize a learning problem as a set of engineering decisions: boundary engineering, label engineering, learner choice, hyper-parameter engineering, and instance engineering. We recommend that practitioners rethink learning problems as these different challenges and use a combination of tools that works for their problem from these different areas.

RQ7: *How does fuzzy sampling help optimization?*

We seek to find why fuzzy sampling is so important to our approach. To do so, we show that using fuzzy sampling improves the $\beta-$smoothness of the loss surface, making it easier to optimize.

5.4.2 Results

Our results are shown in Figure 5.3. A1 is our recommended approach, and uses 9.5% of labels. Looking at the graphs, it is clear that A1 is the strongest contender. For example, against B1, A1 wins (or ties) in precision in all cases, and wins (or ties) in AUC in 7/8 datasets. In false alarm rate, A1 wins/ties 6/8 times, and in recall, it wins/ties 7/8 times. Meanwhile, the method recommended by the original study (SVMs with balanced class weights and normalizing), which is shown as D1, performs poorly across the board.

More interesting than the comparison between the treatments, however, is a trend that can be seen with respect to dataset size. In Figure 5.3, the datasets are sorted by size. In several cases, A1 loses to other treatments on derby and tomcat, the two largest datasets, while winning across the other datasets. For example, against A6, in

Table 5.12: Design of our ablation study. In the learner choice column, **F** = feedforward networks, **T** = traditional learners, **C** = CNN, **B** = CodeBERT.

Treatment	Boundary eng.	Label eng.	Learner choice	Parameter eng.	Instance eng.	% 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 [234]
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. [232] and Kang et al. [110] studies.
CodeBERT			B			100	CodeBERT without modifications

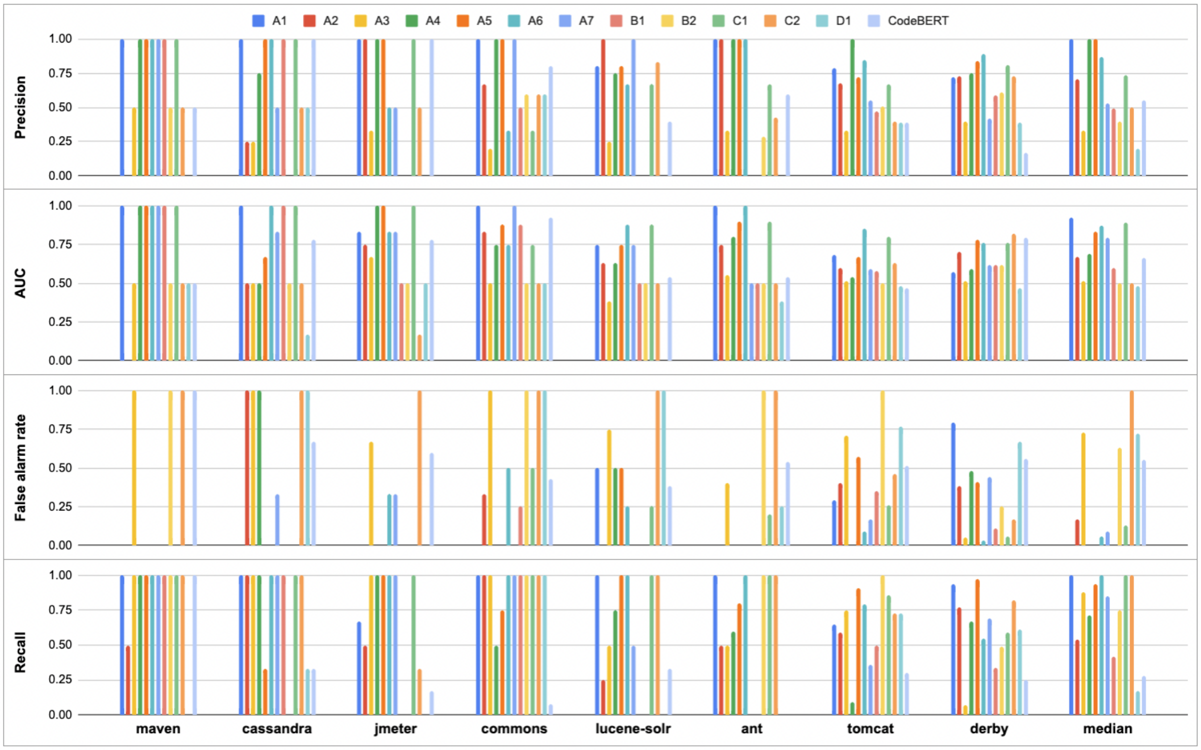


Figure 5.3: Results of our ablation study

both AUC and precision, A1 wins in all datasets except in derby and tomcat (on AUC, it also loses on lucene-solr). Similarly, on precision, A1 matches or outperforms A5 in all datasets except derby. This is perhaps a "small data" result where feedforward networks, i.e., neural approaches, outperform other approaches on small datasets, even though conventional wisdom is that more data is better for neural networks. Going deeper into A1 vs. A5, Table 5.12 shows that the difference between the two is that A5 does not use the semi-supervised learning. This could be the reason it does better on large data. In detail, when there is less data available, a neural network is susceptible to change its boundary more drastically with each training sample. When more samples are available, however, it must accommodate all of them, making changes to the decision boundary more subtle. Therefore, on small datasets, our semi-supervised learning technique has a regularization effect where by not trusting the labels given and instead using the nearest neighbors, we reduce the shift in the decision boundary. On larger datasets, however, this semi-supervised technique, where we modify the labels *a priori*, has the negative effect of corrupting the data and confusing the learner. Our lesson from this is that for datasets with less than 30 training samples (which is all the datasets till and including ant), semi-supervised learning is advisable.

At this point, we digress to make a clarification. Although we explain the superiority of A5 in the larger datasets as a consequence of the effect of semi-supervised learning on the boundary, that does *not* make it a boundary engineering technique; rather, the change in labels is the cause of the boundary shifts. Our definition of boundary engineering covers changes to the *independent variables* that affect the boundary, such as fuzzy sampling and kernel methods.

We now discuss our results in the context of seven research questions:

RQ1: *Is recognizing static code warnings (still) intrinsically easy?*

The intrinsic dimensionality of each project after applying a box-counting dimension reduction algorithm are summarized in in Table 3.4, and we sort the project in a ascending order of their intrinsic dimension. The results illustrate our datasets are still intrinsically easy.

> Static code analysis datasets are intrinsically easy.

RQ2: *Can deep learning help recognize static code warnings?*

Our results with A5 (GHOST), B2 and C2 (CNNs), and CodeBERT show that deep learners cannot be used as-is for detecting actionable static code warnings. Indeed, their performance leaves more to be desired. Our conclusion is:

> Deep learning cannot be used as-is for recognizing static code warnings.

RQ3: *How can we recognize static code warnings?*

We showed that using a combination of engineering techniques, leading up to one of our recommendations (A5) worked well to recognize actionable static code warnings. While it uses all the labels, it is fast and computationally cheap.

> A combination of engineering techniques such as boundary engineering, learner choice, and hyper-parameter engineering can work together to achieve strong results.

RQ4: *Can we recognize static code warnings with fewer labels?*

Our label engineering method allows us to achieve strong results using just 9.5% of labels on average. It is certainly possible that better semi-supervised learning techniques might produce better results; however, we do not explore that in this paper.

> Semi-supervised learning can help achieve strong results using only 9.5% of labels.

RQ5: *Are neural networks practical on small datasets in SE?*

Our results with A1 (GHOST2) show that neural approaches are certainly practical and useful for achieving high performance on small datasets, even as low as 4 samples (in the case of maven). Therefore, we say:

> At least in SE, neural networks are practical on small data.

We also comment here that as shown by Zhang et al. [248], neural networks are often *over-parameterized*, i.e. they have more parameters than data samples. This is not a problem: Du et al. [59] show that stochastic gradient descent can optimize such networks. In fact, over-parameterization is typically seen as *beneficial*, see [7, 38, 111, 161, 195]. We believe this is one of the reasons that our feedforward networks worked for small datasets.

RQ6: *How does fuzzy sampling help optimization?*

To understand this, we took inspiration from a seminal paper that explained how batch normalization helps optimization [181]. In that paper, they showed that batch normalization improved the $\beta-$smoothness of the loss surface, making optimization easier. We use a similar approach. First, using the technique by Li et al. [124], we showed that the loss surface is indeed, smoother. Next, we formalized the smoothness of the loss surface. We extended the work of Yedida et al. [240], who derived the Lipschitz constants of loss functions. Recalling that the $\beta-$smoothness is the maximum gradient of the Lipschitz constant, we used their equations of the Lipschitz constant to derive the $\beta-$smoothness. Finally, we implemented it and show that the smoothness is improved by a median of 33.85%.

> Fuzzy sampling helps smoothen the loss surface, making optimization easier.

RQ7: *What lessons can practitioners learn from our results?*

We offer several lessons from our work:

> 1. Mistrust your data more. Labels may not always be accurate, and data can have errors.

2. Attempt to use fewer labels when possible to lower manual effort required to label data.
3. Our empirical methods may need to be revised for small data.
4. Do not use tools off-the-shelf; use hyper-parameter optimization and consider the engineering decisions of the problem carefully.
5. We learn from failures as well as successes. Work with your critics.

5.5 Automated microservice partitioning

5.5.1 Research Questions

We aim to answer the following research questions:
RQ1: How prevalent is hyper-parameter brittleness in automated microservice partitioning?

We verify Yedida et al. [239]'s results, who showed that hyper-parameter optimizers are "brittle", i.e., they work well for a few metrics and datasets, but are not useful across multiple goals and data sets. The verification is crucial to set the motivation for our contribution.
RQ2: Is hyper-parameter optimization enough to curb the aforementioned optimizer brittleness?

Here we will show that standard hyper-parameter optimization methods are insufficient for solving brittleness.
RQ3: How else might we fix the aforementioned brittleness problem?
RQ4: Does `DEEPLY` generate "dust" (where all functionality is loaded into one partition) or "boulders" (where all partitions contain only one class each)?

5.5.2 Results

We ran the approaches from Table 3.7, 30 times each, then compared results from those treatments with the statistical methods of Section 4.10. The median results are shown in Table 5.13a-5.13b and the statistical analysis is shown in Table 5.13c-5.13d.

In those results, we see that across different datasets and metrics, those ranks vary widely, with little consistency across the space of datasets and metrics. For example:

Table 5.13: Results on all datasets. The top row shows our median performance scores over 30 runs, while the bottom row shows the Scott-Knott ranks (lower is better). Gray cells indicate the best result according to the Scott-Knott test. All algorithms were tuned by hyperopt.

(a) Median performance scores over 30 runs for jpetstore and daytrader

Algorithm	jpetstore			daytrader		
	BCS [-]	**ICP** [-]	**SM** [+]	**BCS** [-]	**ICP** [-]	**SM** [+]
Bunch	2.43	0.48	0.21	1.86	0.57	**0.27**
Mono2Micro	**1.67**	0.53	0.05	**1.58**	0.4	0.07
CO-GCN	2.79	0.27	0.2	2.09	0.05	0.13
FoSCI	1.99	0.66	0.05	2.04	0.48	0.07
MEM	2.74	0.48	0.22	2	0.59	0.09
DEEPLY	2.35	**0.06**	0.2	1.74	**0.03**	0.17

(b) Median performance scores over 30 runs for acmeair and plants

Algorithm	acmeair			plants		
	BCS [-]	**ICP** [-]	**SM** [+]	**BCS** [-]	**ICP** [-]	**SM** [+]
Bunch	1.67	0.55	**0.17**	2.29	0.56	0.22
Mono2Micro	**1.58**	0.4	0.06	2.49	0.39	0.07
CO-GCN	1.89	0.47	0.15	2.46	0.35	0.24
FoSCI	2.08	0.54	0.07	1.94	0.45	0.07
MEM	2.13	0.69	0.03	2.04	0.42	0.25
DEEPLY	**1.41**	**0.18**	0.16	**1.87**	**0.02**	**0.27**

(c) Scott-Knott ranks for jpetstore and daytrader. For all metrics, *lower* is *better*.

Algorithm	jpetstore			daytrader		
	BCS [-]	**ICP** [-]	**SM** [+]	**BCS** [-]	**ICP** [-]	**SM** [+]
Bunch	4	3	2	3	5	**1**
Mono2Micro	**1**	5	5	**1**	3	6
CO-GCN	6	2	2	6	2	3
FoSCI	2	6	4	5	4	5
MEM	5	4	1	4	6	4
DEEPLY	3	**1**	3	2	**1**	2

(d) Scott-Knott ranks for acmeair and plants. For all metrics, *lower* is *better*.

