ABSTRACT

FU, WEI. Simpler Software Analytics: When? When not?. (Under the direction of Dr. Timothy Menzies.)

Software engineering researchers and industrial practitioners routinely make extensively use of software analytics to discover insights about their projects: how long it will take to develop their code, or where bugs are most likely to occur. Significant research effort has been put forth to improve the quality of model software analytics. There has been continuous development of new feature selection, feature discovering and even new modeling techniques for software analytics. These exciting innovations have the potential to dramatically improve the quality of software analytics tools. The problem is that most of these methods are complex (e.g.,): not easy to explain, hard to reproduce, and require large amount of computational resources.

In this dissertation, we want to investigate the potential of simpler software analytics. The thesis of this dissertation is that software analytics should be simpler; Software analytics can be simpler; However, oversimplification can be harmful. To defend the claim of the thesis, we have conducted several studies to explore simple tools in different tasks, like software defect prediction and software text mining.

First of all, we discuss why software analytics should be simpler. From the perspectives of effectiveness, economy, explainability and reproducibility, we argue that simpler software analytics have more potential to impact the software engineering industry. To show software analytics can be simpler, we conduct a study to simplify hyper-parameter tuning techniques on software defect prediction tasks. According to our literature review, hyper-parameter tuning was ignored in most software defect prediction works. Even though some researchers proposed methods to tune learners' hyper-parameters, those methods usually take at least hours to complete, which would not motivate researchers to tune hyper-parameters in every single task. In this dissertation, we proposed to apply search-based software engineering method, like differential evolution algorithm, to tune defect predictors. According to the experiment, we usually get better performance than predictors with default parameters. Meanwhile, tuning with differential evolution can be terminated within seconds or minutes on defect prediction tasks, which requires far less resources than most frequently used hyper-parameter tuning techniques proposed by other researchers.

Over years, many software analytics tasks have been solved by using various techniques. However, these techniques are becoming more and more complex and resource consuming. For example, deep learning methods have been proposed to solve tasks like defect prediction and effort estimation. In this dissertation, to show that software analytics can be simplified and simpler methods can reach parity with complex methods with much lower execution costs and less understanding barriers, we reproduced a case study where deep learning methods were proposed to predict similarities
between different questions posted on Stack Overflow. First of all, it is hard to select a case study like the one we studied to explore simplicity versus complexity because most papers using complex methods (e.g., deep learning) do not share either the source code or data, which could introduce bias if we explore the same topic with different datasets or implement the proposed method. In this study, even though we got the same testing data from the author we compared with, we are still missing training data. Therefore, to fully reproduce the experiment, we spend months of time in collecting data, pre-processing data and reproducing the prior study. Based on the experimental results, we find that tuning simple learner with differential evolution algorithm outperforms the deep learning method for this task while our proposed method is 84 times faster. This conclusion further confirms that our first claim that software analytics can be simpler and complex software analytics methods can be improved and outperformed by simpler methods. Since most existing software analytics research tends to be more complex and not fully open-sourced, showing any single software analytics tasks can be simplified is not simple at all.

Thirdly, we show that simplification for software analytics can be done in a harmful way. Specifically, over-simplification without considering the local information can generate a less effective model. By reproducing a recent study in FSE’16 about just-in-time effort aware defect prediction, we find that the unsupervised predictors proposed in the original paper can not get a consistent prediction among all proposed unsupervised learners. To improve their method, we investigated the use of local data to prune weak predictors and only select the best one as the final predictor. Experimental results show that our proposed simple method, OneWay, performs better than the unsupervised learners as well as more complex standard supervised learners. Through this study, we defend our claim that software analytics can be simpler but can not be oversimplified.

Overall, since we have shown that simpler software analytics methods work pretty well on tasks like software defect prediction and text classification tasks, we recommend software analytics researchers should consider simple techniques in the future.
© Copyright 2018 by Wei Fu

All Rights Reserved
Simpler Software Analytics: When? When not?

by

Wei Fu

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

2018

APPROVED BY:

Dr. Min Chi

Dr. Christopher Parnin

Dr. Ranga Vatsavai

Dr. Timothy Menzies
Chair of Advisory Committee
DEDICATION

To my parents: Decai Fu and Chun'e Tang.
BIOGRAPHY

Wei Fu was born in a small town called Fengcheng, Liaoning, China. He obtained a Bachelor of Science in Electrical Engineering from Nanjing Tech University in 2009. After that, he continued his graduate study at Beijing University of Posts and Telecommunications and received a Master of Science in Electrical Engineering in 2012. During his gap year (2012 to 2013), he was a visiting student of Automation Department at Tsinghua University, where he worked with Dr. Feifei Gao, mainly focusing on signal processing in wireless communication networks. In 2013, Wei began his PhD journey at NC State University and started exploring software engineering research under the direction of Dr. Tim Menzies since 2014.
ACKNOWLEDGEMENTS

A Five-year PhD study is a special journey in my life. I could not finish it without the support and encouragement from many nice people I meet over this time. Therefore, I would like to thank the following people.

I am grateful to my advisor, Dr. Tim Menzies, for the time and effort he took in introducing me to software engineering research, for all his enthusiasm, patience, advice, and funding. That means a lot to me. Our story informally started from the moment that I followed him on Facebook in 2014 summer. Back then, I was down and about to quit my PhD program while he just started his career at NC State. As his first NC State graduate student who happened to have no computer science background, he always patiently explains every single knowledge/idea that I could not understand. Sometimes, even repeating English words until I get them. Over past four years, he raised me up, so that I can stand on mountains; he raised me up to more than I can be. This thesis could not have been possible without his help on every single step during my PhD study.

I am grateful to my PhD committee members, Dr. Min Chi, Dr. Chris Parnin, and Dr. Raju Vatsavai, for all the time spent, all the help and advice on my research and career.

I am grateful to the members of Dr. Tim Menzies’ RAISE lab: Vivek Nair, Rahul Krishna, George Mathew, Jianfeng Chen, Zhe Yu, Amritanshu Agrawal, Di Chen, Tianpei Xia, Huy Tu, Suvodeep Majumder, and our distinguished visiting scholar, Dr. Junjie Wang. Thanks for all the critiques on my work, project collaboration, and even paper proof reading. Those tears, beers, woes, and joys that we shared together are great moments that will never be forgot. I am grateful to the RAISE Lab founding members: Vivek Nair, Rahul Krishna and George Mathew, who taught me not only the new computer skills but also the old Indian culture. I will miss those days and nights that I stood around and listened to random research discussions in room 3231, even though most of them turned out to not work at all. Special thanks to Vivek Nair, who has been serving as my “vice-advisor” over past 4 years when the actual one is not on. I wish I could put him on my committee even though our first co-authored work went nowhere.

I am grateful to my friends I made at NC State. This list can only be partial: Qiang(Jack) Zhang, Junjiamin Mu, Chen(Liana) Lin, Feifei Wang, Shengpei Zhang, Qiong Tao, Kelei Gong, Jiaming Li, Pei Deng, Xing Pan, Peipei Wang, Hui Guan, Liang Dong, Xiaozhou Fang, Hong Xiong, Xin Xu, Akhilesh Tanneeru, Adam Gillfillan. Thanks for your invaluable support, advice, and inspiring ideas. Those are really important during my whole PhD study, especially my tough time.

I am grateful to some old friends, who are not even in US but always stay with me virtually whenever I was down and need help. I am grateful to the “Eight Immortals”, who were my lab mates of Beijing University of Posts and Telecommunications: Zhen Li, Linjie Hu, Renfeng Liu, Wangbo Deng, Fei Deng, Ying Deng and Minggang Chen. Even though it has been 6 years since we graduated,
we still keep in touch and the tremendous support from this group of friends keeps me moving forward. Special thanks to Dr. Feifei Gao from Tsinghua University, I could not even start my PhD study without his advice, encouragement and recommendation.

Most of all, I am grateful to the family I came from: as the first-generation college graduate, I am proud of who I am and where I come from. Thanks to my parents - Decai Fu and Chun’e Tang, brother - Jishuang Fu, sister-in-law - Yang Xu, and nephew - Boji Fu, for their abundant love and support over years, which motivates me to keep fighting.
TABLE OF CONTENTS

LIST OF TABLES .............................................................................................................. ix

LIST OF FIGURES ......................................................................................................... xi

Chapter 1 Introduction ........................................................................................................ 1
  1.1 Background .............................................................................................................. 1
    1.1.1 Software Analytics ......................................................................................... 1
    1.1.2 Defect Prediction ............................................................................................ 2
  1.2 Contributions .......................................................................................................... 5

I Software Analytics Should Be Simpler ........................................................................ 8

Chapter 2 The Quest for Simplicity in Software Analytics ............................................. 9

II Software Analytics Can Be Simpler ............................................................................. 12

Chapter 3 Tuning for Software Analytics ....................................................................... 13
  3.1 Introduction ............................................................................................................ 14
  3.2 Preliminaries .......................................................................................................... 15
    3.2.1 Tuning: Important and Ignored ...................................................................... 15
    3.2.2 You Can't Always Get What You Want ......................................................... 16
    3.2.3 Notes on Data Miners .................................................................................. 18
    3.2.4 Learners and Their Tunings ......................................................................... 18
    3.2.5 Tuning Algorithms ...................................................................................... 21
  3.3 Experimental Design .............................................................................................. 23
    3.3.1 Data Sets ...................................................................................................... 23
    3.3.2 Optimization Goals ..................................................................................... 24
  3.4 Experimental Results .............................................................................................. 24
    3.4.1 RQ1: Does Tuning Improve Performance? .................................................. 24
    3.4.2 RQ2: Does Tuning Change a Learner's Ranking? ....................................... 25
    3.4.3 RQ3: Does Tuning Select Different Project Factors? ................................. 28
    3.4.4 RQ4: Is Tuning Easy? .................................................................................. 29
    3.4.5 RQ5: Is Tuning Impractically Slow? .............................................................. 30
    3.4.6 RQ6: Should we use "off-the-shelf" Tunings? .............................................. 32
  3.5 Reliability and Validity ............................................................................................ 33
  3.6 Conclusions ............................................................................................................ 33

Chapter 4 Differential Evolution V.S. Grid Search .......................................................... 35
  4.1 Introduction ............................................................................................................ 36
  4.2 Background ............................................................................................................ 38
    4.2.1 Data Mining .................................................................................................. 38
LIST OF TABLES

| Table 1.1 | OO Measures used in our defect data sets. | 4 |
| Table 3.1 | List of parameters tuned by this study. | 19 |
| Table 3.2 | Data used in this experiment. E.g., the top left data set has 20 defective classes out of 125 total. See §3.3.1 for explanation of training, tuning, testing sets. | 23 |
| Table 3.3 | Precision results (best results shown in bold). | 26 |
| Table 3.4 | F-measure results (best results shown in bold). | 26 |
| Table 3.5 | Features selected by tuned WHERE with different goals: bold features are those found useful by the tuned WHERE. Also, features shown in plain text are those found useful by the untuned WHERE. | 28 |
| Table 3.6 | Kolmogorov-Smirnov Tests for distributions of Figure 3.3 | 30 |
| Table 3.7 | Number of evaluations for tuned learners, optimizing for precision and F-Measure. | 31 |
| Table 3.8 | Runtime for tuned and default learners (in sec), optimizing for precision and F-Measure. | 31 |
| Table 4.1 | List of parameters tuned by this study. | 39 |
| Table 4.2 | Data used in this case study. | 43 |
| Table 5.1 | Classes of knowledge unit pairs. | 64 |
| Table 5.2 | List of parameters tuned by this study. | 64 |
| Table 5.3 | Confusion matrix. | 70 |
| Table 5.4 | Comparison of our baseline method with XU's. The best scores are marked in bold. | 72 |
| Table 5.5 | Comparison of the tuned SVM with XU's CNN method. The best scores are marked in bold. | 74 |
| Table 5.6 | Comparison of experimental environment | 76 |
| Table 6.1 | Change metrics used in our data sets. | 87 |
| Table 6.2 | Statistics of the studied data sets | 90 |
| Table 6.3 | Comparison in $P_{opt}$: Yang's method (A) vs. our implementation (B) | 94 |
| Table 6.4 | Comparison in Recall: Yang's method (A) vs. our implementation (B) | 94 |
| Table 6.5 | Best unsupervised predictor (A) vs. OneWay (B). The colorful cell indicates the size effect: green for large; yellow for medium; gray for small. | 97 |
| Table 6.6 | Best supervised predictor (A) vs. OneWay (B). The colorful cell indicates the size effect: green for large; yellow for medium; gray for small. | 100 |
| Table 7.1 | 32 defect predictors clustered by their performance rank by Ghotra et al. (using a Scott-Knot statistical test) [Gho15]. | 108 |
| Table 7.2 | Statistics of the studied data sets. | 118 |
| Table 7.3 | DART vs. state-of-the-art defect predictors from Ghotra et al. [Gho15] for dis2heaven and $P_{opt}$. Gray cells mark best performances on each project (so DART is most-often best). | 118 |
| Table 7.4 | List of parameters tuned in this study. | 120 |
### Table 7.5
DART v.s. tuning Random Forests for *dis2heaven* and *P*\textsubscript{opt}. Gray cells mark best performances on each project. Note that even when DART does not perform best, it usually performs very close to the best.

### Table 7.6
DART v.s. data-savvy learners in *dis2heaven* and *P*\textsubscript{opt}. Gray cells mark best performances on each project.
**LIST OF FIGURES**

| Figure 3.1 | Deltas in performance seen in Table 3.3 (left) and Table 3.4 (right) between tuned and untuned learners. Tuning improves performance when the deltas are above zero. | 25 |
| Figure 3.2 | Comparison between Logistic Regression and Random Forest before and after tuning. | 27 |
| Figure 3.3 | Deltas in performance between $np = 10$ and the recommended np's. The recommended np is better when deltas are above zero. $np = 90, 50 \text{ and } 60$ are recommended population size for WHERE, CART and Random Forest by Storn. | 29 |
| Figure 3.4 | Four representative tuning values in WHERE with precision and F-measure as the tuning goal, respectively. | 32 |
| Figure 4.1 | Literature review about parameter tuning on 52 top cited defect prediction papers | 40 |
| Figure 4.2 | Tuning to improve $F$. Median results from 20 repeats. | 45 |
| Figure 4.3 | Tuning to improve Precision. Median results from 20 repeats. | 46 |
| Figure 4.4 | Tuning to improve AUC. Median results from 20 repeats. | 47 |
| Figure 4.5 | Tuning to improve $AUC_{20}$. Median results from 20 repeats. | 47 |
| Figure 4.6 | Time (in seconds) required to run 20 repeats of parameter tunings for learners to generate Figure 4.2 to Figure 4.5. | 48 |
| Figure 4.7 | Runtime relative to just running the learners using their default tunings. | 49 |
| Figure 4.8 | Intrinsic dimensionality of our data. | 52 |
| Figure 4.9 | Illustration for data D=dimensionality | 52 |
| Figure 5.1 | Procedure TUNER: strives to find “good” tunings which maximizes the objective score of the model on training and tuning data. TUNER is based on Storn's differential evolution optimizer [SP97]. | 67 |
| Figure 5.2 | The overall workflow of building knowledge units predictor with tuned SVM. | 69 |
| Figure 5.3 | Score delta between our SVM with XU's SVM in [Xu16] in terms of precision, recall and F1-score. Positive values mean our SVM is better than XU's SVM in terms of different measures; Otherwise, XU's SVM is better. | 72 |
| Figure 5.4 | Score delta between tuned SVM and CNN method [Xu16] in terms of precision, recall and F1-score. Positive values mean tuned SVM is better than CNN in terms of different measures; Otherwise, CNN is better. | 74 |
| Figure 5.5 | Score delta between the tuned SVM and XU's baseline SVM in terms of precision, recall and F1-score. Positive values mean the tuned SVM is better than XU's SVM in terms of different measures; Otherwise, XU's SVM is better. | 75 |
| Figure 5.6 | Score delta between tuned SVM and Our Untuned SVM in terms of precision, recall and F1-score. Positive values mean the tuned SVM is better than our untuned SVM in terms of different measures; Otherwise, our SVM is better. | 75 |
| Figure 6.1 | Example of an effort-based cumulative lift chart [Yan16b]. | 92 |
| Figure 6.2 | Performance comparisons between supervised and unsupervised predictors over six projects (from top to bottom are Bugzilla, Platform, Mozilla, JDT, Columba, PostgreSQL). | 96 |
Figure 6.3  Performance comparisons between the proposed OneWay learner and unsupervised predictors over six projects (from top to bottom are Bugzilla, Platform, Mozilla, JDT, Columba, PostgreSQL). .................................................. 98

Figure 6.4  Performance comparisons between the proposed OneWay learner and supervised predictors over six projects (from top to bottom are Bugzilla, Platform, Mozilla, JDT, Columba, PostgreSQL). .................................................. 100

Figure 7.1  Illustration of $P_{opt}$ metric. .......................................................... 109

Figure 7.2  Illustration of $dis2heaven$ metric. .................................................. 110

Figure 7.3  Results space when recall is greater than false alarms (see blue curve). .... 111

Figure 7.4  Grids in results space. ................................................................. 113

Figure 7.5  $\epsilon$ can vary across results space. 100 experiments with LUCENE results using 90% of the data, for training, and 10%, for testing. The x-axis sorts the code base, first according to predicted defective or not, then second on lines of code. The blue line shows the distribution of the 100 results across this space. For example, 70% of the results predict defect for up to 20% of the code (see the blue curve). The y-axis of this figures shows mean recall (in red); the standard deviation $\sigma$ of the recall (in yellow); and $\epsilon$ is defined as per standard t-tests that says at the 95% confidence level, two distributions differ when they are $\epsilon = 2 \times 1.96 \times \sigma$ apart. (in green). ........................................................................................................ 114

Figure 7.6  DART: an ensemble algorithm to sample results space, $M$ number of times. . 116

Figure 7.7  A simple model for software defect prediction ........................................ 116

Figure 7.8  TUNER is an evolutionary optimization algorithm based on Storn's differential evolution algorithm [SP97; FM17b]. ................................................................. 121

Figure 7.9  Different ways to reason about software quality prediction. ...................... 125
1.1 Background

1.1.1 Software Analytics

In the 21st century, it is impossible to manually browse all available software project data. The SeaCraft repository of SE data has grown to 200+ projects [Sea] and this is just one of over a dozen open-source repositories that are readily available to researchers [Rod12]. Faced with this data overload, researchers in empirical software engineering field and software practitioners used analytics, like data mining tools, to generate models to get better understanding of the software projects. According to Menzies et al. [MZ13a], software analytics is defined as: “software analytics is analytics on software data for managers and software engineers with the aim of empowering software development individuals and teams to gain and share insight from their data to make better decisions.”

