ABSTRACT

AGRAWAL, AMRITANSHU. On the Nature of Software Engineering Data (Implications of ϵ-Dominance in Software Engineering). (Under the direction of Dr Timothy Menzies).

Software Analytic has become increasingly important to identify better optimal solutions. One of the "black arts" of analytics is how to tune the many parameters that control the inference of the data mining algorithms. Hyperparameter optimizers (HPO) can automatically find useful parameter settings, but they can be very CPU expensive. Worse yet, recent results show that selecting which HPO to apply is itself a black art, raining the spectre of hyperparameter optimizers needing hyper-hyperparameter optimizers (and so, in a regress of increasing computational complexity).

This thesis asks, is there a way to avoid hyper-hyperparameter optimization? We speculate that any HPO is one point within the space of decision options on how to configure a learner. If there was a way to divide that decision space into "chunks", then avoid sampling any chunk more than once, would we perform hyperparameter optimization as effective as other approaches? In past, software engineering (SE) data has shown to have "chunks" where redundant options can be found, but there hasn't been any method proposed to sample these "chunks". It can also be said that these "chunks" can only be found in SE field, whether it holds true in other fields needs to be validated as well.

To test this hypothesis, we propose **DODGE(ϵ)**, a very fast self-tuning optimization tool. The objective space of the learners (two-dimensional grids of, say, recall vs false alarms) can be divided into cells ("chunks") of size ϵ where 1 decision option is harder to distinguish result from another decision option. **DODGE(ϵ)** (a) negatively weigh regions of the decision space that fall within ϵ of old objective space; and (b) selects next samples from regions with least negative weights.

When tested on a variety of case studies from software analytics, **DODGE(ϵ)** generates optimizations better than the state of the art in HPO in software analytics tasks such as 1) Defect prediction; 2) SE text mining classification; 3) Bad Code Smell; and 4)Issue Close Time. More importantly, it does so after very few samples (fewer than **50**) than other state-of-the-art methods.

Performance was scored using $p = 2$ metrics: 1) for all 4 tasks: *recall* vs *false alarms* trade-offs; 2) for defect prediction: how *few* modules must be inspected to find *most* bugs. In the comparison, **DODGE(.2)** produced best performance scores. Further, **DODGE(.2)** terminated after far fewer evaluations than standard HPO. We conjecture this is because **DODGE(.2)** explores a very small space of $1/\epsilon^p$ cells while standard optimizers struggle to cover billions of decision options (most of which yield indistinguishable results).

Also, when **DODGE(ϵ)** was applied in other domains (mainly the UCI datasets), we did not see the same pattern as observed in SE. We conclude that, the data of SE is different in nature, and it contains "chunks" where redundant options lead to same indistinguishable results. These "chunks" can be utilized to build an optimizer for a Software Analytics.

On the Nature of Software Engineering Data (Implications of ϵ-Dominance in Software Engineering)

by
Amritanshu Agrawal

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

2019

APPROVED BY:

Dr. Matthias Stallmann	Dr. Min Chi
Dr. Jamie Jennings	Dr Timothy Menzies Chair of Advisory Committee

BIOGRAPHY

Amritanshu Agrawal was born in Patna, Bihar, India. He grew up there and completed his high school from Kota, Rajasthan, India.

He attended PES University from 2011 to 2015 and graduated with a B.E in Computer Science Engineering. While at PES, he performed many research projects which resulted in a research publication. He also carried out research in Computer Vision at Durham University, UK under the supervision of Dr Toby Breckon.

Based on his past research experiences, he entered into Ph.D. program at North Carolina State University in 2015. His advisor was Dr Tim Menzies. He became interested in Machine learning and Artificial Intelligence for Software Engineering. Since then, he has authored and published many research articles at top conferences and journals. He will be finishing his Ph.D. in 2019 at a record time of 4 years after coming directly from his bachelors degree.

ACKNOWLEDGEMENTS

Firstly, I would like to express my sincere gratitude to my advisor Dr Tim Menzies for the continuous support of my Ph.D study and related research, for his patience, motivation, and immense knowledge. His guidance helped me in all the time of research and writing of this thesis. I could not have imagined having a better advisor and mentor for my Ph.D study.

Besides my advisor, I would like to thank the rest of my thesis committee: Dr Matthias Stallmann, Dr Min Chi, and Dr. Jamie Jennings, for their insightful comments and encouragement, but also for the hard question which incented me to widen my research from various perspectives.

I thank my fellow labmates in for the stimulating discussions, for the sleepless nights we were working together before deadlines, and for all the fun we have had in the last four years.

Last but not the least, I would like to thank my family for supporting me throughout this Ph.D journey and my life in general.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER

1

INTRODUCTION

This thesis argues on the nature of Software engineering data and how is it different than other domains. We believe *Software analytics can be inherently simple* which means that much prior research [Fu16a; Men07a; AM18] has *needlessly complicated an inherently simple task*. Here by "simple" we mean that very little CPU is required to build software quality prediction models. We show that this is true for four SE tasks:

- Software defect prediction; i.e. the classification of software modules into "buggy" or otherwise based on static code attributes [Fu16a; Tan16; Men07a; AM18];
- Software bug report text mining; i.e. text mining algorithms that rank or predict issue reports severity [Agr18b; Oli10a];
- Detecting bad code smells; i.e., the prediction of defect modules due to the presence of bad code smells [KM18];
- Issue lifetime estimation; i.e., predicting the time it takes to close an issue [Che18; KM18].

Our success in simplifying these tasks is most surprising. Recent trends in software analytics favors very CPU-intensive *hyperparameter optimization (HPO)* methods to automatically tune control options for data mining. Assuming that one parameter options takes a float value between a range of 0 to 1, then it holds over a billion tuning options. With enough CPU, automatic HPOs can prune those options to find tunings that improve the performance of software quality predictors [AM18;

Fu16a; Agr18b; TW18; Liu10; Sar12; Tan16; Zho04; Oli10a]. The problem with HPO is that tuning requires high number of evaluations to test hundreds to millions of different tuning options and the cost of running a data miner through all those options is very high, requiring days to weeks to decades of CPU [Wan13; TW18].

For many years, we have addressed these long CPU times via cloud-based CPU farms. For example, many of the experiments conducted in software analytics can be easily parallelized just by running (e.g.) multiple sub-samples of the data on separate cores. But that process can be very expensive. Fisher et al. [Fis12] comment that cloud computation is a heavily monetized environment that charges for all their services (storage, uploads, downloads, and CPU time). While each small part of that service is cheap, the total annual cost to an organization can be exorbitant.

Though, recently we developed faster HPO methods like SMOTUNED for defect prediction task and LDADE for SE text mining task which took 50 and 30 evaluations respectively to achieve better performance than other HPOs for similar tasks. More recently we developed FFtrees [Phi17; Che18; Fu18], and surprisingly, FFtrees (a) save most of CPU cost while at the same time (b) find better tunings. FFtrees out-perform many of state-of-the-art SE HPOs, and our own developed LDADE and SMOTUNED, for different SE tasks understudy. This was a very strange observation since tuning sampled hundreds to thousands of options, while FFtrees explored only 16.

These few examples motivated us further to conjecture that

1. Something might be grouping the output space into "chunks" such that
2. looking at a few samples is just as effective as looking at many more.

To explain what causes that grouping, we note that analytics is a probabilistic process where different treatments produce different distributions of results. When comparing distributions to show that, say, tuning1 is better than tuning2, there is always some delta ϵ in the results below which it is hard to distinguish different treatments. As shown in Figure 1.1, ϵ divides the output space of a learner into $1/\epsilon^p$ cells, where p is the number of performance scores being monitored. For $p=2$, and $\epsilon=0.2$, that output space divides into just 25 cells (where green is preferred over red).

To exploit these groupings, we propose **DODGE(ϵ)**, a very fast self-tuning analytics tool. **DODGE(ϵ)** executes randomly selected tuning options. When new results fall within ϵ of old results, from then onwards, **DODGE(ϵ)** will avoid them. It is to be seen how much number of evaluations (N) will be needed by **DODGE(ϵ)** to find no interesting options. To test **DODGE(ϵ)**, we compared it against state-of-the-art results for the SE tasks mentioned earlier [Gho15; Fu16a; AM18; Agr18b; KM18]. Also, multi-goal optimizations such as 1) achieving better recall with minimal cost; 2) achieving better area under the curve of recall and precision, with reduced runtime; 3) reducing the cost with maximum customer satisfaction, are much more complex task than single goal optimization such as recall, accuracy, precision, and many more. We scored **DODGE(ϵ)** on $p=2$ goals:

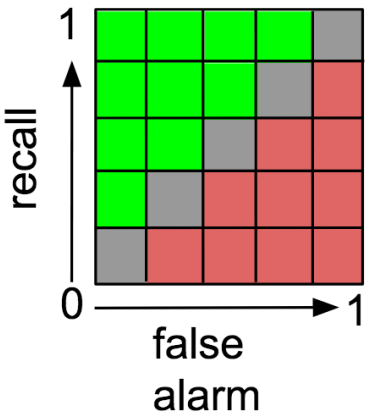

Figure 1.1 Grids in results space.

- For all four SE tasks: *recall* vs *false alarms* trade-offs.
- For defect prediction: how *few* modules must be inspected to find *most* bugs.

To certify the value of this approach, this thesis explores 3 research questions:

- **RQ1**: Is **DODGE(ϵ)** too complicated? Is it difficult to determine its $\{N, \epsilon\}$ values?

This thesis propose simplifications to software analytics (where lesser evaluations define simplicity), it is important to check if the new proposed method is itself simple to apply. Accordingly, this research question asks if it is difficult to find useful values for N (the number of evaluations) or the ϵ value ("chunk" size) used in the search.

- **RQ2**: How does **DODGE(ϵ)** compare to recent prominent defect prediction and hyperparamter optimization results?

Defect prediction is studied extensively among SE researchers using many complex methods, it is important to validate how **DODGE(ϵ)** performs to those methods.

- **RQ3**: How does **DODGE(ϵ)** compare to recent prominent text mining and hyperparamter optimization results?

To show how prominent **DODGE(ϵ)** is for the SE tasks, we explored text mining as well. Text mining has become another interesting direction for researchers due to the complexity of the data in use.

- **RQ4**: How does **DODGE(ϵ)** compare to recent prominent bad code smell detection and hyperparamter optimization results?

DODGE(ϵ) was also applied to detect defective modules due to the presence of bad code smells.

- **RQ5**: How does **DODGE(ϵ)** compare to recent prominent issue lifetime prediction and hyper-paramter optimization results?

We also used **DODGE(ϵ)** to predict different issues lifetime (1 day, 7 days, 14 days, 30 days, 90 days, 180 days, 365 days).

- **RQ6**: What inference can be drawn about SE data compared against other domain data (UCI datasets) after applying **DODGE(ϵ)**?

We also applied **DODGE(ϵ)** to non-SE domain where the data came from popular UCI repository [AN07; DG17]. We wanted to see whether our hypothesis of objective space being grouped into "chunks" hold true for non-SE domain or not. If not, then how is the data different between the SE and non-SE domain?

We found that $\epsilon = 0.2$ which comes under large ϵ variabilities of SE data, making SE data more peculiar and different. **DODGE(.2)** produced best performance scores than FFtrees, SMOTUNED, LDADE, FLASH [Nai18] and other state-of-the-art HPOs. Further, **DODGE(.2)** terminated after far fewer evaluations than standard optimizers. Figure 1.2 represents number of evaluations taken by different frameworks and HPOs for SE tasks. **DODGE(ϵ)** explores billions of choices through preprocessors, and machine learners, while performing the best in only 30 evaluations. FFtree builds 16 trees but only explores few ranges. SMOTUNED takes about 50 evaluations to tune only few choices of SMOTE. On another hand, LDADE and LDA-GA takes 300 and 1000 to 10,000 evaluations respectively, to tune only the choices of LDA and performing worse than **DODGE(ϵ)**. **DODGE(ϵ)** explore billions of choices, the most among other frameworks, while performing the best with just 30 evaluations. We conjecture this is because **DODGE(.2)** explores a very small space of $1/\epsilon^p$ cells while standard optimizers struggle to cover billions of tuning options (most of which yield indistinguishably different results).

This thesis talks about first the motivation which led us to believe there exists ϵ-dominance in goal space where we worked with complex methods like LDADE for text mining [Agr18b] (our own developed framework), SMOTUNED (our own developed framework) and FLASH for defect prediction [AM18; Nai18] to show we can achieve great performance with only 100s of evaluations not 1,000s or 10,000s. We then further explored FFT (our own developed framework), a simpler method which takes only 10s of trees to achieve similar performances in all 4 tasks understudy.

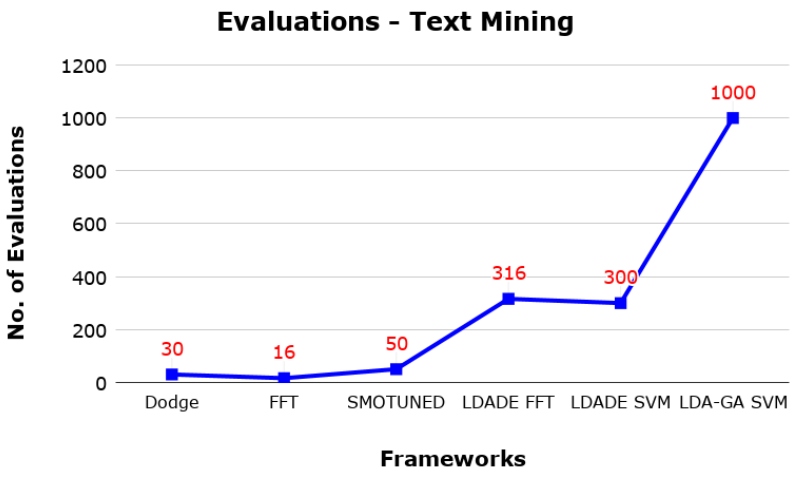

Figure 1.2 Evaluations taken by different Frameworks for Text Mining

The rest of this thesis discusses the why and how of software analytics simplification and the existence of ϵ-domination in SE data giving the simplified nature of SE data. When we say SE data is different in ***nature***, we say that there exists ϵ or have "chunks" where redundant options lead to indistinguisable results. It was also argued that does ϵ size "chunks" exists in non-SE domain as well or not. We applied **DODGE(.2)** for UCI datasets which are all non-SE domain and found that we didn't achieve better performance than the tradition machine learning methods. We tried looking at the feature distribution for both SE and non-SE data, and found that **DODGE(ϵ)** works really well when data is 1) highly imbalanced, and 2) if discretization of features works. Finally, it can be concluded that SE data contains ϵ and not the non-SE data. Also, **DODGE(ϵ)** can be applied with highly imbalanced datasets and good discretization methods like Fayyad & Irani's [FI93].

The next section provide background, and argues that such simplifications is useful. Next, we discuss the design and methodology of some of our faster developed HPOs like LDADE and SMOTUNED, FFtree, FLASH and then the framework of **DODGE(ϵ)**. All these frameworks provide better improvement and at the same time reduction in number of evaluations as we study them. After that, we reports experiments that compare **DODGE(ϵ)** to prior state-of-the-art results. Our conclusion will be that **DODGE(ϵ)** is much better, at reducing the number of evaluations required to find good tunings for software analytics.

1.1 Contribution

Here we showed **DODGE(ϵ)** is a faster optimizer, with the existence of ϵ-domination which was not found in non-SE domain. With **DODGE(ϵ)**, we found that SE data is different in nature which

commonly contains "chunks" of size ϵ. Also, **DODGE(ϵ)** defects our all the recently newly created frameworks (our past 3 years of work), namely:

- SMOTUNED [AM18] which was faster and better than all the previous state-of-the-art methods in defect prediction before **DODGE(ϵ)**.
- LDADE [Agr18b] which was faster and better than all the previous state-of-the-art methods in text mining before **DODGE(ϵ)**.
- FFtree [Agr18a; Che18] which was faster and better than all the previous state-of-the-art methods in both defect prediction and text mining before **DODGE(ϵ)**.
- Explored 2 more tasks such as Bad code smell detection, and issue lifetime estimation to compare **DODGE(ϵ)** against previous state-of-the-art methods.

But we would also like to propose a new characterization of software analytics:

> *Software analytics is that branch of machine learning that studies problems with large amounts of variability ϵ in their nature.*

(For reasons why SE has such large ϵ variability , see 2.3.)

This new characterization is a significant contribution since it means that every new machine learning algorithm developed in the AI community might not apply to SE as proved by applying **DODGE(ϵ)** on non-SE data. Perhaps understanding SE is a different problem to understanding other problems that are more precisely controlled and restrained. Also, perhaps, it is time to design new machine learning algorithms (like **DODGE(ϵ)**) that are better suited to the large ϵ variabilities of SE data. As shown in this thesis, such new algorithms can exploit the peculiarities of SE data to dramatically simplify software analytics.

At last, we have open sourced all the data and code scripts for other students/researchers to refute and reproduce results. Our work is distributed across multiple repositories which are:

- https://github.com/ai-se/e-dom contains our code and data which was used for **DODGE(ϵ)** framework.
- https://github.com/ai-se/Smote_tune contains our code and data which was used for SMOTUNED framework.
- https://github.com/amritbhanu/LDADE-package contains only our reproduction package for LDADE.
- https://github.com/ai-se/FFT_Jack contains our code and data which was used for FFtree framework.

1.2 External Validity and Future work

This thesis's results are based on an analysis of 4 SE tasks and many non-SE tasks which begs the question: how many other domains can be simplified in a similar manner?.

Though it will be needed to explore more tasks with different problems like regression before the methods of this paper can be broadly applied. This would be a fruitful direction for future work.

1.3 Statement of Thesis

Software Engineering is much **simpler** when measured in terms of CPU, than other Machine Learning tasks since SE data often conforms to ϵ**-domination** which can be exploited to produce better learners. Due to the existence of ϵ**-domination**, we can avoid spending time, effort and resources on trying billions of ways to tune learners to produce optimal performance.

1.4 Research Articles

1.4.1 Accepted Papers from this thesis

- **Amritanshu Agrawal**, Wei Fu, Di Chen, Xipeng Shen and Tim Menzies. "Can we **DODGE(ϵ)** Complex Software Analytics?". IEEE Transactions on Software Engineering (2019).

- **Amritanshu Agrawal**, Wei Fu, and Tim Menzies. "What is wrong with topic modeling? And how to fix it using search-based software engineering." Information and Software Technology (2018). It contributed to the case study on LDADE for text mining as a motivation for this thesis.

- **Amritanshu Agrawal**, and Tim Menzies. "Is better data better than better data miners?: on the benefits of tuning SMOTE for defect prediction." International Conference on Software Engineering (2018). It contributed to the case study on SMOTUNED for Defect Prediction as a motivation for this thesis.

- George Mathew, **Amritanshu Agrawal**, and Tim Menzies. "Finding Trends in Software Research." IEEE Transactions on Software Engineering (2018). It showed LDADE to improve topic stability in bibliometric SE text mining study.

- Vivek Nair, **Amritanshu Agrawal**, Jianfeng Chen, Wei Fu, George Mathew, Tim Menzies, Leandro Minku, Markus Wagner, and Zhe Yu. "Data-Driven Search-based Software Engineering". Mining Software Repository (2018). This paper talks about the impact of data driven faster LDADE and SMOTUNED methods.

- George Mathew, **Amritanshu Agrawal**, and Tim Menzies. "Trends in topics at SE conferences (1993-2013)" ICSE-C (2016). It showed LDADE to improve topic stability in bibliometric SE text mining study.

1.4.2 Other PhD papers not part of this thesis

- **Amritanshu Agrawal**, Akond Rahman, Rahul Krishna, Alexander Sobran, and Tim Menzies. "We don't need another hero?: the impact of heroes on software development." International Conference on Software Engineering: Software Engineering in Practice (2018).
- Rahul Krishna, **Amritanshu Agrawal**, Akond Rahman, Alexander Sobran, and Tim Menzies. "What is the connection between issues, bugs, and enhancements?: lessons learned from 800+ software projects." International Conference on Software Engineering: Software Engineering in Practice (2018).
- Akond Rahman, **Amritanshu Agrawal**, Rahul Krishna, and Alexander Sobran. "Characterizing The Influence of Continuous Integration: Empirical Results from 250+ Open Source and Proprietary Projects." Swan Workshop, FSE 2018
- Rahul Krishna, Zhe Yu, **Amritanshu Agrawal**, et al. "The 'BigSE' Project: Lessons Learned from Validating Industrial Text Mining" BIGDSE Workshop, ICSE 2016.
- Suvodeep Majumder, Joymallya Chakraborty, **Amritanshu Agrawal**, and Tim Menzies. "Why Software Projects need Heroes (Lessons Learned from 1100+ Projects)" Under Review, IEEE Transactions on Software Engineering 2019.

1.4.3 Papers Under Review part of this thesis

- **Amritanshu Agrawal** and Tim Menzies. "**DODGE(ϵ)** works for Bad Code Smell and Issue Lifetime Prediction".
- **Amritanshu Agrawal**, Huy Tu and Tim Menzies. "Can You Explain That, Better? Comprehensible Text Analytics for SE Applications".

1.5 Structure of this Thesis

The rest of this report is structured as follows:

- Chapter 2 gives a brief background about optimizers and different SE tasks studied in this thesis. It also provides the motivation for the existence of ϵ.
- Chapter 3 introduces the methodology and experimentation of this thesis.

- Chapter 4 talks about results and answers to 6 research questions.
- Chapter 5 discusses about threats to validity of this study.
- Conclusion and future work is discussed in Chapter 6.

CHAPTER

2

BACKGROUND AND MOTIVATION

In this chapter we will give a brief background about hyperparameter tuning and literature review on different optimizers and what optimizers did we select for our study and why. It will be followed up with the discussion on defect prediction, text mining, bad code smell detection and issue lifetime estimation SE tasks, and why did we study these SE tasks, its related past work. We will also talk about why we chose UCI datasets as the source for non-SE domain. We will explain how did we come up with our hypothesis of large ϵ or existence of ϵ-domination in the nature of SE data.

2.1 Hyperparameter Tuning

Hyperparameter tuning is a subset to an optimization problem. Data mining is a problem that involves finding an approximation $\hat{h}(\mathbf{x})$ of a function of the following format:

$$\mathbf{y} = h(\mathbf{x}) \tag{2.1}$$

where $\mathbf{x} = (x_1, \cdots, x_p) \in \mathbf{X}$ are the input variables, $\mathbf{y} = (y_1, \cdots, y_q) \in \mathbf{Y}$ are the output variables of the function $h(\mathbf{x}) : \mathbf{X} \rightarrow \mathbf{Y}$, $\mathbf{X}$ is the input space and $\mathbf{Y}$ is the output space. The input variables $\mathbf{x}$ are frequently referred to as the independent variables or input features, whereas $\mathbf{y}$ are referred to as the dependent variables or output features.

An example of a data mining problem in software engineering is software defect prediction [Hal12]. Here, the input features could be a software component's size and complexity, and the output feature

could be a label identifying the component as defective or non-defective. Many different machine learning algorithms can be used for data mining.

The functions $h(\mathbf{x})$ and $\hat{h}(\mathbf{x})$ may *not* necessarily correspond to the optimization functions. However, the true function $h(\mathbf{x})$ is unknown. Therefore, data mining frequently relies on machine learning algorithms to learn an approximation $\hat{h}(\mathbf{x})$ based on a set $D = \{(x_i, y_i)\}_{i=1}^{|D|}$ of known examples (data points) from $h(\mathbf{x})$. And, learning this approximation typically consists of searching for a function $\hat{h}(\mathbf{x})$ that minimizes the error (or other predictive performance metrics) on examples from D. These machine learning algorithms also come up with their own parameters which can give rise to different approximations while learning about the data. Hyperparameter tuning is an art of tuning the parameters that control the choices within a data miner. They also try to minimizes or maximizes the performance metric. In our case, we used a multi-goal optimization problem where we either minimized or maximized the goals for our 4 SE tasks understudy.

The impact of hyperparameter tuning is well understood in the theoretical machine learning literature [BB12]. When we tune a data miner, what we are really doing is changing how a learner applies its heuristics. This means tuned data miners use different heuristics, which means they ignore different possible models, which means they return different models, i.e., *how* we learn changes *what* we learn.

Tuning has been addressed in many software analytics literature. Fu et al. [Fu16a] surveyed hundreds of recent SE papers in the area of software defect prediction from static code attributes. Tantithamthavorn et al. [Tan16] explored tunings for many learners. In general, the importance of tuning is extensively mentioned.

Bergstra et al. [BB12] commented that *grid search*[1] is very popular since (a) such a simple search to gives researchers some degree of insight; (b) grid search has very little technical overhead for its implementation; (c) it is simple to automate and parallelize; (d) on a computing cluster, it can find better tunings than sequential optimization (in the same amount of time). That said, grid search is not efficient due to extensive options that it explores.

The literature mentions many faster optimizers than grid search like simulated annealing [FM02; Men07c]; various genetic algorithms [Gol79] augmented by techniques such as DE (differential evolution [SP97]), tabu search and scatter search [GM86; Bea06; Mol07; Neb08]; particle swarm optimization [Pan08]; numerous decomposition approaches that use heuristics to decompose the total space into small problems, then apply a response surface methods [Kra15; Zul13]. Table 2.1 shows the optimizers that were used in this thesis.

2.1.1 Why Explore Faster Software Analytics?

This section argues that avoiding slow methods for software analytics is an open and urgent issue.

[1] For N tunable option, run N nested for-loops to explore their ranges.

Table 2.1 Notes on different optimizers used in our case study

Genetic Algorithms (GAs) execute over multiple generations. Generation zero is usually initialized at random. After that, in each generation, candidate items are subject to *select* (prune away the less interesting solutions), *crossover* (build new items by combining parts of selected items), and *mutate* (randomly perturb part of the the new solutions). Modern GAs take different approaches to the *select* operator (see https://raw.githubusercontent.com/txt/ase16/master/img/rankvscountvsdepth.png). Notable exceptions are MOEA/D that use a decomposition operator to divide all the solutions into many small neighborhoods where if anyone finds a better solution, all its neighbors move there as well [AC05; HC06; CH07; Sar16; Oli10a; Pan13; Du15; MY13; Sar12].
Different evolution (DE) execute over multiple generations. Generation zero is usually initialized at random. After that, in each generation, candidate items are subject to *select* (prune away the less interesting solutions), *mutate* (build new items by combining with 3 other random candidates from the same generation) [SP97; Fu16a; AM18; Agr18b; FM17; Men18].
FLASH, a sequential model-based method such as Bayesian optimization, is a useful strategy to find extremes of an unknown objective. FLASH is efficient because of its ability to incorporate prior belief as already measured solutions (or configurations), to help direct further sampling. Here, the prior represents the already known areas of the search (or performance optimization) problem. The prior can be used to estimate the rest of the points (or unevaluated configurations). Once one (or many) points are evaluated based on the prior, the posterior can be defined. The posterior captures the updated belief in the objective function. This step is performed by using a machine learning model, also called surrogate model [Nai18].

Researchers and industrial practitioners now routinely make extensive use of software analytics to discover (e.g.) how long it will take to integrate the new code [Cze11], where bugs are most likely to occur [Ost04], who should fix the bug [Anv06], or how long it will take to develop their code [Koc12b; Koc12a; MJ03]. Large organizations like Microsoft routinely practice data-driven policy development where organizational policies are learned from an extensive analysis of large data sets collected from developers [BZ14; The15].

But the more complex the method, the harder it is to apply the analysis. Fisher et al. [Fis12] characterizes software analytics as a work flow that distills large quantities of low-value data down to smaller sets of higher value data. Due to the complexities and computational cost of SE analytics, "the luxuries of interactivity, direct manipulation, and fast system response are gone" [Fis12]. They characterize modern cloud-based analytics as a throwback to the 1960s-batch processing mainframes where jobs are submitted and then analysts wait, wait, and wait for results with "little insight into what is really going on behind the scenes, how long it will take, or how much it is going to cost" [Fis12]. Fisher et al. [Fis12] document the issues seen by 16 industrial data scientists, one of whom remarks

> "Fast iteration is key, but incompatible with the jobs are submitted and processed in the cloud. It is frustrating to wait for hours, only to realize you need a slight tweak to your feature set".

Methods for improving the quality of modern software analytics have made this issue even more serious. There has been continuous development of new feature selection [HH03] and feature discovering [Jia13] techniques for software analytics, with the most recent ones focused on deep learning methods. These are all exciting innovations with the potential to dramatically improve the quality of our software analytics tools. Yet these are all CPU/GPU-intensive methods. For instance:

- Learning control settings for learners can take days to weeks to years of CPU time [Fu16b; Tan16; Wan13].
- Lam et al. needed weeks of CPU time to combine deep learning and text mining to localize buggy files from bug reports [Lam15].
- Gu et al. spent 240 hours of GPU time to train a deep learning based method to generate API usage sequences for given natural language query [Gu16].

Note that the above problem is not solvable by waiting for faster CPUs/GPUs. We can no longer rely on Moore's Law [Moo98] to double our computational power every 18 months. Power consumption and heat dissipation issues effect block further exponential increases to CPU clock frequencies [Kum03]. Cloud computing environments are extensively monetized so the total financial cost of training models can be prohibitive, particularly for long running tasks. For example, it would take 15 years

of CPU time to learn the tuning parameters of software clone detectors proposed in [Wan13]. Much of that CPU time can be saved if there is a faster way.

2.2 Background

This section we will provide background on each case study for this thesis, which are: defect prediction, text mining, bad code smell detection and issue lifetime estimation as well as non-SE tasks. This section justifies why these are worthy of inspection and what have been done in the past.

2.2.1 SE: Defect Prediction

This section argues that defect prediction is a useful area of research, worthy of exploration and simplification. Defect prediction has been more than a decade long study and researchers have come up with different methods, to improve performances in which some are heavy CPU intensive methods like [Tan16] and some are simple. Table 2.2 gives top 22 highly cited Software defect prediction studies done since as early as 2007. This shows that the field of Software defect prediction is largely studied.