Algorithm	acmeair			plants		
	BCS [-]	**ICP** [-]	**SM** [+]	**BCS** [-]	**ICP** [-]	**SM** [+]
Bunch	2	4	**1**	4	6	4
Mono2Micro	**1**	2	4	6	3	5
CO-GCN	3	3	3	5	2	3
FoSCI	4	4	5	2	5	6
MEM	5	5	6	3	4	2
DEEPLY	**1**	**1**	2	**1**	**1**	**1**

- For jpetstore, we see that the best algorithms for the three metrics are Mono2Micro, CO-GCN, and MEM respectively;
- For acmeair, the best algorithms are Mono2Micro, and Bunch;
- For plants, the best algorithms are FoSCI, CO-GCN, and MEM;

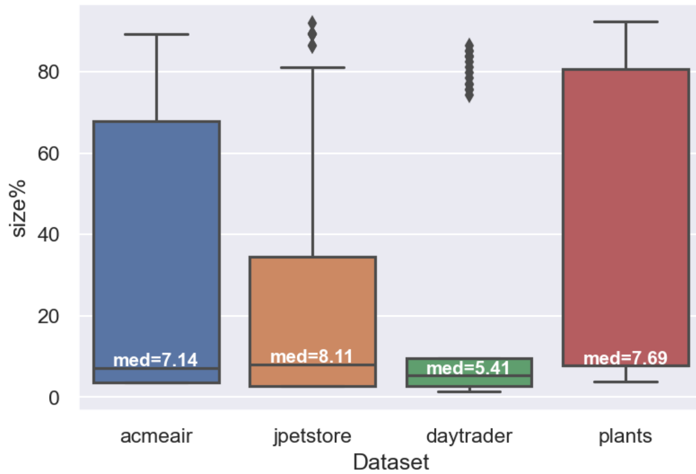


Figure 5.4: Analysis of partitions generated by our approach. Distribution of class size percentages across datasets (median results).

- For daytrader, the best are Mono2Micro, CO-GCN, and Bunch.

Hence we say:

> For all the data sets and goals studied here, there is widespread performance brittleness.

RQ2: Is hyper-parameter optimization enough to curb the aforementioned optimizer brittleness?

We check of brittleness can be solved by tuning the partitioning methods.

Since all the algorithms in our study were tuned by hyperopt, the results of Table 5.13 show that even post-tuning, there is a large brittleness problem. Accordingly, we say:

> Hyper-parameter optimization does not suffice to fix performance brittleness.

RQ3: How else might we fix the aforementioned brittleness problem?

When we added the novel loss function (discussed above in Section 4.7.5), we obtained the results shown in the last line of the two tables of Table 5.13c and Table 5.13d. When measured in terms of ranks seen amongst the different populations, the last lines are most insightful. Here we see that in general, `DEEPLY` performs better compared to all other optimizers across all our case studies.

Group	Clustering	LR	HPO	Loss	jpetstore			acmeair			plants			daytrader		
					BCS [-]	ICP [-]	SM [+]	BCS [-]	ICP [-]	SM [+]	BCS [-]	ICP [-]	SM [+]	BCS [-]	ICP [-]	SM [+]
1	KMeans	Exponential	Yes	Unweighted	3.32	0.26	0.08	1.88	0.31	0.11	3.38	0.12	0.12	2.44	0.03	0.09
	KMeans	1cycle	Yes	Unweighted	3.3	0.25	0.09	1.91	0.31	0.11	3.38	0.12	0.12	2.39	0.03	0.11
	Spectral	Exponential	Yes	Unweighted	3.32	0.23	0.08	1.91	0.31	0.11	3.38	0.12	0.12	2.29	0.03	0.11
	Spectral	1cycle	Yes	Unweighted	3.13	0.24	0.08	1.91	0.31	0.11	3.38	0.12	0.12	2.35	0.03	0.11
2	KMeans	Exponential	No	Unweighted	2.79	0.27	0.2	1.89	0.47	0.15	2.46	0.35	0.24	2.09	0.05	0.13
	KMeans	1cycle	No	Unweighted	2.27	0.17	0.08	1.5	0.32	0.08	3.26	0.38	0.12	2.13	0.03	0.09
	Spectral	Exponential	No	Unweighted	2.98	0.25	0.15	1.65	0.33	0.09	3.29	0.34	0.15	2.29	0.07	0.08
	Spectral	1cycle	No	Unweighted	2.95	0.28	0.12	1.8	0.32	0.1	3.28	0.33	0.12	2.44	0.11	0.13
3	KMeans	Exponential	No	Weighted	2.44	0.17	0.08	1.68	0.34	0.09	3.19	0.32	0.15	2.07	0.03	0.1
	KMeans	1cycle	No	Weighted	2.53	0.17	0.1	1.36	0.26	0.09	3.34	0.26	0.12	2.08	0.05	0.08
	Spectral	Exponential	No	Weighted	2.77	0.25	0.11	1.65	0.39	0.09	3.29	0.32	0.13	2.45	0.12	0.12
	Spectral	1cycle	No	Weighted	2.9	0.22	0.13	1.53	0.31	0.1	3.18	0.35	0.13	2.28	0.08	0.11
4	KMeans	Exponential	Yes	Weighted	**1.76**	**0.03**	0.18	**1**	**0.05**	0.13	**1.87**	0.05	**0.27**	1.67	**0.01**	0.14
	KMeans	1cycle	Yes	Weighted	**1.76**	**0.03**	0.19	**1**	**0.05**	**0.16**	**1.87**	0.09	**0.27**	**1.58**	**0.01**	0.14
	Spectral	Exponential	Yes	Weighted	2.29	**0.03**	**0.26**	1.36	0.11	**0.16**	**1.87**	0.04	**0.27**	1.64	0.03	**0.19**
	Spectral	1cycle	Yes	Weighted	2.35	0.06	0.2	1.41	0.18	**0.16**	**1.87**	**0.02**	**0.27**	1.74	0.03	0.17
				abs(best-worst)	1.56	0.25	0.18	0.91	0.42	0.08	1.51	0.36	0.15	0.87	0.11	0.11

Figure 5.5: Results of the ablation study. Medians across 20 studies.

> Weighted losses together with hyper-parameter optimization fix the brittleness problem.

RQ4: Do we generate "dust" or "boulders"?

Figure 5.4 shows the distribution of partition sizes seen in 30 random samples. From that figure we say that our methods usually generate small number of partitions. Also, since we are usually dealing with classes ranging from 33 in acmeair to 109 in daytrader, Figure 5.4 tells us that we are not generating partitions with only one item per partition (for example, on plants, the median size is $5.4\% \times 111 = 6$ classes).

> Our partitions are neither "boulders" nor "dust".

At this point, it is instructive to ask what components of `DEEPLY` contribute to its performance. Rather than use a simple ablation study (where one removes one component at a time to measure performance changes), we run a more comprehensive suite of experiments by permuting between the different options. Specifically, we test {k-means, spectral} $\times$ {exponential, 1cycle} $\times$ {hyper-parameter optimization, no hyper-parameter optimization} $\times$ {unweighted loss, weighted loss}. This leads to the results of Figure 5.5.

The numbers in Figure 5.5 are medians over 20 repeats. For each column, we color based on performance on a red-white-green scale, where red indicates the worst performance, and green indicates best. The best scores of the column are in bold and colored white.

The results broadly divide into four groups based on the combinations of hyper-parameter optimization and the use of a weighted loss function. In detail, treatments using hyper-parameter optimization and a weighted loss function to choose from it perform the best. Treatments not using hyper-parameter optimization have dubious performance, as expected. More surprising, however, is that the worst category of performers use hyper-parameter optimization with an unweighted loss function. That is, in the case where a practitioner would have to choose between hyper-parameter optimization and a weighted loss, they should use the weighted loss, which is contrary to intuition. We explain this in the following manner. Consider the *true* Pareto frontier, made from thousands of configurations, of which hyper-parameter optimization samples a few. We posit that the weighted loss function better approximates the Pareto frontier, and therefore chooses better performers.

Among the top performing category, k-means performs better than spectral clustering. This is likely because the variance of the Gaussian kernel used by spectral clustering was set incorrectly. Because a significant part of this paper advocates for hyper-parameter optimization, we conclude that the spectral clustering results would likely have been better had the kernel parameters been tuned.

5.6 Hyper-parameter optimization

We divide our results for hyper-parameter optimization for the two methods proposed.

5.6.1 `SMOOTHIE`

Research Questions

RQ1: What are the benefits of increasing smoothness in SE datasets (measured in metrics like F1, recall, etc)?
RQ2: What are the costs of smoothing?
RQ3: Should we use different kinds of algorithms for SE data?

Results

RQ1: What are the benefits of increasing smoothness in SE datasets (measured in metrics like F1, recall, etc)?

Table 5.14 shows our results on **static code analysis false alarm detection**. Recall that the task here was to recognize static code warnings that, in the historical log, were ignored by developers. In the table, gray denotes "best results in this column, for that

Table 5.14: **RQ1 results:** Hyper-parameter optimization results for the static code warnings datasets. Anything that ties with the statistically best results are shown in **bold** (and such ties are determined using the statistics of Section 4.10). For Algorithm 1, we use $N_1 = 50, N_2 = 10$. FF = Feedforward network. NB = Naive Bayes. LR = Logistic Regression.

Dataset	Method (Learner)	#evals	pd	pf	precision
ant	Baseline (SVM)		0	0.25	0
	SMOOTHIE (FF)	10	**0.65**	**0**	**0.8**
	SMOOTHIE (NB)	10	**0.6**	**0**	**0.8**
	SMOOTHIE (LR)	10	**1**	**0**	**1**
	Random	10	0.2	0.5	0.5
	BOHB (FF)	10	**1**	**0.2**	0.5
	BOHB (FF)	50	**0.6**	1	0.6
cassandra	Baseline (SVM)		0.33	1	0.5
	SMOOTHIE (FF)	10	**0.83**	**0**	**0.75**
	SMOOTHIE (NB)	10	**1**	0.5	**0.83**
	SMOOTHIE (LR)	10	**1**	0.67	0.33
	Random	10	0	0.66	0
	BOHB (FF)	10	**1**	0.67	0.25
	BOHB (FF)	50	0	**0**	0
commons	Baseline (SVM)		**1**	1	**0.6**
	SMOOTHIE (FF)	10	**1**	**0.25**	**0.5**
	SMOOTHIE (NB)	10	0	**0**	0
	SMOOTHIE (LR)	10	**1**	1	**0.8**
	Random	10	0	**0**	0
	BOHB (FF)	10	**0.88**	1	**0.65**
	BOHB (FF)	50	**1**	1	0.2
derby	Baseline (SVM)		0.61	0.67	0.39
	SMOOTHIE (FF)	10	0.74	0.46	0.44
	SMOOTHIE (NB)	10	**0.90**	0.60	0.4
	SMOOTHIE (LR)	10	0.59	**0.06**	**0.8**
	Random	10	**1.0**	1.0	**0.68**
	BOHB (FF)	10	**0.93**	0.84	**0.68**
	BOHB (FF)	10	**1**	1	**0.68**
jmeter	Baseline (SVM)		0	**0**	0
	SMOOTHIE (FF)	10	**1**	**0**	**1**
	SMOOTHIE (NB)	10	**1**	0.33	0.5
	SMOOTHIE (LR)	10	0.67	**0**	0.63
	Random	10	0	1	0
	BOHB (FF)	10	0	**0**	0
	BOHB (FF)	50	0	**0**	0
lucene-solr	Baseline (SVM)		0	1	0
	SMOOTHIE (FF)	10	**0.5**	**0.25**	**0.5**
	SMOOTHIE (NB)	10	**1**	**0.5**	**0.5**
	SMOOTHIE (LR)	10	**1**	**0.25**	**0.67**
	Random	10	**1**	**0.25**	**0.67**
	BOHB (FF)	10	**0.75**	**0.5**	**0.33**
	BOHB (FF)	50	0	0	0
tomcat	Baseline (SVM)		0.73	0.77	0.39
	SMOOTHIE (FF)	10	0.14	**0.09**	**0.47**
	SMOOTHIE (NB)	10	0.43	**0.13**	**0.63**
	SMOOTHIE (LR)	10	0.71	0.3	**0.61**
	Random	10	0.43	0.5	**0.59**
	BOHB (FF)	10	0.7	0.48	**0.6**
	BOHB (FF)	50	**1**	1	**0.62**

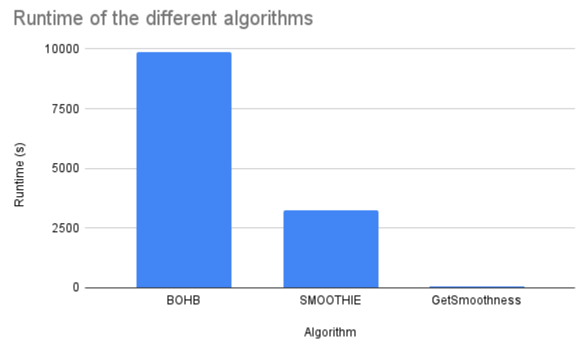


Figure 5.6: **RQ2 Results:** Runtime results for `SMOOTHIE`.

dataset". Note that `SMOOTHIE` (and more specifically, `SMOOTHIE` (FF) has much more gray than anything else. More specifically: all the non-`SMOOTHIE` methods have far more white cells (i.e. non best) than the `SMOOTHIE` methods. (for an exact count of the wins and ties, see Table 5.17).