Software analytics have been applied to many applications in Software Engineering (SE). For example, it has been used to estimate how long it would take to integrate new code into an existing project [Cze11a], where defects are most likely to occur [Ost04; Men07c], which two questions asked by software developers are related [Xu16], or how long will it take to develop a project [Tur11; Koc12a], etc. Large organizations, like Microsoft, routinely practice data-driven policy development where organizational policies are learned from an extensive analysis of large data sets [BZ14; The15a].

Despite these successes, there exist several drawbacks with current software analytic tools. Recently, many complex methods have been proposed to improve software analytics, which makes
it harder for the users to reproduce models, understand models or operationalize models. For example, there are some evidence to show that deep learning methods are very effective for building software analytics models based on high dimensional data [Xu16; Wan16; Gu16]. However, since most deep learning methods have very complex structures and a tiny difference may lead to a huge difference in the results. For example, different weights initialization schemes will have great impacts on the speed of network convergence. Therefore, it is hard to reproduce the invented models by a third-party if the original research paper did not share the model; furthermore, most deep learning methods require weeks of hours of GPU/CPU time to train a model on a moderate size of data set [Xu16], much of that GPU/CPU time can be saved if there is a faster method for this analytics. Therefore, the total financial cost of training deep learning models can be prohibitive, particularly for long running tasks. In addition, the most obvious drawback of deep learning methods is that they do not readily support explainability, they have been criticizing as “data mining alchemy” [Syn17]. At a recent workshop on “Actionable Analytics” at ASE’15, business users were very vocal in their complaints about analytics [HM15], saying that there are rarely producible models that business users can understand or operationalize.

In software analytics, if a model is to be used to persuade software engineers to change what they are doing, it needs to be comprehensible so humans can debate the merits of its conclusions. Several researchers demand that software analytics models needs to be expressed in a simple way that is easy for software practitioners to interpret [Men14a; Lip16; Dam18]. According to Kim et al. [Kim16], software analytics aim to obtain actionable insights from software artifacts that help practitioners accomplish tasks related to software development, systems, and users. Other researchers [TC16] argue that for software vendors, managers, developers and users, such comprehensible insights are the core deliverable of software analytics. Sawyer et al. comments that actionable insight is the key driver for businesses to invest in data analytics initiatives [Saw13]. Accordingly, much research focuses on the generation of simple models, or making blackbox models more explainable, so that human engineers can understand and appropriately trust the decisions made by software analytics models [FM17b; AN16].

Based the drawbacks we discussed above, we think that we need simple, actionable, and explainable software analytics models. Therefore, for software researchers and practitioners, it is the time to simplify software analytics from the perspectives of effectiveness, economy, explainability and reproducibility. In this dissertation, I will show what I have learned about simplicity in software analytics.

1.1.2 Defect Prediction

This section discusses defect prediction, which has been served as a case study in this dissertation across several chapters.
Human programmers are clever, but flawed. Coding adds functionality, but also defects. Hence, software sometimes crashes (perhaps at the most awkward or dangerous moment) or delivers the wrong functionality. For a very long list of software-related errors, see Peter Neumann’s “Risk Digest” at catless.ncl.ac.uk/Risks.

Since programming inherently introduces defects into programs, it is important to test them before they are used. Testing is expensive. Software assessment budgets are finite while assessment effectiveness increases exponentially with assessment effort. For example, for black-box testing methods, a linear increase in the confidence $C$ of finding defects can take exponentially more effort:

- A randomly selected input to a program will find a fault with probability $p$.
- After $N$ random black-box tests, the chances of the inputs not revealing any fault is $(1 - p)^N$.
- Hence, the chances $C$ of seeing the fault is $1 - (1 - p)^N$ which can be rearranged to $N(C, p) = \log(1 - C)/\log(1 - p)$.
- For example, $N(0.90, 10^{-3}) = 2301$ but $N(0.98, 10^{-3}) = 3901$; i.e. nearly double the number of tests.

Exponential costs quickly exhaust finite resources so standard practice is to apply the best available methods on code sections that seem most critical. But any method that focuses on parts of the code can blind us to defects in other areas. Some lightweight sampling policy should be used to explore the rest of the system. This sampling policy will always be incomplete. Nevertheless, it is the only option when resources prevent a complete assessment of everything.

One such lightweight sampling policy is defect predictors learned from static code attributes. Given software described in the attributes of Table 1.1, data miners can learn where the probability of software defects is highest. The rest of this section argues that such defect predictors are easy to use, widely-used, and useful to use.

**Easy to use:** Static code attributes can be automatically collected, even for very large systems [NB05a]. Other methods, like manual code reviews, are far slower and far more labor-intensive. For example, depending on the review methods, 8 to 20 LOC/minute can be inspected and this effort repeats for all members of the review team, which can be as large as four or six people [Men02].

**Widely used:** Researchers and industrial practitioners use static attributes to guide software quality predictions. Defect prediction models have been reported at Google [Lew13]. Verification and validation (V&V) textbooks [Rak01] advise using static code complexity attributes to decide which modules are worth manual inspections.

**Useful:** Defect predictors often find the location of 70% (or more) of the defects in code [Men07b]. Defect predictors have some level of generality: predictors learned at NASA [Men07b] have also been found useful elsewhere (e.g. in Turkey [Tos10; Tos09]). The success of this method in predictors
### Table 1.1 OO Measures used in our defect data sets.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>amc</td>
<td>average method complexity</td>
<td>Number of Java byte codes</td>
</tr>
<tr>
<td>avg_cc</td>
<td>average McCabe</td>
<td>Average McCabe's cyclomatic complexity seen in class</td>
</tr>
<tr>
<td>ca</td>
<td>afferent couplings</td>
<td>How many other classes use the specific class</td>
</tr>
<tr>
<td>cam</td>
<td>cohesion amongst classes</td>
<td>#different method parameters types divided by (#different method parameter types in a class)/*#methods).</td>
</tr>
<tr>
<td>cbm</td>
<td>coupling between methods</td>
<td>Total number of new/redefined methods to which all the inherited methods are coupled</td>
</tr>
<tr>
<td>cbo</td>
<td>coupling between objects</td>
<td>Increased when the methods of one class access services of another.</td>
</tr>
<tr>
<td>ce</td>
<td>efferent couplings</td>
<td>How many other classes is used by the specific class</td>
</tr>
<tr>
<td>dam</td>
<td>data access</td>
<td>Ratio of private (protected) attributes to total attributes</td>
</tr>
<tr>
<td>dit</td>
<td>depth of inheritance tree</td>
<td>It’s defined as the maximum length from the node to the root of the tree</td>
</tr>
<tr>
<td>ic</td>
<td>inheritance coupling</td>
<td>Number of parent classes to which a given class is coupled (includes counts of methods and variables inherited)</td>
</tr>
<tr>
<td>lcom</td>
<td>lack of cohesion in methods</td>
<td>Number of pairs of methods that do not share a reference to an instance variable.</td>
</tr>
<tr>
<td>locm3</td>
<td>another lack of cohesion measure</td>
<td>m, a count the methods, attributes in a class. μ(a) is the number of methods accessing an attribute. locm3=((1/2)∑jμ(a_j))−m)/(1−m).</td>
</tr>
<tr>
<td>loc</td>
<td>lines of code</td>
<td>Total lines of code in this file or package</td>
</tr>
<tr>
<td>max_cc</td>
<td>Maximum McCabe</td>
<td>maximum McCabe's cyclomatic complexity seen in class</td>
</tr>
<tr>
<td>mfa</td>
<td>functional abstraction</td>
<td>Number of methods inherited by a class plus number of methods accessible by member methods of the class</td>
</tr>
<tr>
<td>moa</td>
<td>aggregation</td>
<td>Count of the number of data declarations (class fields) whose types are user-defined classes</td>
</tr>
<tr>
<td>nmc</td>
<td>number of children</td>
<td>Number of direct descendants (subclasses) for each class</td>
</tr>
<tr>
<td>npm</td>
<td>number of public methods</td>
<td>npm metric simply counts all the methods in a class that are declared as public.</td>
</tr>
<tr>
<td>rfc</td>
<td>response for a class</td>
<td>Number of methods invoked in response to a message to the object.</td>
</tr>
<tr>
<td>wmc</td>
<td>weighted methods per class</td>
<td>A class with more member functions than its peers is considered to be more complex and therefore more error prone</td>
</tr>
<tr>
<td>defect</td>
<td>defect</td>
<td>Boolean: where defects found in post-release bug-tracking systems.</td>
</tr>
</tbody>
</table>
in finding bugs is markedly higher than other currently-used industrial methods such as manual code reviews. For example, a panel at IEEE Metrics 2002 [Shu02] concluded that manual software reviews can find \( \approx 60\% \) of defects. In another work, Raffo documents the typical defect detection capability of industrial review methods: around 50\% for full Fagan inspections [Fag76] to 21\% for less-structured inspections.

Not only do static code defect predictors perform well compared to manual methods, they also are competitive with certain automatic methods. A recent study at ICSE’14, Rahman et al. [Rah14] compared (a) static code analysis tools FindBugs, Jlint, and Pmd and (b) static code defect predictors (which they called “statistical defect prediction”) built using logistic regression. They found no significant differences in the cost-effectiveness of these approaches. Given this equivalence, it is significant to note that static code defect prediction can be quickly adapted to new languages by building lightweight parsers that find information like Table 1.1. The same is not true for static code analyzers—these need extensive modification before they can be used on new languages.

1.2 Contributions

Over my past five-year PhD study, I was trying to demonstrate how to reach such goal by using simple techniques to make software analytics easy. Based on those studies, I formulated the following thesis:

Software analytics should be simpler;
Software analytics can be simpler;
However, oversimplification can be harmful.

In the above, by “simpler” we mean simple methods over complex methods. Simple methods mean fast, requiring less computational resources, reproducible, and easy to explain and understand; and complex methods mean slow, requiring days to weeks of CPU time to execute, not reproducible, not easy to explain and understand;

This dissertation advances knowledge by several contributions. To defend that Software analytics can be simpler, I offer evidence through two case studies:

- **Tuning for Software Analytics**: The performance of software analytics can be improved from various perspectives and by different ways. On direction to reach this goal is to tune hyper-parameters of existing software analytics methods. In Chapter 3, we show that by applying one simple search-based software engineering method, differential evolution (DE), to tune software quality prediction methods, the detection precision of the resulting models are improved from 0\% to 60\% on different datasets.

- **Differential Evolution V.S. Grid Search**: Hyper-parameter tuning is well studied in other community, like pure machine learning. However, in software engineering, especially software...
analytics, it is still a quite under-explored problem. After the success with differential evolution for tuning software analytics, we did a literature review on parameter tuning methods for software quality prediction. We found that 94% papers we surveyed did not conduct parameter tuning and 4% papers mentioned that grid search was applied to to tune parameters. Since it’s easy to understand and implement and, to some extent, also has good performance, grid search has been available in most popular data mining and machine learning tools, like caret [Kuh14] package in the R and GridSearchCV module in Scikit-learn [Ped11]. In Chapter 3, we want to evaluate which method can find better parameters in terms of effectiveness (performance score) and economy (runtime cost). Experimental results show that the seemingly complete approach of grid search does no better, and sometimes worse, than differential evolution method. Since the best results from grid search are largely depending on grid points set by the user. Grid search is prone to skip optimal configurations and waste time exploring useless space. On the other hand, with the DE method, users do not need to use expert knowledge to set the grid points and the method itself will explore the searching space automatically. We are not going to sell DE as the best parameter tuning, but at least, it’s a good candidate for considering trade off between performance and runtime. Through this case study, we show that, simple hyper-parameter tuning methods, like differential evolution, perform no worse than resource consuming grid search method for hyper-parameter tuning.

To defend the claim that simpler methods can reach parity with complex methods with much lower execution costs and less understanding barriers, I offer the evidence through the following case study on text mining tasks:

**Easy Over Hard:** In Chapter 4, we extend a prior result from ASE’16 by Xu et al. [Xu16]. In their work, they described a deep learning method to explore large programmer discussion forums, then uncover related, but separate, entries. In this study, we spend weeks of time to collect, clean, and transform data from Stack Overflow. We strictly followed Xu et al.’s description to split data and train the Word2vec model. To show that simple techniques work better than complex ones, we apply DE to tune SVM model (which was used to serve as baseline method in Xu’s paper). We show here that applying a very simple optimizer DE to fine tune SVM, it can achieve similar (and sometimes better) results. The DE approach terminated in 10 minutes (while that deep learning system took 14 hours to execute.); i.e. 84 times faster hours than deep learning method.

To defend the claim that However, oversimplification can be harmful, I offer the evidence through the following case study on software quality prediction tasks:

**Unsupervised Learners:** Most software analytics methods require a large amount of training data to explore the targeting tasks. Collecting data from software projects can be time-consuming and expensive. Hence, some researchers explore “unsupervised” approaches to build prediction models that does not require labeled data. An alternate technique is to use “supervised” approaches
that learn models from project data labeled with, say, “defective” or “not-defective” for software quality prediction tasks. Most researchers use these supervised models since, it is argued, they can exploit more knowledge of the projects. At FSE’16, Yang et al. reported startling results where unsupervised defect predictors outperformed supervised predictors for effort-aware just-in-time defect prediction. If confirmed, these results would lead to a dramatic simplification of a seemingly complex task (data mining) that is widely explored in the software engineering literature. In Chapter 5, we repeat and refute Yang et al. results: (1) There is much variability in the efficacy of the Yang et al. unsupervised predictors so even with their approach, some supervised data is required to prune weaker predictors away. (2) Their findings were grouped across \( N \) projects. When we repeat their analysis on a project-by-project basis, supervised predictors are seen to work better. Therefore, when taking actions to simplify software analytics, we need to use caution to avoid pitfalls and make models more applicable.

Simplifying software analytics is not an easy task itself. Sometimes, we find that some methods work, most of time, it does not. Therefore, it is necessary to have a criterion or indicator to tell us when simple methods could work. We were trying to explore this question, and in the following study, we show some preliminary results in this direction:

**DART:** When we were trying to simplify software analytics, in other words, apply simple techniques, we find that many methods still exhibit some degree of uncertainty in their results. For example, Lessmann et al. reported that 17 of 22 studied data mining algorithms for defect prediction had statistically indistinguishable performance [Les08]. Also, Ghotra et al. reported that the performance of 32 data mining algorithms for defect prediction clustered into just four groups [Gho15]. In Chapter 6, we investigate if this uncertainty is a resource that can simplify software quality prediction. For example, Deb’s principle of \( \epsilon \)-dominance states that if there exists some \( \epsilon \) value below which it is useless or impossible to distinguish results, then it is superfluous to explore anything less than \( \epsilon \). We say that for “large \( \epsilon \) problems”, the results space of learning effectively contains just a few regions. If many learners are then applied to such large \( \epsilon \) problems, they would exhibit a “many roads lead to Rome” property; i.e., many different software quality prediction methods would generate a small set of very similar results. In this chapter, we explore DART, an algorithm especially selected to succeed for large \( \epsilon \) software quality prediction problems. DART is remarkable simple yet, on experimentation, it dramatically out-performs three sets of state-of-the-art defect prediction methods.
Part I

Software Analytics Should Be Simpler
In this chapter, we argue that software analytics should be simpler. We study simplicity since it is very useful to replace \( N \) methods with \( M \ll N \) methods, especially when the results from the many are no better than the few. Take a software quality prediction task as an example. Each year, a bewildering array of new methods for software quality prediction are reported (some of which rely on intimidatingly complex mathematical methods) such as deep belief net learning [Wan16], spectral-based clustering [Zha16], and n-gram language models [Ray16]. Ghotra et al. list dozens of different data mining algorithms that might be used for defect predictors [Gho15]. Fu and Menzies argue that these algorithms might require extensive tuning [Fu16a]. There are many ways to implement that tuning, some of which are very slow [Tan16]. And if they were not enough, other computationally expensive methods might also be required to handle issues like (say) class imbalance [AM18].

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; Wan13b].
- 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].

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 [Wan13b]. Much of that CPU time can be saved if there is a faster way.

Given recent advances in cloud computing, it is possible to find the best method for a particular data set via a “shoot out” between different methods. For example, Lessmann et al. [Les08] and Ghotra et al. [Gho15] explored 22 and 32 different learning algorithms (respectively) for software quality defect prediction. Such studies may require days to weeks of CPU time to complete [FM17b]. But are such complex and time-consuming studies necessary?