A variety of approaches have been proposed to recognize defect-prone software components using code metrics (lines of code, complexity) [D'A10; Men07a; Nag06; She14; Men10] or process metrics (number of changes, recent activity) [Has09]. Other work, such as that of Bird et al. [Bir09], indicated that it is possible to predict which components (for e.g., modules) are likely locations of defect occurrence using a component's development history and dependency structure. Prediction models based on the topological properties of components within them have also proven to be accurate [ZN08].

The lesson of all the above is that the probable location of future defects can be guessed using logs of past defects [Hal12; CD09]. These logs might summarize software components using static code metrics such as McCabes cyclomatic complexity, Briands coupling metrics, dependencies between binaries, or the CK metrics [CK94]. One advantage with CK metrics is that they are simple to compute and hence, they are widely used. Radjenović et al. [Rad13] reported that in the static code defect prediction, the CK metrics are used twice as much (49%) as more traditional source code metrics such as McCabes (27%) or process metrics (24%). Note that such attributes can be automatically collected, even for very large systems [NB05]. Other methods, like manual code reviews, are far slower and far more labor intensive.

Static code defect predictors are remarkably fast and effective. Given the current generation of data mining tools, it can be a matter of just a few seconds to learn a defect predictor (see the runtimes in Table 9 of reference [Fu16a]). Further, in a recent study by Rahman et al. [Rah14], found no significant differences in the cost-effectiveness of (a) static code analysis tools FindBugs and

Table 2.2 22 highly cited Software Defect prediction studies. Defect prediction is largely studied field.

Ref	Year	Citations
[Men07a]	2007	855
[Les08]	2008	607
[EE08]	2008	298
[Men10]	2010	178
[Gon08]	2008	159
[Kim11]	2011	153
[Rad13]	2013	150
[Jia08b]	2008	133
[WY13]	2013	115
[MK09]	2009	92
[Li12]	2012	79
[Kam07]	2007	73
[PD07]	2007	66
[Jia09]	2009	62
[Kho10]	2010	60
[Gho15]	2015	53
[Jia08a]	2008	41
[Tan16]	2016	31
[Tan15]	2015	27
[PD12]	2012	23
[Fu16a]	2016	15
[Ben17]	2017	0

Jlint, and (b) static code defect predictors. This is an interesting result since it is much slower to adapt static code analyzers to new languages than defect predictors (since the latter just requires hacking together some new static code metrics extractors).

Software developers are smart, but sometimes make mistakes. In a survey done by NIST in 2002, it was found that software defects and failures can cost upto $60 billion a year in the US itself. [Tas02]. Hence, it is essential to test software before the deployment [OR14; Bar15; YH12; Mye11]. Software quality assurance budgets are finite while assessment effectiveness increases exponentially with assessment effort [Fu16a]. Therefore, standard practice is to apply the best available methods on code sections that seem most critical and bug-prone. Software bugs are not evenly distributed across the project [HGP09; Kor09; Ost04; Mis11]. Hence, a useful way to perform software testing is to allocate most assessment budgets to the more defect-prone parts in software projects. Software defect predictors are never 100% correct. But they can be used to suggest where to focus more expensive methods.

One reason to study defect prediction is that there is much commercial interest in this technology. In their survey of 395 practitioners from 33 countries and five contents, Wan et al. [Wan18] found

that over 90% of the respondents were willing to adopt defect prediction techniques. Other results from commercial deployments show the benefits of defect prediction. When Misirli et al. [Mis11] built a defect prediction model for a telecommunications company, those models could predict 87 percent of code defects. Those models also decreased inspection efforts by 72 percent, and hence reduce post-release defects by 44 percent. Also, when Kim et al. [Kim15] applied defect prediction model, REMI, to API development process at Samsung Electronics, they found they could predicted the bug-prone APIs with reasonable accuracy (0.68 F1 score) and reduced the resources required for executing test cases.

Software defect predictors not only save labor compared with traditional manual methods, but they are also competitive with certain automatic methods. Given this equivalence, it is significant to note that static code defect prediction can be quickly adapted to new languages by building lightweight parsers to extract static code metrics such as CK metrics [CK94]. The same is not true for static code analyzers - these need extensive modification before they can be used in new languages.

Class imbalance is concerned with the situation in where some classes of data are highly under-represented compared to other classes [HG09]. By convention, the under-represented class is called the *minority* class, and correspondingly the class which is over-represented is called the *majority* class. In this paper, we say that class imbalance is *worse* when the ratio of minority class to majority *increases*, that is, *class-imbalance of 5:95* is worse than *20:80*. Menzies et al. [Men07b] reported SE data sets often contain class imbalance. In their examples, they showed static code defect prediction data sets with class imbalances of 1:7; 1:9; 1:10; 1:13; 1:16; 1:249.

The problem of class imbalance is sometimes discussed in the software analytics community. Hall et al. [Hal12] found that models based on C4.5 under-perform if they have imbalanced data while Naive Bayes and Logistic regression perform relatively better. Their general recommendation is to not use imbalanced data. Some researchers offer preliminary explorations into methods that might mitigate for class imbalance. Wang et al. [WY13] and Yu et al. [Yu17] validated the Hall et al. results and concluded that the performance of C4.5 is unstable on imbalanced data sets while Random Forest and Naive Bayes are more stable. Yan et al. [Yan10] performed fuzzy logic and rules to overcome the imbalance problem, but they only explored one kind of learner (Support Vector Machines). Pelayo et al. [PD07] studied the effects of the percentage of oversampling and undersampling done. They found out that different percentage of each helps improve the accuracies of decision tree learner for defect prediction using CK metrics. Menzies et al. [Men08b] undersampled the non-defect class to balance training data and reported how little information was required to learn a defect predictor. They found that throwing away data does not degrade the performance of Naive Bayes and C4.5 decision trees. Other papers [PD07; PD12; Riq08] have shown the usefulness of resampling based on different learners.

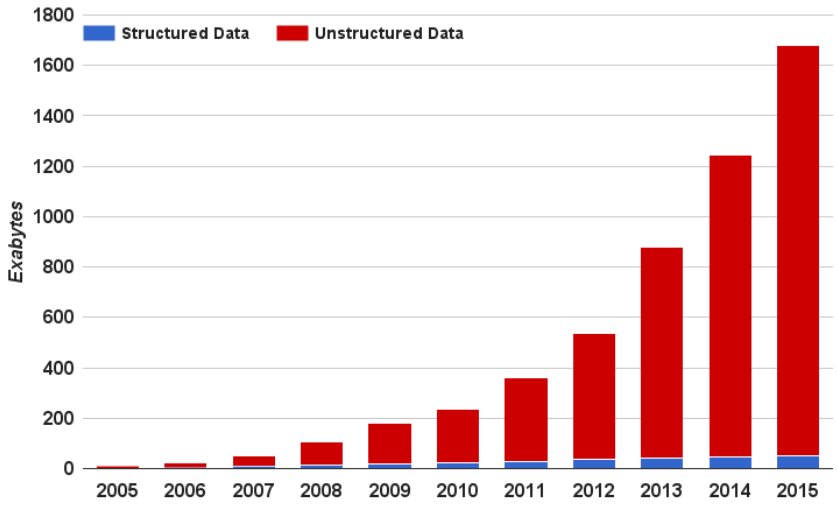

Figure 2.1 Data Growth 2005-2015, from [NY14].

2.2.2 SE: Text Mining

The current great challenge in software analytics is understanding unstructured data. As shown in Figure 2.1, most of the planet's 1600 Exabytes of data does not appear in structured sources (databases, etc) [NY14]. Mostly the data is of *unstructured* form, often in free text, and found in word processing files, slide presentations, comments, etc. Such unstructured data does not have a pre-defined data model and is typically text-heavy. Finding insights among unstructured text is difficult unless we can search, characterize, and classify the textual data in a meaningful way.

The earliest results from software analytics come from relatively simpler problems (e.g. defect prediction applied to structured data). To some extent, this was due to relative simplicity of the problem. In defect prediction, a single example is some unit of code such as module or function or class. Since that code can compile, then by definition, that unit of code has a precisely defined syntax and semantics. In practice, such code modules are described in just a few dozen attributes.

Lately, there has been much more interest in SE text mining [Men08a; MM08; Pan13; Agr18b; Xu16; Men18] since this covers a much wider range of SE activities. Text mining is a harder problem than defect prediction. Free form natural language is semantically very complex and may not conform to any known grammar. In practice, such text documents may be described using tens of thousands of attributes (one for each word in the natural language of the author of those documents). For example, consider NASA's software project and issue tracking systems (or PITS) [Men08a; MM08]. PITS contains text that discusses bugs and changes in source code. It also contains and comments on software patches.

We also studied text-mining since it has been widely studied in recent research papers appearing

Table 2.3 Top SE venues that published on Text Mining using Topic Modeling from 2009 to 2016.

Venue	Full Name	Count
ICSE	International Conference on Software Engineering	4
CSMR-WCRE / SANER	International Conference on Software Maintenance, Reengineering, and Reverse Engineering / International Conference on Software Analysis, Evolution, and Reengineering	3
ICSM / ICSME	International Conference on Software Maintenance / International Conference on Software Maintenance and Evolution	3
ICPC	International Conference on Program Comprehension	4
ASE	International Conference on Automated Software Engineering	3
ISSRE	International Symposium on Software Reliability Engineering	2
MSR	International Working Conference on Mining Software Repositories	8
OOPSLA	International Conference on Object-Oriented Programming, Systems, Languages, and Applications	1
FSE/ESEC	International Symposium on the Foundations of Software Engineering / European Software Engineering Conference	1
TSE	IEEE Transaction on Software Engineering	1
IST	Information and Software Technology	3
SCP	Science of Computer Programming	2
ESE	Empirical Software Engineering	4

in prominent SE venues. Tables 2.3 [Sun16] and 2.4 show top SE venues that published SE results and a sample of those papers respectively.

As to *how* text mining have been used, it is important to understand the distinction between supervised and unsupervised data mining algorithms. In the general sense, most data mining is "supervised" when you have data samples associated with labels and then use machine learning tools to make predictions. For example, in the case of [Loh13; Sun15], the authors used LDA as a feature extractor to build feature vectors which, subsequently, were passed to a learner to predict for a target class. Note that such supervised LDA processes can be fully automated and do not require human-in-the-loop insight to generate their final conclusions.

However, in case of, "unsupervised learning", the final conclusions are generated via a manual analysis and reflection over, e.g., the topics generated by topic modeling. Such cases represent [Bar14; Hin12] who used a manual analysis of the topics generated from some SE tasks textual data by LDA as part of the reasoning within the text of that paper. It is possible to combine manual and automatic methods, please see the "Semi-supervised" paper of Le et al. [Le14].

However, as shown in our sample, one observation could be made that out of the 28 studies in Table 2.4, 23 of them talks for the purposes of *unsupervised exploration*. As witnessed in Table 2.4, many papers have been using text mining extensively for many SE tasks.

Table 2.4 A sample of the recent literature on using Text Mining in SE. Sorted by number of citations (in column3).

REF	Year	Citations	Venues	Tasks / Use cases	Unsupervised or Supervised
[RK11]	2011	112	WCRE	Bug Localisation	Unsupervised
[Oli10b]	2010	108	MSR	Traceability Link recovery	Unsupervised
[Bar14]	2014	96	ESE	Stackoverflow Q&A data analysis	Unsupervised
[Pan13]	2013	75	ICSE	Finding near-optimal configurations	Semi-Supervised
[GCW13]	2013	61	ICSE	Software RequirementsAnalysis	Unsupervised
[Hin11]	2011	52	MSR	Software Artifacts Analysis	Unsupervised
[GM14]	2014	44	RE	Requirements Engineering	Unsupervised
[Tho11]	2011	44	ICSE	A review on LDA Mining software repositories using topic models	Unsupervised
[Tho14b]	2014	35	SCP	Software Artifacts Analysis	Unsupervised
[Che12]	2012	35	MSR	Software Defects Prediction	Unsupervised
[Tho14a]	2014	31	ESE	Software Testing	Unsupervised
[BL09]	2009	29	MSR	Software History Comprehension	Unsupervised
[Loh13]	2013	27	ESEC/FSE	Traceability Link recovery	Supervised
[Bin14]	2014	20	ICPC	Source Code Comprehension	Unsupervised
[LV13]	2013	20	MSR	Stackoverflow Q&A data analysis	Unsupervised
[Kol14]	2014	15	WebSci	Social Software Engineering	Unsupervised
[Gra13]	2013	13	SCP	Source Code Comprehension	Unsupervised
[Hin12]	2012	13	ICSM	Software Requirements Analysis	Unsupervised
[Fu15]	2015	6	IST	Software re-factoring	Supervised
[GM16]	2016	5	CS Review	Bibliometrics and citations analysis	Unsupervised
[Le14]	2014	5	ISSRE	Bug Localisation	Semi-Supervised
[Nik15]	2015	3	JIS	Social Software Engineering	Unsupervised
[Sun15]	2015	2	IST	Software Artifacts Analysis	Supervised
[Che16]	2016	0	/JSS	Software Defects Prediction	Unsupervised

2.2.3 SE: Bad Code Smell Detection

According to Fowler [Fow99], bad smells (i.e., code smells) are "a surface indication that usually corresponds to a deeper problem". Studies suggest a relationship between code smells and poor maintainability or defect proneness [YM13; YC13; Zaz11] and therefore, smell detection has become an established method to discover source code (or design) problems to be removed through refactoring steps, with the aim to improve software quality and maintenance. Research on software refactoring endorses the use of code-smells as a guide for improving the quality of code as a preventative maintenance. Consequently, code smells are captured by popular static analysis tools, like PMD [2], CheckStyle [3], FindBugs [4], and SonarQube [5].

Now, most detection tools for code smells make use of detection rules based on the computation of a set of metrics, e.g., well-known object-oriented metrics such as naming [Moh10], structural rules [Moh10], or even version history [Pal13]. Though these rule-based approaches rely heavily on human created rules that must be manually specified. For example, DECOR [Moh10] requires that the rules are specified in the form of domain specific language and this specification process must be undertaken by domain experts, engineers or quality experts. Naturally, this rule creation requires effort from these individuals that could be spent in some other tasks. Whether the metrics based ML-approaches require less effort than rule-based approaches is however not clear, and it depends on two factors; (a) how complex rules one needs for the rule based approaches, and (b) how many training samples are needed for the metrics based machine learning approaches. At the moment, we are not aware of any studies comparing the approaches effort-wise. However, what remains a clear benefit for the metrics based ML- approach is the reduction of cognitive load required from the engineers. The rule-based approach requires that the engineers create specific rules of defining each smell. For the machine learning based approached the rule creation is left for the ML-algorithms requiring the engineers only to provide information whether a piece of code has a smell or not. Also, these object-oriented metrics are used to set some thresholds for the detection of a code smell. But these rules lead to far too many false positives making it difficult for practitioners to refactor code [Kri17].

Recently, the research community is changing rapidly in terms of defining novel methodologies that incorporate additional information to detect code-smells. Much progress has been made in towards adopting machine learning tools to classify code smells from examples, easing the build of automatic code smell detectors, and also due to the problems face in manual detection using object-oriented metrics, thereby providing a better-targeted detection. Maiga et al. [Mai12b] introduce SVMDetect, an approach to detect anti- patterns or bad code smell, based on support vector

[2] https://github.com/pmd/pmd
[3] http://checkstyle.sourceforge.net/
[4] http://findbugs.sourceforge.net/
[5] http://www.sonarqube.org/

machines (SVM). The subjects of their study are the Blob, Functional Decomposition, Spaghetti Code and Swiss Army Knife antipatterns, extracted from three open-source programs: ArgoUML, Azureus, and Xerces. Maiga et al. [Mai12a] extended the previous paper by introducing SMURF, which takes into account practitioner's feedback.

Kreimer [Kre05] proposed an adaptive detection to combine known methods for finding design flaws Large Class and Long Method on the basis of metrics. Khomh et al. [Kho09] proposed a Bayesian approach to detect occurrences of the Blob antipattern on open-source programs. Khomh et al. [Kho11] also presented BDTEX, a GQM approach to build Bayesian Belief Networks from the definitions of antipatterns. Yang et al. [Yan12] study the judgment of individual users by applying machine learning algorithms on code clones. These studies were not included in our comparison as the data was not readily available for us to reuse.

More recently, Fontana et al. [Arc16] in their study of several code smells, considered 74 systems for their analysis and validation. They experimented with 16 different machine learning algorithms. They made available their dataset, which we have adapted for our applications in this study. These datasets were generated using the Qualitas Corpus (QC) of systems [Tem10]. The Qualitas corpus is composed of 111 systems written in Java, characterized by different sizes and belonging to different application domains. Fontana et al. [Arc16] selected a subset of 74 systems for their analysis. The authors computed a large set of object-oriented metrics belonging to class, method, package, and project level. A detailed list of metrics and their definitions are available in appendices of [Arc16].

2.2.4 SE: Issue Lifetime Estimation

Open source projects use issue tracking systems to enable effective development and maintenance of their software systems. Typically, issue tracking systems collect information about system failures, feature requests, and system improvements. Based on this information and actual project planing, developers select the issues to be fixed. Predicting the time it may take to close an issue has multiple benefits for the developers, managers, and stakeholders involved in a software project. Predicting issue lifetime helps software developers better prioritize work; helps managers effectively allocate resources and improve consistency of release cycles; and helps project stakeholders understand changes in project timelines and budgets. It is also useful to be able to predict issue lifetime specifically when the issue is created, since it is found earlier that delaying to resolve issues can become harder and costlier [Men17]. An immediate prediction can be used, for example, to auto-categorize the issue or send a notification if it is predicted to be an easy fix.

As an initial attempt, Panjer [Pan07] used logistic regression models to classify bugs as closing in 1.4, 3.4, 7.5, 19.5, 52.5, and 156 days, and greater than 156 days. He was able to achieve an accuracy of 34.9%. Giger et al. [Gig10] used models constructed with decision trees to predict for issue lifetimes (binary classification) in Eclipse, Gnome, and Mozilla. They were able obtain a peak precision of

65by dividing time into 1, 3, 7, 14, 30 days. Zhang et al. [Zha13] developed a comprehensive system to predict lifetime of issues. They used a Markov model with a kNN-based classifier to perform their prediction. More recently, Rees-Jones et al. [Ree17] showed that using Hall's CFS feature selector and C4.5 decision tree learner a very reliable prediction of issue lifetime could be made.

Bhattacharya and Neamtiu [BN11] studied how existing models "fail to validate on large projects widely used in bug studies". In a comprehensive study, they find that the predictive power (measured by the coefficient of determination R^2) of existing models is 30-49%. That study found that there is no correlation between bug-fix likelihood, bug-opener reputation, and time required to close the bug for the three datasets used in their study. Guo et al. [Guo10] evaluated the most predictive factors that affect bug fix time for Windows Vista and Windows 7 software bugs. Unlike [BN11], they found that bug-opener reputation affected fix time; an issue with a high-reputation creator was more likely to get fixed. Bugs are also more likely to get fixed if the bug fixer is in the same team as or in geographical proximity of the bug creator. Guo et. al. conclude that the factors most important in bug fix time are social factors such as history of submitting high-quality bug reports and trust between teams interacting over bug reports.

Marks et al. [Mar11] found that time and location of filing the bug report were the most important factors in predicting Mozilla issues, but severity was most important for Eclipse issues. Priority was not found to be an important factor for either Mozilla or Eclipse. Their models produced lower performance metrics (65% misclassification rate) than subsequent work. Kikas et al. [Kik16] 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.

2.2.5 NON-SE: UCI Repository

Till this section, we have only been looking at tasks and data related to SE. We also wanted to explore non-SE tasks in which the data came from popular UCI repository [AN07; DG17].

UCI Machine Learning Data Repository was created in 1987 to foster experimental research in machine learning. It contains over 429 databases from a broad range of problem areas including engineering, molecular biology, medicine, finance and politics. The Machine Learning (ML) Repository is commonly used by industrial and academic researchers. It is widely cited in the artificial intelligence literature and the UCI data sets are the most widely used benchmark for empirical evaluation of new and existing learning algorithms. Over 3800 papers available on the web, cite the repository.

UCI contains data with varying attributes type, sample size, and attribute size. It has data for classification, regression and clustering tasks. Due to the existence of large number of datasets from multiple fields, UCI is a good resource to verify our hypothesis on non-SE domain.

2.3 Existence and Exploitation of ϵ

AI researchers have repeatedly concluded that a small number of *key* variables determine the behavior of the rest of the system. **Keys** have been discovered and rediscovered many times and given different names, including feature subset selection [KJ97], narrows [Mic13], master variables [CB94], and back doors [Wil03]. We mention this since the problem of mining data with keys reduces to just the mining the key variables. For many years, we have been believing the Machine learning problems to be complex and due to that new complex methods (which take high CPU resources) have been proposed like deep learning, HPOs and many more taking days, weeks, years of CPU.

For over a decade now, we have been documenting examples were solutions to SE analytics problems were surprisingly **simple** (but until **DODGE(ϵ)**, we had no explanation for that phenomenon nor a method to exploit that simplicity). For example, when optimizing requirements engineering, we have often found that most requirements are dependent on just a few *key* choices. We have exploited this effect to simplify reasoning about requirements engineering models [Has05]. In another case, Panichella et al. [Pan13] took 1,000 to 10,000 of key evaluations to tune the parameters of topic modeling where as Agrawal et al. [Agr18b] achieved the same in just 30 key evaluations.

In software effort estimation, COCOMO model takes many attributes to calculate effort but it was found that only few parameters have the most impact on effort [MS12]. For defect prediction, the variance in performance increases (become more unstable) with more number of samples. Also, the performance optimization of a data imbalance technique was achieved in only 50 evaluations [AM18].

Above examples show that minimalist, or less have been better. With this, we should expect that the results from software analytics have some large degree of variability ϵ. One way to characterize this thesis is to say:

- Stop treating large ϵ variability as a problem;
- Instead, treat large ϵ variability as a resource that can be used to simplify software analytics.

According to Menzies and Shepperd [MS12], the variability or uncertainity of software quality predictors come from different choices in the training data and many other factors.

Sampling Bias: Any data mining algorithm inputs multiple examples to make its conclusions. The more diverse the input examples, the greater the variance in the conclusions. And software engineering is a very diverse discipline:

- Software is built by people with variable skills.
- That construction process is performed using a wide range of languages and tools.
- The product is delivered to many platforms.

- Languages, tools & platforms keep changing.
- Within one project, the problems faced and the purpose served by each software module may be very different (e.g., GUI, database, network connections, business logic, etc.).
- Within the career of one developer, the problem domain, goal, and stakeholders of their software can change dramatically from one project to the next.

Pre-processing: Real world data usually requires some clean-up before it can be used effectively by data mining algorithms. There are many ways to transform the data favorably. The numeric data may be discretized into smaller bins. Discretization can greatly affect the results of the learning: since there may be ways to implement discretization [FI93]; Feature selection is sometimes useful to prune uncorrelated features to the target variable [Che05a]. Also, it can be helpful to prune data points that are very noisy or are outliers [Koc10]. Note that the choices made during pre-processing can introduce some variability in the final results.

Stochastic algorithms: Numerous methods in software quality predictors employ stochastic algorithms that use random number generators. For example, the key to scalability is usually (a) build a model on the randomly selected small part of the data then (b) see how well that works over the rest of the data [Scu10]. Also, when evaluating data mining algorithms, it is standard practice to divide the data randomly into several bins as part of a cross-validation experiment [WF02]. For all these stochastic algorithms, the conclusions are adjusted, to some extent, by the random numbers used in the cross-validation experiments.

All the above examples, complex or simple methods which were proposed is to reduce the uncertainties or in other words try to make $\epsilon = 0$ (uncertainty to nil).

Now consider the implications of large ϵ for exploring HPO. Recall that such optimizers find a few useful options by exploring a very large number of options. Given a results space like Figure 1.1, many of those options will fall into the same regions. If we mark which options generate results that fall within ϵ of other results, then we strive to avoid those options in the future, then theoretically we might be able to search across Figure 1.1, very quickly. This large ϵ variability is utilized by **DODGE(ϵ)**, our framework. Next chapters we will describe our methodlogy and framework used in thesis as well as our result findings.

2.4 Summary

In this chapter we provided a brief background on hyperparameter tuning and what optimizers did we select for our study and why. We also showed that studying defect prediction, text mining, bad code smell detection and issue lifetime estimation SE tasks is important due to its extensive popularity as well as their direct impact in industries, in terms of cost, time and resources. We also

showed why it was needed to study non-SE domain using the datasets available in UCI machine learning repository. We also explained what is so different in nature about SE data by showing examples of ϵ-domination existence, which can already be seen in many research articles, but no one thought about it or took any measures to exploit this existence.

In the next chapter, we will dive deep into the dataset details for both SE and NON-SE tasks, and what are the different frameworks which were implemented by us (SMOTUNED, LDADE, FFtree) as well as other researchers (FLASH, SMOTE, LDA, LDA-GA). We will finally introduce how our framework **DODGE(ϵ)** exploits the existence of ϵ-dominance to have simpler and better model.

CHAPTER

3

DISCUSSION OF THE METHODOLOGY

This chapter, we will introduce the datasets used for all 4 SE tasks and non-SE tasks. All these tasks and related datasets try to solve for binary classification. After that we describe our own developed frameworks LDADE, SMOTUNED, FFtrees and finally **DODGE(ϵ)**. We also compared LDADE against LDA-GA, LDA which were implemented by other researcher. LDADE and SMOTUNED were implemented to motivate that optimal performance can be achieved in smaller number of key evaluations than other Genetic algorithms. After observing so low number of evaluations, FFtrees was developed to further reduce the number of evaluations. FLASH was introduced by Nair et al. [Nai18] which we incorporated to compare against our recently developed frameworks. These frameworks helped us to concur that there must be something different in the nature of SE data which we can utilize to further simplify our methods. That is where **DODGE(ϵ)** was introduced to explore billions of choices but evaluate only few to achieve optimal performance which none of the other frameworks could do. We will describe our evaluation criteria and the billions of choices that were explored by **DODGE(ϵ)**. Also, Table 3.1 shows a summarized comparison of frameworks for each SE and non-SE tasks.

Table 3.1 Frameworks comparison used with different SE and NON-SE tasks.

	Traditional ML	SMOTUNED	FFTree	FLASH	LDA-GA	LDA	DODGE(ϵ)
Defect Prediction	✓	✓	✓	✓			✓
Text Mining	✓		✓		✓	✓	✓
Code Smell Detection	✓		✓				✓
Predict Issue Lifetime	✓		✓				✓
Non-SE (UCI)	✓		✓				✓

3.1 Datasets

3.1.1 Defect Prediction

Defect prediction can input a range of attributes. For example, table 3.2 and 3.3 describes some data widely used in this area of research. As shown in table 3.3, this data is available for multiple versions of the same software (from http://tiny.cc/seacraft). This is important since, when applying data mining algorithms to build predictive models, one important principle is not to test on the data used in training. There are many ways to design a experiment that satisfies this principle. Some of those methods have limitations; e.g., leave-one-out is too slow for large data sets and cross-validation mixes up older and newer data (such that data from the past may be used to test on future data). In this work, for each project data, we set the latest version of project data as the testing data and all the older data as the training data. For example, we use *poi1.5*, *poi2.0*, *poi2.5* data for training predictors, and the newer data, *poi3.0* is left for testing.

Table 3.3 illustrates the variability of SE data. When we compare the % of Defects in the training and test data, we see that the past can be very different to the future. Observe how the median defect percentage in the training data is 29% but in the test data, it is 49% (i.e. nearly doubled). This tells us that software analytics will forever be an imprecise science (and one of the lessons of this paper is that imprecision can be used to simplify complex tasks like hyperparameter optimization).

One issue with some of the data in table 3.3 is imbalanced class frequencies. If the target class is not common (as in camel, ivy, jedit and to a lesser extent velocity and synapse), it can be difficult for a data mining algorithm to generate a model that can locate it. A standard method for addressing class imbalance is the SMOTE pre-processor [Cha02]. SMOTE randomly deletes members of the majority class while synthesizing artificial members of the minority class. SMOTE is controlled by the parameters shown in Table 3.4.

As to machine learning algorithms for defect prediction, these are many and varied. At ICSE'15, Ghotra et al. [Gho15] applied 32 different machine learning algorithms to defect prediction. In a result consistent with the theme of this article, they found that those 32 algorithms formed in the four groups (and the performance of two learners in any one group were statistically indistinguishable

Table 3.2 OO CK code metrics used for all studies in this paper. The last line shown, denotes the dependent variable.

amc	average method complexity	e.g., number of JAVA byte codes
avg, cc	average McCabe	average McCabe's cyclomatic complexity seen in class
ca	afferent couplings	how many other classes use the specific class.
cam	cohesion amongst classes	summation of number of different types of method parameters in every method divided by a multiplication of number of different method parameter types in whole class and number of methods.
cbm	coupling between methods	total number of new/redefined methods to which all the inherited methods are coupled
cbo	coupling between objects	increased when the methods of one class access services of another.
ce	efferent couplings	how many other classes is used by the specific class.
dam	data access	ratio of the number of private (protected) attributes to the total number of attributes
dit	depth of inheritance tree	
ic	inheritance coupling	number of parent classes to which a given class is coupled
lcom	lack of cohesion in methods	number of pairs of methods that do not share a reference to an case variable.
locm3	another lack of cohesion measure	if m,a are the number of $methods,attributes$ in a class number and $\mu(a)$ is the number of methods accessing an attribute, then $lcom3 = ((\frac{1}{a}\sum, j^a \mu(a,j)) - m)/(1-m)$.
loc	lines of code	
max, cc	maximum McCabe	maximum McCabe's cyclomatic complexity seen in class
mfa	functional abstraction	no. of methods inherited by a class plus no. of methods accessible by member methods of the class
moa	aggregation	count of the number of data declarations (class fields) whose types are user defined classes
noc	number of children	
npm	number of public methods	
rfc	response for a class	number of methods invoked in response to a message to the object.
wmc	weighted methods per class	
defect	defect	Boolean: where defects found in post-release bug-tracking systems.