Table 5.15 shows our results on **issue lifetime prediction**. Recall that the task here is to classify a GitHub issue according to how long it will take to resolve. As done in prior work [121], for 3-class classification, the number of days taken to fix bugs is binned into 0 to 1 days, 2 to 5 days, and 5 to 100 days. For 2-class, the number of days taken to fix bugs is binned into less than (or greater than) 4 days for Firefox, less than (or greater than) 5 days for Chromium, and less than (or greater than) 6 days for Eclipse. Note that:

- For two class prediction (eclipse(2), firefox(2), chromium(2)) `SMOOTHIE` does not perform well (as shown in Table 5.15, 1 tie and 2 losses). Indeed, we found that these datasets have the highest smoothness across all our SE datasets: the 2-class Chromium, Eclipse, and Firefox datasets had smoothness values of 9.4, 14.9, and 13.8 respectively. This explains why `SMOOTHIE` did not work well for the 2-class binned version of the data.

- For three-class problems (eclipse(3), firefox(3), chromium(3)) , `SMOOTHIE` wins in all cases.

Table 5.17 summarizes the **RQ1** results. In that table, two methods **tie** if, they achieve the same statistical rank (using the methods of Section 4.10). Otherwise, a **win** is when one measure has a different rank to another *and* one measure has better median values (where "better" means "less" for false alarms and "greater" for other metrics).

> **RQ1 (answer)**: In the usual case, `SMOOTHIE` rarely does worse and often performs better than the prior state of the art in hyper-parameter optimization.

RQ2: What are the costs of smoothing?

Figure 5.6 shows the runtimes for our method. We take care to distinguish the overheads associated with finding smoothness, versus the time required for another process to use smoothness to make a recommendation about what are good hyperparameter options.

As shown in Figure 5.6, the time required to compute `Get-Smoothness` smoothness is negligible. Those results are then used by `SMOOTHIE` for its decision making (and here, `SMOOTHIE` repeats its processing $N_1 + N_2 = 30 + 5$ times). Even with those repeats, `SMOOTHIE` is four times faster than the prior state of the art (see BOHB in Figure 5.6).

Figure 5.6 suggests that further optimizations might be possible if `SMOOTHIE` could reduce how many repeats its executes. This is a matter for further work, in another paper.

> **RQ2 (answer)**: `SMOOTHIE` works as well as the prior state of the art, and does so four times faster.

RQ3: Should we use different kinds of algorithms for SE data?

Apart from performance and runtime details, what does the above mean for the challenge problem sketched out in the introduction to this paper? To answer that question, we first recall Figure 4.4 where the mean smoothness for our SE and AI data has values 2 and 5, respectively.

Turning now to Figure 4.3, that figure reports the ratio of datasets for which `SMOOTHIE` outperforms BOHB (measured in terms of the F1 metric).

We note that, in those results, different algorithms are the best choice for low and high smoothness values:

- `SMOOTHIE` usually wins at the low values typically associated with SE data;
- While BOHB usually wins at the higher values, seen only in the non-SE data.

5.6.2 `AHSC`

Research Questions

We are primarily interested in the following research question:

RQ1: Does optimizing for flatness improve the performance of ML models?

Results

Table 5.18 shows a comparison of `AHSC` on 14 datasets with five popular hyper-parameter optimization methods. For these results, we compared the means over 20 repeats using pairwise Mann-Whitney U-tests [138], followed by a p-value correction using the two-stage Benjamini-Hochberg procedure, as endorsed by Farcomeni [68]. We use a 5% significance level to determine the best-performing methods.

Table 5.19 summarizes the wins, ties, and losses of each method over the 14 datasets. Notably, `AHSC` is the only method with any win (this is on the *wine* dataset). Moreover, our method is between 200% (compared to TuRBO) to 700% (compared to BOHB) faster.

Table 5.15: **RQ1 results:** Hyper-parameter optimization for Bugzilla issue lifetime prediction. For Algorithm 1, we use $N_1, N_2 = 30, 5$. Anything that ties with the statistically best results are shown in **bold** (and such ties are determined using the statistics of Section 4.10). Our baseline is DeepTriage [137]. In the dataset column, the number of classes the data is binned into is in parentheses.

Dataset	Learner	HPO method	#evals	Accuracy
eclipse (2)	Baseline			61
	Feedforward	SMOOTHIE	5	69.3
	Naive Bayes	SMOOTHIE	5	63
	Logistic	SMOOTHIE	5	62.5
	Best (Feedforward)	Random	5	**80.2**
	Best (Feedforward)	BOHB	5	45.8
	Best (Feedforward)	BOHB	30	45.8
firefox (2)	Baseline			67
	Feedforward	SMOOTHIE	5	**69**
	Naive Bayes	SMOOTHIE	5	66.6
	Logistic	SMOOTHIE	5	66.4
	Best (Feedforward)	Random	5	**76.5**
	Best (Feedforward)	BOHB	5	36
	Best (Feedforward)	BOHB	30	35.9
chromium (2)	Baseline			63
	Feedforward	SMOOTHIE	5	62.2
	Naive Bayes	SMOOTHIE	5	63
	Logistic	SMOOTHIE	5	63.1
	Best (Logistic)	Random	5	**67.1**
	Best (Logistic)	BOHB	5	44.5
	Best (Logistic)	BOHB	30	44.4
eclipse (3)	Baseline			44
	Feedforward	SMOOTHIE	5	**71.2**
	Naive Bayes	SMOOTHIE	5	51.6
	Logistic	SMOOTHIE	5	51.2
	Best (Feedforward)	Random	5	36.3
	Best (Feedforward)	BOHB	5	29.2
	Best (Feedforward)	BOHB	30	46.2
firefox (3)	Baseline			44
	Feedforward	SMOOTHIE	5	**69.2**
	Naive Bayes	SMOOTHIE	5	49.5
	Logistic	SMOOTHIE	5	49.3
	Best (Feedforward)	Random	5	46.1
	Best (Feedforward)	BOHB	5	32.9
	Best (Feedforward)	BOHB	30	44.4
chromium (3)	Baseline			43
	Feedforward	SMOOTHIE	5	**71.9**
	Naive Bayes	SMOOTHIE	5	49.1
	Logistic	SMOOTHIE	5	49.8
	Best (Feedforward)	Random	5	53.5
	Best (Feedforward)	BOHB	5	27.5
	Best (Feedforward)	BOHB	30	44.6

Table 5.16: **RQ1 results:** Hyper-optimization for defect prediction results. For Algorithm 1, we use $N_1, N_2 = 30, 5$. Anything that ties with the statistically best results are shown in **bold** (and such ties are determined using the statistics of Section 4.10).

Dataset	**Learner**	**HPO method**	**#evals**	**F1**
camel	Baseline (GHOST)			**90.9**
	Feedforward	SMOOTHIE	5	**90.9**
	Naive Bayes	SMOOTHIE	5	**90.9**
	Logistic	SMOOTHIE	5	**90.9**
	Best (NB)	Random	5	**90.9**
	Best (NB)	BOHB	5	**90.9**
	Best (NB)	BOHB	30	**90.9**
ivy	Baseline (GHOST)			83.4
	Feedforward	SMOOTHIE	5	**94**
	Naive Bayes	SMOOTHIE	5	**94**
	Logistic	SMOOTHIE	5	**94**
	Best (NB)	Random	5	**94**
	Best (NB)	BOHB	5	**94**
	Best (NB)	BOHB	30	**94**
log4j	Baseline (GHOST)			65.9
	Feedforward	SMOOTHIE	5	**79.6**
	Naive Bayes	SMOOTHIE	5	**79.6**
	Logistic	SMOOTHIE	5	**79.6**
	Best (NB)	Random	5	**79.6**
	Best (NB)	BOHB	5	**79.6**
	Best (NB)	BOHB	30	**79.6**
synapse	Baseline (GHOST)			**84.6**
	Feedforward	SMOOTHIE	5	**84.4**
	Naive Bayes	SMOOTHIE	5	**84.4**
	Logistic	SMOOTHIE	5	**84.4**
	Best (NB)	Random	5	**84.4**
	Best (NB)	BOHB	5	74.5
	Best (NB)	BOHB	30	0
velocity	Baseline (GHOST)			50.3
	Feedforward	SMOOTHIE	5	55.4
	Naive Bayes	SMOOTHIE	5	**79.8**
	Logistic	SMOOTHIE	5	**79.8**
	Best (NB)	Random	5	**79.8**
	Best (NB)	BOHB	5	**79.8**
	Best (NB)	BOHB	30	**79.8**
xalan	Baseline (GHOST)			66
	Feedforward	SMOOTHIE	5	**68.3**
	Naive Bayes	SMOOTHIE	5	**68.3**
	Logistic	SMOOTHIE	5	**68.3**
	Best (Logistic)	Random	5	**68.2**
	Best (Logistic)	BOHB	5	0
	Best (Logistic)	BOHB	30	**68.3**

Table 5.17: **RQ1 results (summary):** Hyper-parameter optimization via `SMOOTHIE` from 20 repeats over 19 data sets.

	Frequency counts of:		
Datasets	**wins**	**ties**	**losses**
Static code warnings	4	3	0
Issue lifetime prediction (binary)	0	1	2
Issue lifetime prediction (multi-class)	3	0	0
Defect prediction	0	6	0

Table 5.18: Experimental results on various classification datasets. `AHSC` is our method. Values shown are medians over 20 repeats. Statistically best results are highlighted in **bold**.

Image		
Dataset	**HPO method**	**Accuracy**
MNIST	`AHSC`	**98.65**
	Hyperopt	97.22
	Random	**98.95**
	TuRBO	**98.99**
	HEBO	**98.94**
	BOHB	98.85
SVHN	`AHSC`	**86.63**
	Hyperopt	67.20
	Random	**91.86**
	TuRBO	80.41
	HEBO	92.89
	BOHB	79.67

Bayesmark		
Dataset	**HPO method**	**Score**
breast	`AHSC`	**93.98**
	Hyperopt	**92.68**
	Random	90.45
	TuRBO	88.97
	HEBO	90.84
	BOHB	92.37
digits	`AHSC`	84.74
	Hyperopt	**96.85**
	Random	87.39
	TuRBO	89.51
	HEBO	**95.24**
	BOHB	91.50
iris	`AHSC`	**91.00**
	Hyperopt	79.83
	Random	82.98
	TuRBO	83.52
	HEBO	78.27
	BOHB	**92.30**
wine	`AHSC`	**92.35**
	Hyperopt	83.98
	Random	76.59
	TuRBO	81.15
	HEBO	74.93
	BOHB	81.68

OpenML		
Dataset	**HPO method**	**AUC**
vehicle	`AHSC`	**0.885**
	Hyperopt	**0.883**
	Random	0.873
	TuRBO	**0.882**
	HEBO	**0.884**
	BOHB	**0.883**
blood-transf...	`AHSC`	0.721
	Hyperopt	**0.728**
	Random	0.708
	TuRBO	0.720
	HEBO	0.718
	BOHB	**0.725**
Australian	`AHSC`	**0.934**
	Hyperopt	**0.932**
	Random	0.928
	TuRBO	**0.932**
	HEBO	**0.935**
	BOHB	**0.928**
car	`AHSC`	1.0
	Hyperopt	**1.0**
	Random	1.0
	TuRBO	1.0
	HEBO	1.0
	BOHB	**1.0**
phoneme	`AHSC`	0.560
	Hyperopt	**0.564**
	Random	0.561
	TuRBO	**0.564**
	HEBO	0.563
	BOHB	**0.563**
segment	`AHSC`	**0.961**
	Hyperopt	**0.960**
	Random	0.955
	TuRBO	**0.960**
	HEBO	**0.961**
	BOHB	**0.960**
credit-g	`AHSC`	0.778
	Hyperopt	**0.782**
	Random	0.766
	TuRBO	0.763
	HEBO	0.763
	BOHB	**0.752**
kcl	`AHSC`	**0.816**
	Hyperopt	**0.817**
	Random	0.769
	TuRBO	0.775
	HEBO	**0.816**
	BOHB	**0.785**

Table 5.19: Summary of results for `AHSC`.