- If there exists some way to dramatically simplify software quality predictors; then those cloud-based resources would be better used for other tasks.
- Also, Lessmann and Ghotra et al. [Les08; Gho15] report that many defect prediction methods have equivalent performance.
- Further, as we show here; there are very simple methods that perform even better than the methods studied by Lessmann and Ghotra et al.
Another reason to study simplification is that studies can reveal the underlying nature of seemingly complex problems. In terms of core science, we argue that the better we understand something, the better we can match tools to SE. Tools which are poorly matched to task are usually complex and/or slow to execute.

Seeking simpler and/or faster solutions is not just theoretically interesting. It is also an approach currently in vogue in contemporary software engineering. Calero and Pattini [CP15] comments that “redesign for greater simplicity” also motivates much contemporary industrial work. In their survey of modern SE companies, they find that many current organizational redesigns are motivated (at least in part) by arguments based on “sustainability” (i.e., using fewer resources to achieve results). According to Calero and Pattini, sustainability is now a new source of innovation. Managers used sustainability-based redesigns to explore cost-cutting opportunities. In fact, they say, sustainability is now viewed by many companies as a mechanism for gaining a complete advantage over their competitors. Hence, a manager might ask a programmer to assess simple methods to generate more interesting products.

For all these reasons, we assert that it is high time to explore how to simplify software analytics methods.
Part II

Software Analytics Can Be Simpler
This chapter firstly appeared at *Information and Software Technology 76* (2016): 135-146 titled “Tuning for software analytics: is it really necessary?”. In this chapter, by using this study, we want to show that software analytics can be easier.

Data miners have been widely used in different software analytics tasks, like building defect predictors from static code measures. Such static code defect predictors perform well compared to manual methods, and they are easy to use and useful to use. There are several ways to improve the performance of such defect prediction models. In this chapter, we show how to achieve this goal by simply tuning hyper-parameters of models, which is to support our claim that software analytics can be simpler. Specifically, we seek a simple, automatic, and very effective method (DE) for finding those tunings. Contrary to our prior expectations, we found these tunings were remarkably simple: it only required tens, not thousands, of attempts to obtain very good results. For example, when learning software defect predictors, this method can quickly find tunings that alter detection precision from 0% to 60%. Since (1) the improvements are so large, and (2) the tuning is so simple, we need to change standard methods in software analytics. At least for defect prediction, it is no longer enough to just run a data miner and present the result *without* conducting a tuning optimization study. The implication for other kinds of analytics is now an open and pressing issue.
3.1 Introduction

One of the “black arts” of data mining is setting the tuning parameters that control the choices within a data miner. Prior to this work, our intuition was that tuning would change the behavior or a data miner, to some degree. Nevertheless, we rarely tuned our defect predictors since we reasoned that a data miner’s default tunings have been well-explored by the developers of those algorithms (in which case tuning would not lead to large performance improvements). Also, we suspected that tuning would take so long time and be so CPU intensive that the benefits gained would not be worth effort.

The results of this study show that the above points are false since, at least for defect prediction from code attributes: (1) Tuning defect predictors is remarkably simple; (2) And can dramatically improve the performance. Those results were found by exploring six research questions:

- **RQ1:** Does tuning improve the performance scores of a predictor? We will show below examples of truly dramatic improvement: usually by 5 to 20% and often by much more (in one extreme case, precision improved from 0% to 60%).

- **RQ2:** Does tuning change conclusions on what learners are better than others? Recent SE papers [Les08; Hal12] claim that some learners are better than others. Some of those conclusions are completely changed by tuning.

- **RQ3:** Does tuning change conclusions about what factors are most important in software engineering? Numerous recent SE papers (e.g. [Bel13; RD13; MS03; Mos08; Zim07; Her13]) use data miners to conclude that this is more important than that for reducing software project defects. Given the tuning results of this study, we show that such conclusions need to be revisited.

- **RQ4:** Is tuning easy? We show that one of the simpler multi-objective optimizers (differential evolution [SP97]) works very well for tuning defect predictors.

- **RQ5:** Is tuning impractically slow? We achieved dramatic improvements in the performance scores of our data miners in less than 100 evaluations (!); i.e., very quickly.

- **RQ6:** Should data miners be used “off-the-shelf” with their default tunings? For defect prediction from static code measures, our answer is an emphatic “no” (and the implication for other kinds of analytics is now an open and urgent question).

Based on our answers to these questions, we strongly advise that: (1) Data miners should not be used “off-the-shelf” with default tunings; (2) Any future paper on defect prediction should include a tuning study. Here, we have found an algorithm called differential evolution to be a useful method
for conducting such tunings; (3) Tuning needs to be repeated whenever data or goals are changed. Fortunately, the cost of finding good tunings is not excessive since, at least for static code defect predictors, tuning is easy and fast.

3.2 Preliminaries

3.2.1 Tuning: Important and Ignored

This section argues that tuning is an under-explored software analytics—particularly in the apparently well-explored field of defect prediction.

In other fields, the impact of tuning is well understood [BB12]. Yet issues of tuning are rarely or poorly addressed in the defect prediction literature. 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.

Are the impacts of tuning addressed in the defect prediction literature? To answer that question, in Jan 2016 we searched scholar.google.com for the conjunction of “data mining” and “software engineering” and “defect prediction” (more details can be found at https://goo.gl/Inl9nF). After sorting by the citation count and discarding the non-SE papers (and those without a pdf link), we read over this sample of 50 highly-cited SE defect prediction papers. What we found in that sample was that few authors acknowledged the impact of tunings (exceptions: [Gao11; Les08]). Overall, 80% of papers in our sample did not adjust the “off-the-shelf” configuration of the data miner (e.g. [Men07b; Mos08; EE08]). Of the remaining papers:

- Some papers in our sample explored data super-sampling [PD07] or data sub-sampling techniques via automatic methods (e.g. [Gao11; Men07b; PD07; Kim11]) or via some domain principles (e.g. [Mos08; Nag08; Has09]). As an example of the latter, Nagappan et al. [Nag08] checked if metrics related to organizational structure were relatively more powerful for predicting software defects. However, it should be noted that these studies varied the input data but not the “off-the-shelf” settings of the data miner.

- A few other papers did acknowledge that one data miner may not be appropriate for all data sets. Those papers tested different “off-the-shelf” data miners on the same data set. For example, Elish et al.[EE08] compared support vector machines to other data miners for the purposes of defect prediction. SVM’s execute via a “kernel function” which should be specially selected for different data sets and the Elish et al. paper makes no mention of any SVM tuning study. To be fair to Elish et al., we hasten to add that we ourselves have published papers using “off-the-shelf” tunings [Men07b] since, prior to this study it was unclear to us how to effectively
navigate the large space of possible tunings.

Over our entire sample, there was only one paper that conducted a somewhat extensive tuning study. Lessmann et al.[Les08] tuned parameters for some of their algorithms using a grid search; i.e. divide all $C$ configuration options into $N$ values, then try all $N^C$ combinations. This is a slow approach—we have explored grid search for defect prediction and found it takes days to terminate [Men07b]. Not only that, we found that grid search can miss important optimizations [Bak07]. Every grid has “gaps” between each grid division which means that a supposedly rigorous grid search can still miss important configurations [BB12]. Bergstra and Bengio [BB12] comment that for most data sets only a few of the tuning parameters really matter– which means that much of the runtime associated with grid search is actually wasted. Worse still, Bergstra and Bengio comment that the important tunings are different for different data sets– a phenomenon that makes grid search a poor choice for configuring data mining algorithms for new data sets.

Since the Lessmann et al. paper, much progress has been made in configuration algorithms and we can now report that finding useful tunings is very easy. This result is both novel and unexpected. A standard run of grid search (and other evolutionary algorithms) is that optimization requires thousands, if not millions, of evaluations. However, in a result that we found startling, that differential evolution (described below) can find useful settings for learners generating defect predictors in less than 100 evaluations (i.e. very quickly). Hence, the “problem” (that tuning changes the conclusions) is really an exciting opportunity. At least for defect prediction, learners are very amenable to tuning. Hence, they are also very amenable to significant performance improvements. Given the low number of evaluations required, then we assert that tuning should be standard practice for anyone building defect predictors.

### 3.2.2 You Can't Always Get What You Want

Having made the case that tuning needs to be explored more, but before we get into the technical details of this paper, this section discusses some general matters about setting goals during tuning experiments.

This study characterizes tuning as an optimization problem (how to change the settings on the learner in order to best improve the output). With such optimizations, it is not always possible to optimize for all goals at the same time. For example, the following text does not show results for tuning on recall or false alarms since optimizing only for those goals can lead to some undesirable side effects:

- **Recall** reports the percentage of predictions that are actual examples of what we are looking for. When we tune for **recall**, we can achieve near 100% recall– but at the cost of a near 100% false alarms.
• *False alarms* is the percentage of other examples that are reported (by the learner) to be part of the targeted examples. When we tune for *false alarms*, we can achieve near zero percent false alarm rates by effectively turning off the detector (so the recall falls to nearly zero).

Accordingly, this study explores performance measures that comment on all target classes: see the precision and “F” measures discussed below: see *Optimization Goals*. That said, we are sometimes asked what good is a learner if it optimizes for (say) precision at the expense of (say) recall.

Our reply is that software engineering is a very diverse enterprise and that different kinds of development need to optimize for different goals (which may not necessarily be “optimize for recall”):

• Anda, Sjoberg and Mockus are concerned with *reproducibility* and so assess their models using the the “coefficient of variation” \( CV = \frac{\text{stddev}}{\text{mean}} \) [And09].

• Arisholm & Briand [AB06], Ostrand & Weyeuker [Ost04] and Rahman et al. [Rah12] are concerned with reducing the work load associated with someone else reading a learned model, then applying it. Hence, they assess their models using *reward*; i.e. the fewest lines of code containing the most bugs.

• Yin et al. are concerned about *incorrect bug fixes*; i.e. those that require subsequent work in order to complete the bug fix. These bugs occur when (say) developers try to fix parts of the code where they have very little experience [Yin11]. Hence, they assess a learned model using a measure that selects for the most number of bugs in regions that the most programmers have worked with before.

• For safety critical applications, high false alarm rates are acceptable if the cost of overlooking critical issues outweighs the inconvenience of inspecting a few more modules.

• When rushing a product to market, there is a business case to avoid the extra rework associated with false alarms. In that business context, managers might be willing to lower the recall somewhat in order to minimize the false alarms.

• When Dr. Menzies worked with contractors at NASA’s software independent verification and validation facility, he found new contractors only reported issues that were most certainly important defects; i.e. they minimized false alarms even if that damaged their precision (since, they felt, it was better to be silent than wrong). Later on, once those contractors had acquired a reputation of being insightful members of the team, they improved their precision scores (even if it means some more false alarms).

Accordingly, this study does not assume that (e.g.) minimizing false alarms is more important than maximizing precision or recall. Such a determination depends on business conditions.
Rather, what we can show examples where changing optimization goals can also change the conclusions made from that learner on that data. More generally, we caution that it is important not to overstate empirical results from analytics. Those results need to be expressed *along with* the context within which they are relevant (and by “context”, we mean the optimization goal).

### 3.2.3 Notes on Data Miners

There are several ways to make defect predictors using CART [Bre01], Random Forest [Bre84], WHERE [Men13] and LR (logistic regression). For this study, we use CART, Random Forest and LR versions from SciKitLearn [Ped11] and WHERE, which is available from github.com/ai-se/where. We use these algorithms for the following reasons.

CART and Random Forest were mentioned in a recent IEEE TSE paper by Lessmann et al. [Les08] that compared 22 learners for defect prediction. That study ranked CART worst and Random Forest as best. In a demonstration of the impact of tuning, this study shows we can refute the conclusions of Lessmann et al. in the sense that, after tuning, CART performs just as well as Random Forest.

LR was mentioned by Hall et al. [Hal12] as usually being as good or better as more complex learners (e.g. Random Forest). In a finding that endorses the Hall et al. result, we show that untuned LR performs better than untuned Random Forest (at least, for the data sets studied here). However, we will show that tuning raises doubts about the optimality of the Hall et al. recommendation.

Finally, this paper uses WHERE since, as shown below, it offers an interesting case study on the benefits of tuning.

### 3.2.4 Learners and Their Tunings

Our learners use the tuning parameters of Table 3.1 The default parameters for CART and Random Forest are set by the Scikit-learn authors and the default parameters for WHERE-based learner are set via our own expert judgment. When we say a learner is used “off-the-shelf”, we mean that they use the defaults shown in Table 3.1.

As to the value of those defaults, it could be argued that these defaults are not the best parameters for practical defect prediction. That said, prior to this study, two things were true:

- Many data scientists in SE use the standard defaults in their data miners, without tuning (e.g. [Men07b; Mos08; Her13; Zim07]).
- The effort involved to adjust those tunings seemed so onerous, that many researchers in this field were content to take our prior advice of “do not tune... it is just too hard” [Men14b].

As to why we used the ”Tuning Range” shown in Table 3.1, and not some other ranges, we note that (1) those ranges included the defaults; (2) the results shown below show that by exploring those ranges, we achieved large gains in the performance of our defect predictors. This is not to say that
Table 3.1 List of parameters tuned by this study.

<table>
<thead>
<tr>
<th>Learner Name</th>
<th>Parameters</th>
<th>Default</th>
<th>Tuning Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Where-based Learner</td>
<td>threshold</td>
<td>0.5</td>
<td>[0.01,1]</td>
<td>The value to determine defective or not.</td>
</tr>
<tr>
<td></td>
<td>infoPrune</td>
<td>0.33</td>
<td>[0.01,1]</td>
<td>The percentage of features to consider for the best split to build its final decision tree.</td>
</tr>
<tr>
<td></td>
<td>min_sample_split</td>
<td>4</td>
<td>[1,10]</td>
<td>The minimum number of samples required to split an internal node of its final decision tree.</td>
</tr>
<tr>
<td></td>
<td>min_Size</td>
<td>0.5</td>
<td>[0.01,1]</td>
<td>Finds min_samples_leaf in the initial clustering tree using n_samples/min_Size.</td>
</tr>
<tr>
<td></td>
<td>wriggle</td>
<td>0.2</td>
<td>[0.01,1]</td>
<td>The threshold to determine which branch in the initial clustering tree to be pruned.</td>
</tr>
<tr>
<td></td>
<td>depthMin</td>
<td>2</td>
<td>[1,6]</td>
<td>The minimum depth of the initial clustering tree below which no pruning for the clustering tree.</td>
</tr>
<tr>
<td></td>
<td>depthMax</td>
<td>10</td>
<td>[1,20]</td>
<td>The maximum depth of the initial clustering tree.</td>
</tr>
<tr>
<td></td>
<td>wherePrune</td>
<td>False</td>
<td>T/F</td>
<td>Whether or not to prune the initial clustering tree.</td>
</tr>
<tr>
<td></td>
<td>treePrune</td>
<td>True</td>
<td>T/F</td>
<td>Whether or not to prune the final decision tree.</td>
</tr>
<tr>
<td>CART</td>
<td>threshold</td>
<td>0.5</td>
<td>[0.1]</td>
<td>The value to determine defective or not.</td>
</tr>
<tr>
<td></td>
<td>max_feature</td>
<td>None</td>
<td>[0.01,1]</td>
<td>The number of features to consider when looking for the best split.</td>
</tr>
<tr>
<td></td>
<td>min_sample_split</td>
<td>2</td>
<td>[2,20]</td>
<td>The minimum number of samples required to split an internal node.</td>
</tr>
<tr>
<td></td>
<td>min_samples_leaf</td>
<td>1</td>
<td>[1,20]</td>
<td>The minimum number of samples required to be at a leaf node.</td>
</tr>
<tr>
<td></td>
<td>max_depth</td>
<td>None</td>
<td>[1,50]</td>
<td>The maximum depth of the tree.</td>
</tr>
<tr>
<td>Random Forests</td>
<td>threshold</td>
<td>0.5</td>
<td>[0.01,1]</td>
<td>The value to determine defective or not.</td>
</tr>
<tr>
<td></td>
<td>max_feature</td>
<td>None</td>
<td>[0.01,1]</td>
<td>The number of features to consider when looking for the best split.</td>
</tr>
<tr>
<td></td>
<td>max_leaf_nodes</td>
<td>None</td>
<td>[1,50]</td>
<td>Grow trees with max_leaf_nodes in best-first fashion.</td>
</tr>
<tr>
<td></td>
<td>min_sample_split</td>
<td>2</td>
<td>[2,20]</td>
<td>The minimum number of samples required to split an internal node.</td>
</tr>
<tr>
<td></td>
<td>min_samples_leaf</td>
<td>1</td>
<td>[1,20]</td>
<td>The minimum number of samples required to be at a leaf node.</td>
</tr>
<tr>
<td></td>
<td>n_estimators</td>
<td>100</td>
<td>[50,150]</td>
<td>The number of trees in the forest.</td>
</tr>
<tr>
<td>Logistic Regression</td>
<td>This study uses untuned LR in order to check a conclusion of [Hal12].</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

_larger_ tuning ranges might not result in _greater_ improvements. However, for the goals of this study (to show that some tunings do matter), exploring just these ranges shown in Table 3.1 will suffice.

As to the details of these learners, LR is a parametric modeling approach. Given \( f = \beta_0 + \sum \beta_j x_j \), where \( x_j \) is some measurement in a data set, and \( \beta_j \) is learned via regression, LR converts that into a function \( 0 \leq g \leq 1 \) using \( g = 1/(1 + e^{-f}) \). This function reports how much we believe in a particular class.