Table 3.3 Some open-source JAVA systems. Used for training and testing, showing different details for each.

	Training			Testing		
Data Set	Versions	Cases	% Defective	Versions	Cases	% Defective
jedit	3.2, 4.0, 4.1, 4.2	1257	23	4.3	492	2
ivy	1.1, 1.4	352	22	2.0	352	11
camel	1.0, 1.2, 1.4	1819	21	1.6	965	19
synapse	1.0, 1.1	379	20	1.2	256	34
velocity	1.4, 1.5	410	70	1.6	229	34
lucene	2.0, 2.2	442	53	2.4	340	59
poi	1.5, 2, 2.5	936	46	3.0	442	64
xerces	1.0, 1.2, 1.3	1055	16	1.4	588	74
log4j	1.0, 1.1	244	29	1.2	205	92
xalan	2.4, 2.5, 2.6	2411	38	2.7	909	99

Table 3.4 SMOTE parameters

Para	**Defaults used by** SMOTE	**Tuning Range (Explored by** (SMOTUNED)	**Description**
k	5	[1,20]	Number of neighbors
m	50%	{50, 100, 200, 400}	Number of synthetic examples to create. Expressed as a percent of final training data.
r	2	[0.1,5]	Power parameter for the Minkowski distance metric.

Table 3.5 Classifiers used by SMOTUNED. Rankings from Ghotra et al. [Gho15].

RANK	LEARNER	NOTES
1 "best"	RF= random forest	Random forest of entropy-based decision trees.
	LR=Logistic regression	A generalized linear regression model.
2	KNN= K-means	Classify a new instance by finding "k" examples of similar instances. Ghortra et al. suggested $K = 8$.
	NB= Naive Bayes	Classify a new instance by (a) collecting mean and standard deviations of attributes in old instances of different classes; (b) return the class whose attributes are statistically most similar to the new instance.
3	DT= decision trees	Recursively divide data by selecting attribute splits that reduce the entropy of the class distribution.
4 "worst"	SVM= support vector machines	Map the raw data into a higher-dimensional space where it is easier to distinguish the examples.

from each other). We took couple of learners from each group for the case study of SMOTUNED as shown in table 3.5.

We compared **DODGE(ϵ)** to traditional machine learning algorithms [Gho15], HPO tuning a classifier [Fu16a], HPO tuning a preprocessor [AM18], FFtree [Che18], and FLASH [Nai18]. These all comparisons have shown to work really well in the past.

3.1.2 Text Mining

Table 3.6 describe our PITS data. **PITS** is a text mining data set generated from NASA software project and issue tracking system (PITS) reports [Men08a; MM08]. This text discusses bugs and changes found in big reports and review patches. Such issues are used to manage quality assurance, to support communication between developers. Topic modeling in PITS can be used to identify the top topics which can identify each severity separately. The dataset can be downloaded from the PROMISE repository [Men15]. Note that, this data comes from six different NASA projects, which we label as PitsA, PitsB, etc.

For this study, all datasets were preprocessed using the usual text mining filters [Fel06]. We implemented stop words removal using NLTK toolkit[1] [Bir06] (to ignore very common short words such as "and" or "the"). Next, Porter's stemming filter [Por80] was used to delete uninformative word endings (e.g., after performing stemming, all the following words would be rewritten to "connect": "connection", "connections", "connective", "connected", "connecting").

Tf-idf word reduction: focus on the 5% of words that occur frequently, but only in small numbers of documents. If a word occurs w times and is found in d documents and there are W, D total

[1] http://www.nltk.org/book/ch02.html

Table 3.6 Text Mining Dataset statistics. Data comes from the SEACRAFT repository: `http://tiny.cc/seacraft`

Dataset	No. of Documents	No. of Words	Severe %
PitsA	965	155,165	39
PitsB	1650	104,052	40
PitsC	323	23,799	56
PitsD	182	15,517	92
PitsE	825	93,750	63
PitsF	744	28,620	64

number of words and documents respectively, then tf-idf is scored as follows:

$$tfidf(w,d) = \frac{w}{W} * \log \frac{D}{d}$$

After these data-preprocessing methods, the textual data is converted into numerical features using some vectorization methods such as LDA, TFIDF, TF, and HASHING methods which we will be talked about in later Chapter.

Also, unlike the defect prediction data of Table 3.3, the PITS data is not conveniently divided into versions. Hence, to generate separate train and test data sets, we use a $x * y$ stratified cross-validation study where, $x = 5$ times, we randomize the order of the data then divide into into $y = 5$ bins. Then, we test on that bin after training on all the others.

We compared **DODGE(ϵ)** to traditional machine learning algorithms [Kri16], GA tuning LDA [Pan13], DE tuning a LDA [AM18], and FFtree [Che18; Agr18a]. These all comparisons have shown to work really well in the past.

3.1.3 Bad Code Smell Detection

Fontana et al. [Arc16] studied several code smells from 74 systems for their analysis and validation. They made their dataset available publicly, which we have adapted for our applications in this study. These datasets were generated using the Qualitas Corpus (QC) of systems [Tem10]. The Qualitas corpus is composed of 111 systems written in Java, characterized by different sizes and belonging to different application domains. Fontana et al. [Arc16] selected a subset of 74 systems for their analysis. The authors computed a large set of object-oriented metrics belonging to class, method, package, and project level. A detailed list of metrics and their definitions are available in appendices of [Arc16].

The code smells repository we use comprises of 22 datasets for 2 different code smells: class and method level. For each level we used 2 datasets each, 1) For class level, God Class and Data

Table 3.7 Bad code smell detection Dataset statistics. Data comes from [Arc16]

Nature	Dataset	No. of instances	No. of attributes	Smelly %
Method	Feature Envy	109	82	45
Method	Long Method	109	82	43.1
Class	God Class	139	61	43.9
Class	Data Class	119	61	42

Class; and 2) For method level, Long Method, Feature Envy. Data statistics for these datasets could be found in Table 3.7. The attributes in this dataset which are OO oriented metrics is described in Table 3.8. Descriptive information on what these datasets are given below:

- **Data Class** code smell refers to classes that store data without providing complex functionality, and having other classes strongly relying on them. A Data Class reveals many attributes, it is not complex, and exposes data through accessor methods.
- **God Class** code smell refers to classes that tend to centralize the intelligence of the system. A God Class tends to be complex, to have too much code, to use large amounts of data from other classes and to implement several different functionalities.
- **Feature Envy** code smell refers to methods that use much more data from other classes than from their own class. A Feature Envy tends to use many attributes of other classes (considering also attributes accessed through accessor methods), to use more attributes from other classes than from its own class, and to use many attributes from few different classes.
- **Long Method** code smell refers to methods that tend to centralize the functionality of a class. A Long Method tends to have too much code, to be complex, to be difficult to understand and to use large amounts of data from other classes.

Also, unlike the defect prediction data of Table 3.3, the Bad Smell datasets are not conveniently divided into versions. Hence, to generate separate train and test data sets, we use a $x * y$ stratified cross-validation study where, $x = 5$ times, we randomize the order of the data then divide into into $y = 5$ bins. Then, we test on that bin after training on all the others.

We compared **DODGE(ϵ)** to traditional machine learning algorithms [KM18], and FFtree [Che18]. For bad code smell detection, only traditional machine learning algorithms have been used in the past. To the best of our knowledge there has not been any case study using HPO. We used FFtree following the recommendations from the past studies.

Table 3.8 Static code metrics used in code smells data sets.

Size		Complexity		Cohesion		Coupling		Encapsulation		Inheritance	
Label	Description	Label	Description	Label	Description	Label	Description	Label	Description	Label	Description
LOC	Lines Of Code	CYCLO	Cyclomatic Complexity	LCOM	Lack of Cohesion	FANOUT/IN	Fan Out/In	LAA	Locality of Attribute Accesses	DIT	Depth of Inheritance Tree
LOCNAMM	LOC (without accessor or mutator)	WMC	Weighted Methods Count	TCC	Tight Class Cohesion	ATFD	Access to Foreign Data	NOAM	Number of Accessor Methods	NOI	Number of Interfaces
NOM	No. of Methods	WMCNAMM	Weighted Methods Count (without accessor or mutator)	CAM	Cohesion Among classes	FDP	Foreign Data Providers	NOPA	Number of Public Attribute	NOC	Number of Children
NOPK	No. of Packages	AMW	Average MethodsWeight			RFC	Response for a Class			NMO	Number of Methods Overridden
NOCS	No. of Classes	AMWNAMM	Average Methods Weight (without accessor or mutator)			CBO	Coupling Between Objects			NIM	Number of Inherited Methods
NOMNAMM	Number of Not Accessor or Mutator Methods	MAXNESTING	Max Nesting			CFNAMM	Called Foreign Not Accessor or Mutator Methods			NOII	Number of Implemented Interfaces
NOA	Number of Attributes	CLNAMM	Called Local Not Accessor or Mutator Methods			CINT	Coupling Intensity				
		NOP	Number of Parameters			MaMCL	Maximum Message Chain Length				
		NOAV	Number of Accessed Variables			MeMCL	Mean Message Chain Length				
		ATLD	Access to Local Data			CA/CE/IC	Afferent/ Efferent/ Inheritance coupling				
		NOLV	Number of Local Variable			CM	Changing Methods				
		WOC	Weight Of Class			CBM	Coupling between Methods				
		MAX_CC/AVG_CC	Maximum/ Average McCabe								

3.1.4 Issue Lifetime Estimation

Tables 3.9 and 3.10 show list of 8 projects used to study issue lifetimes. These datasets have class imbalance problem as our other SE tasks. The goal here is to classify an issue according to how long it will take to close; i.e. less than 1 day, greater than 1 and less than 7 days, and so on. We used this data which was provided by Rees-Jones et al. [Ree17]. The attributes/metrics used by these dataset is described in Table 3.11.

In raw form, the data consisted of sets of JSON files for each repository, each file contained one type of data regarding the software repository (issues, commits, code contributors, changes to

specific files). In order to extract data specific to issue lifetime, we did similar preprocessing and feature extraction on the raw datasets as suggested by [Ree17].

Also, unlike the defect prediction data of Table 3.3, the Issue Lifetime Estimation datasets are not conveniently divided into versions. Hence, to generate separate train and test data sets, we use a $x * y$ stratified cross-validation study where, $x = 5$ times, we randomize the order of the data then divide into into $y = 5$ bins. Then, we test on that bin after training on all the others.

We compared **DODGE(ϵ)** to traditional machine learning algorithms [Ree17; KM18], and FFtree [Che18]. For estimating issue lifetime, only traditional machine learning algorithms and FFtree have been used in the past. To the best of our knowledge there has not been any other case study using HPO.

3.1.5 NON-SE: UCI Datasets

The UCI Machine Learning Repository is a collection of databases, domain theories, and data generators that are used by the machine learning community for the empirical analysis of machine learning algorithms. The archive was created as an ftp archive in 1987 by David Aha and fellow graduate students at UC Irvine. It is hosted and maintained by the Center for Machine Learning and Intelligent Systems at the University of California, Irvine. Since that time, it has been widely used by students, educators, and researchers all over the world as a primary source of machine learning data sets. As an indication of the impact of the archive, it has been cited over 1000 times, making it one of the top 100 most cited "papers" in all of computer science. The current version of the web site was designed in 2007 by Arthur Asuncion and David Newman, and this project is in collaboration the University of Massachusetts Amherst. They receive funding support from the National Science Foundation. Also, it is a database that can be accessed for free.

UCI is a popular archive storage platform which contains over 429 databases from a broad range of problem areas including engineering, molecular biology, medicine, finance and politics. The Machine Learning (ML) Repository is commonly used by industrial and academic researchers. It is widely cited in the artificial intelligence literature and the UCI data sets are the most widely used benchmark for empirical evaluation of new and existing learning algorithms. Over 3800 papers available on the web, cite the repository.

UCI contains data with varying attributes type, sample size, and attribute size. Almost all datasets are drawn from the domain (as opposed to being synthetic), meaning that they have real-world qualities. It has data for classification, regression and clustering tasks. Datasets cover a wide range of subject matter from biology to particle physics. The details of datasets are summarized by aspects like attribute types, number of instances, number of attributes and year published that can be sorted and searched. Datasets are well studied which means that they are well known in terms of interesting properties and expected “good” results. This can provide a useful baseline for comparison. Most datasets are small (hundreds to thousands of instances) meaning that it can be readily loaded in a

Table 3.9 Issue Lifetime Estimation Dataset statistics. Data comes from [Ree17]

Project Name	Dataset	# of instances		# metrics
		Total	Closed (%)	
camel	1 day	5056	698 (14.0)	18
	7 days		437 (9.0)	
	14 days		148 (3.0)	
	30 days		167 (3.0)	
	90 days		298 (6.0)	
	180 days		657 (13.0)	
	365 days		2052 (41.0)	
cloudstack	1 day	1551	658 (42.0)	18
	7 days		457 (29.0)	
	14 days		101 (7.0)	
	30 days		107 (7.0)	
	90 days		133 (9.0)	
	180 days		65 (4.0)	
	365 days		23 (2.0)	
cocoon	1 day	2045	125 (6.0)	18
	7 days		92 (4.0)	
	14 days		32 (2.0)	
	30 days		45 (2.0)	
	90 days		86 (4.0)	
	180 days		51 (3.0)	
	365 days		73 (3.5)	
node	1 day	6207	2426 (39.0)	18
	7 days		1800 (29.0)	
	14 days		521 (8.0)	
	30 days		453 (7.0)	
	90 days		552 (9.0)	
	180 days		254 (4.0)	
	365 days		180 (3.0)	
deeplearning	1 day	1434	931 (65.0)	18
	7 days		214 (15.0)	
	14 days		76 (5.0)	
	30 days		72 (5.0)	
	90 days		69 (5.0)	
	180 days		39 (3.0)	
	365 days		32 (2.0)	

Table 3.10 Issue Lifetime Estimation Dataset statistics Continued. Data comes from [Ree17]

Project Name	Dataset	# of instances		# metrics
		Total	Closed (%)	
hadoop	1 day	12191	40 (0.0)	18
	7 days		65 (1.0)	
	14 days		107 (1.0)	
	30 days		396 (3.0)	
	90 days		1743 (14.0)	
	180 days		2182 (18.0)	
	365 days		2133 (17.5)	
hive	1 day	5648	18 (0.0)	18
	7 days		22 (0.0)	
	14 days		58 (1.0)	
	30 days		178 (3.0)	
	90 days		1050 (19.0)	
	180 days		1356 (24.0)	
	365 days		1440 (25.0)	
ofbiz	1 day	6177	1515 (25.0)	18
	7 days		1169 (19.0)	
	14 days		467 (8.0)	
	30 days		477 (8.0)	
	90 days		574 (9.0)	
	180 days		469 (7.5)	
	365 days		402 (6.5)	
qpid	1 day	5475	203 (4.0)	18
	7 days		188 (3.0)	
	14 days		84 (2.0)	
	30 days		178 (3.0)	
	90 days		558 (10.0)	
	180 days		860 (16.0)	
	365 days		531 (10.0)	

Table 3.11 Metrics used in Issue lifetime data

Commit	Comment	Issue
nCommitsByActorsT	meanCommentSizeT	issueCleanedBodyLen
nCommitsByCreator	nComments	nIssuesByCreator
nCommitsByUniqueActorsT		nIssuesByCreatorClosed
nCommitsInProject		nIssuesCreatedInProject
nCommitsProjectT		nIssuesCreatedInProjectClosed
		nIssuesCreatedProjectClosedT
		nIssuesCreatedProjectT
Misc.	nActors, nLabels, nSubscribedByT	

text editor or Excel and reviewed, and then can easily be modeled into any local workstation.

Though there are some criticisms over the UCI repository. These datasets are cleaned, meaning that the researchers that prepared them have often already performed some pre-processing in terms of the the selection of attributes and instances. The datasets are small, this is not helpful if anyone is interested in investigating larger scale problems and techniques. There are so many to choose from that we can be frozen by indecision and over-analysis. It can be hard to just pick a dataset and get started when you are unsure if it is a "good dataset" for what youâĂŹre investigating. Datasets are limited to tabular data, primarily for classification (although clustering and regression datasets are listed). This is limiting for those interested in natural language, computer vision, recommendation system and other data.

But we still chose to use the UCI respository is that all the above criticisms do not hold true for our research. Our SE data is also public, user-provided, cleaned, smaller in size giving us a consistent comparison if we used UCI repository as our NON-SE analysis. Also, due to the existence of large number of datasets from multiple fields, UCI is a good resource to verify our hypothesis on non-SE domain.

Table 3.12 provides the list of datasets which were used in this study. These datasets are from varying domain such as Computer, Physical, Financial, Medicine, Social, Environment and many more. We picked varying domains to validate our hypothesis by using **DODGE(ϵ)** as an HPO. Many of these data have class imbalance problem as SE tasks ensuring we have similar data to make a successful comparison between SE and non-SE tasks.

Also, unlike the defect prediction data of Table 3.3, the UCI datasets are not conveniently divided into versions. Hence, to generate separate train and test data sets, we use a $x * y$ stratified cross-validation study where, $x = 5$ times, we randomize the order of the data then divide into into $y = 5$ bins. Then, we test on that bin after training on all the others.

Table 3.12 NON-SE: 37 UCI Datasets statistics.

Area	Dataset	No. of instances	No. of attributes	Class %
Computer	optdigits	1143	64	50
Physical	satellite	2159	36	28
Physical	climate-sim	540	18	91
Financial	credit-approval	653	15	45
Medicine	cancer	569	30	37
Business	shop-intention	12330	17	15
Computer Vision	image	660	19	50
Life	covtype	12240	54	22
Computer	hand	29876	15	47
Social	drug-consumption	1885	30	23
Environment	biodegrade	1055	41	34
Social	adult	45222	14	25
Physical	crowdsource	1887	28	24
Medicine	blood-transfusion	748	4	24
Financial	credit-default	30000	23	22
Medicine	cervical-cancer	668	33	7
Social	autism	609	19	30
Marketing	bank	3090	20	12
Financial	bankrupt	4769	64	3
Financial	audit	775	25	39
Life	contraceptive	1473	9	56
Life	mushroom	5644	22	38
Computer	pendigits	2288	16	50
Security	phishing	11055	30	56
Automobile	car	1728	6	30
Medicine	diabetic	1151	19	53
Physical	hepmass	2000	27	50
Physical	htru2	17898	8	9
Computer	kddcup	3203	41	69
Automobile	sensorless-drive	10638	48	50
Physical	waveform	3304	21	50
Physical	annealing	716	10	13
Medicine	cardiotocography	2126	40	22
Phyical	shuttle	54489	9	16
Electrical	electric-stable	10000	12	36
Physical	gamma	19020	10	35
Medicine	liver	579	10	72

Here, we only compared **DODGE(ϵ)** to traditional machine learning algorithms, and FFtree [Che18].

3.2 Evaluation Criteria

We choose not to evaluate a classifier on any single criteria (e.g. not just recall) since succeeding on one criteria can damage another [Fu16a]. Also, we ignored accuracy since these can be inaccurate for data sets where the target class is rare (which is common in class imbalanced data sets) [Men07b]. Instead, we evaluated our predictors on metrics that aggregate multiple metrics. We used two other *score* measures: *dis2heaven (d2h)* and P_{opt}.

D2h is short for "distance to heaven" and ideally, a perfect learner will have perfect recall (100%) with no false alarms [Che18].

$$Recall = \frac{TruePositive}{TruePositive + FalseNegative} \tag{3.1}$$

$$FAR = \frac{FalsePositive}{FalsePositive + TrueNegative} \tag{3.2}$$

We combine these two into a "distance to heaven" measure called *dis2heaven* that reports how far a learner falls away from the ideal point of *Recall=1* and *FAR=0*:

$$dis2heaven = \sqrt{\frac{(1 - Recall)^2 + FAR^2}{2}} \tag{3.3}$$

Here, the $\sqrt{2}$ term normalizes *d2h* to the range zero to one.

As to P_{opt}, Ostrand et al. [Ost05] report that their quality predictors can find 20% of the files containing 80% of all defects in the project. Although there is nothing magical about the number 20%, it has been used as a cutoff value to set the efforts required for the defect inspection when evaluating the defect learners [Kam13; MK10; Mon13; Yan16]. That is, P_{opt} reports how many defects have been found after (a) the code is sorted by the learner from "most likely to be buggy" to "least likely"; then (b) humans inspect 20% of the code (measured in lines of code), where that code has , how many defects can be detected by the learner. This measure is widely used in defect prediction literature [Kam13; Men07a; Men10; Mon13; Yan16; Zim07].

P_{opt} is defined as $1 - \Delta_{opt}$, where Δ_{opt} is the area between the effort cumulative lift charts of the optimal model and the prediction model (as shown in Figure 3.1). In this chart, the x-axis is the percentage of required effort to inspect the code and the y-axis is the percentage of defects found in the selected code. In the optimal model, all the changes are sorted by the actual defect density in descending order, while for the predicted model, all the changes are sorted by the actual predicted value in descending order. According to Kamei et al. and Xu et al. [Kam13; Mon13; Yan16] P_{opt} can

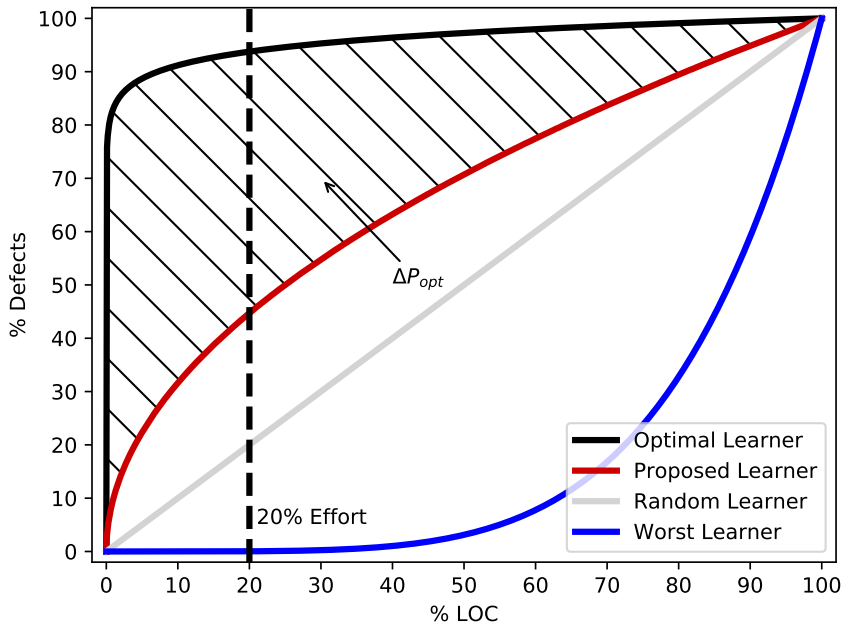

Figure 3.1 Effort-based cumulative lift chart [Yan16].

be normalized as follows:

$$score_2 = P_{opt}(m) = 1 - \frac{S(optimal) - S(m)}{S(optimal) - S(worst)} \tag{3.4}$$

where $S(optimal)$, $S(m)$ and $S(worst)$ represent the area of curve under the optimal model, predicted model, and worst model, respectively. This worst model is built by sorting all the changes according to the actual defect density in ascending order.

Note that for our two *score* functions:

- For *dis2heaven*, the *lower* values are *better*.
- For P_{opt}, the *higher* values are *better*.

Finally, when comparing one result to N others, we will sometimes see "small effects" (which can be ignored). To define "small effect", we use Cohen's delta [Coh88]:

$$d = small\ effect = 0.2 * \sqrt{\frac{\sum_i^x (x_i - (\sum x_i / n))^2}{n-1}} \tag{3.5}$$

i.e. 20% of the standard deviation.

Note one final difference to the evaluation procedures used for text mining. Before, in defect

prediction, to rule out trivially small effects we were comparing one number to a set of numbers. For that task, we used Cohen's delta. But when comparing the $x * y$ cross validation results between two treatments, we need a statistical significance test (to certify that the distributions are indeed different) and an effect size test (to check that the differences are more than a "small effect"). For text mining, we used Scott-knott tests used by many researchers [Agr18b; AM18] which uses Efron's 95% bootstrap procedure [ET93] confidence and the A12 effect size test [AB11].

3.3 Frameworks

This section we will explain in detail about SMOTE, LDA, DE, SMOTUNED, LDADE, LDA-GA, FFtree, FLASH and **DODGE(ϵ)**.

3.3.1 Traditional Machine Learning without hyperparameter optimization

Traditional machine learning algorithms are those which are financially and computationally cheaper and easier to implement. They do not require heavy CPUs or GPUs to implement and can be done over local machine with limited resources. They are also easier to interpret. Due to them being cheaper to implement we can spend resources on tuning these algorithms using an HPO. Some of the traditional machine learning algorithms which we have used in this thesis can be found in table 3.13.

3.3.2 Synthetic Minority Oversampling Technique (SMOTE)

SMOTE handles class imbalance by changing the frequency of different classes of the training data [Cha02]. The algorithm's name is short for "Synthetic Minority Oversampling Technique". When applied to data, SMOTE sub-samples the majority class (i.e., deletes some examples) while super-sampling the minority class until all classes have the same frequency. In the case of *software defect prediction task*, the minority class is usually the defective class. If you look at Table 3.3, we have few datasets where %Defective classes are really less.

Figure 3.2 shows how SMOTE works. During super-sampling, a member of the minority class finds k nearest neighbors. It builds an artificial member of the minority class at some point in-between itself and one of its random nearest neighbors. During that process, some distance function is required which is the *minkowski_distance* function.

SMOTE's control parameters are (a) k that selects how many neighbors to use (defaults to $k = 5$), (b) m is how many examples of each class which need to be generated (defaults to $m = 50\%$ of the total training samples), and (3) r which selects the distance function (default is $r = 2$, i.e., use Euclidean distance).

Table 3.13 Traditional Machine Learning Algorithms

LEARNER	NOTES
RF= Random Forest	Random forest is an ensemble algorithm using entropy-based decision trees. RF uses multiple decision trees to obtain better predictive performance than could be obtained from any of the constituent learning algorithms alone [Bre01]. Usually, it builds multiple weak models that are independently trained and combines the output of each weak model in some way to make the overall prediction
LR=Logistic Regression	Regression use predefined functions to model the mapping from input space to output space. Parameters of the predefined function are estimated by minimizing the error between the ground truth outputs and the function outputs. LR uses a generalized linear regression function to build the model.
KNN= K-means	KNN is a Instance-based algorithm. Instead of fitting a model on the training data, it stores all the training data in a memory. When a new test item comes, the similarities between the test item and "k" training item are measured. The test item is then classified to the class where most of its similar training items belong to.
NB= Naive Bayes	Classify a new instance by (a) collecting mean and standard deviations of attributes in old instances of different classes; (b) return the class whose attributes are statistically most similar to the new instance.
DT= Decision Trees	DTs such as CART and C4.5 seek attributes which, if we split on their ranges, most *reduces* the reduce the entropy of the class distribution.. These algorithms then recurse over each split to find further useful divisions. For numeric classes, diversity may be measured in terms of variance. For discrete classes the Gini or entropy measures can assess diversity.
SVM= Support Vector Machines	SVM are supervised learning trying to separate training items from two classes by a clear gap [SC08]. It maps the raw data into a higher-dimensional space where it is easier to distinguish the examples. For linearly non-separable problems, SVMs either allow but penalize misclassification of training items (soft-margin) or utilize kernel tricks to map input data to a higher-dimensional feature space before learning a hyper-plane to separate the two classes.

```
def SMOTE(k=2, m=50%, r=2): # defaults
  while Majority > m do
    delete any majority item # random
  while Minority < m do
    add something_like(any minority item)

def something_like(X0):
  relevant = emptySet
  k1 = 0
  while(k1++ < 20 and size(found) < k) {
      all = k1 nearest neighbors
      relevant += items in "all" of X0 class}
  Z = any of found
  Y = interpolate (X0, Z)
  return Y

def minkowski_distance(a,b,r):
  return (Σ_i abs(a_i − b_i)^r)^(1/r)
```

Figure 3.2 Pseudocode of SMOTE

In the software analytics literature, there are contradictory findings on the value of applying SMOTE for class imbalance problem. Van et al. [VH07], Pears et al. [Pea14], Tan et al. [Tan15] and Bennin et al. [Ben17] found SMOTE to be advantageous, while others, such as Pelayo et al. [PD07] did not.

Further, some researchers report that some learners respond better than others to SMOTE. Kamei et al. [Kam07] evaluated the effects of SMOTE applied to four fault-proneness models (linear discriminant analysis, logistic regression, neural network, and decision tree) by using two module sets of industry legacy software. They reported that SMOTE improved the prediction performance of the linear and logistic models, but not neural network and decision tree models. Similar results, that the value of SMOTE was dependent on the learner, was also reported by Van et al. [VH07].

Due to the past success of SMOTE, we tried tuning SMOTE (SMOTUNED described in Section 3.3.5) for our defect prediction study.

3.3.3 Latent Dirichlet allocation (LDA)

Latent Dirichlet Allocation (LDA) is a generative statistical model that allows sets of observations to be explained by unobserved groups that explain why some parts of the data are similar. It learns the various distributions (the set of topics, their associated word probabilities, the topic of each word, and the particular topic mixture of each document). What makes topic modeling interesting is that these algorithms scale to very large text corpus.