Algorithm	**Wins**	**Ties**	**Losses**
AHSC	1	8	5
TPE	0	10	4
Random	0	2	12
TuRBO	0	5	9
HEBO	0	6	8
BOHB	0	8	6

CHAPTER

6

ERRATA

During the final writing stage of this thesis, the following inaccuracies were found.

An error was identified in the proof of Theorem 4.4 and Theorem 4.7 presented in Chapter 3. The error arose from an incorrect computation of gradients. Due to time constraints, we were unable to conduct a thorough re-evaluation of the proofs. Below, we report a corrected version that more rigorously analyzes the gradients and the resulting Hessian.

6.1 Smoothness for feedforward networks

We first reiterate a result from user1551 .

Lemma 7 (Norm of Moore-Penrose pseudo-inverse). *Let B be the left Moore-Penrose pseudo-inverse of some matrix A. Then*

$$\|B\|_2 = \frac{1}{\sigma_{min}(A)}$$

where $\sigma_{min}(A)$ denotes the least singular value of A.

Proof. It is well-established that AB is the orthogonal projection to the column space of

A, so that for every vector y, $ABy \perp (I - AB)y$. Therefore:

$$\begin{aligned}
\|B\|_2 &= \max_{y \neq 0} \frac{\|By\|}{\|y\|} \\
&= \max_{y \neq 0} \frac{\|By\|}{\|ABy + (I - AB)y\|} \\
&= \max_{By \neq 0} \frac{\|By\|}{\|ABy + (I - AB)y\|} \\
&\leq \max_{By \neq 0} \frac{\|By\|}{\|ABy\|} \\
&\leq \max_{x \neq 0} \frac{\|x\|}{\|Ax\|} \\
&= \left(\min_{x \neq 0} \frac{\|Ax\|}{\|x\|} \right)^{-1} \\
&= \frac{1}{\sigma_{min}(A)}
\end{aligned}$$

□

Lemma 8. *For a deep learner with ReLU activations in the hidden layers, the last two layers being fully-connected, and a softmax activation at the last layer,*

$$\frac{\partial E}{\partial z_h^{[L]}} = a_h^{[L]} - \frac{1}{m} \sum_{i=1}^{m} [y^{(i)} = h]$$

under the cross-entropy loss.

Proof. Consider the Iverson notation version of the general cross-entropy loss:

$$E(\mathbf{a}^{[L]}) = -\frac{1}{m} \sum_{i=1}^{m} \sum_{j=1}^{k} [y^{(i)} = j] \log a_j^{[L]}$$

Then,

$$
\begin{aligned}
\frac{\partial E}{\partial z_h^{[L]}} &= -\frac{1}{m}\sum_{i=1}^{m}\sum_{j=1}^{k}\frac{[y^{(i)}=j]}{a_j^{[L]}}a_j^{[L]}\left(\delta_{hj}-a_h^{[L]}\right) \\
&= -\frac{1}{m}\sum_{i=1}^{m}\sum_{j=1}^{k}[y^{(i)}=j]\left(\delta_{hj}-a_h^{[L]}\right) \\
&= -\frac{1}{m}\sum_{i=1}^{m}\sum_{j=1}^{k}[y^{(i)}=j]\delta_{hj}+\frac{1}{m}\sum_{i=1}^{m}\sum_{j=1}^{k}[y^{(i)}=j]a_h^{[L]} \\
&= -\frac{1}{m}\sum_{i=1}^{m}[y^{(i)}=h]+\frac{a_h^{[L]}}{m}\sum_{i=1}^{m}1 \\
&= a_h^{[L]}-\frac{1}{m}\sum_{i=1}^{m}[y^{(i)}=h] \qquad (6.1)
\end{aligned}
$$

□

Theorem 6.1 (Smoothness for feedforward networks)**.** *For a deep learner with ReLU activations in the hidden layers, the last two layers being fully-connected, and a softmax activation at the last layer, the smoothness of the cross-entropy loss is given by*

$$\sup\|\nabla_W^2 E\| = \left\|a^{[L-1]}\right\|_2\left(\left\|a^{[L]}\right\|_2+\left\|a^{[L]}\right\|_2^2\right)+\frac{\left\|a^{[L]}\right\|_2}{\sigma_{min}(W^{[L]})} \tag{6.2}$$

Proof. From Lemma 8 and $z^{[L]} = W^{[L]}a^{[L-1]} + b^{[L]}$,

$$\nabla_{W^{[L]}}E = a^{[L]}a^{[L-1]T}$$

Now, we work toward the Hessian:

$$\frac{\partial^2 E}{\partial a^{[L]}\partial W^L} = I_{n^{[L]}\times n^{[L]}}\otimes a^{[L-1]T}+a^{[L]}\otimes\left(\frac{\partial a^{[L]}}{\partial a^{[L-1]T}}\right)^{-1}$$

which is of shape $(n^{[L]}, n^{[L]}, n^{[L-1]})$. Note that because the derivative inside the inverse is not square, this inverse is the Moore-Penrose pseudo-inverse. We have:

$$
\begin{aligned}
\frac{\partial a^{[L]}}{\partial a^{[L-1]}} &= \frac{\partial a^{[L]}}{\partial z^{[L]}}\frac{\partial z^{[L]}}{\partial a^{[L-1]}} \\
&= \left(\operatorname{diag}(a^{[L]})-a^{[L]}a^{[L]T}\right)W^{[L]}I_{n^{[L-1]}\times n^{[L-1]}} \\
\frac{\partial a^{[L]}}{\partial a^{[L-1]T}} &= I_{n^{[L-1]}\times n^{[L-1]}}W^{[L]T}\left(\operatorname{diag}(a^{[L]})-a^{[L]}a^{[L]T}\right)
\end{aligned}
$$

so that

$$\frac{\partial^2 E}{\partial a^{[L]} \partial W^L} = I_{n^{[L]} \times n^{[L]}} \otimes a^{[L-1]T} + a^{[L]} \left(I_{n^{[L-1]} \times n^{[L-1]}} W^{[L]T} \left(\text{diag}(a^{[L]}) - a^{[L]} a^{[L]T} \right) \right)^{-1}$$

Next,

$$\begin{aligned} \frac{\partial a^{[L]}}{\partial W^{[L]}} &= \frac{\partial a^{[L]}}{\partial z^{[L]}} \frac{\partial z^{[L]}}{\partial W^{[L]}} \\ &= \left(\text{diag}(a^{[L]}) - a^{[L]} a^{[L]T} \right) \otimes a^{[L-1]} \end{aligned}$$

which is of shape $(n^{[L]}, n^{[L]}, n^{[L-1]})$. This yields the Hessian

$$\begin{aligned} \nabla^2_{W^{[L]}} E &= \left(I_{n^{[L]} \times n^{[L]}} \otimes a^{[L-1]T} \right) \left(\left(\text{diag}(a^{[L]}) - a^{[L]} a^{[L]T} \right) \otimes a^{[L-1]} \right)^T + \\ &\quad a^{[L]} \left(I_{n^{[L-1]} \times n^{[L]}} W^{[L]T} \left(\text{diag}(a^{[L]}) - a^{[L]} a^{[L]T} \right) \right)^{-1} \left(\text{diag}(a^{[L]}) - a^{[L]} a^{[L]T} \right) \otimes a^{[L-1]} \\ \left\| \nabla^2_{W^{[L]}} E \right\|_2 &\leq \left\| a^{[L-1]} \right\|_2^2 \left(\left\| a^{[L]} \right\|_2 + \left\| a^{[L]} \right\|_2^2 \right) + \left\| a^{[L]} \right\|_2 \left\| a^{[L-1]} \right\|_2 \frac{1}{\sigma_{min}(W^{[L]})} \end{aligned}$$

where for the last step we used (i) $\|XY\|_2 \leq \|X\|_2 \|Y\|_2$, and (ii) Lemma 7. □

6.1.1 So Does This Change the Results of this Thesis?

Of course, this raises the obvious question: *why did the experiments based on the incorrect equation work?* This is because the above corrected version simplifies to a term very close to the incorrect version. To see why, note that the last layer uses a softmax activation, so that $0 \leq \left\| a^{[L]} \right\| \leq 1$, where the latter equality only occurs when the classifier is absolutely certain about one class. Therefore, the above simplifies to:

$$\left\| \nabla^2_{W^{[L]}} E \right\|_F \leq 2 \left\| a^{[L]} \right\|_2 \left\| a^{[L-1]} \right\|_2^2 + \left\| a^{[L]} \right\|_2 \left\| a^{[L-1]} \right\|_2 \frac{1}{\sigma_{min}(W^{[L]})}$$

Now, note that since the penultimate layer usually uses the ReLU activation, it is often true that $\left\| a^{[L]} \right\|_2 \leq \left\| a^{[L-1]} \right\|_2$, so that

$$\left\| \nabla^2_{W^{[L]}} E \right\|_F \leq 2 \left\| a^{[L-1]} \right\|_2^3 + \left\| a^{[L-1]} \right\|_2^2 \frac{1}{\sigma_{min}(W^{[L]})} \leq 2 \left\| a^{[L-1]} \right\|_2^3 + \left\| a^{[L-1]} \right\|_2^2 \frac{1}{\left\| W^{[L]} \right\|_2}$$

The original equation does not use the first term, and uses an incorrect constant in the second term; this turns out to be a sufficiently close approximation in practice that the experiments still yield strong results.

6.2 Strong convexity for feedforward networks

The corrected proof for the strong convexity largely follows the same steps as for smoothness, since most steps above used an equality. The main change from the smoothness arises in the last step, where simplifications that were done for the smoothness to achieve an inequality cannot be used here, since we require the opposite inequality.

CHAPTER

7 DISCUSSION

This dissertation discussed the application of deep learning for five software analytics tasks. In doing so, we dismissed the prior belief about deep learning that the automated feature engineering performed by the first few layers suffices. Rather, we found that applying neural networks to software analytics tasks requires an intricate set of engineering techniques that work together to achieve excellent results. Moreover, this set of techniques can outperform more complex neural architectures such as convolutional networks (as seen in code smell detection) or deep belief networks (as seen in defect prediction). We tested our methods on 122 datasets (40 for defect prediction, 15 for issue lifetime prediction, 40 for code smell detection, 9 for false alarm detection in static code warnings, 4 for automated microservice partitioning, 14 for non-SE hyper-parameter optimization)

Of these engineering techniques, this dissertation strongly advocates for hyper-parameter optimization, and proposes two novel methods of doing it in a computationally efficient manner.

7.1 Threats to Validity

Sampling bias. We claim that the sampling bias is mitigated in this dissertation through the use of multiple, varied datasets, including different modalities (tabular, image, and graph). In total, we tested on over 100 datasets, so that the crux of this dissertation (that simpler methods should be tried first, and that hyper-parameter optimization can be performed cheaply) is not limited to a few samples or datasets.

Evaluation bias. For each of the five SE tasks, and for all the non-SE datasets, we used the same evaluation metrics as in prior work. These metrics are well-established and are chosen appropriately for each task (for example, defect prediction does not use accuracy, since the datasets are very imbalanced).

Learner bias. It is important to recognize that this dissertation heavily focuses on feedforward networks. Although the theory developed for the novel hyper-parameter optimization methods do not necessitate a feedforward learner, only a few experiments on convolutional models have been performed. As such, we caution against directly generalizing these conclusions to more models without smaller-scale pilot experiments.

Order bias. Unless strictly necessary, the datasets were shuffled randomly, and all experiments were repeated multiple times using different random seeds. In some cases, such as defect prediction, it is incorrect to train on future versions and test on past versions. In those cases, we did not perform such random shuffling, and we maintain that such order bias was necessary.

External validity. In all our case studies, including for non-SE datasets, we compare against the prior state-of-the-art for each task.

Statistical validity. All experiments in this dissertation were repeated 20 times using different random seeds. In all cases, a statistical test was applied (either Scott-Knott, or pairwise Mann-Whitney U-tests followed by a two-stage Benjamini-Hochberg correction procedure).

7.2 Future Work

This dissertation lays the groundwork for several avenues of future work, whilst filling in various research gaps. We discuss these specific avenues here.

The clearest line of future work, one that is arguably already ongoing, is a deeper theoretical investigation of feedforward networks, and how various data preprocessing operators shape the landscape and possibly make it smoother. As a clear example of this, Santurkar et al. [181] show that batch normalization makes the loss landscape smoother and thus easier to optimize; indeed, this was a motivation for us computing the smoothness to test if our preprocessors also smoothed the loss landscape. Similarly, Li et al. [124] show that the popular ResNet [88] architecture also smooths out the loss landscape by introducing its skip-connections. Much current work investigates the impact of mild or moderate over-parameterization of deep ReLU networks; we urge researchers to also study the effect that preprocessors have on the landscape.