CART, Random Forest, and WHERE-based learners are all tree learners that divide a data set, then recur on each split. All these learners generate numeric predictions which are converted into binary “yes/no” decisions via Equation 3.1.
\[
\text{inspect} = \begin{cases} 
  d_i \geq T \rightarrow \text{Yes} \\
  d_i < T \rightarrow \text{No},
\end{cases}
\]

where \(d_i\) is the number of observed issues and \(T\) is some threshold defined by an engineering judgment; we use \(T = 1\).

The splitting process is controlled by numerous tuning parameters. If data contains more than \texttt{min\_sample\_split}, then a split is attempted. On the other hand, if a split contains no more than \texttt{min\_samples\_leaf}, then the recursion stops. CART and Random Forest use a user-supplied constant for this parameter while WHERE-based learner firstly computes this parameter \(m = \texttt{min\_samples\_leaf}\) from the size of the data sets via \(m = \texttt{size}^{\text{min\_size}}\) to build an initial clustering tree. Note that WHERE builds two trees: the initial clustering tree (to find similar sets of data) then a final decision tree (to learn rules that predict for each similar cluster). A frequently asked question is why does WHERE build two trees—would not a single tree suffice? The answer is, as shown below, tuned WHERE’s twin-tree approach generates very precise predictors. As to the rest of WHERE’s parameters, the parameter \texttt{min\_sample\_split} controls the construction of the final decision tree (so, for WHERE-based learner, \texttt{min\_size} and \texttt{min\_sample\_split} are the parameters to be tuned).

These learners use different techniques to explore the splits: CART finds the attributes whose ranges contain rows with least variance in the number of defects. If an attribute ranges \(r_i\) is found in \(n_i\) rows each with a defect count variance of \(v_i\), then CART seeks the attributes whose ranges minimizes \(\sum_i \left( \sqrt{v_i} \times n_i / (\sum_i n_i) \right)\); Random Forest divides data like CART then builds \(F > 1\) trees, each time using some random subset of the attributes; When building the initial cluster tree, WHERE projects the data on to a dimension it synthesizes from the raw data using a process analogous to principle component analysis \[Jol02\]. WHERE divides at the median point of that projection. On recursion, this generates the initial clustering tree, the leaves of which are clusters of very similar examples. After that, when building the final decision tree, WHERE pretends its clusters are “classes”, then asks the InfoGain algorithm of the Fayyad-Irani discretizer \[FI93\], to rank the attributes, where \texttt{infoPrune} is used. WHERE’s final decision tree generator then ignores everything except the top \texttt{infoPrune} percent of the sorted attributes.

Some tuning parameters are learner specific: \texttt{Max\_feature} is used by CART and Random Forest to select the number of attributes used to build one tree. CART’s default is to use all the attributes while Random Forest usually selects the square root of the number of attributes. \texttt{Max\_leaf\_nodes} is the upper bound on leaf notes generated in a Random Forest. \texttt{Max\_depth} is the upper bound on the depth of the CART tree. WHERE’s tree generation will always split up to \texttt{depthMin} number of branches. After that, WHERE will only split data if the mean performance scores of the two halves is “trivially small” (where “trivially small” is set by the \texttt{wriggle} parameter). WHERE’s \texttt{tree\_prune} setting controls how WHERE prunes back superfluous parts of the final decision tree. If a decision sub-tree
and its parent have the same majority cluster (one that occurs most frequently), then if $tree\_prune$ is enabled, we prune that decision sub-tree.

### 3.2.5 Tuning Algorithms

How should researchers select which optimizers to apply to tuning data miners? Cohen [Coh95] advises comparing new methods against the simplest possible alternative. Similarly, Holte [Hol93] recommends using very simple learners as a kind of "scout" for a preliminary analysis of a data set (to check if that data really requires a more complex analysis). Accordingly, to find our “scout”, we used engineering judgement to sort candidate algorithms from simplest to complex. For example, here is a list of optimizers used widely in research: simulated annealing [FM02; Men07a]; various genetic algorithms [Gol79] augmented by techniques such as differential evolution [SP97], tabu search and scatter search [GM86; Bea06a; 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 [Kra15b; Zul13]. Of these, the simplest are simulated annealing (SA) and differential evolution (DE), each of which can be coded in less than a page of some high-level scripting language. Our reading of the current literature is that there are more advocates for differential evolution than SA. For example, Vesterstrom and Thomsen [VT04] found DE to be competitive with particle swarm optimization and other GAs.

DEs have been applied before for parameter tuning (e.g. see [Omr05; Chi12]) but this is the first time they have been applied to optimize defect prediction from static code attributes. The pseudocode for differential evolution is shown in Algorithm 1. In the following description, superscript numbers denote lines in that pseudocode.

DE evolves a NewGeneration of candidates from a current Population. Our DE’s lose one “life” when the new population is no better than current one (terminating when “life” is zero)$^{14}$. Each candidate solution in the Population is a pair of (Tunings, Scores). Tunings are selected from Table 3.1 and Scores come from training a learner using those parameters and applying it test data$^{23-27}$.

The premise of DE is that the best way to mutate the existing tunings is to Extrapolate$^{28}$ between current solutions. Three solutions $a$, $b$, $c$ are selected at random. For each tuning parameter $i$, at some probability $cr$, we replace the old tuning $x_i$ with $y_i$. For booleans, we use $y_i = \neg x_i$ (see line 36). For numerics, $y_i = a_i + f \times (b_i - c_i)$ where $f$ is a parameter controlling cross-over. The trim function$^{38}$ limits the new value to the legal range min..max of that parameter.

The main loop of DE$^{6}$ runs over the Population, replacing old items with new Candidates (if new candidate is better). This means that, as the loop progresses, the Population is full of increasingly more valuable solutions. This, in turn, also improves the candidates, which are Extrapolated from the Population.

For the experiments of this study, we collect performance values from a data mining model,
Algorithm 1 Pseudocode for DE with Early Termination

Input: \( np = 10, f = 0.75, cr = 0.3, \) life = 5, \( \) Goal \( \in \{ pd, f, \ldots \} \)

Output: \( S_{\text{best}} \)

1: function DE\((np, f, cr, \text{life}, \text{Goal})\)
2: \[ Population \leftarrow \text{InitializePopulation}(np) \]
3: \[ S_{\text{best}} \leftarrow \text{GetBestSolution}(Population) \]
4: while \( \text{life} > 0 \) do
5: \[ \text{NewGeneration} \leftarrow \emptyset \]
6: for \( i = 0 \rightarrow np - 1 \) do
7: \[ S_i \leftarrow \text{Extrapolate}(Population[i], Population, cr, f) \]
8: if \( \text{Score}(S_i) > \text{Score}(Population[i]) \) then
9: \[ \text{NewGeneration}.\text{append}(S_i) \]
10: else
11: \[ \text{NewGeneration}.\text{append}(Population[i]) \]
12: end if
13: end for
14: Population \leftarrow \text{NewGeneration}
15: if \( \neg \text{Improve}(Population) \) then
16: \( \text{life} \leftarrow 1 \)
17: end if
18: \[ S_{\text{best}} \leftarrow \text{GetBestSolution}(Population) \]
19: end while
20: return \( S_{\text{best}} \)
21: end function

function \( \text{SCORE}(\text{Candidate}) \)
22: # set tuned parameters according to \( \text{Candidate} \)
23: \( \text{model} \leftarrow \text{TrainLearner()} \)
24: \( \text{result} \leftarrow \text{TestLearner(model)} \)
25: return \( \text{Goal}(\text{result}) \)
26: end function

function \( \text{EXTRAPOLATE}(\text{old}, \text{pop}, \text{cr}, f) \)
27: \( a, b, c \leftarrow \text{threeOthers}(\text{pop, old}) \)
28: \( \text{newf} \leftarrow \emptyset \)
29: for \( i = 0 \rightarrow np - 1 \) do
30: if \( \text{cr} < \text{random()} \) then
31: \( \text{newf}.\text{append}(\text{old}[i]) \)
32: else
33: if \( \text{typeof}(\text{old}[i]) = \text{bool} \) then
34: \( \text{newf}.\text{append}(\neg \text{old}[i]) \)
35: else
36: \( \text{newf}.\text{append}(\text{trim}(i, a[i] + f \ast (b[i] - c[i]))) \)
37: end if
38: end if
39: end if
40: end for
41: return \( \text{newf} \)
42: end function

from which a \( \text{Goal} \) function extracts one performance value\(^ {26} \) (so we run this code many times, each time with a different \( \text{Goal}^l \)). Technically, this makes a single objective DE (and for notes on multi-objective DEs, see [RF05; ZL07; HL10]).
Table 3.2 Data used in this experiment. E.g., the top left data set has 20 defective classes out of 125 total. See §3.3.1 for explanation of training, tuning, testing sets.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>antV0</th>
<th>antV1</th>
<th>antV2</th>
<th>camelV0</th>
<th>camelV1</th>
<th>ivy</th>
<th>jeditV0</th>
<th>jeditV1</th>
<th>jeditV2</th>
</tr>
</thead>
<tbody>
<tr>
<td>training</td>
<td>20/125</td>
<td>40/178</td>
<td>32/293</td>
<td>13/339</td>
<td>216/608</td>
<td>63/111</td>
<td>90/272</td>
<td>75/306</td>
<td>79/312</td>
</tr>
<tr>
<td>tuning</td>
<td>40/178</td>
<td>32/293</td>
<td>92/351</td>
<td>216/608</td>
<td>145/872</td>
<td>16/241</td>
<td>75/306</td>
<td>79/312</td>
<td>48/367</td>
</tr>
<tr>
<td>testing</td>
<td>32/293</td>
<td>92/351</td>
<td>166/745</td>
<td>145/872</td>
<td>188/965</td>
<td>40/352</td>
<td>79/312</td>
<td>48/367</td>
<td>11/492</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dataset</th>
<th>log4j</th>
<th>lucene</th>
<th>poiV0</th>
<th>poiV1</th>
<th>synapse</th>
<th>velocity</th>
<th>xercesV0</th>
<th>xercesV1</th>
</tr>
</thead>
<tbody>
<tr>
<td>training</td>
<td>34/135</td>
<td>91/195</td>
<td>141/237</td>
<td>37/314</td>
<td>16/157</td>
<td>147/196</td>
<td>77/162</td>
<td>71/440</td>
</tr>
<tr>
<td>tuning</td>
<td>37/109</td>
<td>144/247</td>
<td>37/314</td>
<td>248/385</td>
<td>60/222</td>
<td>142/214</td>
<td>71/440</td>
<td>69/453</td>
</tr>
<tr>
<td>testing</td>
<td>189/205</td>
<td>203/340</td>
<td>248/385</td>
<td>281/442</td>
<td>86/256</td>
<td>78/229</td>
<td>69/453</td>
<td>437/588</td>
</tr>
</tbody>
</table>

§3.3 Experimental Design

3.3.1 Data Sets

Our defect data comes from the PROMISE repository (http://openscience.us/repo/defect) and pertains to open source Java systems defined in terms of Table 1.1: ant, camel, ivy, jedit, log4j, lucene, poi, synapse, velocity and xerces.

An important principle in data mining is not to test on the data used in training. There are many ways to design an 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).

To avoid these problems, we used an incremental learning approach. The following experiment ensures that the training data was created at some time before the test data. For this experiment, we use data sets with at least three consecutive releases (where release \( i + 1 \) was built after release \( i \)).

When tuning a learner, the first release was used on line 24 of Algorithm 1 to build some model using some the tunings found in some Candidate; The second release was used on line 25 of Algorithm 1 to test the candidate model found on line 24; Finally the third release was used to gather the performance statistics reported below from the best model found by DE.

To be fair for the untuned learner, the first and second releases used in tuning experiments will be combined as the training data to build a model. Then the performance of this untuned learner will be evaluated by the same third release as in the tuning experiment.

Some data sets have more than three releases and, for those data, we could run more than one experiment. For example, ant has five versions in PROMISE so we ran three experiments called V0,V1,V2:

- AntV0: first, second, third = versions 1, 2, 3
- AntV1: first, second, third = versions 2, 3, 4
- AntV2: first, second, third = versions 3, 4, 5
These data sets are displayed in Table 3.2.

3.3.2 Optimization Goals

Recall from Algorithm 1 that we call differential evolution once for each optimization goal. This section lists those optimization goals. Let \{A, B, C, D\} denote the true negatives, false negatives, false positives, and true positives (respectively) found by a binary detector. Certain standard measures can be computed from A, B, C, D, as shown below. Note that for \( pf \), the better scores are smaller while for all other scores, the better scores are larger.

\[
\begin{align*}
pd &= \text{recall} = \frac{D}{B + D} \\
pf &= \frac{C}{A + C} \\
prec &= \text{precision} = \frac{D}{D + C} \\
F &= \frac{2 \times pd \times prec}{pd + prec}
\end{align*}
\]

The rest of this study explores tuning for \( prec \) and \( F \). As discussed in §3.2.2, our point is not that these are best or most important optimization goals. Indeed, the list of “most important” goals is domain-specific (see §3.2.2) and we only explore these two to illustrate how conclusions can change dramatically when moving from one goal to another.

3.4 Experimental Results

In the following, we explore the effects of tuning WHERE, Random Forest, and CART. LR will be used, untuned, in order to check one of the recommendations made by Hall et al. [Hal12].

3.4.1 RQ1: Does Tuning Improve Performance?

Figure 3.1 says that the answer to RQ1 is “yes”– tuning has a positive effect on performance scores. This figure sorts deltas in the precision and the F-measure between tuned and untuned learners. Our reading of this figure is that, overall, tuning rarely makes performance worse and often can make it much better.

Table 3.3 and Table 3.4 show the the specific values seen before and after tuning with \( \text{precision} \) and \( “F” \) as different optimization goals (the corresponding “F” and precision values for Table 3.3 and Table 3.4 are not provided for the space limitation). For each data set, the maximum precision or “F” values for each data set are shown in bold. As might have been predicted by Lessmann et al. [Les08], untuned CART is indeed the worst learner (only one of its untuned results is best and bold). And, in \( \frac{12}{17} \) cases, the untuned Random Forest performs better than or equal to untuned CART in terms of precision.
That said, tuning can improve those poor performing detectors. In some cases, the median changes may be small (e.g. the “F” results for WHERE and Random Forests) but even in those cases, there are enough large changes to motivate the use of tuning. For example:

- For “F” improvement, there are two improvements over 25% for both WHERE and Random Forests. Also, in poiV0, all untuned learners report “F” of under 50%, tuning changes those scores by 25%. Finally, note the xercesV1 result for the WHERE learner. Here, tuning changes precision from 32% to 70%.

- Regarding precision, for antV0, and antV1 untuned WHERE reports precision of 0. But tuned WHERE scores 35 and 60 (the similar pattern can seen in “F”).

3.4.2 RQ2: Does Tuning Change a Learner’s Ranking?

Researchers often use performance criteria to assert that one learner is better than another [Les08; Men07b; Hal12]. For example:

1. Lessmann et al. [Les08] conclude that Random Forest is considered to be statistically better than CART.

2. Also, in Hall et al.’s systematic literature review[Hal12], it is argued that defect predictors based on simple modeling techniques such as LR perform better than “complicated” techniques.
Table 3.3 Precision results (best results shown in bold).

<table>
<thead>
<tr>
<th>Data set</th>
<th>WHERE</th>
<th>CART</th>
<th>Random Forest</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>default</td>
<td>Tuned</td>
<td>default</td>
</tr>
<tr>
<td>antV0</td>
<td>0</td>
<td>35</td>
<td>15</td>
</tr>
<tr>
<td>antV1</td>
<td>0</td>
<td>60</td>
<td>54</td>
</tr>
<tr>
<td>antV2</td>
<td>45</td>
<td>55</td>
<td>42</td>
</tr>
<tr>
<td>camelV0</td>
<td>20</td>
<td>30</td>
<td>30</td>
</tr>
<tr>
<td>camelV1</td>
<td>27</td>
<td>28</td>
<td><strong>38</strong></td>
</tr>
<tr>
<td>ivy</td>
<td>25</td>
<td>21</td>
<td>21</td>
</tr>
<tr>
<td>jeditV0</td>
<td>34</td>
<td>37</td>
<td>56</td>
</tr>
<tr>
<td>jeditV1</td>
<td>30</td>
<td>42</td>
<td>32</td>
</tr>
<tr>
<td>jeditV2</td>
<td>4</td>
<td>22</td>
<td>6</td>
</tr>
<tr>
<td>log4j</td>
<td>96</td>
<td>91</td>
<td>95</td>
</tr>
<tr>
<td>lucene</td>
<td>61</td>
<td>75</td>
<td>67</td>
</tr>
<tr>
<td>poiV0</td>
<td>70</td>
<td>70</td>
<td>65</td>
</tr>
<tr>
<td>poiV1</td>
<td>74</td>
<td>76</td>
<td>72</td>
</tr>
<tr>
<td>synapse</td>
<td>61</td>
<td>50</td>
<td>50</td>
</tr>
<tr>
<td>velocity</td>
<td>34</td>
<td><strong>44</strong></td>
<td>39</td>
</tr>
<tr>
<td>xercesV0</td>
<td>14</td>
<td>17</td>
<td>17</td>
</tr>
<tr>
<td>xercesV1</td>
<td>86</td>
<td>54</td>
<td>72</td>
</tr>
</tbody>
</table>

Table 3.4 F-measure results (best results shown in bold).