As shown in Figure 3.3, the LDA topic modelling algorithm [Ble03; Nik15] assumes D documents contain k topics expressed with w different words. Each document $d \in D$ of length N_d is modeled as a

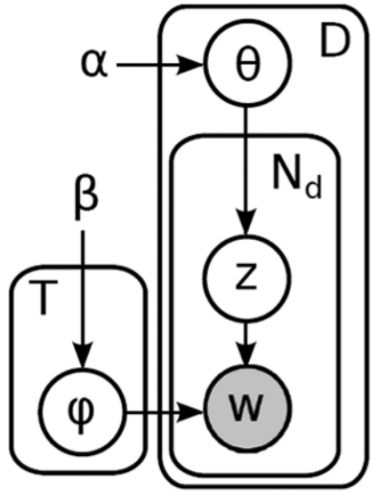

Figure 3.3 LDA

discrete distribution $\theta(d)$ over the set of topics. Each topic corresponds to a multinomial distribution over the words. α is the discrete prior assigned to the distribution of topics vectors(θ); and β for the distributions of words in topics(ψ).

As shown in Figure 3.3, the outer plate spans documents and the inner plate spans word instances in each document (so the w node denotes the observed word at the instance and the z node denotes its topic). The inference problem in LDA is to find hidden topic variables z, a vector spanning all instances of all words in the dataset. LDA is a problem of Bayesian inference. The original method used is a variational Bayes approximation of the posterior distribution [Ble03] and alternative inference techniques use Gibbs sampling [GS04] and expectation propagation [ML02].

Table 3.14 illustrates an example of topic generation from PitsA dataset. To build topic model from *Text Mining task*, LDA has few tunable parameters:

- k, number of topics.
- $\alpha = P(k|d)$, probability of topic k in document d;
- $\beta = P(w|k)$, probability of word w in topic k.

Initially, α and β may be set randomly as follows: each word in a document was generated by first randomly picking a topic (from the documentâĂŹs distribution of topics) and then randomly picking a word (from the topicâĂŹs distribution of words). Successive iterations of the algorithm count the implications of prior sampling which, in turn, incrementally updates α and β.

Binkley et al. [Bin14] performed an extensive study and found that apart from α and β, the other parameters that define LDA are:

Table 3.14 Top 10 topics found by LDA for PitsA dataset. Within each topic, the weight for words decreases exponentially left to right across the order shown here. The words here are truncated (e.g. "software" becomes "softwar") due to stemming.

Topics=	Top words in topic
01=	command engcntrl section spacecraft unit icd tabl point referenc indic
02=	softwar command test flight srobc srup memori script telemetri link
03=	file variabl line defin messag code macro initi use redund
04=	file includ section obc issu fsw code number matrix src
05=	mode safe control state error power attitud obc reset boot
06=	function eeprom send non uplink srup control load chang support
07=	valu function cmd return list ptr curr tss line code
08=	tabl command valu data tlm load rang line count type
09=	flight sequenc link capabl spacecraft softwar provid time srvml trace
10=	line messag locat column access symbol file referenc code bld

- b = number of burn-in iterations;
- si = the sampling interval.

Binkley et al.'s study of the LDA settings was a mostly manual process guided by their considerable background knowledge and expertise and program comprehension. But their method was not automatic, so we proposed an automatic method like LDADE (described in Section 3.3.6). LDADE tuned only k, α, and β because in the past, these parameters have shown to be the most important.

In an ICSE'13 article, Panichella et al. [Pan13] used a genetic algorithm to tune their LDA text miners. More recently, in a IST'18 journal paper, Agrawal et al. [Agr18b] found that differential evolution can out-perform genetic algorithms for tuning LDA.

A standard feature transformation technique for text mining is *vectorization*; i.e. replace the raw observations that wordX appears in documentY with some more informative statistic. For example, LDA converts the textual data of PitsA dataset into the vectors of Table 3.15. The cells in that table shows how much each issue report matches each topic (and the final column shows the issue severity of that report). Other vectorization methods are TF, Hashing, TFIDF. Note that once textual data has been converted to short vectors like Table 3.15, then standard classifiers can be applied to text mining classification tasks.

3.3.4 Differential Evolution (DE)

Different evolution (DE) execute over multiple generations. Generation zero is usually initialized at random. After that, in each generation, candidate items are subject to *select* (prune away the less interesting solutions), *mutate* (build new items by combining with 3 other random candidates from the same generation) [SP97]. Differential Evolutionary algorithm is described in Figure 3.4.

Table 3.15 Document Topic distribution found by LDA for PitsA dataset

Issue report	10 Topics										Severe?
01	.60	.10	.00	.15	.00	.05	.03	.04	.03	.00	y
02	.10	.03	.02	.00	.03	.02	.15	.65	.00	.00	n
03	.00	.20	.05	.05	.00	.60	.02	.03	.03	.02	n
04	.03	.01	.01	.10	.15	.00	.70	.00	.00	.00	y
etc											

INPUT:

- A dataset, such as Table 3.3;
- A tuning goal G (e.g. D2h or Popt20);
- DE parameters: $np = 10$, $f = 0.7$, $cr = 0.3$, $lives = 5$

OUTPUT:

- Best tunings for a learner, found by DE.

PROCEDURE:

- Separate the data into *train* and *tune*;
- Given x options, generate $x \times np$ tunings as the initial population P;
- Score each tuning P_i in the population with goal G;
- While $lives > 0$
  1. Set $\delta = -1$; i.e. we will lose a life (unless we find improvements);
  2. For $i \in P$ do
     (a) Make a mutant m by extrapolating between $\{a, b, c\} \in P$ at probability cr. For decisions $m_k \in m$:
        i. $m_k = a_k + f * (b_k - c_k)$ (continuous values).
        ii. $m_k = a_k \vee (b_k \vee c_k)$ (discrete values).
     (b) Build a learner with parameters m and train data;
     (c) Using the data and G, score that learner;
     (d) If m is preferred to i, then replace i with m in P and set $\delta = 0$ (i.e. do not lose a life).
  3. $lives = lives + \delta$

Figure 3.4 Storn's Differential Evolution algorithm [SP97; Agr18b; AM18].

We choose to use DE after a literature search on search-based SE methods. The literature mentions many optimizers: simulated annealing [FM02; Men07c]; various genetic algorithms [Gol79] augmented by techniques such as DE (differential evolution [SP97]), tabu search and scatter search [GM86; Bea06; Mol07; Neb08]; particle swarm optimization [Pan08]; numerous decomposition approaches that use heuristics to decompose the total space into small problems, then apply a response surface methods [Kra15; Zul13]. Of these, we use DE for two reasons. Firstly, it has been

proven useful in prior SE tuning studies [Omr05; Chi12; Fu16a; FM17]. Secondly, our reading of the current literature is that there are many advocates for differential evolution like Vesterstrom et al. [VT04] showed DE to be competitive with particle swarm optimization and other GAs.

3.3.5 SMOTUNED = auto-tuning SMOTE

We designed SMOTUNED, which is an auto-tuning version of SMOTE for *Defect Prediction* task [AM18]. SMOTUNED uses different control parameters for different data sets. SMOTUNED uses DE (differential evolution [SP97] as described earlier to explore the parameter space of Table 3.4.

From Figure 3.4, DE evolves a *frontier* of candidates from an initial population which is driven by a goal (like maximizing Popt20). In the case of SMOTUNED, each candidate is a randomly selected value for SMOTE's k, m and r parameters. To evolve the frontier, within each generation, DE compares each item to a *new* candidate generated by combining three other frontier items (and better *new* candidates replace older items). To compare them, the new parameter settings candidate is evaluated using Figure 3.2 SMOTE method. This pre-processed training data is then fed into a classifier (one of them from 3.5) to find a particular measure (like Popt20). When our DE terminates, it returns the best candidate ever seen in the entire run.

In our ICSE paper [AM18], we showed how SMOTUNED beats other recent state of the art Defect Prediction methods like MAHAKIL [Ben17] and more.

3.3.6 LDADE = auto-tuning LDA

LDA suffers from "order effects". Langley [Gen89] defines such effects as follows:

> *A learner L exhibits an order effect on a training set T if there exist two or more orders of T for which L produces different knowledge structures.*

Many learners exhibit order effects, e.g., certain incremental clustering algorithms generate different clusters, depending on the order with which they explore the data [Gen89]. Hence, some algorithms survey the space of possible models across numerous random divisions of the data (e.g., Random Forests [Bre01]). DE (differential evolution) was used to tame the instability in LDA.

LDADE is a combination of topic modeling (with LDA) and an optimizer (differential evolution, or DE) that adjusts the parameters of LDA in order to optimize (i.e., maximize) similarity scores for a *Text Mining* task [Agr18b]. LDADE adjusts the parameters of Table 3.16. Most of these parameters were explained above.

From Figure 3.4, DE evolves a *frontier* of candidates from an initial population which is driven by a goal like maximizing Raw Score (for more details on the score please look at Agrawal et al. work [Agr18b]). In the case of LDADE, each candidate is a randomly selected value for LDADE's k, α and β parameters. To evolve the frontier, within each generation, DE compares each item to

Table 3.16 List of LDA parameters tuned by DE

Parameters	Defaults	Tuning Range	Description
k	10	[10,100]	Number of topics or cluster size
α	None	[0,1]	Prior of document topic distribution. This is called alpha
β	None	[0,1]	Prior of topic word distribution. This is called beta

a *new* candidate generated by combining three other frontier items (and better *new* candidates replace older items). To compare them, the new parameter settings candidate is evaluated using Raw score generated by LDA method. When our DE terminates, it returns the best candidate ever seen in the entire run. After the best parameter found, the pre-processed training data is then fed into a classifier like Support Vector Machine (SVM) to find a particular measure (like D2h).

In our IST Journal paper [Agr18b], we showed how LDADE beats other recent state of the art Text Mining methods like TF-IDF and LDA-GA [Pan13] methods and more.

3.3.7 LDA-GA

The GA search starts with a random population of solutions, where each individual from the population represents a solution of the optimization problem. The population evolves through subsequent generations and, during each generation, the individuals are evaluated based on the fitness function (maximizing Popt20 or minimizing D2h) that has to be optimized. For creating the next generation, new individuals are generated by (i) applying a *selection* operator, which is based on the fitness function, for the individuals to be reproduced, (ii) recombining, with a given probability, two individuals from the current generation using the *crossover* operator, and (iii) modifying, with a given probability, individuals using the *mutation* operator. More details about GA can be found in a book by Goldberg [GH88].

Panichella et al. [Pan13] used simple GA with elitism of two individuals (i.e., the two best individuals are kept alive across generations). We replicated this study for our use case of *Text Mining* task. An individual is a particular LDA configuration (k, α and β) and the population is represented by a set of different LDA configurations. The selection operator is the Roulette wheel selection, which assigns to the individuals with higher fitness a higher chances to be selected. The crossover operator is the arithmetic crossover, that creates new individuals by performing a linear combination with random coefficients of the two parents. The mutation operator is the uniform mutation, which randomly changes one of the genes (i.e., one of the 3 LDA parameter values) of an individual, with a different parameter value within a specified range.

INPUT:

- A dataset
- A goal predicate p; e.g., P_{opt} or *dis2Heaven*;
- M, N= number of models, number of ranges used per model

OUTPUT:

- Score of the best model when applied to data not used for training.

PROCEDURE:

- Separate the data into train and test;
- On the train data, build an ensemble and select the best:
  - For $i = 1$ to M do
    1. Divide numeric attributes into ranges;
    2. Find N *extreme* ranges that score highest and lowest on p;
    3. Combine some the extreme ranges into model i;
    4. Score i using p;
    5. Keep the best scoring model.
- On the test data:
  - Return the p score of the best scoring model.

NOTES:

- For training step (2), we use *extreme* ranges in order to maximize the spread of the darts around the results space.
- To keep this simple, the discretizer used in training step (1) just divides the numeric data on its median value.

Figure 3.5 FFT: an ensemble algorithm to sample results space, M number of times.

3.3.8 FFtrees

Psychological scientists have developed FFTs (Fast and Frugal Trees) as one way to generate comprehensible models consisting of separate tiny rules [Phi17; Che18; Mar08]. A FFT is a decision tree made for binary classification problem with exactly two branches extending from each node, where either one or both branches is an exit branch leading to a leaf [Mar08]. That is to say, in an FFT, every question posed by a node will trigger an immediate decision (so humans can read every leaf node as a separate rule). How FFT is built is described in Figure 3.5.

For example, Figure 3.6 shows an FFTtree generated from the log4j JAVA system of Table 3.3. The goal of this tree is to classify a software module as “defective=1” or “non-defective=0”. The four nodes in the Figure 3.6, reference four attributes *cbo, rfc, dam, amc* (defined in Table 3.2).

```
1. if cob <= 4          then 0
2. else if rfc > 32     then 1
3. else if dam > 0      then 1
4. else if amc < 32.25  then 1
5. else 0
```

Figure 3.6 A simple model for software defect prediction

We used the similar implementation of FFT as offered by Fu and Chen et al. [Fu18; Che18]. An FFT of depth d has a choice of two "exit policies" at each level: the existing branch can select for the negation of the target, i.e., non-severe, (denoted "0") or the target (denoted "1"), i.e., severe. The right-hand-side tree in Figure 3.6 is 01110 since:

- The first level found a rule that exits to the negation of the target: hence, "0".
- While the next tree levels found rules that exit first to target; hence, "111".
- And the final line of the model exits to the opposite of the penultimate line; hence, the final "0".

Following the advice of [Fu18; Che18; Phi17], for all the experiments of this paper, we use a depth $d = 4$. For trees of depth $d = 4$, there are $2^4 = 16$ possible trees which can be denoted as 00001, 00010, 00101, ... , 11110. During FFT training, all 2^d trees are generated, then we select the best one (using the training data). This single best tree is then applied to the test data. Note that FFTs of such small depths are very succinct (see example in Figure 3.6).

Standard rule learners select ranges that best select for some goal (e.g., selecting for the "1" examples). This can lead to overfitting. To avoid overfitting, FFtrees use a somewhat unique strategy: at each level of the tree, FFtrees builds two trees using the ranges that *most* and *least* satisfy some goal; e.g., *d2h* or *Popt20*. That is, half the time, FFtrees will try to avoid the target class by building a leaf node that exits to "0". Assuming a maximum tree depth of $d = 4$ and two choices at each level, then FFtree builds $2^d = 16$ trees then prunes away all but one, as follows, firstly, it selects a goal predicate; e.g., *d2h* or *Popt20*. Next, while *building one tree*, at each level of the tree, FFtree scores each range according to how well that range {does, does not} satisfy that goal. These selected range becomes a leaf note. FFtree then calls itself recursively on all examples that do not fall into that range. Finally, while *assessing 16 trees*, we run the training data through each tree to find what examples are selected by that tree. Each tree is scored by passing the selected examples through the goal predicate. The tree with the best score is then applied to the test data.

In summary, FFtrees *explore around* a few dozen times, trying different options for how to best model the data (i.e., what exit node to use at each level of the tree). After a few *explorations*, FFtrees deletes the worst models, and uses the remaining model on the test data.

3.3.9 FLASH

FLASH, a Sequential Model-based Optimization (SMBO), is a useful strategy to find extremes of an unknown objective [Nai18]. FLASH is efficient because of its ability to incorporate prior belief as already measured solutions (or configurations), to help direct further sampling. Here, the prior represents the already known areas of the search (or performance optimization) problem. The prior can be used to estimate the rest of the points (or unevaluated configurations). Once one (or many) points are evaluated based on the prior, the posterior can be defined. The posterior captures the updated belief in the objective function. This step is performed by using a machine learning model, also called surrogate model. Here, FLASH use a decision tree as a surrogate model. The concept of FLASH can be simply stated as: Given what one knows about the problem and what can be done next?

The "given what one knows about the problem" part is achieved by using a machine learning model whereas "what can be done next" is performed by an acquisition function. Such acquisition function automatically adjusts the *exploration* ("should we sample in uncertain parts of the search space") and *exploitation* ("should we stick to what is already known") behavior of the method.

To tune a data miner, FLASH explores N possible tunings as follows:

1. Set the evaluation budget b. In order to make a fair comparison between FLASH and other methods, we used $b = 200$.

2. Run the data miner using $n = 20$ randomly selected tunings.

3. Build an *archive* of n examples holding pairs of parameter settings and their resulting performance scores (e.g. MRE, SA, etc).

4. Using that archive, learn a *surrogate* to predicts performance. Following the methods of Nair et al. [Nai18], we used decision tree for that surrogate.

5. Use the surrogate to guess M performance scores where $M < N$ and $M \gg n$ parameter settings. Note that this step is very fast since it all that is required is to run M vectors down some very small CART trees.

6. Using some *selection function*, select the most "interesting" setting. After Nair et al. [Nai18] we returned the setting with the nest prediction (i.e. find the most promising possibility).

7. Collect performance scores by evaluating "interesting" using the data miners (i.e. check the most troubling possibility). Set $b = b - 1$.

8. Add "interesting" to the archive. If $b > 0$, goto step 4. Else, halt.

In summary, given what we already know about the tunings (represented in a decision tree tree), FLASH finds the potentially best thing (in Step6); then checks that thing (in Step7); then updates the model with the results of that check.

Here, we used FLASH to tune a Decision Tree and XGBoost [CG16] classifier for *dis2heaven* goal. We used decision tree [Gho15] and XGBoost [Tan16] to optimize as they have shown to have the maximum improvement after tuning. FLASH was initialized with a random initial pool size of 12, and pool size of 10000 with a budget of 30. Initial pool size is to build a surrogate model against to predict on the pool size, as to pick the next best optimal parameters. It is terminated after a budget of 30 similar to how many evaluations were taken by **DODGE(ϵ)**.

3.3.10 DODGE(ϵ)

Deb's principle [Deb05] of "ϵ-dominance" states that if there exists some ϵ value below which it is useless or impossible to distinguish results, then it is superfluous to explore anything less than ϵ.

We say that for "large ϵ problems", the results space of learning effectively contains just a few regions. In such simple result spaces, a few random samples thrown around the output space would sample the results just as well, or better, than more complex methods which are expensive in runtime. We also know that uncertainty is an inherent property of any machine learning algorithms for e.g., "order effects" seen in LDA due to just random input order of data [Agr18b] or sampling of training data introduces uncertainty. That is why we are saying, ***Instead of striving to make*** $\epsilon = 0$***, use*** $\epsilon > 0$ ***as a tool for simplifying machine learning algorithms***.

From the above, we assert that predictors result collected on the same data will vary by some amount ϵ. As mentioned earlier, Deb's principle of ϵ-dominance [Deb05] states that if there exists some ϵ value below which is useless or impossible to distinguish results, then it is superfluous to explore anything less than ϵ.

Note that ϵ effectively clusters the space of possible results. Going back to the same example introduced in Chapter 1 of Figure 1.1, if $\epsilon = 0.2$, then that divides a $p = 2$ dimensional output space of a learner into $1/(0.2^2) = 25$ cells. This means that when $\epsilon = 0.2$, then (a) *recall-vs-false alarm* results space is effectively just the ten green cells of Figure 1.1 , and effectively giving just ten cells if we avoid the red and grey regions; and (b) "many roads lead to Rome" (i.e., if the results of 100 learners were places on this grid, then there could never be more than 10 groups of results).

Similarly, $\epsilon = 0.05, 0.1, 0.2$ could take many values, it is about choosing the right set of ϵ against which we know how many samples are needed. **DODGE(ϵ)** tries to exploit these different set of ϵ values to understand the nature of SE data. This means that we are looking at number of samples required before performance plateaus on the time required to configure parameters of data miners. If ϵ-dominance holds true then we should get plateau at fairly low number of samples.

3.3.10.1 Number of Evaluations Required

Given a $p = 2$ dimensional output space of a learner as in Figure 1.1, and ϵ, we know from binomial distribution, we can put upper and lower bounds on number of samples needed for the given ϵ.

For **lower bound**, we can say that we need to follow, Sampling without replacement, that means 1 sample picks 1 cell in that grid and so on. Sampling without replacement says

$$\binom{N}{n} = \frac{N!}{N-n!} \tag{3.6}$$

The above equation, 1) for $\epsilon = 0.2$ will need 25 samples to cover the whole space; 2) for $\epsilon = 0.1$ will need 100 samples to cover the whole space; 3) for $\epsilon = 0.05$ will need 400 samples to cover the whole space. These number of lower bound samples are shown in Table 3.17.

Now when we consider **upper bound**, we can say that we need to follow, Sampling with replacement which says

$$P(s) = \epsilon^2, \;\; P(s') = 1 - \epsilon^2 \tag{3.7}$$

Here P(s) is the probability needed to pick 1 cell of that grid. With known confidence (c) of reaching a goal after n samples will need,

$$c = 1 - (1 - \epsilon^2)^n, \;\; n = \frac{log(1-c)}{log(1-\epsilon^2)} \tag{3.8}$$

Based on the above equation, the upper bound of number of samples needed are shown in Table 3.17. We expect that if ϵ-dominance holds true for SE data, then we will see number of samples to be within these upper and lower bounds.

Table 3.17 Lower and Upper Bound Samples Needed.

	Lower Bounds			Upper Bounds		
Confidence/ϵ	**0.05**	**0.1**	**0.2**	**0.05**	**0.1**	**0.2**
0.9	400	100	25	459	114	28
0.95	400	100	25	598	148	36
0.99	400	100	25	919	228	55

3.3.10.2 Algorithm

If our data has large ϵ variability in its output space as shown in above examples, then to do better than the old frameworks, we need to:

- Try *thrashing across* a *wider range of options*.
- If some options result in a performance score α, then we will *deprecate options* that lead to $\alpha \pm \epsilon$.

To test that conjecture, we built **DODGE(ϵ)**. To find a *wider range of options*, **DODGE(ϵ)** uses Table 3.18 and 3.19 as a source modelling options. In this approach, learner options are sub-trees of different options. For example, Figure 3.7 shows some sub-tree options for *QuartileTransformer* + {*DecisionTree or NaiveBayes*}. In that tree, the variables a, b, c are scoped to within the pre-processor or learner.

When *evaluating* a branch, the options in that branch are used to configure, then execute, a pre-processor/learner on some test data. Such evaluations result in performance scores. In Figure 3.7, those scores are shown in bottom leaf nodes (here we show *d2h* but it could be anything else, at the discretion of the researcher or an analyst).

To *deprecate options*, **DODGE(ϵ)** initially assign a weight of $w = 0$ to all nodes in the options tree. When a branch is evaluated, if that score is more than ϵ from prior scores, then all nodes in that branch are endorsed (using $w = w + 1$). Otherwise, **DODGE(ϵ)** deprecates (using $w = w - 1$). In Figure 3.7, the green branches have $w > 0$ and the red branches have $w < 0$ and the gray branches have $w = 0$ (see top-most-right-hand-side branches from *Quartile Transformer* down to *c=uniform*). These gray branches arise from the sum of lower branches.

Note that if a branch contains *numeric ranges*, then (as explained in this paragraph), that branch can be evaluated multiple times and produce different scores each time. To see this, note that the blue nodes of Figure 3.7 contain numeric ranges *{lo,ho}*. When a branch is evaluated, a random number $r = random(lo, hi)$ is selected and its weight $w(r)$ is set to zero. This weight of the selected value is endorsed/deprecated as above. When a new value is required (i.e. when the branch is evaluated again) then **DODGE(ϵ)** uses a random number $b..\frac{b+w}{2}$ (where b, w are the previously selected values with highest, lowest weights).

DODGE(ϵ) uses these weights to select options via a recursive *weighted descent* where, at each level, it selects sub-trees whose root has the largest weight. Weighted descent is called after an initial "burn-in" period of N_1 evaluations:

1. **DODGE(ϵ)** selects one option, evaluates it, then weaves it into the options tree. This step is repeated N_1 times. Note that this means that N_1 times, we add a new branch to the options tree.

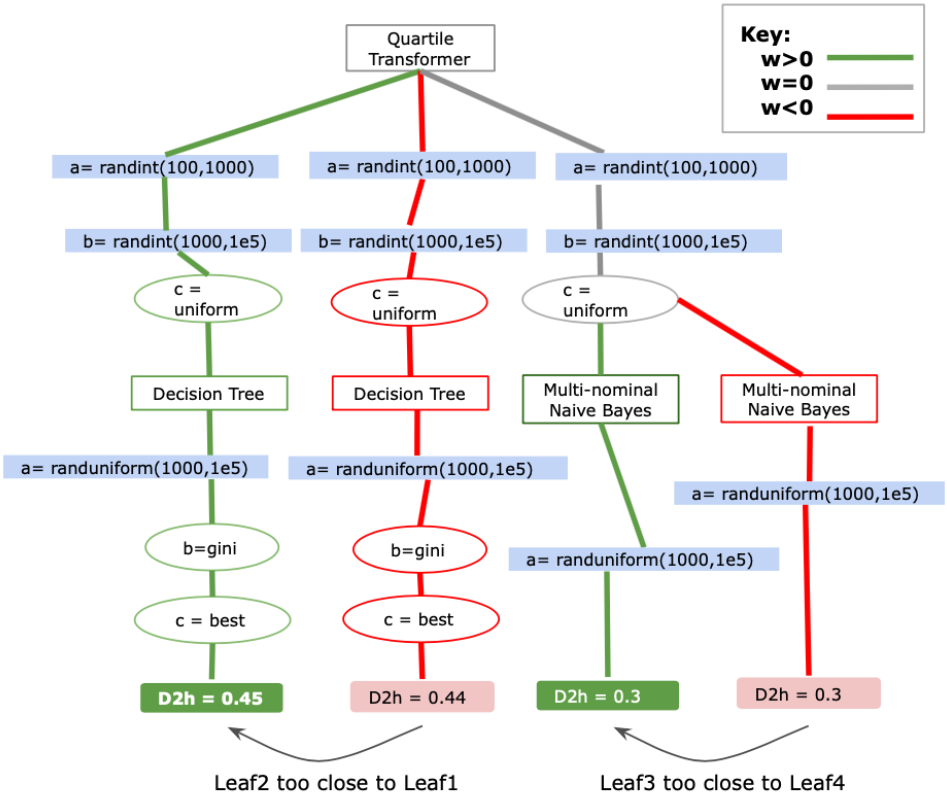

Figure 3.7 A **DODGE(.05)** options tree maintains sepertate sub-trees for the learners underneath the pre-processor branches. Note that this example tree covers all the billion of options since **DODGE(ϵ)** executes by exploring a very small number of randomly selected options of that table.

Table 3.18 Hyperparameter tuning options explored for data preprocessing techniques in this thesis. Note that we make no claim that this is a complete list of options. Rather, we merely claim that a reader of the recent SE literature on hyperparameter optimization might be tempted to try some subset of the following.

DATA PRE-PROCESSING
Software Defect Prediction, Bad Code Smell Detection, Issue Lifetime Estimation, and UCI datasets:

- **Transformations**
  - StandardScaler
  - MinMaxScaler
  - MaxAbsScaler
  - RobustScaler(quantile_range=(a, b))

    a,b= randint(0,50), randint(51,100)
  - KernelCenterer
  - QuantileTransformer(n_quantiles=a, output_distribution=c, subsample=b)

    a, b = randint(100, 1000), randint(1000, 1e5)

    c=randchoice(['normal','uniform'])
  - Normalizer(norm=a)

    a = randchoice(['l1', 'l2','max'])
  - Binarizer(threshold=a)

    a= randuniform(0,100)

Text mining:

- **Feature Extraction**
  - CountVectorizer(max_df=a, min_df=b)

    a, b = randint(100, 1000), randint(1, 10)
  - TfidfVectorizer(max_df=a, min_df=b, norm=c)

    a, b = randint(100, 1000), randint(1, 10)

    c = randchoice(['l1', 'l2', None])
  - HashingVectorizer(n_features=a, norm=b)

    a = randchoice([1000, 2000, 4000, 6000, 8000, 10000])

    b = randchoice(['l1', 'l2', None])
  - LatentDirichletAllocation(n_components=a, doc_topic_prior=b, topic_word_prior=c, learning_decay=d, learning_offset=e,batch_size=f)

    a, b, c = randint(10, 50), randuniform(0, 1), randuniform(0, 1)

    d, e, f = randuniform(0.51, 1.0), randuniform(1, 50), randchoice([150,180,210,250,300])

Table 3.19 Hyperparameter tuning options explored for all the machine learners for the tasks understudy. Note that we make no claim that this is a complete list of options. Rather, we merely claim that a reader of the recent SE literature on hyperparameter optimization might be tempted to try some subset of the following.

LEARNERS
Both SE and NON-SE tasks and data:

- DecisionTreeClassifier(criterion=b, splitter=c, min_samples_split=a)
  - a= randuniform(0.0,1.0)
  - b, c= randchoice(['gini','entropy']), randchoice(['best','random'])
- RandomForestClassifier(n_estimators=a,criterion=b, min_samples_split=c)
  - a,b = randint(50, 150), randchoice(['gini', 'entropy'])
  - c = randuniform(0.0, 1.0)
- LogisticRegression(penalty=a, tol=b, C=float(c))
  - a=randchoice(['l1','l2'])
  - b,c = randuniform(0.0,0.1), randint(1,500)
- MultinomialNB(alpha=a)
  - a= randuniform(0.0,0.1)
- KNeighborsClassifier(n_neighbors=a, weights=b, p=d, metric=c)
  - a, b = randint(2, 25), randchoice(['uniform', 'distance'])
  - c = randchoice(['minkowski','chebyshev'])
  - if c=='minkowski': d= randint(1,15) else: d=2

2. Next, N_2, times, weighted descent is used to select which parts of the options tree should be revisited and reevaluated.

3. Finally, the branch with the best performance score is past to the test data.

Note that this means that $N_1 + N_2$ times, the weights in the tree are adjusted.

Note also that $N = N_1 + N_2$. In the following, we use $N_1 = 12$ then vary N_2. Initially this was an unintentional coding mistake but since that proved to work so well (see below), we never changed it.