Another similar line of future work involves a computationally cheaper version of fuzzy sampling. As-is, fuzzy sampling introduces a significant number of new samples, which slows down the overall runtime by a noticeable factor. It is left as future work whether a stronger exponential decay, or a modified version of the loss can be used instead.

A third avenue of future work would be to test the novel hyper-parameter optimization algorithms from this dissertation on more complex architectures, specifically Transformers, and study the computational benefits offered. Such future studies should also compare against other HPO methods in both runtime and evaluation performance.

It has not escaped our attention that there is a line of work investigating the application of topology to machine learning. Indeed, we can make several connections to topological data analysis (TDA) with our work: a particularly trivial example is to note that the graphs produced by the Mapper algorithm [189] might be useful in identifying the level of complexity of a dataset, and therefore, the complexity of the learning algorithm necessary for that dataset. Some prior studies have investigated the utility of TDA [133, 157, 158, 168], but we argue that further investigation is necessary.

Indeed, more investigation is needed into the claim that SE datasets are simpler than non-SE datasets: there are several studies proposing complexity measures for datasets [28, 31, 93, 117, 123, 132, 144, 159, 215]. A comprehensive study similar to Jiang et al. [103] that establishes "reliable" complexity measures for datasets is necessary.

Further work is also required to extend the two hyper-parameter optimization methods proposed here to other learning algorithms through center-different approximations.

A potentially promising, but speculative line of future work involves topology-inspired approaches to hyper-parameter optimization. Specifically, if all hyper-parameter options are boolean, a Boolean algebra[1] B of options can be constructed. By the Stone representation theorem, every Boolean algebra is isomorphic to a field of clopen subsets of a Stone space[2], each of whose elements are ultrafilters on B. By developing a reasonable metric on this Stone space, one can exploit the properties of the topology to develop HPO methods for such boolean HPO spaces. For example, a pseudo-metric that selects for low loss values and flatness is given below, where a and b are two hyper-parameter options, and $\mathcal{X}$ is the space over which optimization is performed:

[1] An *algebra* (or a field of sets) in this context is a set X paired with a collection of subsets $\mathcal{F}$ containing the empty set that is closed under finite unions and intersections, and complements.

[2] A Stone space is a compact, totally disconnected, Hausdorff space.

$$d(a, b) \triangleq \mathbb{E}\left[\left|\int\limits_{\mathcal{X}} p(x)\left(\mathcal{L}(\cdot; a) - \mathcal{L}(\cdot; b)\right) dx\right|\right]$$

where $p(x)$ is the pdf of the data distribution at x, and any KDE method can be used.

Finally, we urge the software engineering community to reassess the complexity of the models they apply to the learning problems that they encounter, and to match the complexity of the learner to the complexity of the problem.

REFERENCES

[1] Agrawal, A., Fu, W., Chen, D., Shen, X., and Menzies, T. How to "dodge" complex software analytics. *IEEE Transactions on Software Engineering*, 47(10):2182–2194, 2019.

[2] Agrawal, A., Yang, X., Agrawal, R., Yedida, R., Shen, X., and Menzies, T. Simpler hyperparameter optimization for software analytics: why, how, when. *IEEE Transactions on Software Engineering*, 2021.

[3] Akbarinasaji, S., Caglayan, B., and Bener, A. Predicting bug-fixing time: A replication study using an open source software project. *journal of Systems and Software*, 136:173–186, 2018.

[4] Al-Debagy, O. and Martinek, P. A comparative review of microservices and monolithic architectures. In *2018 IEEE 18th International Symposium on Computational Intelligence and Informatics (CINTI)*, pp. 000149–000154. IEEE, 2018.

[5] Allamanis, M., Barr, E. T., Bird, C., and Sutton, C. Suggesting accurate method and class names. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pp. 38–49, 2015.

[6] Allamanis, M., Peng, H., and Sutton, C. A convolutional attention network for extreme summarization of source code. In *International conference on machine learning*, pp. 2091–2100, 2016.

[7] Allen-Zhu, Z., Li, Y., and Song, Z. A convergence theory for deep learning via over-parameterization. In *International conference on machine learning*, pp. 242–252. PMLR, 2019.

[8] Ansel, J., Kamil, S., Veeramachaneni, K., Ragan-Kelley, J., Bosboom, J., O'Reilly, U.-M., and Amarasinghe, S. Opentuner: An extensible framework for program autotuning. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pp. 303–316, 2014.

[9] Ansel, J., Kamil, S., Veeramachaneni, K., Ragan-Kelley, J., Bosboom, J., O'Reilly, U.-M., and Amarasinghe, S. Opentuner: An extensible framework for program autotuning. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Edmonton, Canada, Aug 2014. URL `http://groups.csail.mit.edu/commit/papers/2014/ansel-pact14-opentuner.pdf`.

[10] Arango, S. P., Jomaa, H. S., Wistuba, M., and Grabocka, J. Hpo-b: A large-scale reproducible benchmark for black-box hpo based on openml. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*, 2021.

[11] Arcuri, A. and Briand, L. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pp. 1–10, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450304450. doi: 10.1145/1985793.1985795. URL `https://doi.org/10.1145/1985793.1985795`.

[12] Arcuri, A. and Fraser, G. Parameter tuning or default values? an empirical investigation in search-based software engineering. *Empirical Software Engineering*, 2013.

[13] Ayewah, N. and Pugh, W. The google findbugs fixit. In *Proceedings of the 19th international symposium on Software testing and analysis*, pp. 241–252, 2010.

[14] Azeem, M. I., Palomba, F., Shi, L., and Wang, Q. Machine learning techniques for code smell detection: A systematic literature review and meta-analysis. *Information and Software Technology*, 108:115–138, 2019.

[15] Bahdanau, D., Cho, K., and Bengio, Y. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[16] Baker, B., Gupta, O., Naik, N., and Raskar, R. Designing neural network architectures using reinforcement learning. *arXiv preprint arXiv:1611.02167*, 2016.

[17] Baldassi, C., Ingrosso, A., Lucibello, C., Saglietti, L., and Zecchina, R. Subdominant dense clusters allow for simple learning and high computational performance in neural networks with discrete synapses. *Physical review letters*, 115(12):128101, 2015.

[18] Baldassi, C., Borgs, C., Chayes, J. T., Ingrosso, A., Lucibello, C., Saglietti, L., and Zecchina, R. Unreasonable effectiveness of learning neural networks: From accessible states and robust ensembles to basic algorithmic schemes. *Proceedings of the National Academy of Sciences*, 113(48):E7655–E7662, 2016.

[19] Banerjee, S., Clapp, L., and Sridharan, M. 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*, pp. 740–750, 2019.

[20] Bansiya, J. and Davis, C. G. A hierarchical model for object-oriented design quality assessment. *IEEE Transactions on software engineering*, 28(1):4–17, 2002.

[21] Barkan, E., Hazan, A., and Ratner, V. Reduce discrepancy of human annotators in medical imaging by automatic visual comparison to similar cases, February 9 2021. US Patent 10,916,343.

[22] Bartlett, P. L., Harvey, N., Liaw, C., and Mehrabian, A. Nearly-tight vc-dimension and pseudodimension bounds for piecewise linear neural networks. *The Journal of Machine Learning Research*, 20(1):2285–2301, 2019.

[23] Beck, K., Fowler, M., and Beck, G. Bad smells in code. *Refactoring: Improving the design of existing code*, 1(1999):75–88, 1999.

[24] Beller, M., Bholanath, R., McIntosh, S., and Zaidman, A. 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, pp. 470–481. IEEE, 2016.

[25] Bergstra, J. and Bengio, Y. Random search for hyper-parameter optimization. *Journal of machine learning research*, 13(2), 2012.

[26] Bergstra, J., Bardenet, R., Bengio, Y., and Kégl, B. Algorithms for hyper-parameter optimization. *Advances in neural information processing systems*, 24, 2011.

[27] Bergstra, J., Yamins, D., and Cox, D. D. Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. 2013.

[28] Birdal, T., Lou, A., Guibas, L. J., and Simsekli, U. Intrinsic dimension, persistent homology and generalization in neural networks. *Advances in Neural Information Processing Systems*, 34:6776–6789, 2021.

[29] Bischl, B., Binder, M., Lang, M., Pielok, T., Richter, J., Coors, S., Thomas, J., Ullmann, T., Becker, M., Boulesteix, A.-L., et al. Hyperparameter optimization: Foundations, algorithms, best practices, and open challenges. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 13(2):e1484, 2023.

[30] Box, G. E. and Cox, D. R. An analysis of transformations. *Journal of the Royal Statistical Society Series B: Statistical Methodology*, 26(2):211–243, 1964.

[31] Branchaud-Charron, F., Achkar, A., and Jodoin, P.-M. Spectral metric for dataset complexity assessment. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 3215–3224, 2019.

[32] Bray, A. J. and Dean, D. S. Statistics of critical points of gaussian fields on large-dimensional spaces. *Physical review letters*, 98(15):150201, 2007.

[33] Briand, L. C., Brasili, V., and Hetmanski, C. J. Developing interpretable models with optimized set reduction for identifying high-risk software components. *IEEE Transactions on Software Engineering*, 19(11):1028–1044, 1993.

[34] Brown, W. H., Malveau, R. C., McCormick, H. W. S., and Mowbray, T. J. *AntiPatterns: refactoring software, architectures, and projects in crisis*. John Wiley & Sons, Inc., 1998.

[35] Buda, M., Maki, A., and Mazurowski, M. A. A systematic study of the class imbalance problem in convolutional neural networks. *Neural Networks*, 106:249–259, 2018.

[36] Byrd, R. H., Lu, P., Nocedal, J., and Zhu, C. A limited memory algorithm for bound constrained optimization. *SIAM Journal on scientific computing*, 16(5): 1190–1208, 1995.

[37] Calcagno, C., Distefano, D., Dubreil, J., Gabi, D., Hooimeijer, P., Luca, M., O'Hearn, P., Papakonstantinou, I., Purbrick, J., and Rodriguez, D. Moving fast with software verification. In *NASA Formal Methods Symposium*, pp. 3–11. Springer, 2015.

[38] Cao, Y. and Gu, Q. Generalization error bounds of gradient descent for learning over-parameterized deep relu networks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pp. 3349–3356, 2020.

[39] Chaudhari, P., Choromanska, A., Soatto, S., LeCun, Y., Baldassi, C., Borgs, C., Chayes, J., Sagun, L., and Zecchina, R. Entropy-sgd: Biasing gradient descent into wide valleys. *Journal of Statistical Mechanics: Theory and Experiment*, 2019(12): 124018, 2019.

[40] Chawla, N. V., Bowyer, K. W., Hall, L. O., and Kegelmeyer, W. P. Smote: Synthetic minority over-sampling technique. *J. Artif. Int. Res.*, 16(1):321–357, June 2002. ISSN 1076-9757.

[41] Chen, C., Xing, Z., Liu, Y., and Ong, K. L. X. Mining likely analogical apis across third-party libraries via large-scale unsupervised api semantics embedding. *IEEE Transactions on Software Engineering*, 2019.

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

[43] Choetkiertikul, M., Dam, H. K., Tran, T., Pham, T. T. M., Ghose, A., and Menzies, T. A deep learning model for estimating story points. *IEEE Transactions on Software Engineering*, 2018.

[44] Choromanska, A., Henaff, M., Mathieu, M., Arous, G. B., and LeCun, Y. The loss surfaces of multilayer networks. In *Artificial intelligence and statistics*, pp. 192–204. PMLR, 2015.

[45] Christakis, M. and Bird, C. 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*, pp. 332–343. ACM, 2016.

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

[47] Cordeiro, F. R. and Carneiro, G. 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)*, pp. 9–16. IEEE, 2020.

[48] Cowen-Rivers, A. I., Lyu, W., Tutunov, R., Wang, Z., Grosnit, A., Griffiths, R. R., Maraval, A. M., Jianye, H., Wang, J., Peters, J., et al. Hebo: pushing the limits of sample-efficient hyper-parameter optimisation. *Journal of Artificial Intelligence Research*, 74:1269–1349, 2022.

[49] Cybenko, G. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314, 1989.

[50] DasGupta, B. and Schnitger, G. The power of approximating: a comparison of activation functions. *Advances in neural information processing systems*, 5, 1992.

[51] Dauphin, Y. N., Pascanu, R., Gulcehre, C., Cho, K., Ganguli, S., and Bengio, Y. Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. *Advances in neural information processing systems*, 27, 2014.