<table>
<thead>
<tr>
<th>Data set</th>
<th>WHERE</th>
<th>CART</th>
<th>Random Forest</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>default</td>
<td>Tuned</td>
<td>default</td>
</tr>
<tr>
<td>antV0</td>
<td>0</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>antV1</td>
<td>0</td>
<td>38</td>
<td>37</td>
</tr>
<tr>
<td>antV2</td>
<td>47</td>
<td>50</td>
<td>45</td>
</tr>
<tr>
<td>camelV0</td>
<td>31</td>
<td>28</td>
<td>39</td>
</tr>
<tr>
<td>camelV1</td>
<td>34</td>
<td>34</td>
<td>38</td>
</tr>
<tr>
<td>ivy</td>
<td>39</td>
<td>34</td>
<td>28</td>
</tr>
<tr>
<td>jeditV0</td>
<td>45</td>
<td>47</td>
<td>56</td>
</tr>
<tr>
<td>jeditV1</td>
<td>43</td>
<td>44</td>
<td>44</td>
</tr>
<tr>
<td>jeditV2</td>
<td>8</td>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td>log4j</td>
<td>47</td>
<td>50</td>
<td>53</td>
</tr>
<tr>
<td>lucene</td>
<td>73</td>
<td>73</td>
<td>65</td>
</tr>
<tr>
<td>poiV0</td>
<td>50</td>
<td>74</td>
<td>31</td>
</tr>
<tr>
<td>poiV1</td>
<td>75</td>
<td><strong>78</strong></td>
<td>68</td>
</tr>
<tr>
<td>synapse</td>
<td>49</td>
<td>56</td>
<td>43</td>
</tr>
<tr>
<td>velocity</td>
<td>51</td>
<td>53</td>
<td>53</td>
</tr>
<tr>
<td>xercesV0</td>
<td>19</td>
<td>22</td>
<td>19</td>
</tr>
<tr>
<td>xercesV1</td>
<td>32</td>
<td>70</td>
<td>34</td>
</tr>
</tbody>
</table>

such as Random Forest. To explain that comment, we note that by three measures, Random Forest is more complicated than LR:

(a) CART builds one model while Random Forest builds many models.

(b) LR is just a model construction tool while Random Forest needs both a tool to construct its forest and a second tool to infer some conclusion from all the members of that forest.

(c) the LR model can be printed in a few lines while the multiple models learned by Random Forest model would take up multiple pages of output.

Given tuning, how stable are these conclusions? Before answering this issue, we digress for two comments.
Figure 3.2 Comparison between Logistic Regression and Random Forest before and after tuning.

Firstly, it is important to comment on why it is so important to check the conclusions of these particular papers. These papers are prominent publications (to say the least). Hall et al. [Hal12] is the fourth most-cited IEEE TSE paper for 2009 to 2014 with 176 citations (see goo.gl/MGrGr7) while the Lessmann et al. paper [Les08] has 394 citations (see goo.gl/khTpf7) – which is quite remarkable for a paper published in 2009. Given the prominence of these papers, researchers might believe it is appropriate to use their advice without testing that advice on local data sets.

Secondly, while we are critical of the results of Lessmann et al. and Hall et al., it needs to be said that their analysis was excellent and exemplary given the state-of-the-art of the tools used when those papers were written. While Hall et al. did not perform any new experiments, their summarization of so many defect prediction papers has not been equalled before (or since). As to the Lessmann et al. paper, they compared 22 data miners using various data sets (mostly from NASA) [Les08]. In that study, some learners were tuned using manual methods (C4.5, CART and Random Forest) and some, like SVM-Type learners, were tuned by automatic grid search (for more on grid search, see §2.1).

That said, our tuning results show that it is time to revise the recommendations of those papers. Figure 3.2 comments on the advice from Hall et al. (that LR is better than Random Forest)

- In a result that might have been predicted by Hall et al., untuned Random Forests performs comparatively worse than Logistic Regression. Specifically, untuned Random Forest performs worse than Logistic regression in 13 out of 17 data sets.
Table 3.5 Features selected by tuned WHERE with different goals: bold features are those found useful by the tuned WHERE. Also, features shown in plain text are those found useful by the untuned WHERE.

<table>
<thead>
<tr>
<th>Data set</th>
<th>Precision</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>antV0</td>
<td>rfc</td>
<td>mfa, loc, cam, dit, dam, lcom3</td>
</tr>
<tr>
<td></td>
<td>mfa, loc, cam, dit, dam, lcom3</td>
<td>None</td>
</tr>
<tr>
<td></td>
<td>mfa, loc, cam, dit, dam, lcom3</td>
<td>mfa, loc, cam, dit, dam, lcom3</td>
</tr>
<tr>
<td>camelV0</td>
<td>mfa, wmc, lcom3</td>
<td>mfa, wmc, rfc, loc, cam, lcom3</td>
</tr>
<tr>
<td></td>
<td>mfa, wmc, rfc, loc, cam, lcom3</td>
<td>None</td>
</tr>
<tr>
<td></td>
<td>mfa, wmc, rfc, loc, cam, lcom3</td>
<td>mfa, wmc, rfc, loc, cam, lcom3</td>
</tr>
<tr>
<td>ivy</td>
<td>cam, dam, npm, loc, rfc, wmc</td>
<td>cam, dam, npm, loc, rfc, wmc</td>
</tr>
<tr>
<td></td>
<td>loc, cam, dam, wmc, lcom3</td>
<td>loc, cam, dam, wmc, lcom3</td>
</tr>
<tr>
<td></td>
<td>cam, dam, npm, loc, rfc, wmc</td>
<td>loc, cam, dam, wmc, lcom3</td>
</tr>
<tr>
<td>jeditV0</td>
<td>mfa, dam, loc</td>
<td>mfa, dam, loc</td>
</tr>
<tr>
<td></td>
<td>mfa, lcom3, dam, dit, ic</td>
<td>mfa, dam, loc</td>
</tr>
<tr>
<td></td>
<td>mfa, lcom3, dam, dit, ic</td>
<td>mfa, dam, lcom3, loc, ic</td>
</tr>
<tr>
<td>log4j</td>
<td>loc, ic, dit</td>
<td>mfa, wmc, rfc, loc, npm</td>
</tr>
<tr>
<td></td>
<td>mfa, lcom3, loc, ic</td>
<td>mfa, wmc, rfc, loc, npm</td>
</tr>
<tr>
<td></td>
<td>mfa, lcom3, loc, ic</td>
<td>mfa, lcom3, loc, ic</td>
</tr>
<tr>
<td>lucene</td>
<td>dit, cam, wmc, lcom3, dam, rfc, cbm, mfa, lc</td>
<td>dit, lcom3, dam, mfa</td>
</tr>
<tr>
<td></td>
<td>dit, cam, dam, ic</td>
<td>dit, cam, dam, cbm, ic</td>
</tr>
<tr>
<td>poiV0</td>
<td>mfa, amc, dam</td>
<td>mfa, amc, dam</td>
</tr>
<tr>
<td></td>
<td>mfa, loc, amc, dam, wmc, lcom</td>
<td>mfa, amc, dam</td>
</tr>
<tr>
<td></td>
<td>mfa, loc, amc, dam, wmc, lcom</td>
<td>mfa, loc, amc, dam, wmc, lcom</td>
</tr>
<tr>
<td>synapse</td>
<td>loc, dit, rfc, cam, wmc, dam, lcom, mfa, lcom3</td>
<td>dam</td>
</tr>
<tr>
<td></td>
<td>loc, mfa, cam, lcom, lcom3</td>
<td>loc, mfa, cam, lcom, lcom3</td>
</tr>
<tr>
<td></td>
<td>loc, mfa, cam, lcom, lcom3</td>
<td>loc, mfa, cam, lcom, lcom3</td>
</tr>
<tr>
<td>velocity</td>
<td>dit, wmc, cam, rfc, cbo, moa, dam</td>
<td>mfa, dit</td>
</tr>
<tr>
<td></td>
<td>dit, dam, lcom3, ic, mfa, cbm</td>
<td>dit, dam, lcom3, ic, mfa</td>
</tr>
<tr>
<td></td>
<td>dit, dam, lcom3, ic, mfa, cbm</td>
<td>dit, dam, lcom3, ic, mfa</td>
</tr>
<tr>
<td>xercesV0</td>
<td>wmc</td>
<td>cam, dam, avg, cc, loc, wmc, dit, mfa, ce, lcom3</td>
</tr>
<tr>
<td></td>
<td>wmc, mfa, lcom3, cam, dam</td>
<td>wmc, mfa, lcom3, cam, dam</td>
</tr>
<tr>
<td></td>
<td>wmc, mfa, lcom3, cam, dam</td>
<td>wmc, mfa, lcom3, cam, dam</td>
</tr>
</tbody>
</table>

However, it turns out that advice is sensitive to the tunings used with Random Forest. After tuning, we find that tuned Random Forest loses to Logistic Regression in only 6 out of 17 data sets.

As to Lessmann et al.'s advice (that Random Forest is better than CART), in Table 3.3 and Table 3.4, we saw those counter-examples to that statement. Recall in those tables, tuned CART are better than or equal to tuned Random Forest in \(\frac{12}{17}\) and \(\frac{7}{17}\) data sets in terms of precision and F-measure, respectively. Prior to tuning experiments, those numbers are \(\frac{5}{17}\) and \(\frac{1}{17}\). Results from the non-parametric Kolmogorov-Smirnov(KS) Test show that the performance scores of tuned CART and tuned Random Forest are not statistically different. Note that Random Forest is not significantly better than CART, which would not have been predicted by Lessmann et al.

Hence we answer RQ2 as “yes”: tuning can change how data miners are comparatively ranked.

3.4.3 RQ3: Does Tuning Select Different Project Factors?

Researchers often use data miners to test what factors have most impact on software projects [Bel13; RD13; MS03; Mos08; Zim07; Her13]. Table 3.5 comments that such tests are unreliable since the factors selected by a data miner are much altered before and after tuning.

Table 3.5 shows what features are found in the trees generated by the WHERE algorithm (bold shows the features found by the trees from tuned WHERE; plain text shows the features seen in the untuned study). Note that different features are selected depending on whether or not we tune an algorithm.
Figure 3.3 Deltas in performance between \( np = 10 \) and the recommended \( np \)'s. The recommended \( np \) is better when deltas are above zero. \( np = 90, 50 \) and \( 60 \) are recommended population size for WHERE, CART and Random Forest by Storn.

For example, consider \( mfa \) which is the number of methods inherited by a class plus the number of methods accessible by member methods of the class. For both goals (precision and “F”) \( mfa \) is selected for 8 and 5 data sets, for the untuned and tuned data miner (respectively). Similar differences are seen with other attributes.

As to why different tunings select for different features, recall from §2.1 that tuning changes how data miners heuristically explore a large space of possible models. As we change how that exploration proceeds, so we change what features are found by that exploration.

In any case, our answer to RQ3 is “yes”, tuning changes our conclusions about what factors are most important in software engineering. Hence, many old papers need to be revisited and perhaps revised [Bel13; RD13; MS03; Mos08; Zim07; Her13]. For example, one of us (Menzies) used data miners to assert that some factors were more important than others for predicting successful software reuse [MS03]. That assertion should now be doubted since Menzies did not conduct a tuning study before reporting what factors the data miners found were most influential.

3.4.4 RQ4: Is Tuning Easy?

In terms of the search space explored via tuning, optimizing defect prediction from static code measures is much smaller than the standard optimization.

To see this, recall from Algorithm 1 that DE explores a Population of size \( np = 10 \). This is a very
small population size since Rainer Storn (one of the inventors of DE) recommends setting \( np \) to be ten times larger than the number of attributes being optimized [SP97].

From Table 3.1, we see that Storn would therefore recommend \( np \) values of 90, 50, 60 for WHERE, CART and Random Forest (respectively). Yet we achieve our results using a constant \( np = 10 \); i.e. \( 10 \frac{90}{10}, 10 \frac{50}{10}, 10 \frac{60}{10} \) of the recommended search space.

To justify that \( np = 10 \) is enough, we did another tuning study, where all the settings were the same as before but we set \( np = 90, np = 50 \) and \( np = 60 \) for WHERE, CART and Random Forest, respectively (i.e. the settings as recommended by Storn). The tuning performance of learners was evaluated by precision and “F” as before. To compare performance of each learner with different \( np \)’s, we computed the delta in the performance between \( np = 10 \) and \( np \) using any of \{90, 50, 60\}. Those deltas, shown in Figure 3.3, are sorted along the x-axis. In those plots, a zero or negative y value means that \( np = 10 \) performs as well or better than \( np \in \{90, 50, 60\} \). One technical aside: the data set orderings in Figure 3.3 on the x-axis are not the same (that is, if \( np > 10 \) was useful for optimizing one data set’s precision score, it was not necessary for that data set’s F-measure score).

Figure 3.3 shows that the median improvement is zero; i.e. \( np = 10 \) usually does as well as anything else. This observation is supported by the KS results of Table 3.6. At a 95% confidence, the KS threshold is \( 1.36 \sqrt{34/(17\times17)} = 0.46 \), which is greater than the values in Figure 3.3. That is, no result in Figure 3.3 is significantly different to any other– which is to say that there is no evidence that \( np = 10 \) is a poor choice of search space size.

Another measure showing that tuning is easy (for static code defect predictors) is the number of evaluations required to complete optimization (see next section). That is, we answer RQ4 as “yes”, tuning is surprisingly easy– at least for defect predictors and using DE.

### 3.4.5 RQ5: Is Tuning Impractically Slow?

The number of evaluations and runtime used by our optimizers are shown in Table 3.7 and Table 3.8. WHERE’s runtime are slower than CART and Random Forest since WHERE has yet to benefit from decades of implementation experience with these older algorithms. For example, SciKitLearn’s CART and Random Forest make extensive use of an underlying C library whereas WHERE is a purely interpreted Python.

Looking over Table 3.7, the general pattern is that 50 to 80 evaluations suffice for finding the tuning improvements reported in this study. 50 to 80 evaluations are much fewer than our pre-
Table 3.7 Number of evaluations for tuned learners, optimizing for precision and F-Measure.

<table>
<thead>
<tr>
<th>Datasets</th>
<th>Tuned_Where</th>
<th>Tuned_CART</th>
<th>Tuned_RanFst</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>precision F</td>
<td>precision F</td>
<td>precision F</td>
</tr>
<tr>
<td>antV0</td>
<td>50</td>
<td>60</td>
<td>60</td>
</tr>
<tr>
<td>antV1</td>
<td>60</td>
<td>50</td>
<td>60</td>
</tr>
<tr>
<td>antV2</td>
<td>70</td>
<td>50</td>
<td>60</td>
</tr>
<tr>
<td>camelV0</td>
<td>70</td>
<td>50</td>
<td>70</td>
</tr>
<tr>
<td>camelV1</td>
<td>60</td>
<td>60</td>
<td>110</td>
</tr>
<tr>
<td>ivy</td>
<td>60</td>
<td>60</td>
<td>60</td>
</tr>
<tr>
<td>jeditV0</td>
<td>80</td>
<td>80</td>
<td>90</td>
</tr>
<tr>
<td>jeditV1</td>
<td>60</td>
<td>80</td>
<td>80</td>
</tr>
<tr>
<td>jeditV2</td>
<td>90</td>
<td>80</td>
<td>110</td>
</tr>
<tr>
<td>log4j</td>
<td>50</td>
<td>50</td>
<td>50</td>
</tr>
<tr>
<td>lucene</td>
<td>80</td>
<td>70</td>
<td>60</td>
</tr>
<tr>
<td>poiV0</td>
<td>60</td>
<td>70</td>
<td>60</td>
</tr>
<tr>
<td>poiV1</td>
<td>50</td>
<td>70</td>
<td>50</td>
</tr>
<tr>
<td>synapse</td>
<td>70</td>
<td>60</td>
<td>50</td>
</tr>
<tr>
<td>velocity</td>
<td>60</td>
<td>50</td>
<td>60</td>
</tr>
<tr>
<td>velocity</td>
<td>60</td>
<td>50</td>
<td>60</td>
</tr>
<tr>
<td>xercesV0</td>
<td>60</td>
<td>80</td>
<td>70</td>
</tr>
<tr>
<td>xercesV1</td>
<td>80</td>
<td>80</td>
<td>50</td>
</tr>
</tbody>
</table>

Table 3.8 Runtime for tuned and default learners (in sec), optimizing for precision and F-Measure.