3.4 Summary

We described different datasets and its properties used for the 4 SE tasks and non-SE tasks. We showed how variant the attributes are, different number of samples used. Following that we introduced our evaluation goals $D2h$ and P_{opt}. We discussed different frameworks SMOTE, LDA, DE, SMOTUNED,LDADE, LDA-GA FFtrees and FLASH. SMOTE and LDA are algorithms without any HPO. DE is the optimizer used to tune SMOTE and LDA giving LDADE and SMOTUNED as a framework. LDA-GA uses Genetic algorithm introduce by Panichella et al. [Pan13]. Also, to be noted that SMOTUNED, LDADE, and FFtrees are developed by us as part of previous studies which motivated the rest of this thesis where we hypothesize that there is existence of ϵ-dominance making SE data inherently different in nature. Based on our hypothesis we also developed a framework called **DODGE(ϵ)** to utilize large ϵ variablility as a tool. How each of these frameworks are used in different SE and NON-SE tasks is described in Table 3.1

In the following chapter, we will show results for each of our research questions that were introduced in Chapter 1. At the end, we will also discussion the implication of these results and why was this observed.

CHAPTER

4

RESEARCH FINDINGS

We will answer all our research questions in this chapter. Firstly, we will begin by showing the existence of ϵ-domination. We will find that after only few samples, the performance plateaus which will verify our hypothesis of SE data being inherently different in nature. Following this we will talk about how to determine the parameters ($\{N, \epsilon\}$) of **DODGE(ϵ)** in RQ1. We will show results of **DODGE(ϵ)** compared against other frameworks for 4 SE cases studies in RQ2, RQ3, RQ4 and RQ5. RQ6 results are discussed later where we talk about NON-SE domain and the inferences found using **DODGE(ϵ)**. Lastly we will end this chapter by discussing the results and what can be taken away from this thesis.

Firstly, we wanted to verify whether our hypothesis of existence of ϵ-domination holds true or not. For this, we need to find how quickly **DODGE(ϵ)** can find the optimal performance. Please see the corresponding Figure 4.1. In this figure, X-axis represents number of samples needed. Y-axis represents the Max value of P_{opt} seen until that sample. Here *larger* values are *better*. We used ϵ of 0.05, 0.1, and 0.2 values, and look for number of samples needed before the performance plateaus. We also show the variance (which is iqr, 75th percential - 25th percentile) for those ϵ values. The variances are very small showing us that **DODGE(ϵ)** is very stable.

We observed that maximum improvement or change only happens in the beginning 10s of evaluations and after that it plateaus for any ϵ value. This proves our original hypothesis of that SE data shows ϵ-dominance, showing large variability.

Using **DODGE(ϵ)**, we can now answer 6 research questions asked in this thesis's introduction.

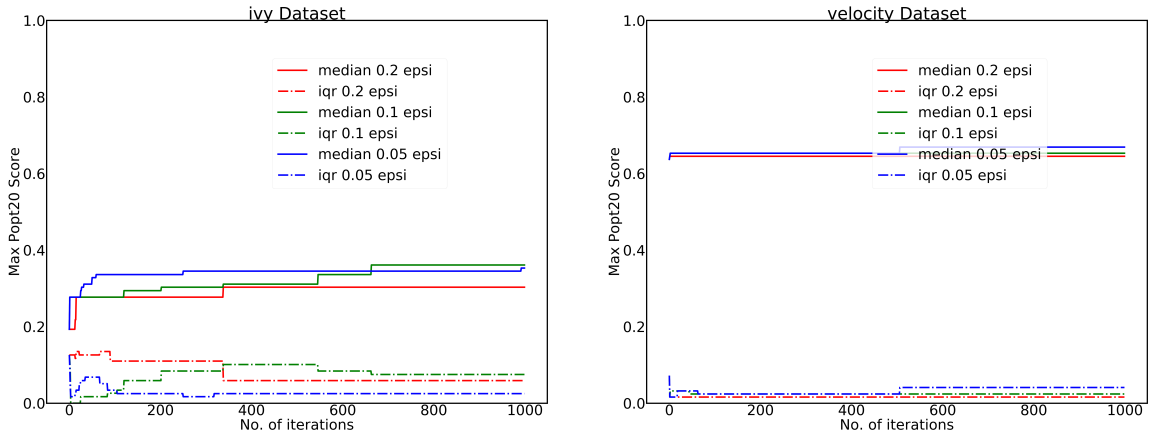

Figure 4.1 DODGE(ϵ) for P_{opt} on 2 datasets. X-axis represents number of samples needed. Y-axis represents the Max value of P_{opt} seen until that sample. Here *larger* values are *better*. For remaining dataset please see `http://tiny.cc/rq1a_tabu`. Performance plateaus only after few samples.

4.1 R1: Is DODGE(ϵ) too complicated? Is it difficult to determine its $\{N, \epsilon\}$ values?

In the thesis proposing simplifications to software analytics (where lesser evaluations define simplicity), it is important to check if the new proposed method is itself simple to apply. Accordingly, this research question asks if it is difficult to find useful values for N (the number of samples) or the ϵ value used in the search.

- Figure 4.2 varies ϵ but keeps N constant for defect prediction.
- Figure 4.3 varies N but keeps ϵ constant for defect prediction.
- Figure 4.4 varies N but keeps ϵ constant as well as it varies ϵ and keeps N constant for text mining.

As shown those figures, these changes to $\{N, \epsilon\}$ alter the performance of ϵ by less than a "small effect". That is, (a) the output performance space for this data falls into a very small number of regions so (b) a large number of samples across a fine-grained division of the output space performs just as well as a few samples over a coarse-grained division.

In summary, our answer to **RQ1** is that, $\{N, \epsilon\}$ can be set very easily.

Based on the results of Figures 4.2, 4.3 and 4.4, for the rest of this thesis we will use $\mathscr{E} = 0.2$ while taking $N = 30$ samples of the options tree. This shows that the SE data is inherently different in nature by showing the existence of ϵ-dominance. SE data contains large ϵ variability which can be utilized as a tool to simplify an optimizer.

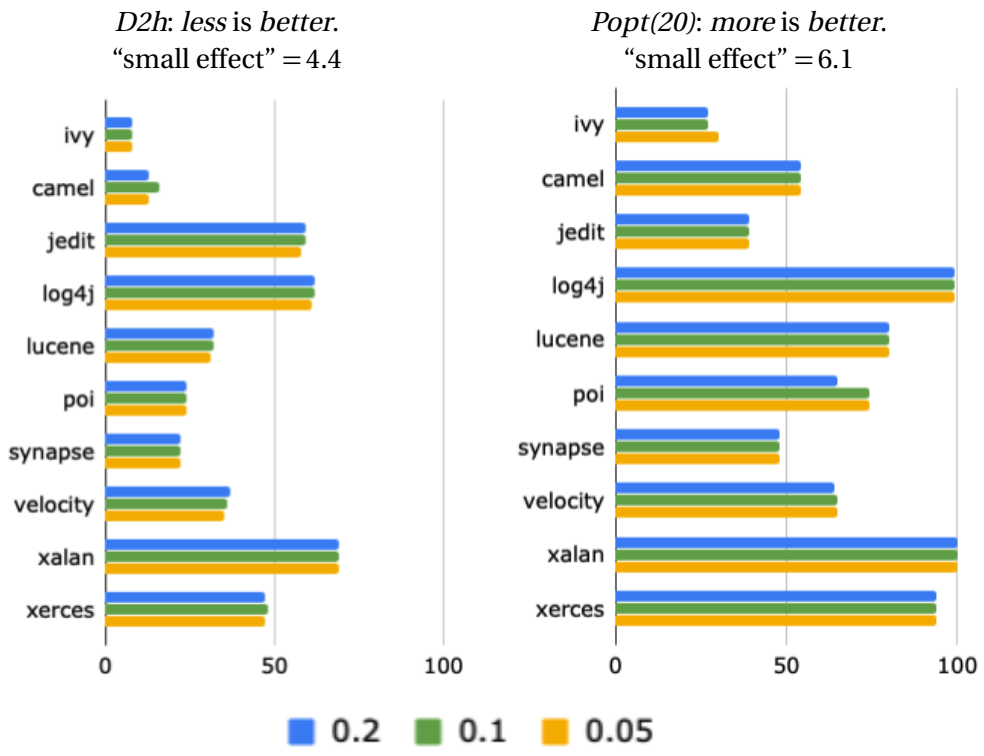

Figure 4.2 Defect prediction results using **DODGE**($\epsilon \in \{0.2, 0.1, 0.05\}$) while keeping the number of samples constant at $N = 30$. As before, changing ϵ does not change learner performance any more than a "small effect". This figure was generated using the same experimental set up as Figure 4.3.

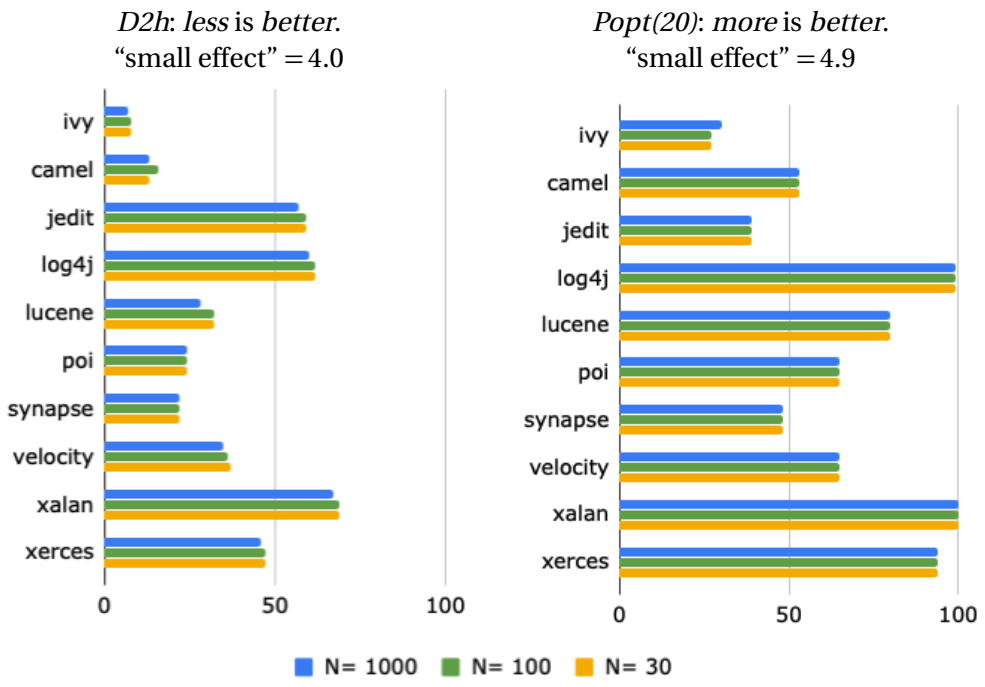

Figure 4.3 Defect prediction results using **DODGE(.2)** while varying number of samples N. Note that for any data set, all these results are very similar; i.e. changing the sample size does not change learner performance any more than a "small effect".

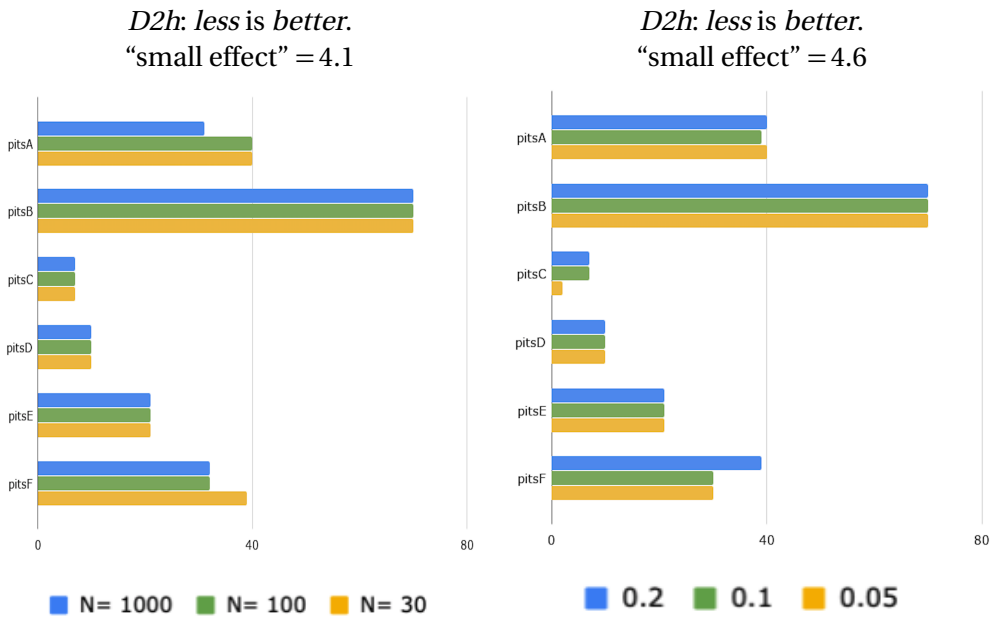

Figure 4.4 Text Mining results using **DODGE(.2)** for constant N and varying ϵ as well as varying N and constant ϵ. Note that for any data set, all these results are very similar; i.e. changing the sample size does not change learner performance any more than a "small effect".

4.2 RQ2: How does DODGE(ϵ) compare to recent prominent defect prediction and hyperparamter optimization results?

In this, we will be comparing our 3 frameworks such as **DODGE(ϵ)**, FFtree, SMOTUNED, FLASH against 2 more methods which are DE+RF (Differential evolution used to tune the parameters of Random forest classifier) and random selection of choices.

DE+RF is a hyperparameter optimizer developed by Fu et al., for an IST'16 journal paper [Fu16b]. It uses differential evolution to tune the control parameters of random forests The premise of RF (which is short for random forests) is "if one tree is useful, why not a hundred?". RF quickly builds many trees, each time using a random selection of the attributes and examples. The final conclusion is then generated by polling across all the trees in the forest. RF's control parameters are listed in Table 3.19.

Figure 4.5 compares these hyperparameter optimizers with **DODGE(.2)**, FFtrees and (just for completeness) a random search method that, 30 times, just picks options at random from Tables 3.18 and 3.19.

- Usually, random performs badly and never defeats **DODGE(ϵ)**. This result tells us that the re-weighting scheme within **DODGE(ϵ)** is useful.
- In 16/20 cases, after discounting "small effects", **DODGE(.2)** is no worse that anything else.
- In the remaining examples, in two cases, **DODGE(.2)** is beaten by FFtrees (see the *d2h* results for jedit and log4j). That is, in 90% of these results, methods that thrash a little around the results space do no do worse than methods that try to extensively explore the space of uncertainty.

We also compared **DODGE(ϵ)** against FLASH which tuned decision tree as well as XGBoost classifier. Figure 4.6 shows the comparison where x-axis represents the D2h value and y-axis represents the datasets. It can be seen that **DODGE(ϵ)** always performs better or similar in both the comparisons for all datasets.

As said in *Chapter 1*, we conjecture that **DODGE(.2)** performs best since the output space divides into just a few regions, so a limited number of samples are enough to explore the space of possible outcomes. Standard optimizers like DE, on the other hand, perform worse since they struggle to cover billions of tuning options (most of which yield indistinguishably different results).

In summary, our answer to **RQ2** is that **DODGE(ϵ)** compares very well to prominent defect prediction and HPOs results.

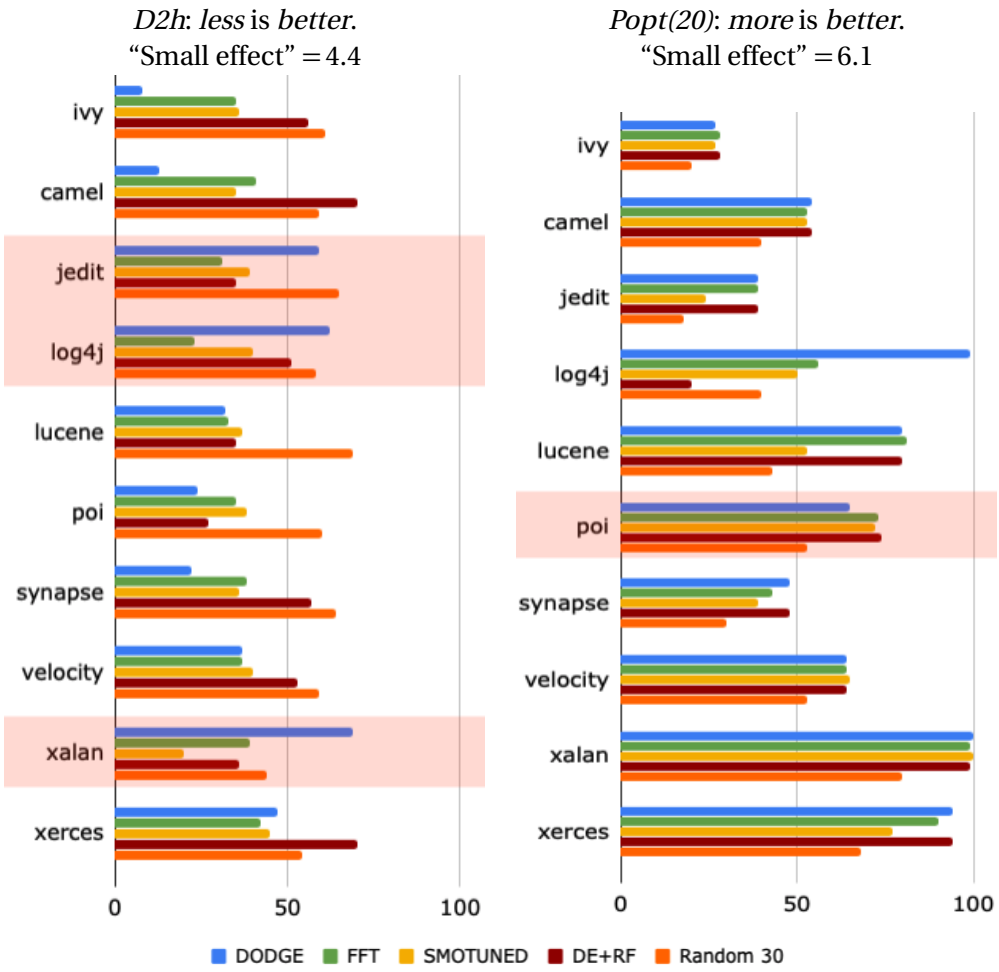

Figure 4.5 Defect prediction results for **DODGE(.2)**, $N = 30$ vs (FFtrees, SMOTUNED, DE+RF, RANDOM). In only a few cases (those highlighted in red) is **DODGE(.2)**'s performance worse than a "small effect" of anything else.

4.3 RQ3: How does DODGE(ϵ) compare to recent prominent text mining and hyperparamter optimization results?

Figure 4.7 shows our text mining results. These divide into several groups. The first group does no hyperparameter optimization. This group includes LDA-FFT and LDA-SVM and uses LDA to vectorize the data, then applies either the FFTree or the SVM classifier. Here, we use LDA-SVM since that was found useful in prior studies [Kri16]. Also, we use FFT since this is analogous to the defect prediction study discussed above.

The second group, tunes the vectorization method (LDA) but not the learner. This group includes

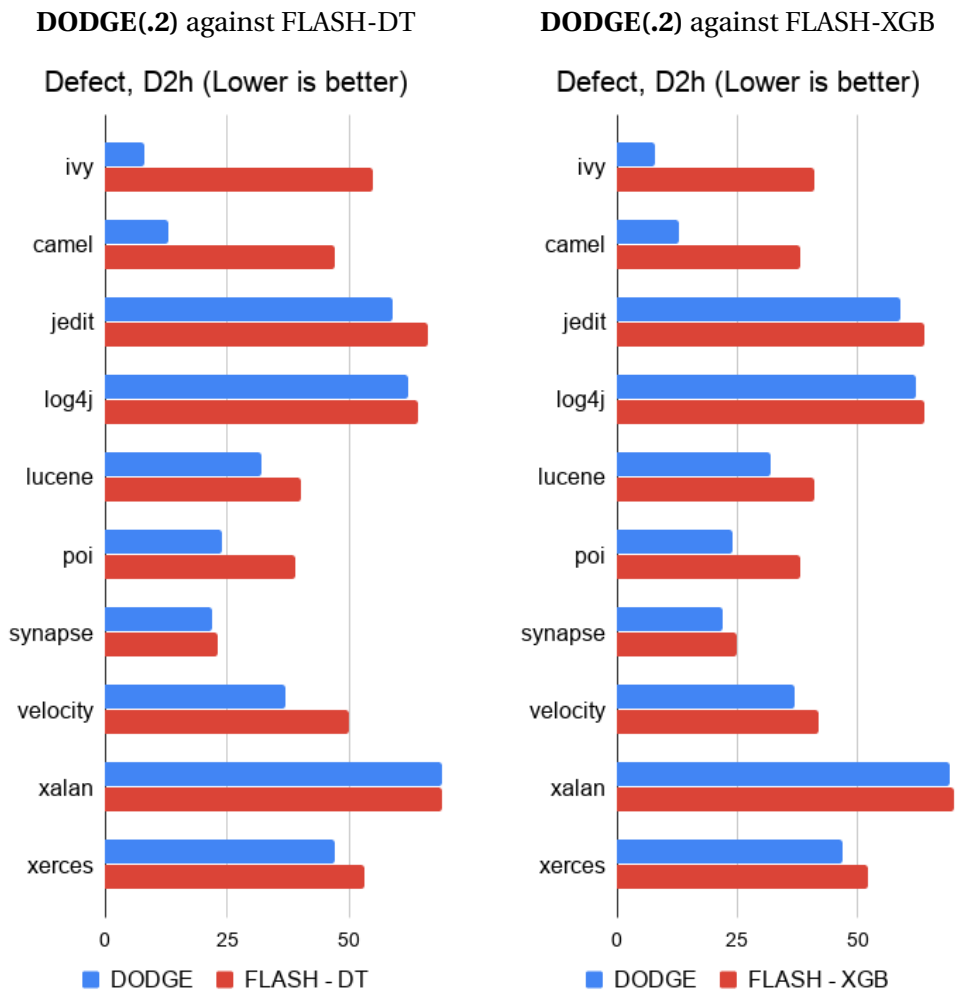

Figure 4.6 Defect prediction results for **DODGE(.2)**, $N = 30$ vs FLASH.

LDADE-FFT, LDADE-SVM and LDA-GA-SVM. In this group, before LDA is used for vectorization, then it is tuned by either DE (our LDADE) or a genetic algorithm (as done by [Pan13]).

The third Group3, which contains **DODGE**(ϵ) and RANDOM tunes both the pre-processor and learner. RANDOM is included, just for completeness. As to **DODGE**(ϵ), we use $N = 30$ samples and $\epsilon = 0.2$. In those results, **DODGE**(ϵ) was free to apply any learner or pre-processing or vectorization procedure of Tables 3.18 and 3.19.

As seen in Figure 4.7, in only one case (PitsB) was anything better than **DODGE**(ϵ). That is, hyperparameter optimization of **DODGE**(ϵ) was usually better than anything else. And, just as with the Figure 4.5 results, when **DODGE**(ϵ) fails, it is beaten by a treatment that uses FFtree (see the PitsB LDA-FFT results). That is, in 100% of these results, methods that thrash a little around the

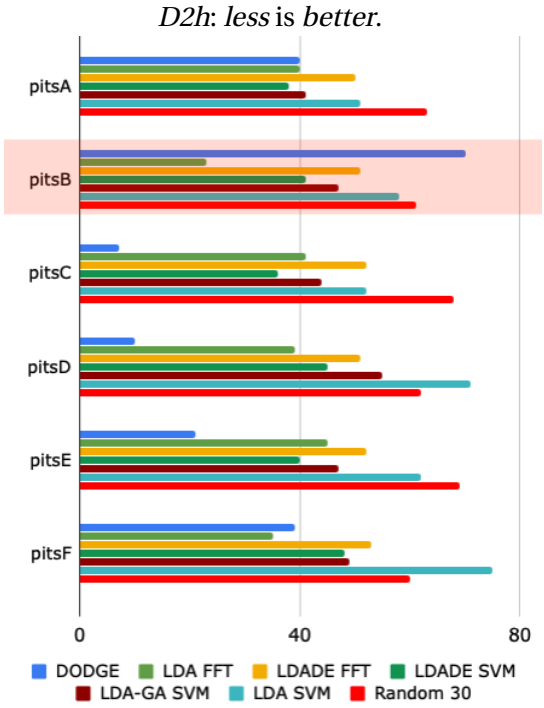

Figure 4.7 Median text mining prediction results using **DODGE(.2)** and $N = 30$ (results seen in 25 repeats of a cross-validation study). In only one case (PitsB) is FFtree's performance worse than anything else. Here, we used Efron's 95% confidence bootstrap procedure [ET93] (to demonstrate significant differences), then the A12 effect size test [AB11] (to demonstrate that the observed delta is bigger than a "small effect").

results space do no worse than methods that try to extensively explore the space of tuning options (e.g., genetic algorithms or differential evolution).

In summary, our answer to **RQ3** is that **DODGE(ϵ)** works well for text mining task making it an interesting candidate for further experimentation with other SE tasks.

4.4 RQ4: How does DODGE(ϵ) compare to recent prominent bad code smell detection and hyperparamter optimization results?

Figure 4.8 shows our bad smell detection results. We compared 3 frameworks, traditional machine learning algorithms (DT, RF, LR, kNN), FFtree and **DODGE(.2)** with N=30. In this figure, x-axis represents different datasets and y-axis represents d2h (where less value is better).

As seen in figure 4.8, **DODGE(ϵ)** performed statistically better (using Efron [ET93] and A12 effect size test [AB11]) and significant than all the other methods against all 4 datasets. That is, hyperparameter optimization using **DODGE(ϵ)** is better than anything else. In summary, our answer

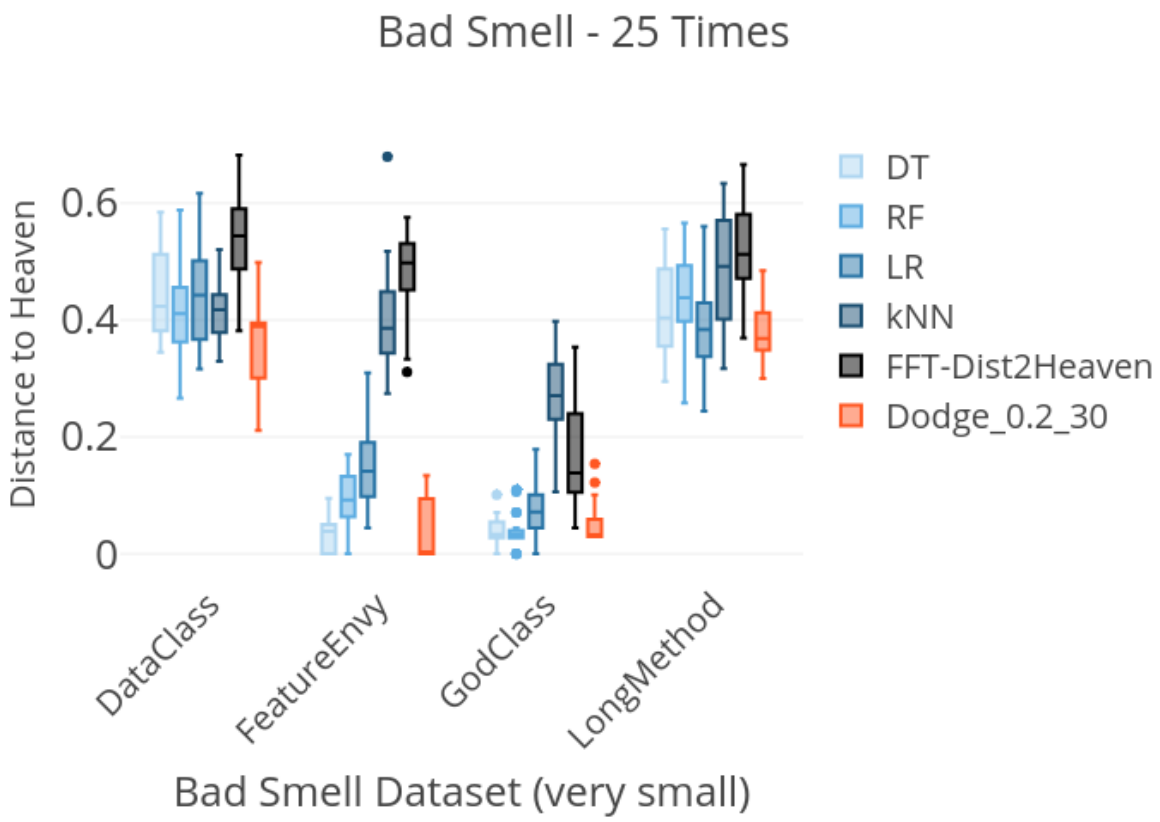

Figure 4.8 Box plot bad smell prediction results using **DODGE(.2)** and $N = 30$ (results seen in 25 repeats of a cross-validation study). Here, we used Efron's 95% confidence bootstrap procedure [ET93] (to demonstrate significant differences), then the A12 effect size test [AB11] (to demonstrate that the observed delta is bigger than a "small effect").

to **RQ4** is that **DODGE(ϵ)** works well for Bad code smell detection task.