[52] Daya, S., Van Duy, N., Eati, K., Ferreira, C., Glozic, D., Gucer, V., Gupta, M., Joshi, S., Lampkin, V., Martins, M., et al. *Microservices from Theory to Practice: Creating Applications in IBM Bluemix Using the Microservices Approach*. IBM Redbooks, 2016. ISBN 9780738440811. URL `https://books.google.com/books?id=eOZyCgAAQBAJ`.

[53] Deb, K., Pratap, A., Agarwal, S., and Meyarivan, T. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE transactions on evolutionary computation*, 6 (2):182–197, 2002.

[54] Debus, C., Piraud, M., Streit, A., Theis, F., and Götz, M. Reporting electricity consumption is essential for sustainable ai. *Nature Machine Intelligence*, 5(11): 1176–1178, 2023.

[55] Desai, U., Bandyopadhyay, S., and Tamilselvam, S. Graph neural network to dilute outliers for refactoring monolith application. *arXiv preprint arXiv:2102.03827*, 2021.

[56] Dhar, P. The carbon impact of artificial intelligence. *Nat. Mach. Intell.*, 2(8): 423–425, 2020.

[57] Dinh, L., Pascanu, R., Bengio, S., and Bengio, Y. Sharp minima can generalize for deep nets. In *International Conference on Machine Learning*, pp. 1019–1028. PMLR, 2017.

[58] Dou, H., Zhang, L., Zhang, Y., Chen, P., and Zheng, Z. Turbo: A cost-efficient configuration-based auto-tuning approach for cluster-based big data frameworks. *Journal of Parallel and Distributed Computing*, 177:89–105, 2023.

[59] Du, S. S., Zhai, X., Poczos, B., and Singh, A. Gradient descent provably optimizes over-parameterized neural networks. *arXiv preprint arXiv:1810.02054*, 2018.

[60] Duchi, J. Lecture notes on statistics and information theory. *https://web.stanford.edu/class/stats311/lecture-notes.pdf*, 2023.

[61] Dziugaite, G. K. and Roy, D. M. Computing nonvacuous generalization bounds for deep (stochastic) neural networks with many more parameters than training data. *arXiv preprint arXiv:1703.11008*, 2017.

[62] Eder, S., Junker, M., Jürgens, E., Hauptmann, B., Vaas, R., and Prommer, K.-H. How much does unused code matter for maintenance? In *2012 34th International Conference on Software Engineering (ICSE)*, pp. 1102–1111. IEEE, 2012.

[63] Eggensperger, K., Müller, P., Mallik, N., Feurer, M., Sass, R., Klein, A., Awad, N., Lindauer, M., and Hutter, F. Hpobench: A collection of reproducible multi-fidelity benchmark problems for hpo. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*, 2021.

[64] Elsken, T., Metzen, J.-H., and Hutter, F. Simple and efficient architecture search for convolutional neural networks. *arXiv preprint arXiv:1711.04528*, 2017.

[65] Elsken, T., Metzen, J. H., and Hutter, F. Neural architecture search: A survey. *The Journal of Machine Learning Research*, 20(1):1997–2017, 2019.

[66] Eriksson, D., Pearce, M., Gardner, J., Turner, R. D., and Poloczek, M. Scalable global optimization via local bayesian optimization. *Advances in neural information processing systems*, 32, 2019.

[67] Falkner, S., Klein, A., and Hutter, F. Bohb: Robust and efficient hyperparameter optimization at scale. In *International Conference on Machine Learning*, pp. 1437–1446. PMLR, 2018.

[68] Farcomeni, A. A review of modern multiple hypothesis testing, with particular attention to the false discovery proportion. *Statistical methods in medical research*, 17(4):347–388, 2008.

[69] Feurer, M. and Hutter, F. Hyperparameter optimization. In *Automated machine learning*, pp. 3–33. Springer, Cham, 2019.

[70] Fontana, F. A., Zanoni, M., Marino, A., and Mäntylä, M. V. Code smell detection: Towards a machine learning-based approach. In *2013 IEEE International Conference on Software Maintenance*, pp. 396–399. IEEE, 2013.

[71] Fontana, F. A., Mäntylä, M. V., Zanoni, M., and Marino, A. Comparing and experimenting machine learning techniques for code smell detection. *Empirical Software Engineering*, 21(3):1143–1191, 2016.

[72] Foret, P., Kleiner, A., Mobahi, H., and Neyshabur, B. Sharpness-aware minimization for efficiently improving generalization. *arXiv preprint arXiv:2010.01412*, 2020.

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

[74] Galke, L. and Scherp, A. Forget me not: A gentle reminder to mind the simple multilayer perceptron baseline for text classification. *arXiv preprint arXiv:2109.03777*, 2021.

[75] Ghotra, B., McIntosh, S., and Hassan, A. E. Revisiting the impact of classification techniques on the performance of defect prediction models. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pp. 789–800. IEEE, 2015.

[76] Giger, E., Pinzger, M., and Gall, H. Predicting the fix time of bugs. In *Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering*, pp. 52–56, 2010.

[77] Greg. Computing the smoothness of gda with respect to covariance matrix. Mathematics Stack Exchange, 2023. URL `https://math.stackexchange.com/q/4647791`. URL:https://math.stackexchange.com/q/4647791 (version: 2023-02-27).

[78] Grünwald, P. D. *The minimum description length principle*. MIT press, 2007.

[79] Gu, X., Zhang, H., Zhang, D., and Kim, S. Deep api learning. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 631–642. ACM, 2016.

[80] Gu, X., Zhang, H., Zhang, D., and Kim, S. Deepam: Migrate apis with multi-modal sequence to sequence learning. *arXiv preprint arXiv:1704.07734*, 2017.

[81] Guo, J., Cheng, J., and Cleland-Huang, J. Semantically enhanced software traceability using deep learning techniques. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pp. 3–14. IEEE, 2017.

[82] Guo, P. J., Zimmermann, T., Nagappan, N., and Murphy, B. Characterizing and predicting which bugs get fixed: an empirical study of microsoft windows. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pp. 495–504, 2010.

[83] Habayeb, M., Murtaza, S. S., Miranskyy, A., and Bener, A. B. On the use of hidden markov model to predict the time to fix bugs. *IEEE Transactions on Software Engineering*, 44(12):1224–1244, 2017.

[84] Habib, A. and Pradel, M. 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)*, pp. 317–328. IEEE, 2018.

[85] Hanam, Q., Tan, L., Holmes, R., and Lam, P. Finding patterns in static analysis alerts: improving actionable alert ranking. In *Proceedings of the 11th working conference on mining software repositories*, pp. 152–161, 2014.

[86] Hardt, M., Recht, B., and Singer, Y. Train faster, generalize better: Stability of stochastic gradient descent. In *International conference on machine learning*, pp. 1225–1234. PMLR, 2016.

[87] Harvey, N., Liaw, C., and Mehrabian, A. Nearly-tight vc-dimension bounds for piecewise linear neural networks. In *Conference on learning theory*, pp. 1064–1068. PMLR, 2017.

[88] He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

[89] Heckman, S. and Williams, L. 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*, pp. 41–50, 2008.

[90] Henderson-Sellers, B., Constantine, L. L., and Graham, I. M. Coupling and cohesion (towards a valid metrics suite for object-oriented analysis and design). *Object oriented systems*, 3(3):143–158, 1996.

[91] Hernández-Lobato, J. M., Hoffman, M. W., and Ghahramani, Z. Predictive entropy search for efficient global optimization of black-box functions. *Advances in neural information processing systems*, 27, 2014.

[92] Hinton, G. E. and Van Camp, D. Keeping the neural networks simple by minimizing the description length of the weights. In *Proceedings of the sixth annual conference on Computational learning theory*, pp. 5–13, 1993.

[93] Ho, T. K. and Basu, M. Complexity measures of supervised classification problems. *IEEE transactions on pattern analysis and machine intelligence*, 24(3):289–300, 2002.

[94] Hochreiter, S. and Schmidhuber, J. Simplifying neural nets by discovering flat minima. *Advances in neural information processing systems*, 7, 1994.

[95] Hochreiter, S. and Schmidhuber, J. Flat minima. *Neural computation*, 9(1):1–42, 1997.

[96] Hochreiter, S. and Schmidhuber, J. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[97] Hornik, K. Approximation capabilities of multilayer feedforward networks. *Neural networks*, 4(2):251–257, 1991.

[98] Hornik, K., Stinchcombe, M., and White, H. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

[99] Huang, Q., Xia, X., Lo, D., and Murphy, G. C. Automating intention mining. *IEEE Transactions on Software Engineering*, 2018.

[100] Huo, X., Thung, F., Li, M., Lo, D., and Shi, S.-T. Deep transfer bug localization. *IEEE Transactions on Software Engineering*, 2019.

[101] Jacot, A., Gabriel, F., and Hongler, C. Neural tangent kernel: Convergence and generalization in neural networks. *arXiv preprint arXiv:1806.07572*, 2018.

[102] Jastrzebski, S., Kenton, Z., Arpit, D., Ballas, N., Fischer, A., Bengio, Y., and Storkey, A. Three factors influencing minima in sgd. *arXiv preprint arXiv:1711.04623*, 2017.

[103] Jiang, Y., Neyshabur, B., Mobahi, H., Krishnan, D., and Bengio, S. Fantastic generalization measures and where to find them. *arXiv preprint arXiv:1912.02178*, 2019.

[104] Jin, W., Liu, T., Cai, Y., Kazman, R., Mo, R., and Zheng, Q. Service candidate identification from monolithic systems based on execution traces. *IEEE Transactions on Software Engineering*, 2019.

[105] Johnson, B., Song, Y., Murphy-Hill, E., and Bowdidge, R. Why don't software developers use static analysis tools to find bugs? In *2013 35th International Conference on Software Engineering (ICSE)*, pp. 672–681. IEEE, 2013.

[106] Kaack, L. H., Donti, P. L., Strubell, E., Kamiya, G., Creutzig, F., and Rolnick, D. Aligning artificial intelligence with climate change mitigation. *Nature Climate Change*, 12(6):518–527, 2022.

[107] Kalia, A. K., Xiao, J., Lin, C., Sinha, S., Rofrano, J., Vukovic, M., and Banerjee, D. Mono2micro: an ai-based toolchain for evolving monolithic enterprise applications to a microservice architecture. In *Proceedings of the 28th ACM Joint Meeting on ESEC/FSE*, pp. 1606–1610, 2020.

[108] Kalia, A. K., Xiao, J., Krishna, R., Sinha, S., Vukovic, M., and Banerjee, D. Mono2micro: a practical and effective tool for decomposing monolithic java applications to microservices. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 1214–1224, 2021.

[109] Kallingal Joshy, A., Chen, X., Steenhoek, B., and Le, W. Validating static warnings via testing code fragments. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 540–552, 2021.

[110] Kang, H. J., Aw, K. L., and Lo, D. Detecting false alarms from automatic static analysis tools: How far are we? *arXiv preprint arXiv:2202.05982*, 2022.

[111] Karhadkar, K., Murray, M., Tseran, H., and Montúfar, G. Mildly overparameterized relu networks have a favorable loss landscape. *arXiv preprint arXiv:2305.19510*, 2023.

[112] Keskar, N. S., Mudigere, D., Nocedal, J., Smelyanskiy, M., and Tang, P. T. P. On large-batch training for deep learning: Generalization gap and sharp minima. *arXiv preprint arXiv:1609.04836*, 2016.

[113] Kikas, R., Dumas, M., and Pfahl, D. Using dynamic and contextual features to predict issue lifetime in github projects. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, pp. 291–302, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450341868. doi: 10.1145/2901739.2901751. URL `https://doi.org/10.1145/2901739.2901751`.

[114] Kim, M., Nam, J., Yeon, J., Choi, S., and Kim, S. Remi: defect prediction for efficient api testing. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pp. 990–993, 2015.

[115] Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[116] Kipf, T. N. and Welling, M. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

[117] Komorniczak, J. and Ksieniewicz, P. problexity—an open-source python library for supervised learning problem complexity assessment. *Neurocomputing*, 521:126–136, 2023.

[118] Kumaraswamy, P. A generalized probability density function for double-bounded random processes. *Journal of hydrology*, 46(1-2):79–88, 1980.

[119] Kunstner, F., Hennig, P., and Balles, L. Limitations of the empirical fisher approximation for natural gradient descent. *Advances in neural information processing systems*, 32, 2019.

[120] Lacoste, A., Luccioni, A., Schmidt, V., and Dandres, T. Quantifying the carbon emissions of machine learning. *arXiv preprint arXiv:1910.09700*, 2019.

[121] Lee, Y., Lee, S., Lee, C.-G., Yeom, I., and Woo, H. Continual prediction of bugfix time using deep learning-based activity stream embedding. *IEEE Access*, 8: 10503–10515, 2020.