<table>
<thead>
<tr>
<th>Datasets</th>
<th>Tuned_Where</th>
<th>Naive_Where</th>
<th>Tuned_CART</th>
<th>Naive_CART</th>
<th>Tuned_RanFst</th>
<th>Naive_RanFst</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>precision F</td>
<td>precision F</td>
<td>precision F</td>
<td>precision F</td>
<td>precision F</td>
<td>precision F</td>
</tr>
<tr>
<td>antV0</td>
<td>1.65</td>
<td>5.64</td>
<td>0.08</td>
<td>9.78</td>
<td>0.20</td>
<td>0.17</td>
</tr>
<tr>
<td>antV1</td>
<td>3.03</td>
<td>6.62</td>
<td>0.12</td>
<td>13.14</td>
<td>0.25</td>
<td>0.25</td>
</tr>
<tr>
<td>antV2</td>
<td>8.24</td>
<td>9.90</td>
<td>0.24</td>
<td>16.75</td>
<td>0.34</td>
<td>0.36</td>
</tr>
<tr>
<td>camelV0</td>
<td>7.93</td>
<td>12.68</td>
<td>0.24</td>
<td>28.49</td>
<td>0.34</td>
<td>0.41</td>
</tr>
<tr>
<td>camelV1</td>
<td>23.56</td>
<td>21.13</td>
<td>0.27</td>
<td>33.96</td>
<td>0.77</td>
<td>0.85</td>
</tr>
<tr>
<td>ivy</td>
<td>0.97</td>
<td>4.26</td>
<td>0.07</td>
<td>8.89</td>
<td>0.19</td>
<td>0.21</td>
</tr>
<tr>
<td>jeditV0</td>
<td>5.33</td>
<td>8.69</td>
<td>0.11</td>
<td>18.40</td>
<td>0.32</td>
<td>0.37</td>
</tr>
<tr>
<td>jeditV1</td>
<td>6.59</td>
<td>9.05</td>
<td>0.12</td>
<td>17.93</td>
<td>0.36</td>
<td>0.34</td>
</tr>
<tr>
<td>jeditV2</td>
<td>6.88</td>
<td>7.90</td>
<td>0.14</td>
<td>27.34</td>
<td>0.38</td>
<td>0.40</td>
</tr>
<tr>
<td>log4j</td>
<td>1.33</td>
<td>2.60</td>
<td>0.06</td>
<td>9.69</td>
<td>0.15</td>
<td>0.17</td>
</tr>
<tr>
<td>lucene</td>
<td>2.00</td>
<td>6.07</td>
<td>0.10</td>
<td>9.77</td>
<td>0.25</td>
<td>0.35</td>
</tr>
<tr>
<td>poiV0</td>
<td>3.92</td>
<td>7.42</td>
<td>0.09</td>
<td>25.86</td>
<td>0.28</td>
<td>0.32</td>
</tr>
<tr>
<td>poiV1</td>
<td>6.94</td>
<td>9.31</td>
<td>0.13</td>
<td>12.67</td>
<td>0.29</td>
<td>0.36</td>
</tr>
<tr>
<td>synapse</td>
<td>1.85</td>
<td>3.88</td>
<td>0.07</td>
<td>8.13</td>
<td>0.19</td>
<td>0.17</td>
</tr>
<tr>
<td>velocity</td>
<td>2.68</td>
<td>4.27</td>
<td>0.07</td>
<td>15.18</td>
<td>0.21</td>
<td>0.27</td>
</tr>
<tr>
<td>xercesV0</td>
<td>1.98</td>
<td>9.17</td>
<td>0.10</td>
<td>14.17</td>
<td>0.22</td>
<td>0.28</td>
</tr>
<tr>
<td>xercesV1</td>
<td>1173.92</td>
<td>1370.89</td>
<td>0.16</td>
<td>18.27</td>
<td>0.40</td>
<td>0.46</td>
</tr>
</tbody>
</table>

Experimental intuition. Prior to this study, the authors have conducted numerous explorations of evolutionary algorithms for search-based SE applications [Kra15b; Kra15a; FM02; Men07a; II09]. Based on that work, our expectations were that non-parametric evolutionary optimization would take thousands, if not millions, of evaluations of candidate tunings. This turned out not to be that case. By comparing the runtime of tuned and default learners shown in Table 3.8, we notice that the actual tuning time for most data sets is not extremely long.

Hence, we answer RQ5 as “no”: tuning is so fast that it could (and should) be used by anyone using defect predictors.

As to why DE can tune defect predictors so quickly, that is an open question. One possibility is that the search space within the control space of these data miners has many accumulative effects.
such that one decision can cascade into another (and the combination of decisions is better than each separate one). DE would be a natural tool for reasoning about such “cascades”, due to the way it mashes candidates together, then inserts the result back into the frontier (making them available for even more mashing at the next step of the inference).

3.4.6 RQ6: Should we use “off-the-shelf” Tunings?

In Figure 3.4, we show how tuning selects the optimal values for tuned parameters. For space limitation, only four parameters from WHERE learner are selected as representatives and all the others can be found in our online support documents (https://goo.gl/aHQKtU). Note that the tunings learned were different in different data sets and for different goals. Also, the tunings learned by DE were often very different to the default (the default values for threshold, infoPrune, min_Size and wriggle are 0.5, 0.33, 0.5 and 0.2, respectively). That is, to achieve the performance improvements seen in the paper, the default tuning parameters required a wide range of adjustments.

Hence, we answer RQ6 as “no” since, to achieve the improvements seen in this study, tuning has to be repeated whenever the goals or data sets are changed. Given this requirement to repeatedly run tuning, it is fortunate that (as shown above) tuning is so easy and so fast (at least for defect predictors from static code attributes).
3.5 Reliability and Validity

Reliability refers to the consistency of the results obtained from the research. For example, how well independent researchers could reproduce the study? To increase external reliability, this study has taken care to either clearly define our algorithms or use implementations from the public domain (SciKitLearn). Also, all the data used in this work is available on-line in the PROMISE code repository and all our algorithms are on-line at github.com/ai-se/where.

External validity checks if the results are of relevance for other cases, or can be generalized from samples to populations. The examples of this study only relate to precision, recall, and the F-measure but the general principle (that the search bias changes the search conclusions) holds for any set of goals. Also, the tuning results shown here only came from one software analytics task (defect prediction from static code attributes). There are many other kinds of software analytics tasks (software development effort estimation, social network mining, detecting duplicate issue reports, etc) and the implication of this study for those tasks is unclear. However, those other tasks often use the same kinds of learners explored in this study so it is quite possible that the conclusions of this study apply to other SE analytics tasks as well.

3.6 Conclusions

Our exploration of the six research questions listed in the introduction show that when learning defect predictors for static code attributes, analytics without parameter tuning are considered harmful and misleading:

- Tuning improves the performance scores of a predictor. That improvement is usually positive (see Figure 3.1) and sometimes it can be quite dramatic (e.g. precision changing from 0 to 60%).

- Tuning changes conclusions on what learners are better than others. Hence, it is time to revisit numerous prior publications of our own [Men07b] and others [Les08; Hal12].

- Also, tuning changes conclusions on what factors are most important in software development. Once again, this means that old papers may need to be revised including those some of our own [MS03] and others [Bel13; RD13; Mos08; Zim07; Her13].

As to future work, it is now important to explore the implications of these conclusions to other kinds of software analytics. this study has investigated some learners using one optimizer. Hence, we can make no claim that DE is the best optimizer for all learners. Rather, our point is that there exists at least some learners whose performance can be dramatically improved by at least one simple
optimization scheme. We hope that this work inspires much future work as this community develops and debugs best practices for tuning software analytics.

Finally, on a more general note, we point out that Fürnkranz [FF05] says data mining is inherently a multi-objective optimization problem that seeks the smallest model with the highest performance, that generalizes best for future examples (perhaps learned in minimal time using the least amount of data). In this view, we are using DE to optimize an optimizer. Perhaps a better approach might be to dispense with the separation of “optimizer” and “learner” and combine them both into one system that learns how to tune itself as it executes. If this view is useful, then instead of adding elaborations to data miners (as done in this study, or by researchers exploring hyper-heuristics [Jia15]), it should be possible to radically simplify optimization and data mining with a single system that rapidly performs both tasks.
This chapter firstly appeared as paper “Why is Differential Evolution Better than Grid Search for Tuning Defect Predictors?”. In this chapter, we show that software analytics can be simpler.

In software analytics, at least for software defect prediction, several methods, like grid search and differential evolution (DE), have been proposed to learn these parameters recently. In this chapter, to further defend our claim that software analytics can be simpler, we want to evaluate which method can find better parameters from the perspectives of effectiveness and economic. Experimental results show that the seemingly complete approach of grid search does no better, and sometimes worse, than the differential evolution method. When repeated 20 times to check for conclusion validity, DE was much faster than grid search to tune Random Forests and CART. Bergstra and Bengio comment that grid search is not more effective than more randomized searchers if the underlying search space is inherently low dimensional. This is significant since recent results show that defect prediction exhibits very low intrinsic dimensionality— an observation that explains why a fast method like DE may work as well as a seemingly more thorough grid search. This suggests, as a future research direction, that it might be possible to peek at data sets before doing any optimization in order to match the optimization algorithm to the problem at hand.
4.1 Introduction

Given the large amount of data now available to analysts, many researchers in empirical software engineering are now turning to automatic data miners to help them explore the data [MZ13b; Men07b; FM17b]. These learners come with many “magic numbers” which, if tuned, can generate better predictors for particular data spaces. For example, Fu et al. [Fu16a] showed that, with parameter tuning, the precision of software defect predictors learned from static code metrics can grow by 20% to 60%.

Since tuning is so effective, it is tempting to enforce tuning for all data mining studies. However, there is a problem: tuning can be slow. For example, in this study, one of our tuners used 27.5 days of CPU time to perform 20 repeats of tuning Random Forests [Bre01] for one tuning goal with 17 test data sets. Other researchers also comment that tuning may require weeks, or more, of CPU time [AF11]. One dramatic demonstration of the slowness of tuning (not for defect prediction) comes from Wang et al. [Wan13b] where the study required 15 years of CPU to explore 9.3 million candidate configurations for software clone detectors.

One way to address the cost of tuning data miners is to use cloud-based CPU farms. The advantage of this approach is that it is simple to implement (just buy the cloud-based CPU time). For example, such cloud resources were used in this paper. But uses of cloud-based resources have several disadvantages:

- Cloud computing environments are extensively monetized so the total financial cost of tuning can be prohibitive.
- The CPU time is wasted if there is a faster and more effective way.

It turns out that the last point is indeed the case—at least for tuning defect predictors learned from static code attributes. The case study of this study compares two tuning methods: the grid search as used by Tantithamthavorn et al. [Tan16] and differential evolution as used by Fu et al. [Fu16a]. Both papers investigate the impacts of tunings on defect predictors and find that parameter tuning can improve the learner performance by different methods. However, little is known which method can find better parameters in terms of performance scores and runtime cost. In this study, we would like to investigate the following questions:

- **RQ1:** Does tuning improve learners’ performance?
  
The case study shows that parameter tuning does improve learners’ performance. Hence, this suggests all the future empirical study involving with data mining methods should not use “off-the-shelf” data mining tools.

- **RQ2:** Is grid search statistically better than DE in terms of performance scores (AUC & AUC$_{20}$)\(^1\) &

\(^1\)This is defined in section 4.3.5
The case study shows that grid search just performs statistically better in 6 out of 17 data sets for CART (Classification And Regression Tree) with precision evaluation measure. Other than that, grid search works just as well as DE for Random Forests and CART.

**RQ3: Is DE a more efficient tuner than grid search in terms of runtime cost?**

The case study show that DE runs much faster than grid search, which terminates faster than grid search.

To answer the research questions, we build defect predictors with CART and Random Forests, and then conduct parameter tuning with DE, grid search and random search. After that, we evaluate the untuned learners, DE tuned learners, grid search tuned learners, and random search tuned learners on 17 test data sets in terms of AUC, \(AUC_{20}\), precision and F-Measure. We also record the start and end time when each tuner running. This study makes the following contributions:

- To the best of our knowledge, this is the first such comparison of these two techniques for the purposes of tuning defect predictors;
- We show that differential evolution is just as effective as grid search for improving defect predictors, while being one to two orders of magnitude faster;
- Hence, we offer a strong recommendation to use evolutionary algorithms (like DE) for tuning defect predictors;
- Lastly, we propose a prediction method that would allow future analysts to match the optimization method to the data at hand.

The rest of this study is organized as follows. Section 4.2 describes background, how defect predictors can be generated by data miners, and how tuning can affect the effectiveness of the learners. Section 4.3 defines and compares differential evolution and grid search as tuners. Section 4.4 presents the results from the case study. Section 4.5 is a discussion on why DE is effective and faster than grid search by showing how intrinsic dimensionality of the tuning search space help stochastic search method, like DE, to converge faster than more exhaustive methods like grid search. Section 4.6 discusses the potential threats to validity of this study. Section 4.7 concludes this study and discusses the future work.

While the specific conclusion of this study relates to defect prediction, this work has broader implications across data science. Many researchers in software engineering (SE) explore hyper-parameter tuning [AF11; Jia15; Pan13; Cor10; MY13; Son13; Les08; Tan16; Fu16a; Wan13b] (for details, see Section 2.3). Within the group, grid search is the most commonly used tuner. Such studies can be tediously slow, often requiring days to weeks of CPU time. The success of differential evolution
to tune defect predictors raises the possibility that tuning research could be greatly accelerated by a better selection of tuning algorithms. This could be a very productive avenue for future research.

4.2 Background

4.2.1 Data Mining

Data miners produce summaries of data and they are efficient since they employ various heuristics in order to reduce their search space for finding summaries. For examples, CART and Random Forests tree learners. These algorithms divide a data set, then recursively split on each node until some stop criterion is satisfied. In the case of building defect predictors, these learners reflect on the number of issue reports $d_i$ raised for each class in a software system where the issue counts are converted into binary “yes/no” decisions via Equation 4.1, where $T$ is a threshold number.

$$\text{inspect} = \begin{cases} 
  d_i \geq T & \rightarrow \text{Yes} \\
  d_i < T & \rightarrow \text{No},
\end{cases} \quad (4.1)$$

For the specific implementation in Scikit-learn [Ped11], the splitting process is controlled by numerous tuning parameters listed in Table 4.1, where the default parameters for CART and Random Forest are set by the Scikit-learn authors except for $n\_estimators$, as recommended by Witten et al. [WF11], we used 100 trees as default instead of 10. If data contains more than $\text{min\_sample\_split}$, then a split is attempted. On the other hand, if a split contains no more than $\text{min\_samples\_leaf}$, then the recursion stops.

These learners use different techniques to explore the splits:

- CART finds the attributes of the dataset whose ranges contain rows (samples of data) with least variance in the number of defects: if an attribute ranges $r_i$ is found in $n_i$ rows, each with a defect count variance of $v_i$, then CART seeks the attributes whose ranges minimizes $\sum_i (\sqrt{v_i} \times n_i / \sum_i n_i)$.

- Random Forests divides data like CART then builds number of $F$ trees ($F > 1$), each time using some random subset of the attributes.

Note that some tuning parameters are learner specific. $\text{max\_feature}$ is used by CART and Random Forests to select the number of attributes used to build one tree. CART’s default is to use all the attributes while Random Forests usually selects the square root of the number of attributes. Also, $\text{max\_leaf\_nodes}$ is the upper bound on leaf nodes generated in a Random Forests. Lastly, $\text{max\_depth}$ is the upper bound on the depth of the CART tree.
Table 4.1 List of parameters tuned by this study.

<table>
<thead>
<tr>
<th>Learner Name</th>
<th>Parameters</th>
<th>Default</th>
<th>Tuning Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CART</td>
<td>threshold</td>
<td>0.5</td>
<td>[0, 1]</td>
<td>The value to determine defective or not.</td>
</tr>
<tr>
<td></td>
<td>max_feature</td>
<td>None</td>
<td>[0.01, 1]</td>
<td>The number of features to consider when looking for the best split.</td>
</tr>
<tr>
<td></td>
<td>min_sample_split</td>
<td>2</td>
<td>[2, 20]</td>
<td>The minimum number of samples required to split an internal node.</td>
</tr>
<tr>
<td></td>
<td>min_samples_leaf</td>
<td>1</td>
<td>[1, 20]</td>
<td>The minimum number of samples required to be at a leaf node.</td>
</tr>
<tr>
<td></td>
<td>max_depth</td>
<td>None</td>
<td>[1, 50]</td>
<td>The maximum depth of the tree.</td>
</tr>
<tr>
<td>Random Forests</td>
<td>threshold</td>
<td>0.5</td>
<td>[0.01, 1]</td>
<td>The value to determine defective or not.</td>
</tr>
<tr>
<td></td>
<td>max_feature</td>
<td>None</td>
<td>[0.01, 1]</td>
<td>The number of features to consider when looking for the best split.</td>
</tr>
<tr>
<td></td>
<td>max_leaf_nodes</td>
<td>None</td>
<td>[1, 50]</td>
<td>Grow trees with max_leaf_nodes in best-first fashion.</td>
</tr>
<tr>
<td></td>
<td>min_sample_split</td>
<td>2</td>
<td>[2, 20]</td>
<td>The minimum number of samples required to split an internal node.</td>
</tr>
<tr>
<td></td>
<td>min_samples_leaf</td>
<td>1</td>
<td>[1, 20]</td>
<td>The minimum number of samples required to be at a leaf node.</td>
</tr>
<tr>
<td></td>
<td>n_estimators</td>
<td>100</td>
<td>[50, 150]</td>
<td>The number of trees in the forest.</td>
</tr>
</tbody>
</table>

4.2.2 Parameter Tuning

Parameter tuning for evolutionary algorithms was studied by Arcuri et al. [AF11] and presented at SSBSE’11. Using the grid search method (described below), they found that different parameter settings for evolutionary programs cause very large variance in their performance. Also, while default parameter settings perform relatively well, they are far from optimal on particular problem instances.

Tuning is now explored in many parts of the SE research literature. Apart for defect prediction, tuning is used in the hyper-parameter optimization literature exploring better combinatorial search methods for software testing [Jia15] or the use of genetic algorithms to explore 9.3 million different configurations for clone detection algorithms [Wan13b].

Other researchers explore the effects of parameter tuning on topic modeling [Pan13]. Like Arcuri et al. [AF11], that work showed that the performance of the LDA (Latent Dirichlet Allocation) topic modeling algorithm was greatly effected by the choice of four parameters that configure LDA. Furthermore, Agrawal et.al [Agr18] demonstrate that more stable topics can be generated by tuning LDA parameters using differential evolution algorithm.