4.5 RQ5: How does DODGE(ϵ) compare to recent prominent issue lifetime prediction and hyperparamter optimization results?

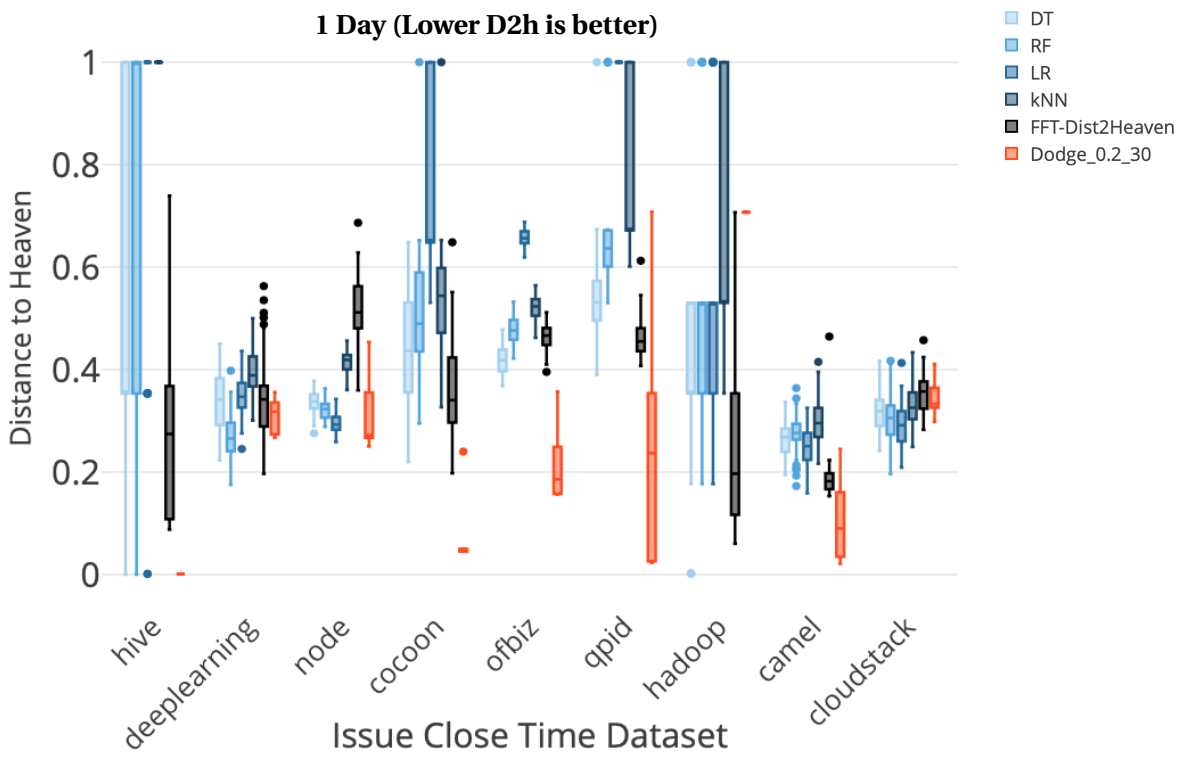

Figure 4.9 RQ5: 1 Day, Issue Lifetime Prediction Box plot comparison using **DODGE(.2)** and $N = 30$ (results seen in 25 repeats of a cross-validation study). Here, we used Efron's 95% confidence bootstrap procedure [ET93] (to demonstrate significant differences), then the A12 effect size test [AB11] (to demonstrate that the observed delta is bigger than a "small effect".

Figures 4.9, 4.10, 4.11, 4.12, 4.13, 4.14 and 4.15 represent all the frameworks comparison for issue lifetime prediction datasets for different classifications which are 1 day, 7, 14, 30, 90, 180, and 365 days respectively. X-axis represents all datasets and y-axis represents *D2h* values. Lower the *D2h* value is better. Legend shows different frameworks understudy. Lets consider an example from Figure 4.9 for 1 day classification. In that we can see that **DODGE(.2)** is statistically significant better or similar in about 6/9 datasets (about 66% times).

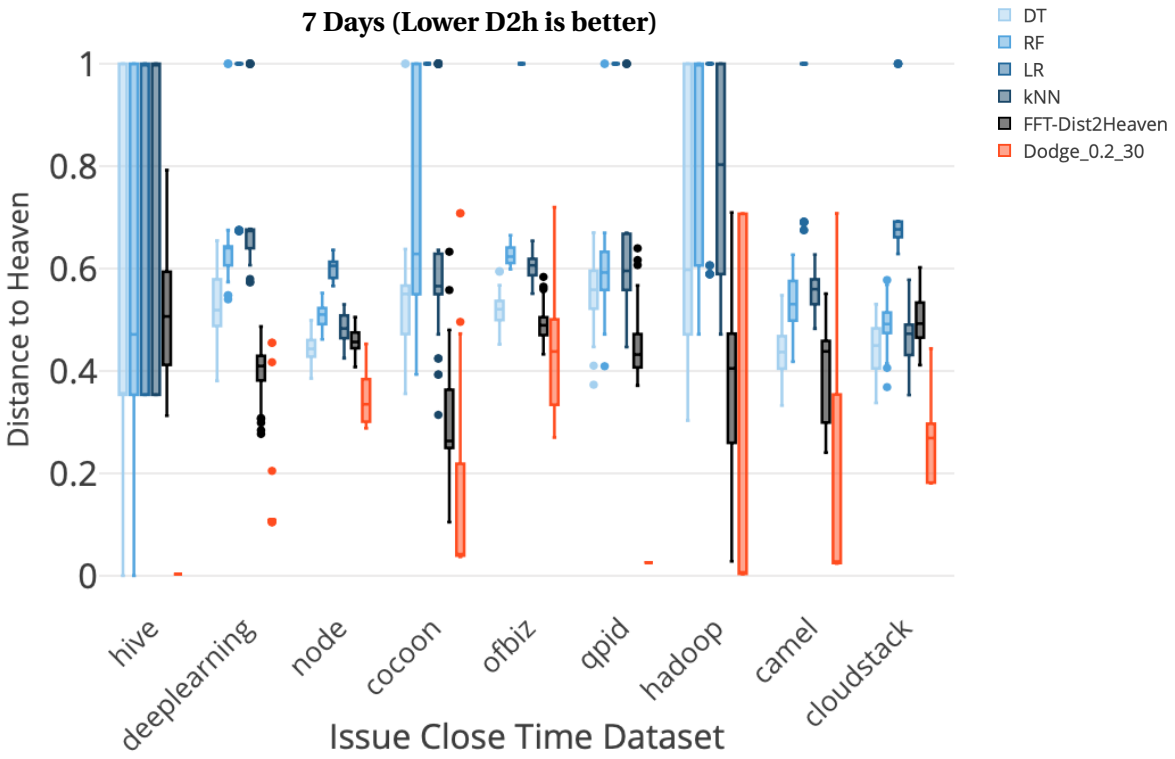

Figure 4.10 RQ5: 7 Days, Issue Lifetime Prediction Box plot comparison using **DODGE(.2)** and $N = 30$ (results seen in 25 repeats of a cross-validation study). Here, we used Efron's 95% confidence bootstrap procedure [ET93] (to demonstrate significant differences), then the A12 effect size test [AB11] (to demonstrate that the observed delta is bigger than a "small effect".

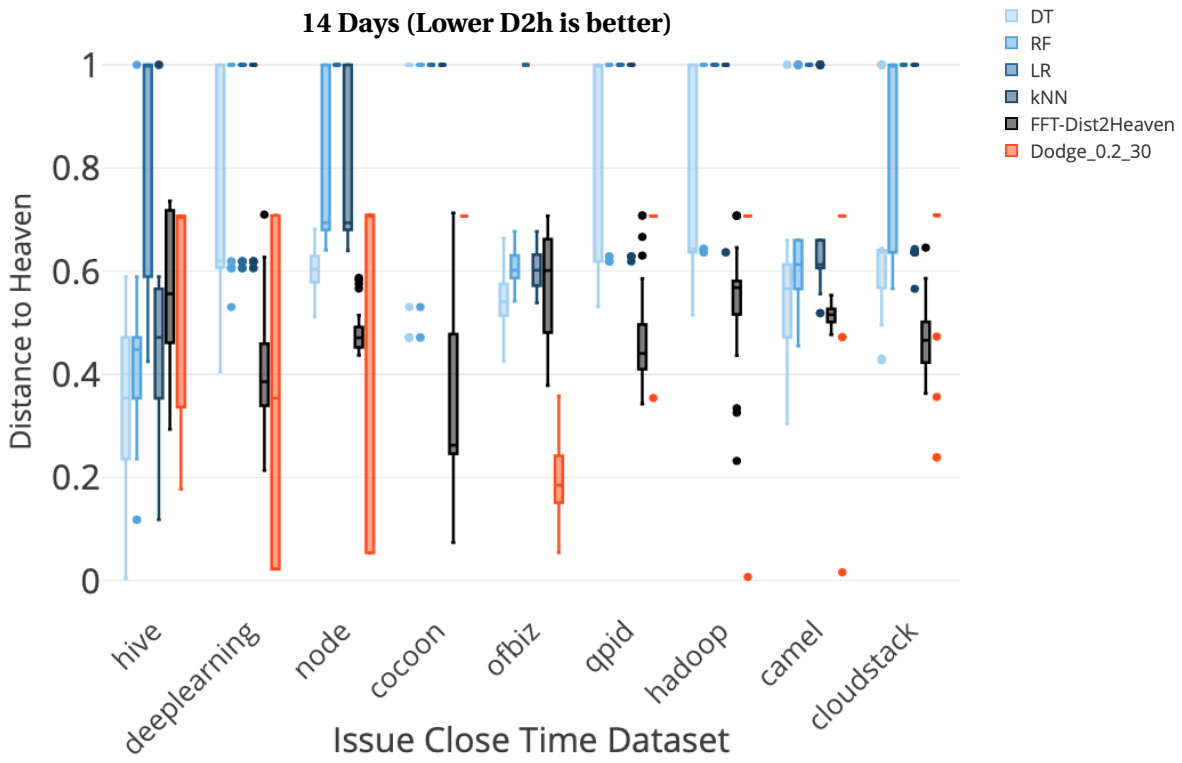

Figure 4.11 RQ5: 14 Days, Issue Lifetime Prediction Box plot comparison using **DODGE(.2)** and $N = 30$ (results seen in 25 repeats of a cross-validation study). Here, we used Efron's 95% confidence bootstrap procedure [ET93] (to demonstrate significant differences), then the A12 effect size test [AB11] (to demonstrate that the observed delta is bigger than a "small effect".

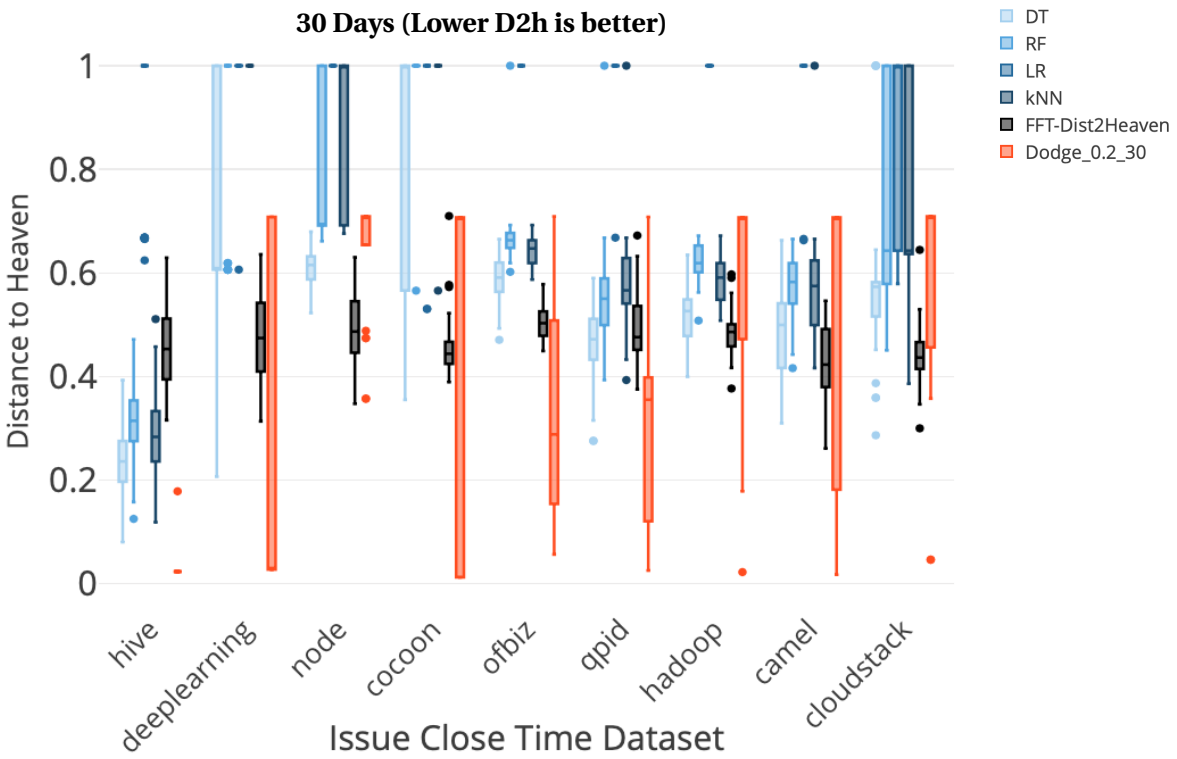

Figure 4.12 RQ5: 30 Days, Issue Lifetime Prediction Box plot comparison using **DODGE(.2)** and $N = 30$ (results seen in 25 repeats of a cross-validation study). Here, we used Efron's 95% confidence bootstrap procedure [ET93] (to demonstrate significant differences), then the A12 effect size test [AB11] (to demonstrate that the observed delta is bigger than a "small effect".

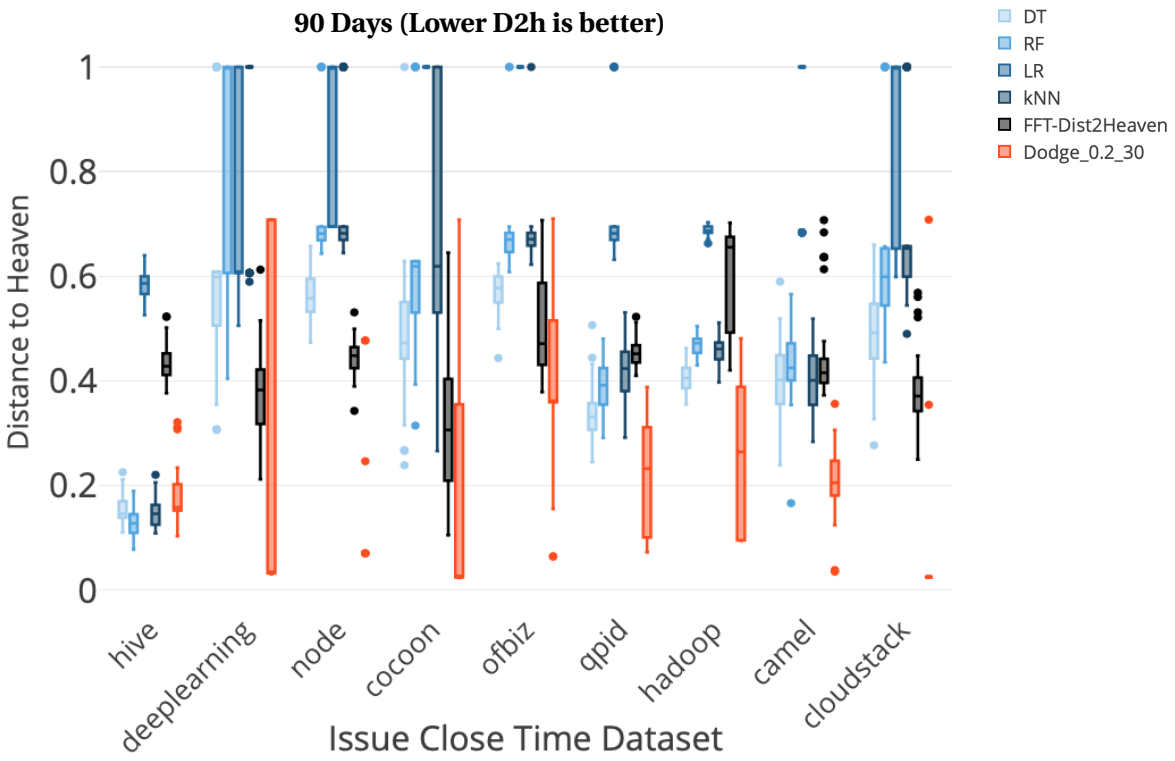

Figure 4.13 RQ5: 90 Days, Issue Lifetime Prediction Box plot comparison using **DODGE(.2)** and $N = 30$ (results seen in 25 repeats of a cross-validation study). Here, we used Efron's 95% confidence bootstrap procedure [ET93] (to demonstrate significant differences), then the A12 effect size test [AB11] (to demonstrate that the observed delta is bigger than a "small effect".

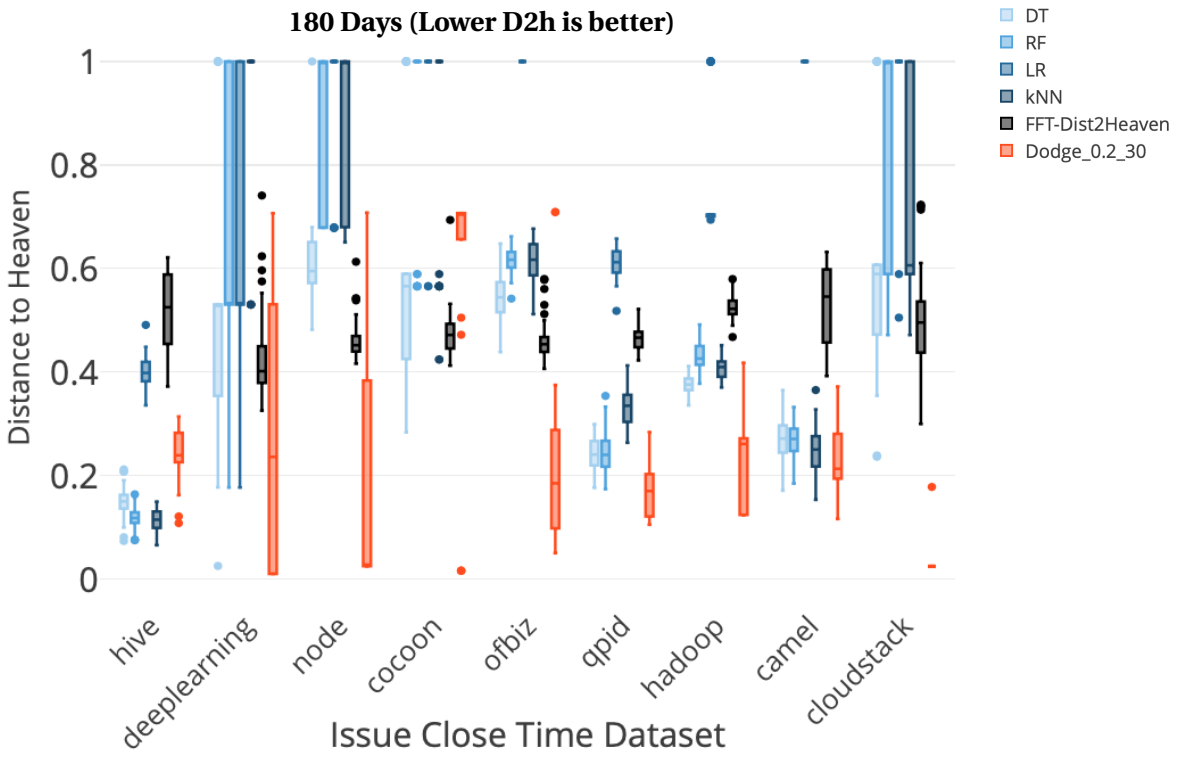

Figure 4.14 RQ5: 180 Days, Issue Lifetime Prediction Box plot comparison using **DODGE(.2)** and $N = 30$ (results seen in 25 repeats of a cross-validation study). Here, we used Efron's 95% confidence bootstrap procedure [ET93] (to demonstrate significant differences), then the A12 effect size test [AB11] (to demonstrate that the observed delta is bigger than a "small effect".

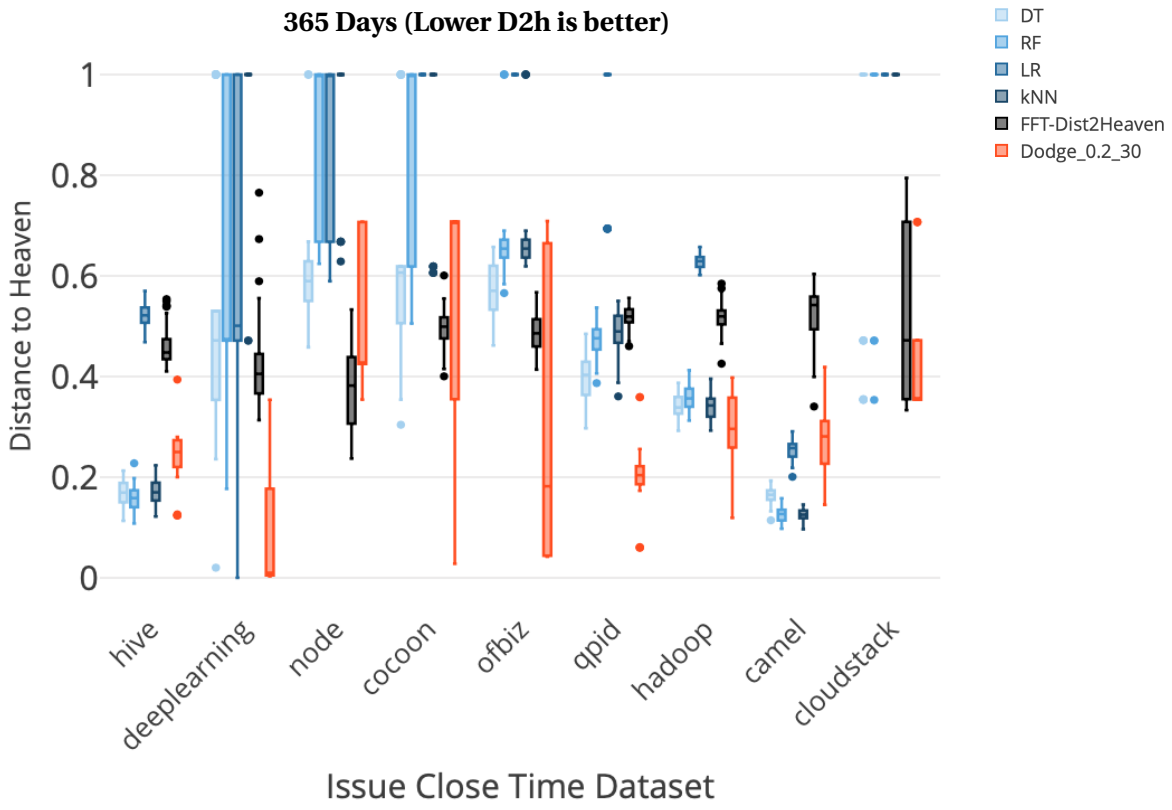

Figure 4.15 RQ5: 365 Days, Issue Lifetime Prediction Box plot comparison using **DODGE(.2)** and $N = 30$ (results seen in 25 repeats of a cross-validation study). Here, we used Efron's 95% confidence bootstrap procedure [ET93] (to demonstrate significant differences), then the A12 effect size test [AB11] (to demonstrate that the observed delta is bigger than a "small effect".

Table 4.1 Summarized Issue lifetime prediction results using **DODGE(.2)** and $N = 30$ (results seen in 25 repeats of a cross-validation study). Here, we used Efron's 95% confidence bootstrap procedure [ET93] (to demonstrate significant differences), then the A12 effect size test [AB11] (to demonstrate that the observed delta is bigger than a "small effect").

	Days till closed						
Data	>365	<180	<90	<30	<14	<7	<1
cloudstack	Dodge	Dodge	Dodge	FFT	FFT	Dodge	RF
node	FFT	Dodge	Dodge	Dodge	FFT	Dodge	Dodge
deeplearning	Dodge	Dodge	Dodge	Dodge	Dodge	Dodge	RF
cocoon	FFT	FFT	Dodge	FFT	FFT	Dodge	Dodge
ofbiz	Dodge	Dodge	Dodge	Dodge	Dodge	Dodge	Dodge
camel	RF	Dodge	Dodge	FFT	FFT	Dodge	Dodge
hadoop	Dodge	Dodge	Dodge	FFT	FFT	Dodge	FFT
qpid	Dodge	Dodge	Dodge	Dodge	FFT	Dodge	Dodge
hive	RF	RF	Dodge	Dodge	Dodge	Dodge	Dodge

The goal here is to classify an issue according to how long it will take to close; i.e. less than 1 day, less than 7 days, and so on.

Cells with a (white, gray) background means Dodge is statistically (better, worse) than (all, any) of the methods understudy respectively.

Table 4.1 shows our summarized issue lifetime detection results on which framework is winning amongst all. We compared 3 frameworks, traditional machine learning algorithms (DT, RF, LR, kNN), FFtree and **DODGE(.2)** with N=30. In this table, rows represents different project names and columns represents different datasets, and each cell correspond to which method performed statistically better.

As can be seen from table 4.1, **DODGE(ϵ)** performed statistically better and significant in about 44 out of 63 possible projects with its datasets. That means, hyperparameter optimization using **DODGE(ϵ)** is 70% better than anything else. In summary, our answer to **RQ5** is that **DODGE(ϵ)** works well for issue lifetime estimation.

4.6 RQ6: What inference can be drawn about SE data compared against other domain data (UCI datasets) after applying DODGE(ϵ)?

Till now, we have only been dealing with data related to SE tasks. We found that SE data have ϵ-dominance and it can be utilized by our new framework **DODGE(ϵ)** to produce simpler and better predictors. But can the same be told for Non-SE domain? To validate our hypothesis, we compared **DODGE(ϵ)** with 2 other frameworks, traditional machine learning algorithms (DT, RF, SVM), and FFtree.

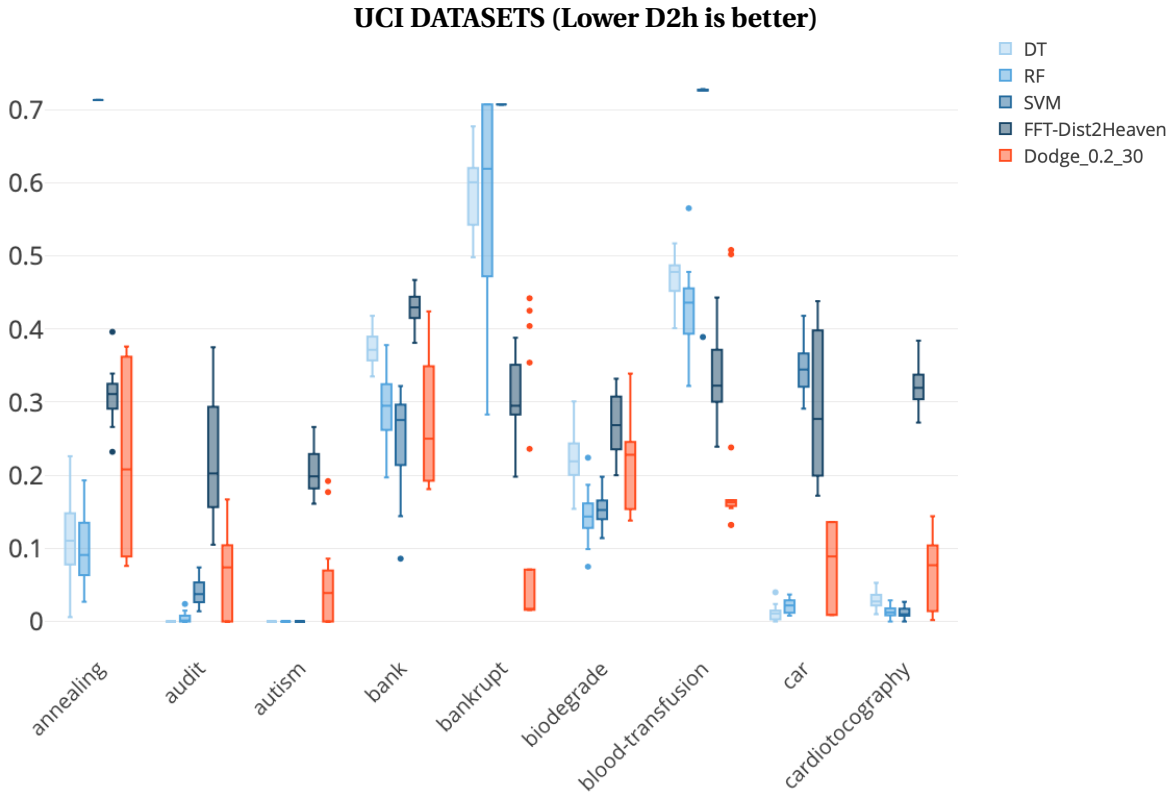

Figure 4.16 RQ6: UCI datasets Box plot comparison using **DODGE(.2)** and $N = 30$ (results seen in 25 repeats of a cross-validation study). Here, we used Efron's 95% confidence bootstrap procedure [ET93] (to demonstrate significant differences), then the A12 effect size test [AB11] (to demonstrate that the observed delta is bigger than a "small effect".