[122] Leshno, M., Lin, V. Y., Pinkus, A., and Schocken, S. Multilayer feedforward networks with a nonpolynomial activation function can approximate any function. *Neural networks*, 6(6):861–867, 1993.

[123] Levina, E. and Bickel, P. Maximum likelihood estimation of intrinsic dimension. *Advances in neural information processing systems*, 17, 2004.

[124] Li, H., Xu, Z., Taylor, G., Studer, C., and Goldstein, T. Visualizing the loss landscape of neural nets. In *Advances in Neural Information Processing Systems*, pp. 6389–6399, 2018.

[125] Li, L., Jamieson, K., DeSalvo, G., Rostamizadeh, A., and Talwalkar, A. Hyperband: A novel bandit-based approach to hyperparameter optimization. *The Journal of Machine Learning Research*, 18(1):6765–6816, 2017.

[126] Li, X., Jiang, H., Ren, Z., Li, G., and Zhang, J. Deep learning in software engineering. *arXiv preprint arXiv:1805.04825*, 2018.

[127] Liang, G., Wu, L., Wu, Q., Wang, Q., Xie, T., and Mei, H. Automatic construction of an effective training set for prioritizing static analysis warnings. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pp. 93–102, 2010.

[128] Lin, B., Zampetti, F., Bavota, G., Di Penta, M., Lanza, M., and Oliveto, R. Sentiment analysis for software engineering: How far can we go? In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pp. 94–104. IEEE, 2018.

[129] Liu, H., Simonyan, K., Vinyals, O., Fernando, C., and Kavukcuoglu, K. Hierarchical representations for efficient architecture search. *arXiv preprint arXiv:1711.00436*, 2017.

[130] Liu, H., Simonyan, K., and Yang, Y. Darts: Differentiable architecture search. *arXiv preprint arXiv:1806.09055*, 2018.

[131] Liu, H., Jin, J., Xu, Z., Bu, Y., Zou, Y., and Zhang, L. Deep learning based code smell detection. *IEEE transactions on Software Engineering*, 2019.

[132] Lorena, A. C., Garcia, L. P., Lehmann, J., Souto, M. C., and Ho, T. K. How complex is your classification problem? a survey on measuring classification complexity. *ACM Computing Surveys (CSUR)*, 52(5):1–34, 2019.

[133] Lum, P. Y., Singh, G., Lehman, A., Ishkanov, T., Vejdemo-Johansson, M., Alagappan, M., Carlsson, J., and Carlsson, G. Extracting insights from the shape of complex data using topology. *Scientific reports*, 3(1):1236, 2013.

[134] Ma, K., Liu, X., Fang, Y., and Simoncelli, E. P. Blind image quality assessment by learning from multiple annotators. In *2019 IEEE International Conference on Image Processing (ICIP)*, pp. 2344–2348. IEEE, 2019.

[135] Majumder, S., Balaji, N., Brey, K., Fu, W., and Menzies, T. 500+ times faster than deep learning. In *Proceedings of the 15th International Conference on Mining Software Repositories. ACM*, 2018.

[136] Mallik, N., Bergman, E., Hvarfner, C., Stoll, D., Janowski, M., Lindauer, M., Nardi, L., and Hutter, F. Priorband: Practical hyperparameter optimization in the age of deep learning. *arXiv preprint arXiv:2306.12370*, 2023.

[137] Mani, S., Sankaran, A., and Aralikatte, R. Deeptriage: Exploring the effectiveness of deep learning for bug triaging. In *COMAD'19: ACM India Joint International Conference on Data Science and Management of Data*, pp. 171–179, 2019.

[138] Mann, H. B. and Whitney, D. R. On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics*, pp. 50–60, 1947.

[139] Marks, L., Zou, Y., and Hassan, A. E. Studying the fix-time for bugs in large open source projects. In *Proceedings of the 7th International Conference on Predictive Models in Software Engineering*, pp. 1–8, 2011.

[140] Mazlami, G., Cito, J., and Leitner, P. Extraction of microservices from monolithic software architectures. In *2017 IEEE International Conference on Web Services (ICWS)*, pp. 524–531. IEEE, 2017.

[141] McAllester, D. A. Pac-bayesian model averaging. In *Proceedings of the twelfth annual conference on Computational learning theory*, pp. 164–170, 1999.

[142] McCabe, T. J. A complexity measure. *IEEE Transactions on software Engineering*, (4):308–320, 1976.

[143] McNicol, D. *A primer of signal detection theory.* Psychology Press, 2005.

[144] Mei, S., Zhao, C., Yuan, S., and Ni, B. Towards bridging sample complexity and model capacity. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 36, pp. 1972–1980, 2022.

[145] Menzies, T., Majumder, S., Balaji, N., Brey, K., and Fu, W. 500+ times faster than deep learning:(a case study exploring faster methods for text mining stackoverflow). In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, pp. 554–563. IEEE, 2018.

[146] Mikolov, T., Chen, K., Corrado, G., and Dean, J. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[147] Mikolov, T., Sutskever, I., Chen, K., Corrado, G. S., and Dean, J. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pp. 3111–3119, 2013.

[148] Misirli, A. T., Bener, A., and Kale, R. Ai-based software defect predictors: Applications and benefits in a case study. *AI Magazine*, 2011.

[149] Mitchell, B. S. and Mancoridis, S. On the automatic modularization of software systems using the bunch tool. *IEEE Transactions on Software Engineering*, 32(3): 193–208, 2006.

[150] Mittas, N. and Angelis, L. Ranking and clustering software cost estimation models through a multiple comparisons algorithm. *IEEE Transactions on software engineering*, 39(4):537–551, 2012.

[151] Montufar, G. F., Pascanu, R., Cho, K., and Bengio, Y. On the number of linear regions of deep neural networks. *Advances in neural information processing systems*, 27, 2014.

[152] Nair, V. and Hinton, G. E. Rectified linear units improve restricted boltzmann machines. In *Icml*, 2010.

[153] Neyshabur, B., Tomioka, R., and Srebro, N. In search of the real inductive bias: On the role of implicit regularization in deep learning. *arXiv preprint arXiv:1412.6614*, 2014.

[154] Neyshabur, B., Bhojanapalli, S., McAllester, D., and Srebro, N. Exploring generalization in deep learning. *Advances in neural information processing systems*, 30, 2017.

[155] Ng, A., Jordan, M., and Weiss, Y. On spectral clustering: Analysis and an algorithm. *Advances in neural information processing systems*, 14, 2001.

[156] Nguyen, T. D., Nguyen, A. T., Phan, H. D., and Nguyen, T. N. Exploring api embedding for api usages and applications. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pp. 438–449. IEEE, 2017.

[157] Nicolau, M., Levine, A. J., and Carlsson, G. Topology based data analysis identifies a subgroup of breast cancers with a unique mutational profile and excellent survival. *Proceedings of the National Academy of Sciences*, 108(17):7265–7270, 2011.

[158] Nielson, J. L., Paquette, J., Liu, A. W., Guandique, C. F., Tovar, C. A., Inoue, T., Irvine, K.-A., Gensel, J. C., Kloke, J., Petrossian, T. C., et al. Topological data analysis for discovery in preclinical spinal cord injury and traumatic brain injury. *Nature communications*, 6(1):8581, 2015.

[159] Ojima, Y., Horiuchi, S., and Ishikawa, F. Model-based data-complexity estimator for deep learning systems. In *2021 IEEE International Conference on Artificial Intelligence Testing (AITest)*, pp. 1–8. IEEE, 2021.

[160] Ott, J., Atchison, A., Harnack, P., Bergh, A., and Linstead, E. A deep learning approach to identifying source code in images and video. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, pp. 376–386. IEEE, 2018.

[161] Oymak, S. and Soltanolkotabi, M. Toward moderate overparameterization: Global convergence guarantees for training shallow neural networks. *IEEE Journal on Selected Areas in Information Theory*, 1(1):84–105, 2020.

[162] Palomba, F. Textual analysis for code smell detection. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pp. 769–771. IEEE, 2015.

[163] Panda, P. and Roy, K. Unsupervised regenerative learning of hierarchical features in spiking deep networks for object recognition. In *2016 International Joint Conference on Neural Networks (IJCNN)*, pp. 299–306. IEEE, 2016.

[164] Panichella, S., Arnaoudova, V., Di Penta, M., and Antoniol, G. Would static analysis tools help developers with code reviews? In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pp. 161–170. IEEE, 2015.

[165] Pecorelli, F., Di Nucci, D., De Roover, C., and De Lucia, A. A large empirical assessment of the role of data balancing in machine-learning-based code smell detection. *Journal of Systems and Software*, 169:110693, 2020.

[166] Pfisterer, F., Schneider, L., Moosbauer, J., Binder, M., and Bischl, B. Yahpo gym-an efficient multi-objective multi-fidelity benchmark for hyperparameter optimization. In *International Conference on Automated Machine Learning*, pp. 3–1. PMLR, 2022.

[167] Platt, J. et al. Probabilistic outputs for support vector machines and comparisons to regularized likelihood methods. *Advances in large margin classifiers*, 10(3):61–74, 1999.

[168] Rabadán, R. and Blumberg, A. J. *Topological data analysis for genomics and evolution: topology in biology*. Cambridge University Press, 2019.

[169] Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., Sutskever, I., et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[170] Rahman, F., Khatri, S., Barr, E. T., and Devanbu, P. Comparing static bug finders and statistical prediction. In *ICSE*. ACM, 2014.

[171] Real, E., Aggarwal, A., Huang, Y., and Le, Q. V. Regularized evolution for image classifier architecture search. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pp. 4780–4789, 2019.

[172] Redmon, J., Divvala, S., Girshick, R., and Farhadi, A. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 779–788, 2016.

[173] Rees-Jones, M., Martin, M., and Menzies, T. Better predictors for issue lifetime. *arXiv preprint arXiv:1702.07735*, 2017.

[174] Rissanen, J. A universal prior for integers and estimation by minimum description length. *The Annals of statistics*, 11(2):416–431, 1983.

[175] Rosenthal, R., Cooper, H., and Hedges, L. Parametric measures of effect size. *The handbook of research synthesis*, 621(2):231–244, 1994.

[176] Rumelhart, D. E., Hinton, G. E., and Williams, R. J. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

[177] Russell, S. and Norvig, P. Artificial intelligence: a modern approach. 2002.

[178] Ruthruff, J., Penix, J., Morgenthaler, J., Elbaum, S., and Rothermel, G. Predicting accurate and actionable static analysis warnings. In *2008 ACM/IEEE 30th International Conference on Software Engineering*, pp. 341–350. IEEE, 2008.

[179] Sadowski, C., Aftandilian, E., Eagle, A., Miller-Cushon, L., and Jaspan, C. Lessons from building static analysis tools at google. *Communications of the ACM*, 61(4): 58–66, 2018.

[180] Sahin, D., Kessentini, M., Bechikh, S., and Deb, K. Code-smell detection as a bilevel problem. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 24(1):1–44, 2014.

[181] Santurkar, S., Tsipras, D., Ilyas, A., and Madry, A. How does batch normalization help optimization? In *Advances in Neural Information Processing Systems*, pp. 2483–2493, 2018.

[182] Saxe, A. M., McClelland, J. L., and Ganguli, S. Exact solutions to the nonlinear dynamics of learning in deep linear neural networks. *arXiv preprint arXiv:1312.6120*, 2013.

[183] Sayyad Shirabad, J. and Menzies, T. The PROMISE Repository of Software Engineering Databases. School of Information Technology and Engineering, University of Ottawa, Canada, 2005. URL `http://promise.site.uottawa.ca/SERepository`.

[184] Schumacher, J., Zazworka, N., Shull, F., Seaman, C., and Shaw, M. Building empirical support for automated code smell detection. In *Proceedings of the 2010 ACM-IEEE international symposium on empirical software engineering and measurement*, pp. 1–10, 2010.

[185] Seong, S., Lee, Y., Kee, Y., Han, D., and Kim, J. Towards flatter loss surface via nonmonotonic learning rate scheduling. In *UAI*, pp. 1020–1030, 2018.

[186] Shalev-Shwartz, S. and Ben-David, S. *Understanding machine learning: From theory to algorithms.* Cambridge university press, 2014.

[187] Shawe-Taylor, J., Bartlett, P. L., Williamson, R. C., and Anthony, M. A framework for structural risk minimisation. In *Proceedings of the ninth annual conference on Computational learning theory*, pp. 68–76, 1996.

[188] Shu, R., Bui, H. H., Narui, H., and Ermon, S. A dirt-t approach to unsupervised domain adaptation. *arXiv preprint arXiv:1802.08735*, 2018.