Tuning is also used for software effort estimation; e.g. using tabu search for tuning SVM [Cor10]; or genetic algorithms for tuning ensembles [MY13]; or as an exploration tool for checking if parameter settings affect the performance of effort estimators (and what learning machines are more sensitive to their parameters) [Son13]. The latter study explored Random Forests, kth-nearest neighbor methods, MLPs (MultiLayer Perceptrons), and bagging. This was another grid search paper that explored a range of tunings parameters and the corresponding value ranges were divided into 5
Figure 4.1 Literature review about parameter tuning on 52 top cited defect prediction papers

For defect prediction, we have performed a literature review starting with the key words “software engineering” and “defect prediction” and “data mining” [Fu16a] (more details can be found at https://goo.gl/Ini9nF). After sorting by the citation count and discarding the non-SE papers (and those without a pdf link), we read over this sample of 52 highly-cited SE defect prediction papers. What we found in that sample was that few authors acknowledged the impact of tunings (exceptions: [Gao11; Les08; Tan16; Fu16a]). As shown in Figure 4.1, about 80% of papers in our sample did not mention any parameter tuning and just use the “off-the-shelf” configuration of the data miner (e.g. [Men07b; Mos08; EE08]). Lessmann et al. [Les08] used grid search to tune parameters as part of their extensive analysis of different algorithms for defect prediction. Strangely, they only tuned a small set of their learners while for most of them, they used the default settings. This is an observation we cannot explain but our conjecture is that the overall cost of their grid search tuning was so expensive that they restricted it to just the hardest choices. Gao et al. [Gao11] acknowledged the impacts of the parameter tuning and they set some parameters within the learner, like set $k = 30$ for KNN. However, they did not provide any further explanation.

There are two recent work investigating the effects of parameter tuning on defect prediction by Tantithamthavorn et al. [Tan16] and Fu et al. [Fu16a]: the former used grid search while the latter used differential evolution (both these techniques are detailed, below). These two teams worked separately using completely different scripts (written in “R” or in Python). Yet for tuning defect predictors, both groups reported the same results as follows:

- Across a range of performance measures (AUC, precision, recall, F-Measure), tuning rarely makes performance worse;
Tuning offers a median improvement of 5% to 15% for most measures;

For a third of data sets exploration, tuning can result in performance improvements of 30% to 50%.

Also, in a result that echoes one of the conclusions of Arcuri & Fraser, Fu et al. [Fu16a] report that different data sets require different tunings.

What was different between Fu et al. [Fu16a] and Tantithamthavorn et al. [Tan16] was the computational costs of the two studies. Fu et al. [Fu16a] used a single desktop machine and all their runs terminated in 2 hours for one tuning goal. On the other hand, the grid search of Tantithamthavorn et al. [Tan16] used 43 high performance computing machines with 24 hyper-threads times 43 machines = 1,032 hyper-threads. Their total runtime were not reported— but as shown below, such tuning with grid search can take over a day just to learn one defect predictor.

4.3 Method

Tantithamthavorn et al. [Tan16] and Fu et al. [Fu16a] use different methods to tune defect predictors. Neither offer a comparison of their preferred tuning method to any other. This section offers such a case study: specifically, a comparison of grid search and different evolution for tuning defect predictors.

4.3.1 Algorithms

Grid search is simply picking a set of values for each configuration parameter and evaluating all the combinations of these values, and then return the best one as the final optimal result, which can be simply implemented by nested for-loops. For example, for Naive Bayes, two loops might explore different values from the Laplace and M-estimator while a third loop might explore what happens when numeric values are divided into, say, $2 \leq b \leq 10$ bins.

Bergstra and Bengio [BB12] comment on the popularity of grid search: (a) Such a simple search gives researchers some degree of insight into it; (b) There is little technical overhead or barrier to its implementation; (c) As to automating grid search, it is simple to implement and parallelization is trivial; (d) According to Bergstra and Bengio [BB12], grid search (on a computing cluster) can find better tunings than sequential optimization (in the same amount of time).

Since it’s easy to understand and implement and, to some extent, also has good performance, grid search has been available in most popular data mining and machine learning tools, like caret [Kuh14] package in the R and GridSearchCV module in Scikit-learn [Ped11].

Differential evolution is included in many optimization toolkits, like JMetal in Java [DN11]. But given its implementation simplicity, it is often written from scratch using the researcher’s preferred
scripting language. Differential evolution just randomly picks three different vectors $B, C, D$ from a list called $F$ (the \textit{frontier}) for each parent vector $A$ in $F$ [SP97]. Each pick generates a new vector $E$ (which replaces $A$ if it scores better according to the tuning goal). $E$ is generated as follows:

$$
\forall i \in A, E_i = \begin{cases} 
B_i + f \ast (C_i - D_i) & \text{if } R < cr \\
A_i & \text{otherwise}
\end{cases}
$$

where $0 \leq R \leq 1$ is a random number, and $f, cr$ are constants (following Storn et al. [SP97], we use $cr = 0.3$ and $f = 0.75$). Also, one $A_i$ value (picked at random) is moved to $E_i$ to ensure that $E$ has at least one unchanged part of an existing vector.

As a sanity check, we also provide \textit{random search} as a third optimizer to tune the parameters. Random search is nothing but randomly generate a set of different candidate parameters, and always evaluate them against the current “best” one. If better, then it will replace the “best” one. The process is repeated until the stop condition meets. In this case study, we set maximum iterations for random search the same as median number of evaluations in DE. The parameter will be randomly generated from the same tuning range as in Table 4.1.

Grid search is much slower than DE since DE explores fewer options. Grid search’s execution of $X$ loops exploring $N$ options takes time $O(N^X)$, where $X$ is the number of parameters being tuned. Hence, Tantithamthavorn et al. [Tan16] required 1000s of hyperthreads to complete their study in less than a day [Tan16]. The grid search of Arcuri & Fraser [AF11] took weeks to terminate, even using a large computer cluster. In the following study, our grid search times took 27.5 days of total CPU time for Random Forests with \textit{F-Measure}.

On the other hand, DE’s runtime is much faster since it is linear on the size of the frontiers, i.e. $O(|F|)$.

\subsection*{4.3.2 Data Miners}

This study uses Random Forests and CART, for the following reason. Firstly, they were two of the learners studied by Tantithamthavorn et al. [Tan16] and Fu et al. [Fu16a]. Secondly, they are interesting learners in that they represent two ends of a performance spectrum for defect predictors. CART and Random Forests were mentioned in a recent IEEE TSE paper by Lessmann et al. [Les08] that compared 22 learners for defect prediction. That study ranked CART as the worst learning technique and Random Forests as the best one. In a demonstration of the impact of tuning, Fu et al. [Fu16a] showed that they could refute the conclusions of Lessmann et al. [Les08] in the sense that, after tuning, CART performs just as well as Random Forests.
4.3.3 Tuning Parameters

The DE and grid search explored the parameter space as described Table 4.1. Specifically, since Tantithamthavorn et al. [Tan16] divide each tuning range into 5 bins (if applicable), we also use the same policy here. For example, we pick values \([50, 75, 100, 125, 150]\) for \(n_{\text{estimators}}\). Other parameters grid will be generated in the same way. As to why we used the “Tuning Range” shown in Table 4.1, and not some other ranges, we note that (1) those ranges included the defaults; (2) the results shown below show that by exploring those ranges, we achieved large gains in the performance of our defect predictors. This is not to say that larger tuning ranges might not result in greater improvements.

### Table 4.2 Data used in this case study.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>antV0</th>
<th>antV1</th>
<th>antV2</th>
<th>camelV0</th>
<th>camelV1</th>
<th>ivyV0</th>
<th>jeditV0</th>
<th>jeditV1</th>
<th>jeditV2</th>
</tr>
</thead>
<tbody>
<tr>
<td>training (release (i))</td>
<td>20/125</td>
<td>40/178</td>
<td>32/293</td>
<td>13/339</td>
<td>216/608</td>
<td>63/111</td>
<td>90/272</td>
<td>75/306</td>
<td>79/312</td>
</tr>
<tr>
<td>tuning (release (i + 1))</td>
<td>40/178</td>
<td>32/293</td>
<td>92/351</td>
<td>216/608</td>
<td>145/872</td>
<td>16/241</td>
<td>75/306</td>
<td>79/312</td>
<td>48/367</td>
</tr>
<tr>
<td>testing (release (i + 2))</td>
<td>32/293</td>
<td>92/351</td>
<td>166/745</td>
<td>145/872</td>
<td>188/965</td>
<td>40/352</td>
<td>79/312</td>
<td>48/367</td>
<td>11/492</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dataset</th>
<th>log4jV0</th>
<th>luceneV0</th>
<th>poiV0</th>
<th>poiV1</th>
<th>synapseV0</th>
<th>velocityV0</th>
<th>xercesV0</th>
<th>xercesV1</th>
</tr>
</thead>
<tbody>
<tr>
<td>training (release (i))</td>
<td>34/135</td>
<td>91/195</td>
<td>141/237</td>
<td>37/314</td>
<td>16/157</td>
<td>147/196</td>
<td>77/162</td>
<td>71/440</td>
</tr>
<tr>
<td>tuning (release (i + 1))</td>
<td>37/109</td>
<td>144/247</td>
<td>37/314</td>
<td>248/385</td>
<td>60/222</td>
<td>142/214</td>
<td>71/440</td>
<td>69/453</td>
</tr>
<tr>
<td>testing (release (i + 2))</td>
<td>189/205</td>
<td>203/340</td>
<td>248/385</td>
<td>281/442</td>
<td>86/256</td>
<td>78/229</td>
<td>69/453</td>
<td>437/588</td>
</tr>
</tbody>
</table>

4.3.4 Data

Our defect data, shown in Table 4.2 comes from SEACRAFT repository so is available for others to replicate our results [FM17a]. This data pertains to open source Java systems defined in terms of Table 1.1: \textit{ant}, \textit{camel}, \textit{ivy}, \textit{jedit}, \textit{log4j}, \textit{lucene}, \textit{poi}, \textit{synapse}, \textit{velocity} and \textit{xerces}.

We selected these data sets since they have at least three consecutive releases (where release \(i + 1\) was built after release \(i\)). This will allow us to build defect predictors based on the past data and then predict (test) defects on future version projects, which will be a more practical scenario.

More specifically, when tuning a learner:

- Release \(i\) was used for training a learner with tunings generated by grid search or differential evolution.
- During the search, each candidate has to be evaluated by some model, which we build using CART or Random Forests from release \(i + 1\).
- After grid search or DE terminated, we tested the tunings found by those methods on CART or Random Forests and applied to release \(i + 2\).
• For comparison purposes, CART and Random Forests were also trained (with default tunings) on releases $i$ and $i+1$, then tested on release $i+2$.

In Table 4.2, we list all the data sets used for our case study. The fractions denote $\text{defects/total}$. E.g., given antV0 data sets, we have training, tuning and testing data. Specifically, the training data data set has 20 defective instances out of 125 total.

### 4.3.5 Optimization Goals

Our optimizers explore tuning improvements for $\text{precision}$, $\text{AUC}$, $\text{AUC}_{20}$ and the $\text{F-Measure}$, defined as follows. Note that all these scores, the better scores are larger.

\[
\begin{align*}
\text{Precision} &= \frac{\text{True Positive}}{\text{True Positive} + \text{False Positive}} \\
\text{Recall} &= \frac{\text{True Positive}}{\text{True Positive} + \text{False Negative}} \\
\text{F-Measure} &= \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Recall} + \text{Precision}}
\end{align*}
\]

$\text{AUC}$ is the area under the receiver operating characteristic (ROC), which is a curve that plots the true positive rate (recall) and false positive rates (false alarm). The AUC is widely used because $\text{AUC}$ is unaffected by class imbalance as well as being independent from the cutoff probability (prediction threshold) that is used to decide whether an instance should be classified as positive or negative [Nam17; Rah12; Son11].

Yet another measure, $\text{AUC(LOC, recall)}$, reflects the effort involved in processing files predicted to be fault, which plot $\text{recall}$ against LOC (lines of code) from the ordered of files (predicted by the model) in ascending order of size. In the SE literature, it is standard practice to report the y-axis measured at 20% effort [MK10; Yan16b; Kam13] (denoted $\text{AUC}_{20}$).

We do not explore all goals since some have trivial, but not useful, solutions. For example, when we tune for $\text{recall}$, we can achieve near 100% recall - but at the cost of a near 100% false alarms. Similarly, when minimize false alarms, we can achieve near 0% false alarms - but at the cost of a near 0% recall. The lesson here is that tuning for defect predictors needs some "brake" effect where multiple goals are in contention (so one cannot be pushed to some extreme value without being "braked" by the other). Precision's definition takes into accounts not only the defective examples but also the none defective ones as well so it has this brake effect. The same is true for the F-Measure (since it uses precision).
4.3.6 20 Random Runs:

All our studies were repeated 20 times to check for the stability of conclusions across different random biases. Initially, we planned for 30 repeats but grid search proved to be so slow that, for pragmatic reasons, we used 20 repeats. To be clear, the random seed is different for each data set in each repeat, but it will be the same across learners built by grid search and DE as well as random search for the same data set.

The reason that we believe this is the right thing to do is the search bias for a particular training/tuning/testing run is always the same. For the same training/tuning/testing data set, searching algorithms will start from the same random seed. With this approach, it is important to note that different triplets have different seed values (so this case study does sample across a range of search biases).

4.3.7 Statistical Tests

For each data set, the results of grid search and DE were compared across the 20 repeats. Statistical differences were tested by the Scott-Knott test [SK74] that used the Efron & Tibshirani bootstrap procedure [ET93] and the Vargha and Delaney A12 effect size test [VD00] to confirm that sub-cluster of the treatments are statistically different by more than a small effect. We used these statistical tests since they were recently endorsed in the SE literature by Mittas & Angelis (for Scott-Knott) in TSE’13 [MA13] and Acura & Briand (for A12) at ICSE’11 [AB11].

![Figure 4.2 Tuning to improve $F$. Median results from 20 repeats.](image-url)
4.4 Results

In this section, we present the results from the above designed case studies. To answer the research questions, we build defect predictors based on CART and Random Forests. Then, we apply DE, random search and grid search on the learners to tune the parameters listed in Table 4.1 according different tuning goals, precision, F-Measure, AUC and $AUC_{20}$, respectively. Then, the performance of these learners are evaluated based on these measures accordingly.

**RQ1: Does tuning improve learners’ performance?**

The performance scores of untuned learner and tuned learners (by DE, random search and grid search) in terms of precision, F-Measure, AUC and $AUC_{20}$ are shown in Figure 4.2, Figure 4.3, Figure 4.4 and Figure 4.5, respectively. As a note, all these four figures are generated by four separate case studies with tuning goals of precision, F-Measure, AUC and $AUC_{20}$, respectively. In those figures:
(a) The blue squares show the DE results; (b) The green diamonds show the results of running a learner using the default parameter settings; (c) The yellow triangles show the random search results; (d) The red dots show the grid search results.

Overall, from Figure 4.2 to Figure 4.5, we can see that random search and DE both improve learners performance in terms of precision, F-Measure, AUC and $AUC_{20}$, respectively. For example, in this case study, a simple random search can improve precision scores for CART and Random Forests in $\frac{1}{17}$ and $\frac{7}{17}$ data sets, respectively. The similar pattern can be found when F-Measure, AUC or $AUC_{20}$ is set as the tuning objective. On the other hand, we observe that DE improves precision, F-Measure, AUC and $AUC_{20}$ scores for CART in $\frac{12}{17}$, $\frac{13}{17}$, $\frac{7}{17}$ data sets, respectively. Similar pattern can be found for Random Forests. This further supports the conclusion from Fu et al. [Fu16a] that parameter tuning is helpful and can not be ignored. The above analysis indicates that:
Figure 4.4 Tuning to improve AUC. Median results from 20 repeats.

Figure 4.5 Tuning to improve AUC. Median results from 20 repeats.
**Tuning can improve learner performance in the majority of cases.**

**RQ2:** Is grid search statistically better than DE in terms of performance improvements (AUC & AUC\(_{20}\) & precision & F-Measure)?

Grid search results are denoted by the red dots. In this study, to compare whether grid search is statistically better than DE, we apply Scott Knott test on evaluation scores over 20 repeats. Based on the Scott Knott results, (a) when tuning CART, grid search performs statistically better than DE only on velocityV0 for AUC and six test data sets (i.e., camelV0, jeditV0, jeditV1, log4jV0, poiV0, xercesV1) for precision. However, grid search is never better than DE for F-Measure. Furthermore, grid search works even worse than the default learner for AUC\(_{20}\) scores on almost all data sets. (b) when tuning Random Forests, grid search only works statistically better than DE on log4jV0 test data for precision score. Other than that, grid search never outperforms DE for any evaluation measure.

![Figure 4.6](image)

**Figure 4.6** Time (in seconds) required to run 20 repeats of parameter tunings for learners to generate Figure 4.2 to Figure 4.5.

Note that Figure 4.2 to Figure 4.5 also contain results that echoes conclusions from Arcuri & Fraser in SSBSE’11 [AF11]: (a) *Default parameter settings perform relatively well.* Note that there exists several cases where the tuned results (blue squares or red dots) are not much different to the results from using the default parameters (the green diamonds). (b) *But they are far from optimal on particular problem instances.* Note all the results where the red dots and blue squares are higher than green, particularly in the results where we are tuning for precision.