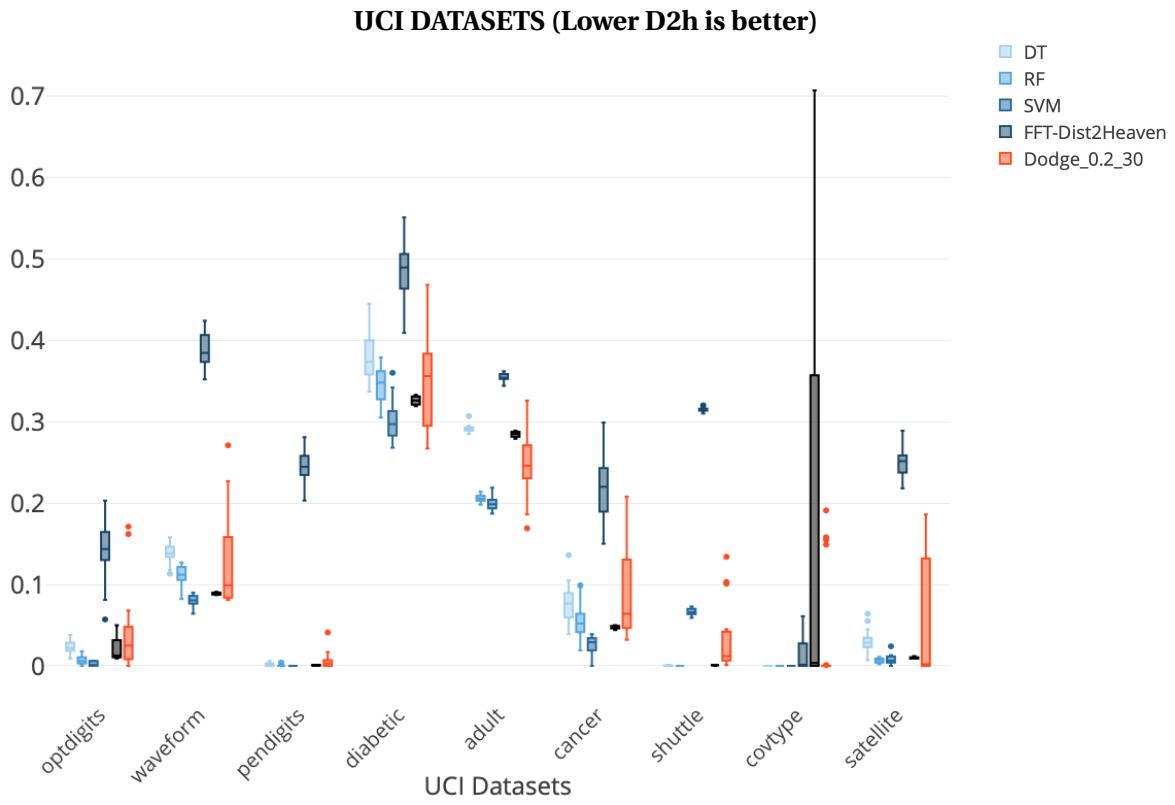

Figure 4.17 RQ6: UCI Datasets-continued, Box plot comparison using **DODGE(.2)** and $N = 30$ (results seen in 25 repeats of a cross-validation study). Here, we used Efron's 95% confidence bootstrap procedure [ET93] (to demonstrate significant differences), then the A12 effect size test [AB11] (to demonstrate that the observed delta is bigger than a "small effect".

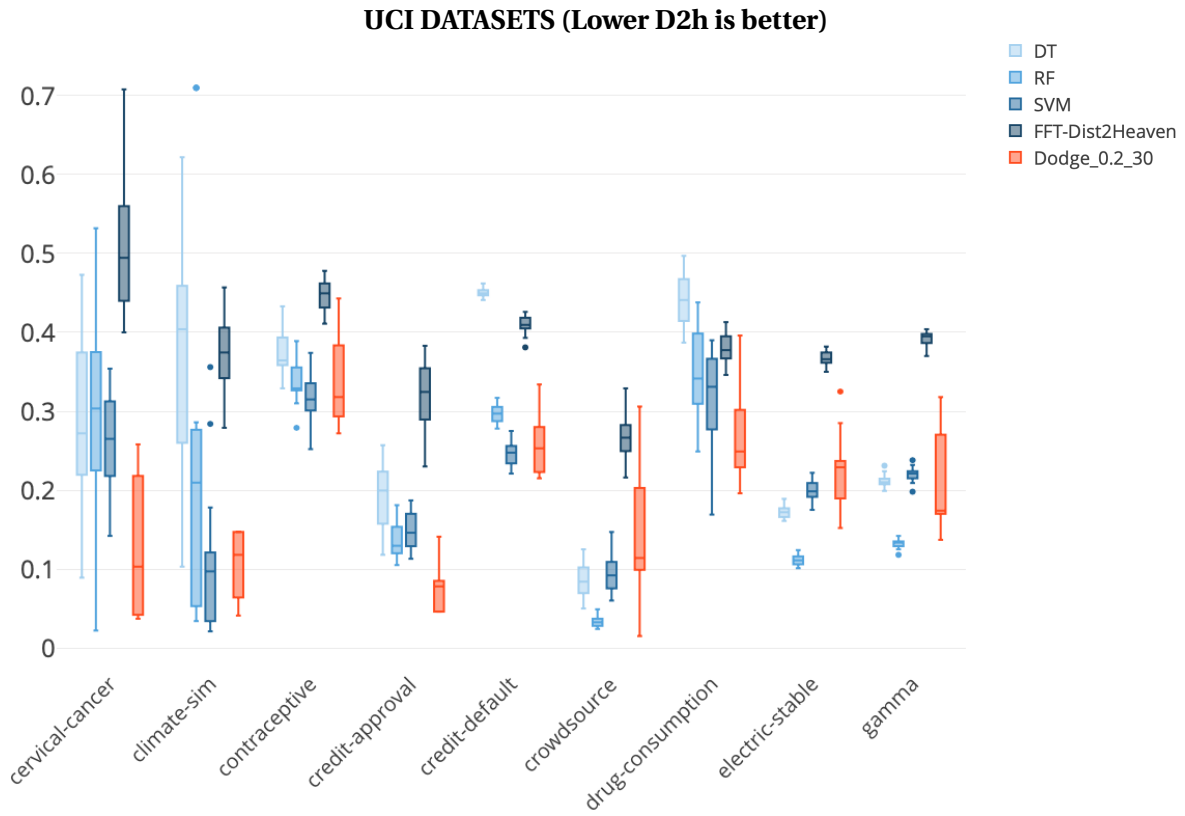

Figure 4.18 RQ6: UCI Datasets-continued, Box plot comparison using **DODGE(.2)** and $N = 30$ (results seen in 25 repeats of a cross-validation study). Here, we used Efron's 95% confidence bootstrap procedure [ET93] (to demonstrate significant differences), then the A12 effect size test [AB11] (to demonstrate that the observed delta is bigger than a "small effect".

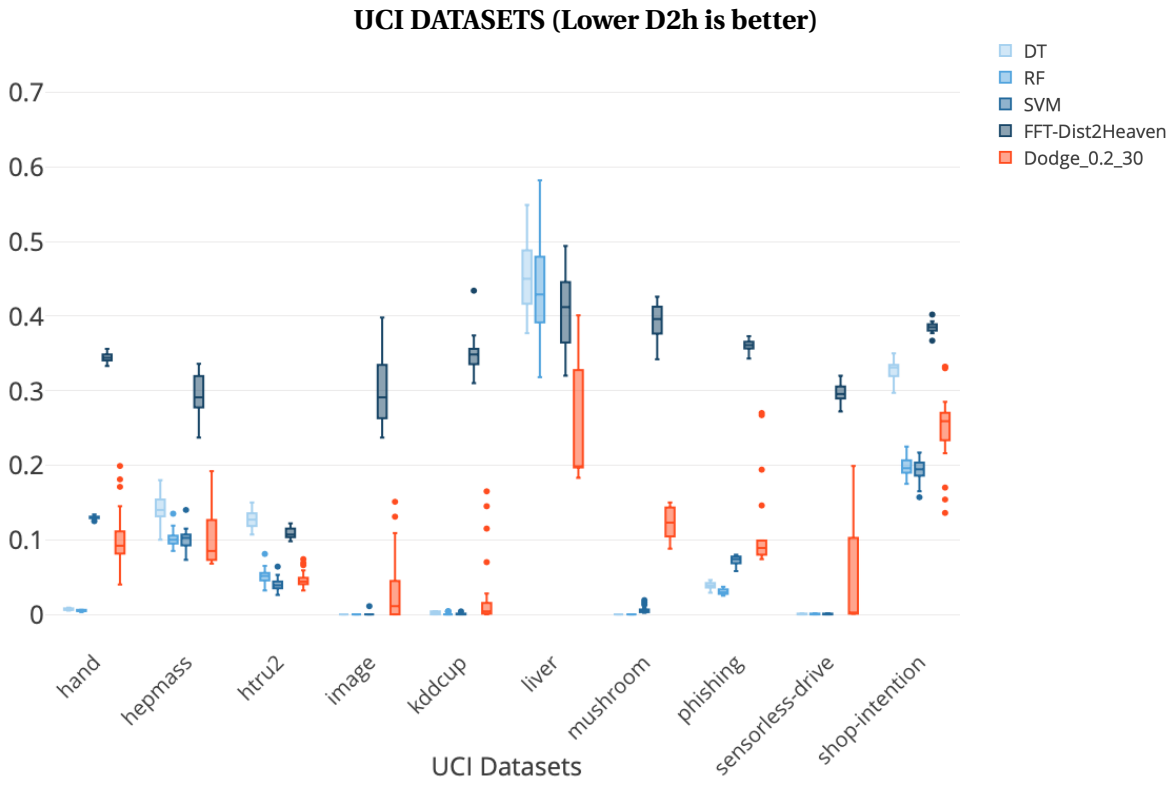

Figure 4.19 RQ6: UCI Datasets-continued, Box plot comparison using **DODGE(.2)** and $N = 30$ (results seen in 25 repeats of a cross-validation study). Here, we used Efron's 95% confidence bootstrap procedure [ET93] (to demonstrate significant differences), then the A12 effect size test [AB11] (to demonstrate that the observed delta is bigger than a "small effect".

Table 4.2 RQ6: NON-SE, Summarized 37 UCI prediction results using **DODGE(.2)** and $N = 30$ (results seen in 25 repeats of a cross-validation study). Here, we used Efron's 95% confidence bootstrap procedure [ET93] (to demonstrate significant differences), then the A12 effect size test [AB11] (to demonstrate that the observed delta is bigger than a "small effect"). Cells with a (gray, white) background means Dodge is statistically (better, worse) than (all, any) of the methods understudy respectively.

Dataset	**Winner**	**Dataset**	**winner**
optdigits	RF	satellite	RF
climate-sim	SVM	credit-approval	Dodge
cancer	SVM	shop-intention	RF
image	RF	covtype	RF
hand	RF	drug-consumption	Dodge
biodegrade	RF	adult	RF
crowdsource	RF	blood-transfusion	Dodge
credit-default	SVM	cervical-cancer	Dodge
autism	RF	bank	SVM
bankrupt	Dodge	audit	RF
contraceptive	SVM	mushroom	RF
pendigits	RF	phishing	RF
car	RF	diabetic	SVM
hepmass	RF	htru2	SVM
kddcup	RF	sensorless-drive	RF
waveform	SVM	annealing	RF
cardiotocography	RF	shuttle	RF
electric-stable	RF	gamma	RF
liver	Dodge		

Figures 4.16, 4.17, 4.18 and 4.19 represent all the frameworks comparison for UCI datasets respectively. X-axis represents all datasets and y-axis represents *D2h* values. Lower the *D2h* value is better. Legend shows different frameworks understudy. Lets consider an example from Figure 4.17 for **kddcup** dataset. In that we can see that **DODGE(.2)** is statistically similar to RF, SVM and DT. We chose the winning framework RF because it is simpler and faster to implement **DODGE(ϵ)**. We chose similarly when we summarized our results in Table 4.2.

Table 4.2 shows our summarized UCI results on which framework is winning amongst all. We observed that **DODGE(ϵ)** performed statistically better and significant only in 6 out 37 datasets understudy. Here our hypothesis did not hold true as we predicted for SE tasks. This tells us that other domain might not have chunks of size ϵ which are redundant options leading to indistinguishable results.

We went ahead to try and justify what might be inherently different between the SE and non-SE data for **DODGE(ϵ)** to work. We hypothesize that probably the feature discretization (if worked

well), or the sample size, or the relevant class percentage could lead us a definition on when to use **DODGE(ϵ)**. We ran Fayyad & Irani's discretization [FI93] to identify the cut points in each feature for each dataset. If we believe the discretization works really well then the expected Entropy [Gra11] for these cut points for each feature would be closer to 0. We ran Fayyad & Irani's discretization and calculated entropy for each attribute for the dataset. If we have 80 attributes then we will 80 entropy values. We then looked at 10th, 30th, 50th, 70th and 90th percentile, combined them with total number of instances in the dataset and relevant class percentage. Now these newly calculated values will act as attributes and the class would be the winning classifiers. This data can be found at `http://tiny.cc/variance_defense`. In total, we had 37 UCI and 77 SE datasets.

We ran a decision tree on this data (where all 37 UCI datasets were picked and 38 random SE datasets were picked) to try interpret when can dodge be used and when can some other framework be used.

Table 4.3 shows 2 tree one without feature selection and another with feature selection respectively. Though 3 inferences can be drawn from both that

- **DODGE(ϵ)** can be used when relevant class percentage is low in number (less than 9 or 20%)
- **DODGE(ϵ)** can be used when discretization works really well (when 10th percentile of entropies is less than 0.31).
- First use simpler implementation of FFtree and if it provides better performances against traditional machine learning, then use **DODGE(ϵ)**, a complex method.

Please note, that the performance score for both trees will lie somewhere between 61% to 82% recall and precision as we got about 80% performance with full training set and when we used cross validation, then we got about 80%. This does provide only weak inferences on when can we use **DODGE(ϵ)** or not above. This needs to be explored further. But our initial inferences say that we might explain when **DODGE(ϵ)** can be used.

In summary, our answer to **RQ6** is that only SE data shows the existence of ϵ, and it may not hold true for non-SE data. Also, use **DODGE(ϵ)** when you have highly imbalanced dataset or if discretization works well.

4.7 Discussion on Results

These findings where we showed that only few options could lead to dramatic improvements are not new. In the past we already have such examples but until now no other researchers have tried utilizing this large variablility ($\epsilon > 0$) as a tool to terminate an optimizer faster.

For example, in the past, researchers in constraint satisfaction found "random search with retries" was a very effective strategy. Crawford and Baker reported that such searches took less time than a

Table 4.3 Decision Tree Built for explaining when to use **DODGE**(ϵ)

Without Feature Selection	With Feature Selection, only 2 attributes were selected
82% recall and precision as full training set	80% recall and precision as full training set
65% recall and precision as cross validation set	61% recall and precision as cross validation set

`% class <= 8.9: Dodge`
`% class > 8.9`
`  30 <= 0.19: RF`
`  30 > 0.19`
`  samples <= 12191`
`  90 <= 0.97: Dodge`
`  90 > 0.97`
`  90 <= 0.98`
`  samples <= 139: Dodge`
`  samples > 139`
`  70 <= 0.89: FFT`
`  70 > 0.89: SVM`
`  90 > 0.98`
`  samples <= 782: Dodge`
`  samples > 782`
`  samples <= 3303`
`  30 <= 0.87`
`  samples <= 2287: RF`
`  samples > 2287: SVM`
`  30 > 0.87: Dodge`
`  samples > 3303`
`  30 <= 0.87: Dodge`
`  30 > 0.87: RF`
`  samples > 12191: RF`
	`10 <= 0.31`
	`  % class <= 20.2: Dodge`
	`  % class > 20.2: RF`
	`10 > 0.31: Dodge`

complete search to find more solutions using just a small number of retries [CB94]. Their ISAMP "iterative sampler" makes random choices within a model until it gets "stuck"; i.e. until further choices do not satisfy expectations. When "stuck", ISAMP does not waste time fiddling with current choices (as was done by older chronological backtracking algorithms). Instead, ISAMP logs what decisions were made before getting "stuck". It then performs a "retry"; i.e. resets and starts again, this time making other random choices to explore.

Crawford and Baker explain the success of this strange approach by assuming models contain a small set of *master variables* that set the remaining variables (and this article calls such master variables *keys*).

A similar conclusion comes from the work of Williams et al. [Wil03]. They found that if a randomized search is repeated many times, that a small number of variable settings were shared by all solutions. They also found that if they set those variables before conducting the rest of the search, then formerly exponential runtimes collapsed to low-order polynomial time. They called these shared variables the *backdoor* to reducing computational complexity.

The obvious question from the above is this: if software analytics is so simple, why has it taken so long to find the simplifications found in this thesis? There are several answers to this question. Firstly, we might look to our research culture that seems to best reward novel, and increasing complex new things.

Secondly, there might be a belief amongst SE researchers that designing new AI algorithms is not an SE task. Why bother, it might be argued, if those algorithms exist in abundance already in the AI literature? What we hope we have shown here is that, it might be a property of SE problems (large ϵ of the output space) that requires specialized algorithms such as **DODGE(ϵ)**.

Thirdly, at least in past 15 years of SE literature, we have been reporting that effective SE analytics models can be constructed from surprisingly few attributes:

- Feature selection results of where two to three attributes was enough to predict software defects or development effort [Men07a; Che05b; MH03].
- Incremental learning results of [Men08b; Men10] where a few dozen examples of defective and non-defective modules performed as well as anything else.
- Maths model at the end of [Nam17] that showed, in the expected case, that very few examples are required to learn SE defect models.

The problem with those reports was that they merely reported an effect. Unlike this thesis, that work did not explain these simplicity results or on the nature of SE data nor propose any way to better exploit that effect. In this thesis we explained how SE data is different in nature by showing the existence of ϵ-domination. We also provided weak inferences on when can we use **DODGE(ϵ)**

through our RQ6. It was weak argument due to lower recall and precision value. We do propose that this needs to be extensively studied in future to identify when and where can we use **DODGE(ϵ)**.

In the next chapters, we will discuss threats to validity for this study and the conclusion which can be drawn from this. We also provided what are our current limitations and the future work.

CHAPTER

5

THREATS TO VALIDITY

5.1 Sampling Bias

This thesis shares the same sampling bias problem as every other data mining paper. Sampling bias threatens any classification experiment; what matters in one case may or may not hold in another case. For example, even though we use many SE and NON-SE open-source datasets in this study which come from several sources, they were all supplied by individuals.

But in our case, our sampling bias is smaller since, we applied our frameworks to about 78 SE datasets and 37 NON-SE datasets giving us more conclusive results. Though, it is possible that what worked in these datasets may not work somewhere else. We propose that when any new data becomes available, then it is important to test our methods on the new data. But we showed that there exists more than one domain where **DODGE(ϵ)** is a useful approach and this makes us to keep away from huge sampling bias.

5.2 Learner Bias

For building different classifiers in this study, we used many preprocessors (about 13 of them) and learners (about 6 of them). We chose these learners because past studies have shown that, these have been extensively used [Gho15; AM18; Tan16]. Thus they are selected as the state-of-the-art learns to be compared with **DODGE(ϵ)**. Though there are other learners which we have not explored

and could result in changes to our final conclusions. But due to extensive use of so many learners and preprocessors our study is suffered less due to any learner bias.

5.3 Evaluation Bias

This thesis uses two performance measures, i.e., P_{opt} and $dist2heaven$. Other quality measures are often used in software engineering to quantify the effectiveness of prediction [Men07b; Men05; Jor04]. But we wanted to show the success of **DODGE**(ϵ) for multi-goals and these two measures are more prominent. Though, other measures can easily be explored if needed.

5.4 Order Bias

For the performance evaluation part, the order that the data trained and predicted affects the results.

For the defect prediction datasets, we deliberately choose an ordering that mimics how our software projects releases versions so, for those experiments, we would say that bias was a required and needed.

For our other SE and NON-SE datasets, to mitigate this order bias, we ran our rig in a the 5-bin cross validation 5 times, randomly changing the order of the data each time.

5.5 Construct Validity

At various stages of data collection by different researchers, they must have made engineering decisions about what attributes to be extracted from Github for Issue lifetime datasets, or what object-oriented metrics need to be extracted. Though all these decisions have been verified by other researchers to make sure the dataset collection do not suffer from any construct validity.

5.6 Statistical Validity

To increase the validity of our results, we applied two statistical tests, bootstrap and the a12 effect size test. Hence, anytime in this paper we reported that "X was different from Y" then that report was based on both an effect size and a statistical significance test.

5.7 External Validity

DODGE(ϵ) self-selects the tunings used in the pre-processors and data miners. Hence, by its very nature, this article avoids one threat to external validity (i.e., that important control parameter settings are explored).

One threat to external validity is that this article compares **DODGE(ϵ)** against existing baselines for traditional machine learning algorithms and HPO in the SE analytics literature. We do not compare our new approach against the kinds of optimizers we might find in search-based SE literature [PM18]. There are two reasons for this. Firstly, search-based SE methods are typically CPU intensive and so do not address our simplicity goal. Secondly, the main point of this thesis is to document a previous unobserved feature of the output space of software analytics. It is an open question whether or not **DODGE(ϵ)** is the best way to explore output space In order to motivate the community to explore that space, some work must demonstrate its existence and offer baseline results that, using the knowledge of output space, it is possible to do better than past work. Hence, this work.

CHAPTER

6

CONCLUSION

This thesis has discussed ways to reduce the CPU cost associated with hyperparameter optimization for software analytics. We found that SE data is inherently different in nature by have ϵ-domination. Tool like **DODGE(ϵ)** were shown to work as well, or better, than numerous recent SE results which utilizes the ϵ-domination. As stated in the introduction, we assert that other methods perform worse than **DODGE(ϵ)** since they do not appreciate the simplicity of the output space. Hence, those other methods waste much CPU as they struggle to cover billions of tuning options like Tables 3.19 and 3.18 (most of which yield indistinguishably different results).

One way to characterize this thesis is to say:

- Stop treating large ϵ variability as a problem;
- Instead, treat large ϵ variability as a resource that can be used to simplify software analytics.

This new characterization is a significant contribution since it means that every new machine learning algorithm developed in the AI community might not apply to SE. Perhaps understanding SE is a different problem to understanding other problems that are more precisely controlled and restrained. Also, perhaps, it is time to design new machine learning algorithms (like **DODGE(ϵ)**) that are better suited to the large ϵ variabilities of SE data. As shown in this thesis, such new algorithms can exploit the peculiarities of SE data to dramatically simplify software analytics which is not true for Non-SE data.

This thesis's results are based on an analysis of four SE tasks. More tasks need to be explored before the methods of this paper can broadly applied. However, our success for these tasks begs the question: how many other domains can **DODGE(ϵ)** be applied?

6.1 Limitations of Current Methodologies

The limitations of current study includes:

- We showed **DODGE(ϵ)** to work well for 4 SE tasks. There could be many other tasks which needs to be explored further and also it needs to be explored for regression tasks.
- Currently we only explored 2 goals together such as *dis2heaven* and P_{opt}. There could be a possibility it may not work well for 3 goals such as IFA [Hua17].
- We did show that **DODGE(ϵ)** works better against FLASH for 2 goals but there is likelihood that when we move from 2 goals to N goals, FLASH might work better.
- **DODGE(ϵ)** is in its early form, and there could be many more information to incorporate which can be utilized to supercharge the search to terminate it faster.

6.2 Future Work

Considering the limitations, in future, it can be explored further:

- A useful extension is to explore problems with three or more goals (e.g., reduce false alarms while at the same time improving precision and recall).
- Currently, only classification tasks were explored, **DODGE(ϵ)** needs to be applied for other tasks such as regression task.
- Right now, in **DODGE(ϵ)** we only deprecate tunings that lead to similar results. Another approach would be to also depreciate tunings that lead to similar *and worse* results (perhaps to rule out larger parts of the output space, sooner).
- Further, for pragmatically reasons it would be useful if the Table 3.19 list could be reduced to a smaller, faster to run, set of learners. That is, here we would select learners that run fastest while generating the most variable kinds of models.
- **DODGE(ϵ)** works better against FLASH for 2 goals it will be interesting to compare the both frameworks when we move from 2 goals to N goals.

- Also, we provided weak inferences due to lower recall and precision value, on when can **DODGE**(ϵ) be used given certain data attributes. This needs to be explored further to provide a strong argument to it.

REFERENCES

[AM18] Agrawal, A. & Menzies, T. "Is better data better than better data miners?: on the benefits of tuning SMOTE for defect prediction". *Proceedings of the 40th International Conference on Software Engineering*. ACM. 2018, pp. 1050–1061.

[Agr18a] Agrawal, A. et al. "Can You Explain That, Better? Comprehensible Text Analytics for SE Applications". *arXiv preprint arXiv:1804.10657* (2018).

[Agr18b] Agrawal, A. et al. "What is wrong with topic modeling? And how to fix it using search-based software engineering". *Information and Software Technology* **98** (2018), pp. 74–88.

[AC05] Anderson-Cook, C. M. *Practical genetic algorithms*. 2005.

[Anv06] Anvik, J. et al. "Who should fix this bug?" *Proceedings of the 28th international conference on Software engineering*. ACM. 2006, pp. 361–370.

[Arc16] Arcelli Fontana, F. et al. "Comparing and experimenting machine learning techniques for code smell detection". *Empir. Softw. Eng.* **21**.3 (2016), pp. 1143–1191.

[AB11] Arcuri, A. & Briand, L. "A Practical Guide for Using Statistical Tests to Assess Randomized Algorithms in Software Engineering". *Proceedings of the 33rd International Conference on Software Engineering*. ICSE '11. 2011, pp. 1–10.

[AN07] Asuncion, A. & Newman, D. *UCI machine learning repository*. 2007.

[BL09] Bajracharya, S. K. & Lopes, C. V. "Mining search topics from a code search engine usage log." *MSR*. Citeseer. 2009, pp. 111–120.

[Bar15] Barr, E. T. et al. "The oracle problem in software testing: A survey". *IEEE transactions on software engineering* **41**.5 (2015), pp. 507–525.

[Bar14] Barua, A. et al. "What are developers talking about? an analysis of topics and trends in stack overflow". *Empirical Software Engineering* **19**.3 (2014), pp. 619–654.

[Bea06] Beausoleil, R. P. ""MOSS" multiobjective scatter search applied to non-linear multiple criteria optimization". *European Journal of Operational Research* **169**.2 (2006), pp. 426–449.

[BZ14] Begel, A. & Zimmermann, T. "Analyze this! 145 questions for data scientists in software engineering". *Proceedings of the 36th International Conference on Software Engineering*. ACM. 2014, pp. 12–23.

[Ben17] Bennin, K. E. et al. “MAHAKIL: Diversity based Oversampling Approach to Alleviate the Class Imbalance Issue in Software Defect Prediction”. *IEEE Transactions on Software Engineering* (2017).

[BB12] Bergstra, J. & Bengio, Y. “Random search for hyper-parameter optimization”. *Journal of Machine Learning Research* **13**.Feb (2012), pp. 281–305.

[BN11] Bhattacharya, P. & Neamtiu, I. “Bug-fix time prediction models: can we do better?” *Proceedings of the 8th Working Conference on Mining Software Repositories*. ACM. 2011, pp. 207–210.

[Bin14] Binkley, D. et al. “Understanding LDA in source code analysis”. *Proceedings of the 22nd International Conference on Program Comprehension*. ACM. 2014, pp. 26–36.

[Bir09] Bird, C. et al. “Putting it all together: Using socio-technical networks to predict failures”. *2009 20th ISSRE*. IEEE. 2009, pp. 109–119.

[Bir06] Bird, S. “NLTK: the natural language toolkit”. *Proceedings of the COLING/ACL on Interactive presentation sessions*. Association for Computational Linguistics. 2006, pp. 69–72.

[Ble03] Blei, D. M. et al. “Latent dirichlet allocation”. *the Journal of machine Learning research*. Vol. 3. JMLR. org, 2003, pp. 993–1022.

[Bre01] Breiman, L. “Random forests”. *Machine learning* **45**.1 (2001), pp. 5–32.

[CD09] Catal, C. & Diri, B. “A systematic review of software fault prediction studies”. *Expert systems with applications* **36**.4 (2009), pp. 7346–7354.

[Cha02] Chawla, N. V. et al. “SMOTE: synthetic minority over-sampling technique”. *Journal of artificial intelligence research* **16** (2002), pp. 321–357.

[Che18] Chen, D. et al. “Applications of Psychological Science for Actionable Analytics”. *Foundations of Software Engineering* (2018).

[CG16] Chen, T. & Guestrin, C. “Xgboost: A scalable tree boosting system”. *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*. ACM. 2016, pp. 785–794.

[Che12] Chen, T.-H. et al. “Explaining software defects using topic models”. *Mining Software Repositories (MSR), 2012 9th IEEE Working Conference on*. IEEE. 2012, pp. 189–198.

[Che16] Chen, T.-H. et al. “Topic-based software defect explanation”. *Journal of Systems and Software* (2016).

[Che05a] Chen, Z. et al. “Feature subset selection can improve software cost estimation accuracy”. *ACM SIGSOFT Software Engineering Notes*. Vol. 30. 4. ACM. 2005, pp. 1–6.

[Che05b] Chen, Z. et al. “Finding the right data for software cost modeling”. *IEEE software* **22**.6 (2005), pp. 38–46.

[CK94] Chidamber, S. R. & Kemerer, C. F. “A metrics suite for object oriented design”. *IEEE Transactions on software engineering* **20**.6 (1994), pp. 476–493.

[Chi12] Chiha, I et al. “Tuning PID controller with multi-objective differential evolution”. *Communications Control and Signal Processing (ISCCSP), 2012 5th International Symposium on*. IEEE. 2012, pp. 1–4.

[CH07] Chiu, N.-H. & Huang, S.-J. “The adjusted analogy-based software effort estimation based on similarity distances”. *Journal of Systems and Software* **80**.4 (2007), pp. 628–640.

[Coh88] Cohen, J. “Statistical power analysis for the behavioral sciences. 1988, Hillsdale, NJ: L”. *Lawrence Earlbaum Associates* **2** (1988).

[CB94] Crawford, J. M. & Baker, A. B. “Experimental results on the application of satisfiability algorithms to scheduling problems”. *AAAI*. Vol. 2. 1994, pp. 1092–1097.

[Cze11] Czerwonka, J. et al. “Crane: Failure prediction, change analysis and test prioritization in practice–experiences from windows”. *Proceedings of the 4th IEEE International Conference on Software Testing, Verification and Validation*. IEEE. 2011, pp. 357–366.