[189] Singh, G., Mémoli, F., Carlsson, G. E., et al. Topological methods for the analysis of high dimensional data sets and 3d object recognition. *PBG@ Eurographics*, 2: 091–100, 2007.

[190] Smith, L. N. Cyclical learning rates for training neural networks. In *2017 IEEE winter conference on applications of computer vision (WACV)*, pp. 464–472. IEEE, 2017.

[191] Smith, L. N. A disciplined approach to neural network hyper-parameters: Part 1–learning rate, batch size, momentum, and weight decay. *arXiv preprint arXiv:1803.09820*, 2018.

[192] Smith, L. N. and Topin, N. Super-convergence: Very fast training of neural networks using large learning rates. In *Artificial Intelligence and Machine Learning for Multi-Domain Operations Applications*, volume 11006, pp. 1100612. International Society for Optics and Photonics, 2019.

[193] Snoek, J., Larochelle, H., and Adams, R. P. Practical bayesian optimization of machine learning algorithms. *Advances in neural information processing systems*, 25, 2012.

[194] Snoek, J., Rippel, O., Swersky, K., Kiros, R., Satish, N., Sundaram, N., Patwary, M., Prabhat, M., and Adams, R. Scalable bayesian optimization using deep neural networks. In *International conference on machine learning*, pp. 2171–2180. PMLR, 2015.

[195] Soltanolkotabi, M., Javanmard, A., and Lee, J. D. Theoretical insights into the optimization landscape of over-parameterized shallow neural networks. *IEEE Transactions on Information Theory*, 65(2):742–769, 2018.

[196] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

[197] Stanley, K. O. and Miikkulainen, R. Evolving neural networks through augmenting topologies. *Evolutionary computation*, 10(2):99–127, 2002.

[198] Suk, H.-I., Lee, S.-W., Shen, D., Initiative, A. D. N., et al. Hierarchical feature representation and multimodal fusion with deep learning for ad/mci diagnosis. *NeuroImage*, 101:569–582, 2014.

[199] Sun, S.-H., Noh, H., Somasundaram, S., and Lim, J. Neural program synthesis from diverse demonstration videos. In *International Conference on Machine Learning*, pp. 4790–4799, 2018.

[200] Sutskever, I., Martens, J., Dahl, G., and Hinton, G. On the importance of initialization and momentum in deep learning. In *International conference on machine learning*, pp. 1139–1147. PMLR, 2013.

[201] Swersky, K., Snoek, J., and Adams, R. P. Freeze-thaw bayesian optimization. *arXiv preprint arXiv:1406.3896*, 2014.

[202] Tang, M.-H., Kao, M.-H., and Chen, M.-H. An empirical study on object-oriented metrics. In *Proceedings sixth international software metrics symposium (Cat. No. PR00403)*, pp. 242–249. IEEE, 1999.

[203] Tantithamthavorn, C., McIntosh, S., Hassan, A. E., and Matsumoto, K. Automated parameter optimization of classification techniques for defect prediction models. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pp. 321–332, 2016. doi: 10.1145/2884781.2884857.

[204] Tempero, E., Anslow, C., Dietrich, J., Han, T., Li, J., Lumpe, M., Melton, H., and Noble, J. The qualitas corpus: A curated collection of java code for empirical studies. In *2010 Asia Pacific Software Engineering Conference*, pp. 336–345. IEEE, 2010.

[205] Thung, F., Lo, D., Jiang, L., Rahman, F., Devanbu, P. T., 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*, pp. 50–59. IEEE, 2012.

[206] Tomassi, D. A. and Rubio-González, C. 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)*, pp. 292–303. IEEE, 2021.

[207] Tran, N., Schneider, J.-G., Weber, I., and Qin, A. K. Hyper-parameter optimization in classification: To-do or not-to-do. *Pattern Recognition*, 103:107245, 2020.

[208] user1551 (https://math.stackexchange.com/users/1551/user1551). L2 norm of pseudo-inverse relation with minimum singular value. Mathematics Stack Exchange, 2020. URL `https://math.stackexchange.com/q/3549936`. URL:https://math.stackexchange.com/q/3549936 (version: 2020-02-17).

[209] Vassallo, C., Panichella, S., Palomba, F., Proksch, S., Gall, H. C., and Zaidman, A. How developers engage with static analysis tools in different contexts. *Empirical Software Engineering*, 25(2):1419–1457, 2020.

[210] Verma, V., Kawaguchi, K., Lamb, A., Kannala, J., Bengio, Y., and Lopez-Paz, D. Interpolation consistency training for semi-supervised learning. *arXiv preprint arXiv:1903.03825*, 2019.

[211] Vieira, R., da Silva, A., Rocha, L., and Gomes, J. P. From reports to bug-fix commits: A 10 years dataset of bug-fixing activity from 55 apache's open source projects. In *Proceedings of the Fifteenth International Conference on Predictive Models and Data Analytics in Software Engineering*, pp. 80–89, 2019.

[212] Vlassopoulos, G., van Erven, T., Brighton, H., and Menkovski, V. Explaining predictions by approximating the local decision boundary. *arXiv preprint arXiv:2006.07985*, 2020.

[213] Wan, Z., Xia, X., Hassan, A. E., Lo, D., Yin, J., and Yang, X. Perceptions, expectations, and challenges in defect prediction. *IEEE Transactions on Software Engineering*, 2018.

[214] Wang, J., Wang, S., and Wang, Q. 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*, pp. 1–10, 2018.

[215] Wang, S., Li, D., Song, X., Wei, Y., and Li, H. A feature selection method based on improved fisher's discriminant ratio for text sentiment classification. *Expert Systems with Applications*, 38(7):8696–8702, 2011.

[216] Wang, S., Liu, T., and Tan, L. Automatically learning semantic features for defect prediction. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pp. 297–308. IEEE, 2016.

[217] Wang, S., Liu, T., Nam, J., and Tan, L. Deep semantic feature learning for software defect prediction. *IEEE Transactions on Software Engineering*, 46(12):1267–1293, 2018.

[218] Watson, C. A. *Deep Learning in Software Engineering.* PhD thesis, College of William & Mary, 2020.

[219] Weiss, C., Premraj, R., Zimmermann, T., and Zeller, A. How long will it take to fix this bug? In *Fourth International Workshop on Mining Software Repositories (MSR'07: ICSE Workshops 2007)*, pp. 1–1. IEEE, 2007.

[220] Welling, M. and Teh, Y. W. Bayesian learning via stochastic gradient langevin dynamics. In *Proceedings of the 28th international conference on machine learning (ICML-11)*, pp. 681–688, 2011.

[221] Wen, K., Li, Z., and Ma, T. Sharpness minimization algorithms do not only minimize sharpness to achieve better generalization. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023.

[222] White, C., Safari, M., Sukthanker, R., Ru, B., Elsken, T., Zela, A., Dey, D., and Hutter, F. Neural architecture search: Insights from 1000 papers. *arXiv preprint arXiv:2301.08727*, 2023.

[223] White, M., Tufano, M., Vendome, C., and Poshyvanyk, D. Deep learning code fragments for code clone detection. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pp. 87–98. ACM, 2016.

[224] Wigner, E. P. On the distribution of the roots of certain symmetric matrices. *Annals of Mathematics*, 67(2):325–327, 1958.

[225] Wohlin, C. Guidelines for snowballing in systematic literature studies and a replication in software engineering. In *Proceedings of the 18th international conference on evaluation and assessment in software engineering*, pp. 1–10, 2014.

[226] Wojke, N. and Bewley, A. Deep cosine metric learning for person re-identification. In *2018 IEEE winter conference on applications of computer vision (WACV)*, pp. 748–756. IEEE, 2018.

[227] Wolff, E. *Microservices: flexible software architecture*. Addison-Wesley Professional, 2016.

[228] Wu, L. and Su, W. J. The implicit regularization of dynamical stability in stochastic gradient descent. In *International Conference on Machine Learning*, pp. 37656–37684. PMLR, 2023.

[229] Xie, S., Zheng, H., Liu, C., and Lin, L. Snas: stochastic neural architecture search. *arXiv preprint arXiv:1812.09926*, 2018.

[230] Yamashita, A. and Moonen, L. To what extent can maintenance problems be predicted by code smell detection?–an empirical study. *Information and Software Technology*, 55(12):2223–2242, 2013.

[231] Yamashita, R., Nishio, M., Do, R. K. G., and Togashi, K. Convolutional neural networks: an overview and application in radiology. *Insights into imaging*, 9(4): 611–629, 2018.

[232] Yang, X., Chen, J., Yedida, R., Yu, Z., and Menzies, T. Learning to recognize actionable static code warnings (is intrinsically easy). *Empirical Software Engineering*, 26(3):1–24, 2021.

[233] Yang, X., Yu, Z., Wang, J., and Menzies, T. Understanding static code warnings: An incremental ai approach. *Expert Systems with Applications*, 167:114134, 2021.

[234] Yedida, R. and Menzies, T. On the value of oversampling for deep learning in software defect prediction. *IEEE Transactions on Software Engineering*, 2021.

[235] Yedida, R. and Menzies, T. How to improve deep learning for software analytics: (a case study with code smell detection). In *Proceedings of the 19th International Conference on Mining Software Repositories*, pp. 156–166, 2022.

[236] Yedida, R. and Menzies, T. Smoothie: A theory of hyper-parameter optimization for software analytics. *arXiv preprint arXiv:2401.09622*, 2024.

[237] Yedida, R. and Saha, S. Beginning with machine learning: a comprehensive primer. *The European Physical Journal Special Topics*, pp. 1–82, 2021.

[238] Yedida, R. and Saha, S. Strong convexity-guided hyper-parameter optimization for flatter losses. *arXiv preprint arXiv:2402.05025*, 2024.

[239] Yedida, R., Krishna, R., Kalia, A., Menzies, T., Xiao, J., and Vukovic, M. Lessons learned from hyper-parameter tuning for microservice candidate identification. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 1141–1145. IEEE, 2021.

[240] Yedida, R., Saha, S., and Prashanth, T. Lipschitzlr: Using theoretically computed adaptive learning rates for fast convergence. *Applied Intelligence*, 51(3):1460–1478, 2021.

[241] Yedida, R., Yang, X., and Menzies, T. Old but gold: Reconsidering the value of feedforward learners for software analytics. *arXiv preprint arXiv:2101.06319*, 2021.

[242] Yedida, R., Kang, H. J., Tu, H., Yang, X., Lo, D., and Menzies, T. How to find actionable static analysis warnings: A case study with findbugs. *IEEE Transactions on Software Engineering*, 2023.

[243] Yedida, R., Krishna, R., Kalia, A., Menzies, T., Xiao, J., and Vukovic, M. An expert system for redesigning software for cloud applications. *Expert Systems with Applications*, 219:119673, 2023.

[244] Yeo, I.-K. and Johnson, R. A. A new family of power transformations to improve normality or symmetry. *Biometrika*, 87(4):954–959, 2000.

[245] Zampetti, F., Scalabrino, S., Oliveto, R., Canfora, G., and Di Penta, M. 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)*, pp. 334–344. IEEE, 2017.

[246] Zeiler, M. D. and Fergus, R. Visualizing and understanding convolutional networks. In *European conference on computer vision*, pp. 818–833. Springer, 2014.

[247] Zeiler, M. D., Taylor, G. W., and Fergus, R. Adaptive deconvolutional networks for mid and high level feature learning. In *2011 International Conference on Computer Vision*, pp. 2018–2025. IEEE, 2011.

[248] Zhang, C., Bengio, S., Hardt, M., Recht, B., and Vinyals, O. Understanding deep learning requires rethinking generalization (2016). *arXiv preprint arXiv:1611.03530*, 2017.

[249] Zhang, C., Bengio, S., Hardt, M., Recht, B., and Vinyals, O. Understanding deep learning (still) requires rethinking generalization. *Communications of the ACM*, 64 (3):107–115, 2021.

[250] Zhang, H., Cisse, M., Dauphin, Y. N., and Lopez-Paz, D. mixup: Beyond empirical risk minimization. *arXiv preprint arXiv:1710.09412*, 2017.

[251] Zhao, G. and Huang, J. Deepsim: deep learning code functional similarity. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 141–151. ACM, 2018.

[252] Zhou, C., Li, B., Sun, X., and Bo, L. Why and what happened? aiding bug comprehension with automated category and causal link identification. *Empirical Software Engineering*, 26(6):1–36, 2021.

[253] Zoph, B., Vasudevan, V., Shlens, J., and Le, Q. V. Learning transferable architectures for scalable image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 8697–8710, 2018.

[254] Zou, D., Cao, Y., Zhou, D., and Gu, Q. Gradient descent optimizes over-parameterized deep relu networks. *Machine Learning*, 109(3):467–492, 2020.