[D’A10] D’Ambros, M. et al. “An extensive comparison of bug prediction approaches”. *2010 7th IEEE MSR)*. IEEE. 2010, pp. 31–41.

[Deb05] Deb, K. et al. “Evaluating the ε-domination based multi-objective evolutionary algorithm for a quick computation of Pareto-optimal solutions”. *Evolutionary computation* **13**.4 (2005), pp. 501–525.

[Du15] Du, X. et al. “An evolutionary algorithm for performance optimization at software architecture level”. *Evolutionary Computation (CEC), 2015 IEEE Congress on*. IEEE. 2015, pp. 2129–2136.

[DG17] Dua, D. & Graff, C. *UCI Machine Learning Repository*. 2017.

[ET93] Efron, B. & Tibshirani, R. J. *An Introduction to the Bootstrap*. New York: Chapman & Hall, 1993.

[EE08] Elish, K. O. & Elish, M. O. “Predicting defect-prone software modules using support vector machines”. *Journal of Systems and Software* **81**.5 (2008), pp. 649–660.

[FI93] Fayyad, U. & Irani, K. “Multi-interval discretization of continuous-valued attributes for classification learning” (1993).

[FM02] Feather, M. S. & Menzies, T. “Converging on the optimal attainment of requirements”. *Requirements Engineering, 2002. Proceedings. IEEE Joint International Conference on*. IEEE. 2002, pp. 263–270.

[Fel06] Feldman, R.-S. *J, The Text Mining Handbook*. New York: Cambridge University Press, 2006.

[Fis12] Fisher, D. et al. “Interactions with big data analytics”. *interactions* **19**.3 (2012), pp. 50–59.

[Fow99] Fowler, M. et al. *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley Longman, 1999.

[FM17] Fu, W. & Menzies, T. “Easy over hard: A case study on deep learning”. *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM. 2017, pp. 49–60.

[Fu16a] Fu, W. et al. “Tuning for software analytics: Is it really necessary?” *Information and Software Technology* **76** (2016), pp. 135–146.

[Fu16b] Fu, W. et al. “Why is Differential Evolution Better than Grid Search for Tuning Defect Predictors?” *arXiv preprint arXiv:1609.02613* (2016).

[Fu18] Fu, W. et al. “Building Better Quality Predictors Using "ϵ-Dominance"”. *arXiv preprint arXiv:1803.04608* (2018).

[Fu15] Fu, Y. et al. “Automated classification of software change messages by semi-supervised Latent Dirichlet Allocation”. *Information and Software Technology* **57** (2015), pp. 369–377.

[GCW13] Galvis Carreño, L. V. & Winbladh, K. “Analysis of user comments: an approach for software requirements evolution”. *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press. 2013, pp. 582–591.

[GM16] Garousi, V. & Mäntylä, M. V. “Citations, research topics and active countries in software engineering: A bibliometrics study”. *Computer Science Review* **19** (2016), pp. 56–77.

[Gen89] Gennari, J. H. et al. “Models of incremental concept formation”. *Artificial intelligence* **40**.1-3 (1989), pp. 11–61.

[Gho15] Ghotra, B. et al. “Revisiting the impact of classification techniques on the performance of defect prediction models”. *37th ICSE-Volume 1*. IEEE Press. 2015, pp. 789–800.

[Gig10] Giger, E. et al. “Predicting the fix time of bugs”. *Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering*. ACM. 2010, pp. 52–56.

[GM86] Glover, F. & McMillan, C. “The general employee scheduling problem. An integration of MS and AI”. *Computers & operations research* **13**.5 (1986), pp. 563–573.

[Gol79] Goldberg, A. T. “On the complexity of the satisfiability problem”. PhD thesis. New York University, 1979.

[GH88] Goldberg, D. E. & Holland, J. H. “Genetic algorithms and machine learning”. *Machine learning* **3**.2 (1988), pp. 95–99.

[Gon08] Gondra, I. “Applying machine learning to software fault-proneness prediction”. *Journal of Systems and Software* **81**.2 (2008), pp. 186–195.

[Gra13] Grant, S. et al. “Using heuristics to estimate an appropriate number of latent topics in source code analysis”. *Science of Computer Programming* **78**.9 (2013), pp. 1663–1678.

[Gra11] Gray, R. M. *Entropy and information theory*. Springer Science & Business Media, 2011.

[GS04] Griffiths, T. L. & Steyvers, M. “Finding scientific topics”. *Proceedings of the National academy of Sciences* **101**.suppl 1 (2004), pp. 5228–5235.

[Gu16] Gu, X. et al. “Deep API learning”. *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM. 2016, pp. 631–642.

[Guo10] Guo, P. J. et al. “Characterizing and predicting which bugs get fixed: an empirical study of Microsoft Windows”. *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*. Vol. 1. ACM. 2010, pp. 495–504.

[GM14] Guzman, E. & Maalej, W. “How do users like this feature? a fine grained sentiment analysis of app reviews”. *Requirements Engineering Conference (RE), 2014 IEEE 22nd International*. IEEE. 2014, pp. 153–162.

[HH03] Hall, M. A. & Holmes, G. “Benchmarking attribute selection techniques for discrete class data mining”. *IEEE Transactions on Knowledge and Data engineering* **15**.6 (2003), pp. 1437–1447.

[Hal12] Hall, T. et al. “A systematic literature review on fault prediction performance in software engineering”. *IEEE TSE* **38**.6 (2012), pp. 1276–1304.

[HGP09] Hamill, M. & Goseva-Popstojanova, K. “Common trends in software fault and failure data”. *IEEE Transactions on Software Engineering* **35**.4 (2009), pp. 484–496.

[Has09] Hassan, A. E. “Predicting faults using the complexity of code changes”. *31st ICSE*. IEEE Computer Society. 2009, pp. 78–88.

[Has05] Hassan, R. et al. “A comparison of particle swarm optimization and the genetic algorithm”. *46th AIAA/ASME/ASCE/AHS/ASC structures, structural dynamics and materials conference*. 2005, p. 1897.

[HG09] He, H. & Garcia, E. A. "Learning from imbalanced data". *IEEE Transactions on knowledge and data engineering* **21**.9 (2009), pp. 1263–1284.

[Hin11] Hindle, A. et al. "Automated topic naming to support cross-project analysis of software maintenance activities". *Proceedings of the 8th Working Conference on Mining Software Repositories*. ACM. 2011, pp. 163–172.

[Hin12] Hindle, A. et al. "Relating requirements to implementation via topic analysis: Do topics extracted from requirements make sense to managers and developers?" *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*. IEEE. 2012, pp. 243–252.

[Hua17] Huang, Q. et al. "Supervised vs unsupervised models: A holistic look at effort-aware just-in-time defect prediction". *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE. 2017, pp. 159–170.

[HC06] Huang, S.-J. & Chiu, N.-H. "Optimization of analogy weights by genetic algorithm for software effort estimation". *Information and software technology* **48**.11 (2006), pp. 1034–1045.

[Jia13] Jiang, T. et al. "Personalized defect prediction". *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*. IEEE. 2013, pp. 279–289.

[Jia08a] Jiang, Y. et al. "Can data transformation help in the detection of fault-prone modules?" *Proceedings of the 2008 workshop on Defects in large software systems*. ACM. 2008, pp. 16–20.

[Jia08b] Jiang, Y. et al. "Techniques for evaluating fault prediction models". *Empirical Software Engineering* **13**.5 (2008), pp. 561–595.

[Jia09] Jiang, Y. et al. "Variance analysis in software fault prediction models". *Software Reliability Engineering, 2009. ISSRE'09. 20th International Symposium on*. IEEE. 2009, pp. 99–108.

[Jor04] Jorgensen, M. "Realism in assessment of effort estimation uncertainty: It matters how you ask". *IEEE Transactions on Software Engineering* **30**.4 (2004), pp. 209–217.

[Kam07] Kamei, Y. et al. "The effects of over and under sampling on fault-prone module detection". *ESEM 2007*. IEEE. 2007, pp. 196–204.

[Kam13] Kamei, Y. et al. "A large-scale empirical study of just-in-time quality assurance". *IEEE Transactions on Software Engineering* **39**.6 (2013), pp. 757–773.

[Kho09] Khomh, F. et al. "A Bayesian Approach for the Detection of Code and Design Smells". *2009 Ninth International Conference on Quality Software*. 2009, pp. 305–314.

[Kho11] Khomh, F. et al. “BDTEX: A GQM-based Bayesian approach for the detection of antipatterns”. *Journal of Systems and Software* **84**.4 (2011). The Ninth International Conference on Quality Software, pp. 559 –572.

[Kho10] Khoshgoftaar, T. M. et al. “Attribute selection and imbalanced data: Problems in software defect prediction”. *Tools with Artificial Intelligence (ICTAI), 2010 22nd IEEE International Conference on*. Vol. 1. IEEE. 2010, pp. 137–144.

[Kik16] Kikas, R. et al. “Using dynamic and contextual features to predict issue lifetime in GitHub projects”. *Proceedings of the 13th International Conference on Mining Software Repositories*. ACM. 2016, pp. 291–302.

[Kim15] Kim, M. et al. “REMI: defect prediction for efficient API testing”. *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM. 2015, pp. 990–993.

[Kim11] Kim, S. et al. “Dealing with noise in defect prediction”. *Software Engineering (ICSE), 2011 33rd International Conference on*. IEEE. 2011, pp. 481–490.

[Koc10] Kocaguneli, E. et al. “When to use data from other projects for effort estimation”. *Proceedings of the IEEE/ACM international conference on Automated software engineering*. ACM. 2010, pp. 321–324.

[Koc12a] Kocaguneli, E. et al. “Exploiting the essential assumptions of analogy-based effort estimation”. *IEEE Transactions on Software Engineering* **38**.2 (2012), pp. 425–438.

[Koc12b] Kocaguneli, E. et al. “On the value of ensemble effort estimation”. *IEEE Transactions on Software Engineering* **38**.6 (2012), pp. 1403–1416.

[KJ97] Kohavi, R. & John, G. H. “Wrappers for feature subset selection”. *Artificial intelligence* **97**.1-2 (1997), pp. 273–324.

[Kol14] Koltcov, S. et al. “Latent dirichlet allocation: stability and applications to studies of user-generated content”. *Proceedings of the 2014 ACM conference on Web science*. ACM. 2014, pp. 161–165.

[Kor09] Koru, A. G. et al. “An investigation into the functional form of the size-defect relationship for software modules”. *IEEE Transactions on Software Engineering* **35**.2 (2009), pp. 293–304.

[Kra15] Krall, J. et al. “GALE: Geometric Active Learning for Search-Based Software Engineering”. *Software Engineering, IEEE Transactions on* **41**.10 (2015), pp. 1001–1018.

[Kre05] Kreimer, J. “Adaptive Detection of Design Flaws”. *Electronic Notes in Theoretical Computer Science* **141**.4 (2005), pp. 117–136.

[KM18] Krishna, R. & Menzies, T. “Bellwethers: A Baseline Method For Transfer Learning”. *IEEE Transactions on Software Engineering* (2018).

[Kri16] Krishna, R. et al. “The BigSE project: lessons learned from validating industrial text mining”. *Proceedings of the 2nd International Workshop on BIG Data Software Engineering*. ACM. 2016, pp. 65–71.

[Kri17] Krishna, R. et al. “Less is More: Minimizing Code Reorganization using XTREE”. *Information and Software Technology* (2017).

[Kum03] Kumar, R. et al. “Single-ISA heterogeneous multi-core architectures: The potential for processor power reduction”. *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE. 2003, pp. 81–92.

[Lam15] Lam, A. N. et al. “Combining deep learning with information retrieval to localize buggy files for bug reports (n)”. *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. IEEE. 2015, pp. 476–481.

[Le14] Le, T.-D. B. et al. “Predicting effectiveness of ir-based bug localization techniques”. *2014 IEEE 25th International Symposium on Software Reliability Engineering*. IEEE. 2014, pp. 335–345.

[Les08] Lessmann, S. et al. “Benchmarking classification models for software defect prediction: A proposed framework and novel findings”. *IEEE Transactions on Software Engineering* **34**.4 (2008), pp. 485–496.

[Li12] Li, M. et al. “Sample-based software defect prediction with active and semi-supervised learning”. *Automated Software Engineering* **19**.2 (2012), pp. 201–230.

[LV13] Linares-Vásquez, M. et al. “An exploratory analysis of mobile development issues using stack overflow”. *Proceedings of the 10th Working Conference on Mining Software Repositories*. IEEE Press. 2013, pp. 93–96.

[Liu10] Liu, Y. et al. “Evolutionary optimization of software quality modeling with multiple repositories”. *IEEE Transactions on Software Engineering* **36**.6 (2010), pp. 852–864.

[Loh13] Lohar, S. et al. “Improving trace accuracy through data-driven configuration and composition of tracing features”. *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM. 2013, pp. 378–388.

[Mai12a] Maiga, A. et al. “SMURF: A SVM-based incremental anti-pattern detection approach”. *2012 19th Working Conference on Reverse Engineering*. IEEE. 2012, pp. 466–475.

[Mai12b] Maiga, A. et al. “Support vector machines for anti-pattern detection”. *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. IEEE. 2012, pp. 278–281.

[Mar11] Marks, L. et al. “Studying the fix-time for bugs in large open source projects”. *Proceedings of the 7th International Conference on Predictive Models in Software Engineering*. ACM. 2011, p. 11.

[Mar08] Martignon, L. et al. “Categorization with limited resources: A family of simple heuristics”. *Journal of Mathematical Psychology* **52**.6 (2008), pp. 352–361.

[MK09] Mende, T. & Koschke, R. “Revisiting the evaluation of defect prediction models”. *Proceedings of the 5th International Conference on Predictor Models in Software Engineering*. ACM. 2009, p. 7.

[MK10] Mende, T. & Koschke, R. “Effort-aware defect prediction models”. *Software Maintenance and Reengineering (CSMR), 2010 14th European Conference on*. IEEE. 2010, pp. 107–116.

[Men08a] Menzies, T. “Improving IV&V Techniques Through the Analysis of Project Anomalies: Text Mining PITS issue reports-final report”. *Citeseer* (2008).

[MH03] Menzies, T. & Hu, Y. “Data mining for very busy people”. *Computer* **36**.11 (2003), pp. 22–29.

[MM08] Menzies, T. & Marcus, A. “Automated severity assessment of software defect reports”. *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on*. IEEE. 2008, pp. 346–355.

[MS12] Menzies, T. & Shepperd, M. “Special issue on repeatable results in software engineering prediction”. *Empirical Software Engineering* **17**.1 (2012), pp. 1–17.

[Men05] Menzies, T. et al. “Simple software cost analysis: safe or unsafe?” *ACM SIGSOFT Software Engineering Notes*. Vol. 30. 4. ACM. 2005, pp. 1–6.

[Men07a] Menzies, T. et al. “Data mining static code attributes to learn defect predictors”. *IEEE TSE* **33**.1 (2007), pp. 2–13.

[Men07b] Menzies, T. et al. “Problems with Precision: A Response to" comments on’data mining static code attributes to learn defect predictors’"”. *IEEE TSE* **33**.9 (2007).

[Men07c] Menzies, T. et al. “The business case for automated software engineering”. *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. ACM. 2007, pp. 303–312.

[Men08b] Menzies, T. et al. “Implications of ceiling effects in defect predictors”. *Proceedings of the 4th international workshop on Predictor models in software engineering*. ACM. 2008, pp. 47–54.

[Men10] Menzies, T. et al. “Defect prediction from static code features: current results, limitations, new approaches”. *Automated Software Engineering* **17**.4 (2010), pp. 375–407.

[Men15] Menzies, T. et al. *The Promise Repository of Empirical Software Engineering Data*. `http://openscience.us/repo`. 2015.

[Men17] Menzies, T. et al. “Are delayed issues harder to resolve? Revisiting cost-to-fix of defects throughout the lifecycle”. *Empirical Software Engineering* **22**.4 (2017), pp. 1903–1935.

[Men18] Menzies, T. et al. “500+ Times Faster than Deep Learning:(A Case Study Exploring Faster Methods for Text Mining StackOverflow)” (2018), pp. 554–563.

[Mic13] Michalski, R. S. et al. *Machine learning: An artificial intelligence approach*. Springer Science & Business Media, 2013.

[ML02] Minka, T. & Lafferty, J. “Expectation-propagation for the generative aspect model”. *Proceedings of the Eighteenth conference on Uncertainty in artificial intelligence*. Morgan Kaufmann Publishers Inc. 2002, pp. 352–359.

[MY13] Minku, L. L. & Yao, X. “An analysis of multi-objective evolutionary algorithms for training ensemble models based on different performance measures in software effort estimation”. *Proceedings of the 9th international conference on predictive models in software engineering*. ACM. 2013, p. 8.

[Mis11] Misirli, A. T. et al. “Ai-based software defect predictors: Applications and benefits in a case study”. *AI Magazine* **32**.2 (2011), pp. 57–68.

[Moh10] Moha, N. et al. “Decor: A method for the specification and detection of code and design smells”. *IEEE Transactions on Software Engineering* **36**.1 (2010), pp. 20–36.

[Mol07] Molina, J. et al. “SSPMO: A scatter tabu search procedure for non-linear multiobjective optimization”. *INFORMS Journal on Computing* **19**.1 (2007), pp. 91–100.

[MJ03] Molokken, K. & Jorgensen, M. “A review of software surveys on software effort estimation”. *Proceedings of the 2003 International Symposium on Empirical Software Engineering*. IEEE. 2003, pp. 223–230.

[Mon13] Monden, A. et al. “Assessing the cost effectiveness of fault prediction in acceptance testing”. *IEEE Transactions on Software Engineering* **39**.10 (2013), pp. 1345–1357.

[Moo98] Moore, G. E. et al. “Cramming more components onto integrated circuits”. *Proceedings of the IEEE* **86**.1 (1998), pp. 82–85.

[Mye11] Myers, G. J. et al. *The art of software testing*. John Wiley & Sons, 2011.

[NY14] Nadkarni, A. & Yezhkova, N. “Structured Versus Unstructured Data: The Balance of Power Continues to Shift”. *IDC (Industry Development and Models) Mar* (2014).

[NB05] Nagappan, N. & Ball, T. “Static analysis tools as early indicators of pre-release defect density”. *Proceedings of the 27th international conference on Software engineering*. ACM. 2005, pp. 580–586.

[Nag06] Nagappan, N. et al. “Mining metrics to predict component failures”. *Proceedings of the 28th international conference on Software engineering*. ACM. 2006, pp. 452–461.

[Nai18] Nair, V. et al. “Finding faster configurations using Flash”. *arXiv preprint arXiv:1801.02175* (2018).

[Nam17] Nam, J. et al. “Heterogeneous defect prediction”. *IEEE Transactions on Software Engineering* (2017).

[Neb08] Nebro, A. J. et al. “AbYSS: Adapting scatter search to multiobjective optimization”. *Evolutionary Computation, IEEE Transactions on* **12**.4 (2008), pp. 439–457.

[Nik15] Nikolenko, S. I. et al. “Topic modelling for qualitative studies”. *Journal of Information Science* (2015), p. 0165551515617393.

[Oli10a] Oliveira, A. L. et al. “GA-based method for feature selection and parameters optimization for machine learning regression applied to software effort estimation”. *information and Software Technology* **52**.11 (2010), pp. 1155–1166.

[Oli10b] Oliveto, R. et al. “On the equivalence of information retrieval methods for automated traceability link recovery”. *Program Comprehension (ICPC), 2010 IEEE 18th International Conference on*. IEEE. 2010, pp. 68–71.

[Omr05] Omran, M. G. et al. “Differential evolution methods for unsupervised image classification”. *Evolutionary Computation, 2005. The 2005 IEEE Congress on*. Vol. 2. IEEE. 2005, pp. 966–973.

[OR14] Orso, A. & Rothermel, G. “Software testing: a research travelogue (2000–2014)”. *Proceedings of the on Future of Software Engineering*. ACM. 2014, pp. 117–132.

[Ost04] Ostrand, T. J. et al. “Where the bugs are”. *ACM SIGSOFT Software Engineering Notes*. Vol. 29. 4. ACM. 2004, pp. 86–96.

[Ost05] Ostrand, T. J. et al. “Predicting the location and number of faults in large software systems”. *IEEE Transactions on Software Engineering* **31**.4 (2005), pp. 340–355.

[Pal13] Palomba, F. et al. “Detecting bad smells in source code using change history information”. *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press. 2013, pp. 268–278.

[Pan08] Pan, H. et al. “Particle swarm-simulated annealing fusion algorithm and its application in function optimization”. *Computer Science and Software Engineering, 2008 International Conference on*. Vol. 1. IEEE. 2008, pp. 78–81.

[Pan13] Panichella, A. et al. “How to effectively use topic models for software engineering tasks? an approach based on genetic algorithms”. *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press. 2013, pp. 522–531.

[Pan07] Panjer, L. D. “Predicting eclipse bug lifetimes”. *Proceedings of the Fourth International Workshop on mining software repositories*. IEEE Computer Society. 2007, p. 29.

[Pea14] Pears, R. et al. “Synthetic Minority over-sampling technique (SMOTE) for predicting software build outcomes”. *arXiv preprint arXiv:1407.2330* (2014).

[PD07] Pelayo, L. & Dick, S. “Applying novel resampling strategies to software defect prediction”. *NAFIPS 2007-2007 Annual Meeting of the North American Fuzzy Information Processing Society*. IEEE. 2007, pp. 69–72.

[PD12] Pelayo, L. & Dick, S. “Evaluating stratification alternatives to improve software defect prediction”. *IEEE Transactions on Reliability* **61**.2 (2012), pp. 516–525.

[PM18] Petke, J. & Menzies, T. “Guest Editorial for the Special Section from the 9th International Symposium on Search Based Software Engineering”. *Information and Software Technology* (2018).

[Phi17] Phillips, N. D. et al. “FFTrees: A toolbox to create, visualize, and evaluate fast-and-frugal decision trees”. *Judgment and Decision Making* **12**.4 (2017), p. 344.

[Por80] Porter, M. *The Porter Stemming Algorithm*. 1980.

[Rad13] Radjenović, D. et al. “Software fault prediction metrics: A systematic literature review”. *Information and Software Technology* **55**.8 (2013), pp. 1397–1418.

[Rah14] Rahman, F. et al. “Comparing Static Bug Finders and Statistical Prediction”. ICSE. New York, NY, USA: ACM, 2014, pp. 424–434.

[RK11] Rao, S. & Kak, A. “Retrieval from software libraries for bug localization: a comparative study of generic and composite text models”. *Proceedings of the 8th Working Conference on Mining Software Repositories*. ACM. 2011, pp. 43–52.

[Ree17] Rees-Jones, M. et al. “Better Predictors for Issue Lifetime”. *CoRR* **abs/1702.07735** (2017).

[Riq08] Riquelme, J. et al. “Finding defective modules from highly unbalanced datasets”. *Actas de los Talleres de las Jornadas de Ingeniería del Software y Bases de Datos* **2**.1 (2008), pp. 67–74.

[Sar12] Sarro, F. et al. “A further analysis on the use of genetic algorithm to configure support vector machines for inter-release fault prediction”. *Proceedings of the 27th annual ACM symposium on applied computing*. ACM. 2012, pp. 1215–1220.

[Sar16] Sarro, F. et al. “Multi-objective software effort estimation”. *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*. IEEE. 2016, pp. 619–630.

[Scu10] Sculley, D. “Web-scale k-means clustering”. *Proceedings of the 19th international conference on World wide web*. ACM. 2010, pp. 1177–1178.

[She14] Shepperd, M. et al. “Researcher bias: The use of machine learning in software defect prediction”. *IEEE Transactions on Software Engineering* **40**.6 (2014), pp. 603–616.

[SC08] Steinwart, I. & Christmann, A. *Support vector machines*. Springer Science & Business Media, 2008.

[SP97] Storn, R. & Price, K. “Differential evolution–a simple and efficient heuristic for global optimization over continuous spaces”. *Journal of global optimization* **11**.4 (1997), pp. 341–359.

[Sun15] Sun, X. et al. “MSR4SM: Using topic models to effectively mining software repositories for software maintenance tasks”. *Information and Software Technology* **66** (2015), pp. 1–12.

[Sun16] Sun, X. et al. “Exploring topic models in software engineering data analysis: A survey”. *2016 17th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*. IEEE. 2016, pp. 357–362.

[Tan15] Tan, M. et al. “Online defect prediction for imbalanced data”. *ICSE-Volume 2*. IEEE Press. 2015, pp. 99–108.

[Tan16] Tantithamthavorn, C. et al. “Automated parameter optimization of classification techniques for defect prediction models”. *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*. IEEE. 2016, pp. 321–332.

[Tas02] Tassey, G. “The economic impacts of inadequate infrastructure for software testing”. *National Institute of Standards and Technology* (2002).

[Tem10] Tempero, E. et al. “Qualitas Corpus: A Curated Collection of Java Code for Empirical Studies”. *2010 Asia Pacific Software Engineering Conference (APSEC2010)*. 2010, pp. 336–345.

[The15] Theisen, C. et al. “Approximating attack surfaces with stack traces”. *Proceedings of the 37th International Conference on Software Engineering-Volume 2*. IEEE. 2015, pp. 199–208.

[Tho11] Thomas, S. W. “Mining software repositories using topic models”. *Proceedings of the 33rd International Conference on Software Engineering*. ACM. 2011, pp. 1138–1139.

[Tho14a] Thomas, S. W. et al. “Static test case prioritization using topic models”. *Empirical Software Engineering* **19**.1 (2014), pp. 182–212.

[Tho14b] Thomas, S. W. et al. “Studying software evolution using topic models”. *Science of Computer Programming* **80** (2014), pp. 457–479.

[TW18] Treude, C. & Wagner, M. “Per-Corpus Configuration of Topic Modelling for GitHub and Stack Overflow Collections”. *arXiv preprint arXiv:1804.04749* (2018).

[VH07] Van Hulse, J. et al. “Experimental perspectives on learning from imbalanced data”. *Proceedings of the 24th international conference on Machine learning*. ACM. 2007, pp. 935–942.

[VT04] Vesterstrøm, J. & Thomsen, R. “A comparative study of differential evolution, particle swarm optimization, and evolutionary algorithms on numerical benchmark problems”. *Evolutionary Computation, 2004. CEC2004. Congress on*. Vol. 2. IEEE. 2004, pp. 1980–1987.

[Wan18] Wan, Z. et al. “Perceptions, Expectations, and Challenges in Defect Prediction”. *IEEE Transactions on Software Engineering* (2018), pp. 1–1.

[WY13] Wang, S. & Yao, X. “Using class imbalance learning for software defect prediction”. *IEEE Transactions on Reliability* **62**.2 (2013), pp. 434–443.

[Wan13] Wang, T. et al. “Searching for better configurations: a rigorous approach to clone evaluation”. *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM. 2013, pp. 455–465.

[Wil03] Williams, R. et al. “Backdoors to typical case complexity”. *IJCAI*. Vol. 3. Citeseer. 2003, pp. 1173–1178.

[WF02] Witten, I. H. & Frank, E. “Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations”. *SIGMOD Rec.* **31**.1 (2002), pp. 76–77.

[Xu16] Xu, B. et al. “Predicting semantically linkable knowledge in developer online forums via convolutional neural network”. *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM. 2016, pp. 51–62.

[YC13] Yamashita, A. & Counsell, S. “Code smells as system-level indicators of maintainability: An empirical study”. *Journal of Systems and Software* **86**.10 (2013), pp. 2639–2653.

[YM13] Yamashita, A. & Moonen, L. “Exploring the impact of inter-smell relations on software maintainability: An empirical study”. *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press. 2013, pp. 682–691.

[Yan10] Yan, Z. et al. “Software defect prediction using fuzzy support vector regression”. *International Symposium on Neural Networks*. Springer. 2010, pp. 17–24.

[Yan12] Yang, J. et al. “Filtering clones for individual user based on machine learning analysis”. *2012 6th International Workshop on Software Clones (IWSC)*. 2012, pp. 76–77.

[Yan16] Yang, Y. et al. “Effort-aware just-in-time defect prediction: simple unsupervised models could be better than supervised models”. *Proceedings of the 2016 24th ACM SIGSOFT*

International Symposium on Foundations of Software Engineering. ACM. 2016, pp. 157–168.

[YH12] Yoo, S. & Harman, M. "Regression testing minimization, selection and prioritization: a survey". *Software Testing, Verification and Reliability* **22**.2 (2012), pp. 67–120.

[Yu17] Yu, Q. et al. "The Performance Stability of Defect Prediction Models with Class Imbalance: An Empirical Study". *IEICE TRANSACTIONS on Information and Systems* **100**.2 (2017), pp. 265–272.

[Zaz11] Zazworka, N. et al. "Investigating the impact of design debt on software quality". *Proceedings of the 2nd Workshop on Managing Technical Debt*. ACM. 2011, pp. 17–23.

[Zha13] Zhang, H. et al. "Predicting bug-fixing time: an empirical study of commercial software projects". *Proceedings of the 2013 international conference on software engineering*. IEEE Press. 2013, pp. 1042–1051.

[Zho04] Zhong, S. et al. "Analyzing software measurement data with clustering techniques". *IEEE Intelligent Systems* **19**.2 (2004), pp. 20–27.

[ZN08] Zimmermann, T. & Nagappan, N. "Predicting defects using network analysis on dependency graphs". *ICSE*. ACM. 2008, pp. 531–540.

[Zim07] Zimmermann, T. et al. "Predicting defects for eclipse". *Proceedings of the third international workshop on predictor models in software engineering*. IEEE Computer Society. 2007, p. 9.

[Zul13] Zuluaga, M. et al. "Active learning for multi-objective optimization". *Proceedings of the 30th International Conference on Machine Learning*. 2013, pp. 462–470.