The comparisons between DE and grid search in terms of different evaluation measures suggest...
that:

**Differential evolution is just as effective as grid search for improving defect predictors.**

**RQ3:** Is DE a more efficient tuner than grid search in terms of runtime cost?

Figure 4.6 and Figure 4.7 show the runtime cost of grid search, random search and DE over 17 data sets including precision, F-Measure, AUC and AUC$_{20}$ as optimization objectives.

The Figure 4.6 shows the raw runtime (in seconds) of our entire tuning process in terms of 4 evaluation measures. In that plot, firstly, we see that tuning CART is faster than tuning Random Forests. Specifically, 20 repeats of a grid search of Random Forests required 109 days of CPU time for all 4 evaluation measures; Secondly, DE is faster than grid search: it takes $10^4$ seconds for both DE to tune CART and Random Forests, while $10^5$ and $10^7$ seconds for grid search to complete the tuning. DE runs as fast as random search for both CART and RF and there's no significant difference between DE and random search in terms of runtime.

![Figure 4.6](image)

**Figure 4.6** Runtime relative to just running the learners using their default tunings.

The Figure 4.7 shows the same information as Figure 4.6 but in the ratio form. In that plot, each bar represents the runtime of the learner with the corresponding tuning technique to running learner without tuning. From Figure 4.7, we observe that grid search is 1,000 to 10,000 times slower than just running the default learners, while DE adds a factor of $10^2$ to the default (untuned) runtime cost. Overall, both Figure 4.6 and Figure 4.7 suggest that:
**Differential evolution runs much faster than grid search.**

According to the above results, we summarize the conclusions as: (a) Both DE and random search as well as grid search can improve learners’ performance for most cases; (b) Compared to DE, grid search runs far too long for too little additional benefit; (c) Both DE and random search require the same amount of runtime. But for some cases, DE has better performance than random search. (d) There are many cases where DE outperforms grid search.

### 4.5 Discussion

#### 4.5.1 Why does DE perform better than grid search?

How to explain the surprising success of DE over grid search? Surely a thorough investigation of all options (via grid search) must do better than a partial exploration of just a few options (via DEs).

It turns out that grid search is not a thorough exploration of all options. Rather, it jumps through different parameter settings between some min and max value of pre-defined tuning range. If the best options lie in between these jumps, then grid search will skip the critical tuning values. That means, the selected grid points will finally determine what kind of tunings we can get and good tunings require a lot of expert knowledge.

Note that DE is less prone to skip since, as shown in Equation 4.2, tuning values are adjusted by some random amount that is the difference between two randomly selected vectors. Further, if that process results in a better candidate, then this new randomly generated value might be used as the start point of a subsequent random selection of data. Hence DE is more likely than grid search to “fill in the gaps” between an initially selected values.

Another important difference between DE and grid search is the nature of their searches:

- **All the grid points** in the pre-defined grids are independently evaluated. This is useful since it makes grid search highly suited for parallelism (just run some of the loops on different processors). That said, this independence has a drawback: any lessons learned midway by grid search cannot affect (improve) the inferences made in the remaining runs.

- **DE’s candidates** (equivalent to grid points) do “transfer knowledge” to candidates in the new generation. Since DE is an evolutionary algorithm, the better candidates will be inherited by the following generations. That said, DE’s discoveries of better vectors accumulate in the frontier—which means new solutions (candidates) are being continually built from increasingly better solutions cached in the frontier. That is, lessons learned midway through a DE run can improve the inferences made in the remaining runs.
Bergstra and Benigo [BB12] offer a more formal analysis for why random searches (like DEs) can do better than grid search. They comment that grid search will be expected to fail if the region containing the useful tunings is very small. In such a search space: (a) Grid search can waste much time exploring an irrelevant part of the space. (b) Grid search’s effectiveness is limited by the curse of dimensionality.

Bergstra and Benigo reasons for the second point are as follows. They compared deep belief networks configured by a thoughtful combination of manual search and grid search, and purely random search over the same 32-dimensional configuration space. They found statistically equal performance on four of seven data sets, and superior performance on one of seven. A Gaussian process analysis of their systems revealed that for most data sets only a few of the tuning really matter, but that different hyper-parameters are important on different data sets. They comment that a grid with sufficient granularity to tune for all data sets must consequently be inefficient for each individual data set because of the curse of dimensionality: the number of wasted grid search trials is exponential in the number of search dimensions that turn out to be irrelevant for a particular data set. Bergstra and Benigo add:

... in contrast, random search thrives on low effective dimensionality. Random search has the same efficiency in the relevant subspace as if it had been used to search only the relevant dimensions.

Our previous results in Section 4.4 also verified Bergstra and Benigo’s conclusion that random search is much better than grid search for exploring defect prediction data space, where random search is as good as DE in most data sets. But grid search rarely outperformed DE and random search in terms of performance scores (F-Measure, Precision, AUC and AUC₂₀). However, grid search just wastes a lot of time to explore unnecessary space.

4.5.2 When (not) to use DE?

How can we assess the external validity of the above results? Is it possible to build some predictor when DEs might and might not work well?

To explore these questions we use Bergstra and Benigo’s comments to define the conditions when we would expect DEs to work better than grid search for defect prediction. According to the argument above, DE works well for tuning since: (a) DE tends to favor the small number of intrinsic dimensions relevant to tuning; (b) The space of tunings for defect predictors is inherently low dimensional.

In defense of the first point, recall that Equation 4.2 says that DE repeatedly compares an existing tuning \( A \) against another candidate \( E \) that is constructed by taking a small step between three other candidates \( B, C, D \). DE runs over a list of old candidates, \( n \) times. For \( n > 1 \), the invariant is that members of that list are not inferior to at least one other example. If a new candidate \( E \) is created
As to the second point about the low dimensional nature of tuning defect predictors, we first assume that *the dimensionality of the tuning problem is linked to the dimensionality of the data explored by the learners*. Our argument for this assumption is (1) learners like CART and Random Forests divide the data into regions with similar properties (specifically, those with and without defects); (2) when we tune those learners, we are constraining how they make those divisions over that data.

**Figure 4.9** Illustration for data D=dimensionality

Given that assumption, exploring the space of tunings for defect predictors really means *exploring the dimensionality of defect prediction data*. Two studies strongly suggest that this data...
is inherently low-dimensionality. Papakroni [Pap13] combined instance selection and attributes pruning for defect prediction. Using some information theory, he was able to prune 75% of the attributes of Table 1.1 as uninformative. He then clustered the remaining data, replacing each cluster with one centroid. This two-phase pruning procedure generated small data sets with, e.g., 24 columns (attributes) and 800 rows (instances) to a table of 6 columns and 20 rows. To test the efficacy of that reduced space, Papakroni built defect predictors by extrapolating between the two nearest centroids in the reduced space, for each test case. Papakroni found that those estimates from that small space worked just as well as those generated by state-of-the-art learners (Random Forests and Naive Bayes) using all the data [Pap13]. That is, according to Papakroni, the signal in these data sets can be found in a handful of attributes and a few dozen instances.

In order to formalize the findings of the Papakroni study, Provence [Pro15] explored the intrinsic dimensionality of defect data sets. Intrinsic dimensionality measures the number of dimensions \( m \) used by data within an \( n \) dimensional space. For example, the “B” data shown in Figure 4.9 spreads over a \( n = 2 \) dimensional space. but the “A” data does not use all the available dimensions. Hence, the intrinsic dimensionality of the “A” data is \( m = 1 \).

Like Provence, we use correlation dimension to calculate the intrinsic dimensionality of the datasets. Euclidean distance is used to compute the distance between the independent decisions \( d \) within each candidate solution; all \( d_i \) values are normalized by \( \max - \min \). Next, we use the distance measure as part of the correlation dimension defined by by Grassberger and Procaccia [GP83]. This correlation dimension of a data set with \( k \) items is found by plotting the number of items found at distance within radius \( r \) from any other item against \( r \) (where \( r \) is actually a distance, as defined in the last paragraph). Then we normalize this by the number of connections between \( k \) items to find the expected number of neighbors at distance \( r \) is \( C(r) = \frac{2}{k(k-1)} \sum_{i=1}^{k} \sum_{j=i+1}^{k} 1\{\|x_i, x_j\| < r\} \).

Given a dataset with \( k \) items and min, max distance of \( r_{\min} \) and \( r_{\max} \), we estimate the intrinsic dimensionality as the mean value of the slope of \( \ln(C(r)) \) vs \( \ln(r) \) by evaluating \( C(r) \) for \( r \) in \( \{r_0, ..., r_n\} \), such that \( \{r_0, ..., r_n\} \) is sufficient for a good estimation of slope, and \( r_n \ll r_{\max} \).

Figure 4.8 shows the intrinsic dimensions for the data sets used in this study. Note the low intrinsic dimensionality (median value, shown as the dashed line, nearly 1.2). By way of comparison, the intrinsic dimensions reported by other researchers in their data sets (not from SE) is often much larger; e.g. 5 to 10, or more; see [LB04].

From all this, Provence concluded that the effects reported by Papakroni were due to the underlying low dimensionality of the data. Extending his result, we conjecture that our conclusion (that DEs do better than grid search for tuning data miners) are externally valid when the data miners are exploring data with low intrinsic dimensionality.
4.6 Threats to Validity

Threats to **construct validity** concern the extent to which the observed phenomena correspond to what is intended to be observed. Some of evaluation measures (*precision, F-Measure, AUC*) used in this study to access defect predictors are widely used in defect prediction literature. In addition, we use $AUC_{20}$ to characterize how the models could predicts defects when taking account of efforts required to inspect the predicted files. We find that grid search works quite bad under this evaluation measure. Another threat is that the data set used in paper is from SEACRAFT (previously as PROMISE), which might include noise and have errors. To mitigate such threats, we will compare DE with grid search to tune defect predictors on other defect prediction data in the future work.

Threats to **conclusion validity** concern the relationship between the treatment and the results. In addition to compare the median values of 20 repeats of DE and grid search for tuning defect predictors, we also use Scott Knott (non-parametric test) to determine if the difference in the performance of 20 repeats DE and grid search is significant.

Threats to **internal validity** concern the consistency of the results obtained from the result. In our study, to investigate why differential evolution performs better than grid search for parameter tuning, we select defect prediction, which is a well explored research problem in software engineering filed as a case study. To mitigate the threats to internal validity, we carefully examine the data sets used in this study and all the data sets used in this study are also publicly available [FM17a] for other researchers to repeat, improve and refute our results. Another threats to the internal validity might be the raw runtime cost for DE and grid search. Different implementations of data loading and processing methods would have different runtime cost. However, the data loading and processing methods implemented in this study are used by all models(e.g., DE and grid search). Therefore, the relative runtime cost comparison between DE and grid search still hold.

Threats to **external validity** represent if the results are of relevance for other cases, or the ability to generalize the observations in a study. In this study, we use ten widely used open source JAVA software project data from SEACRAFT as the subject. As only the metrics listed in Table 4.1 are used as the attributes to build defect predictors, we can not guarantee that our findings can be directly generalized to other projects that using different metrics, like code change metrics [Kam13]. Meanwhile, we can not guarantee that our conclusion could be generalized to defect prediction on other projects. Therefore, the future work might include verify our findings on other software project with different metrics. Also, we only take defect prediction as a case study to compare the performance of differential evolution and grid search as parameter tuner, we can not guarantee that our conclusions can be generated to other software analytics. However, those other software analytics tasks often apply machine learning methods [Koc12b] and so it is quite possible that the conclusion (DE is better than grid search as a parameter optimizer) would be widely applicable, elsewhere.
4.7 Conclusion

When the parameters of data miners are tuned to the local data, the performance of the resulting learners can be greatly increased. Such tuning are very computationally expensive, especially when done with grid search. For some software engineering tasks, it is possible to avoid those very long runtime. When tuning defect predictors learned from static code attributes, a simple evolutionary strategy (differential evolution) runs one to two orders of magnitude faster that grid search. Further the tunings found in this way work as well, or better, than those found via grid search. We explain this result using the (1) Bergstra and Benigo argument that random search works best in low-dimensional data sets and (2) the empirical results of Papakroni and Provence that defect data sets are very low dimensional in nature.

As to testing the external validity of this study’s argument, the next steps are clear:

- Sort data sets by how well a simple evolutionary algorithm like DE can improve the performance of data miners executing on that data;
- Explore the difference in the worst and best end of that sort.
- If the intrinsic dimensionalities are very different at both ends of this sort, then that would support with the claims of this study.
- Else, this study’s claims would not be supported and we would need to seek other difference between the best and worst data sets.

Note that we offer these four items as future work, rather than reported results, since so far all the defect data sets we have tried had responded best to DE. For a time, we did consider trying this with artificially generated data (where we control how many dimensions were contained in the data). However, prior experience with using artificial data sets [MC00] suggested to us that such arguments are not widely convincing since the issue is not how often an effect holds in artificial data, but how often it holds in real data. Hence, in future work, we will look further afield for radically different (real world) data sets.
This chapter first appeared at 2017 11th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2017), pp. 49-60. ACM. titled “Easy over Hard: A Case Study on Deep Learning”. In this chapter, we show that simpler methods can reach parity with complex methods with much lower execution costs and less understanding barriers.

Recently, more complex methods have been proposed to improve the software analytics, like deep learning methods. While deep learning is an exciting new technique, the benefits of this method need to be assessed with respect to its computational cost. Especially, our goal is to design simple and effective software analytics models. Since these deep learning methods need hours (to weeks) to train the model, that limits the ability of (a) a researcher to test the stability of their conclusion via repeated runs with different random seeds; and (b) other researchers to repeat, improve, or even refute that original work. Therefore, it’s hard for researchers to compare simple techniques with such incredible complex methods. In this chapter, we repeat and extend a prior study by Xu et al [Xu16], which was solved by deep learning methods before. After we spent weeks of time to collect, clean, and pre-process data, we show here that applying a very simple optimizer called DE to fine tune SVM, it can achieve similar (and sometimes better) results. The DE approach terminated in 10 minutes; i.e. 84 times faster hours than deep learning method.

We offer these results as a cautionary tale to the software analytics community and suggest that not every new innovation should be applied without critical analysis. If researchers deploy
some new and expensive process, that work should be baselined against some simpler and faster alternatives.

5.1 Introduction

This study extends a prior result from ASE’16 by Xu et al. [Xu16] (hereafter, XU). XU described a method to explore large programmer discussion forums, then uncover related, but separate, entries. This is an important problem. Modern SE is evolving so fast that these forums contain more relevant and recent comments on current technologies than any textbook or research article.

In their work, XU predicted whether two questions posted on Stack Overflow are semantically linkable. Specifically, XU define a question along with its entire set of answers posted on Stack Overflow as a knowledge unit (KU). If two knowledge units are semantically related, they are considered as linkable knowledge units.

In their paper, they used a convolution neural network (CNN), a kind of deep learning method [LeC15], to predict whether two KUs are linkable. Such CNNs are highly computationally expensive, often requiring network composed of 10 to 20 layers, hundreds of millions of weights and billions of connections between units [LeC15]. Even with advanced hardware and algorithm parallelization, training deep learning models still requires hours to weeks. For example:

- XU report that their analysis required 14 hours of CPU.
- Le [Le13] used a cluster with 1,000 machines (16,000 cores) for three days to train a deep learner.

This study debates what methods should be recommended to those wishing to repeat the analysis of XU. We focus on whether using simple and faster methods can achieve the results that are currently achievable by the state-of-art deep learning method. Specifically, we repeat XU’s study using DE (differential evolution [SP97]), which serves as a hyper-parameter optimizer to tune XU’s baseline method, which is a conventional machine learning algorithm, support vector machine (SVM). Our study asks:

**RQ1**: Can we reproduce XU’s baseline results (Word Embedding + SVM)? Using such a baseline, we can compare our methods to those of XU.

**RQ2**: Can DE tune a standard learner such that it outperforms XU’s deep learning method? We apply differential evolution to tune SVM. In terms of precision, recall and F1-score, we observe that the tuned SVM method outperforms CNN in most evaluation scores.

**RQ3**: Is tuning SVM with DE faster than XU’s deep learning method? Our DE method is 84 times faster than CNN.

We offer these results as a cautionary tale to the software analytics community. While deep learning is an exciting new technique, the benefits of this method need to be carefully assessed.
with respect to its computational cost. More generally, if researchers deploy some new and expensive process (like deep learning), that work should be baselined against some simpler and faster alternatives.

The rest of this chapter is organized as follows. Section 5.2 describes the background and related work on deep learning and parameter tuning in SE. Section 5.3 explains the case study problem and the proposed tuning method investigated in this study, then Section 5.4 describes the experimental settings of our study, including research questions, data sets, evaluation measures and experimental design. Section 5.5 presents the results. Section 5.6 discusses implications from the results and the threats to the validity of our study. Section 5.7 concludes the paper and discusses the future work.

Before beginning, we digress to make two points. Firstly, just because “DE + SVM” beats deep learning in this application, this does not mean DE is always the superior method for all other software analytics applications. No learner works best over all problems [Wol96]—the trick is to try several approaches and select the one that works best on the local data. Given the low computational cost of DE (10 minutes vs 14 hours), DEs are an obvious and low-cost candidate for exploring such alternatives. Secondly, to enable other researchers to repeat, improve, or refute our results, all our scripts and data are freely available on-line Github¹.

5.2 Background and Related Work

5.2.1 Why Explore Faster Software Analytics?

This section argues that avoiding slow methods for software analytics is an open and urgent issue. 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 [Cze11b], 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; The15b].

But the more complex the method, the harder it is to apply the analysis. Fisher et al. [Fis12] characterize 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

¹https://github.com/WeiFoo/EasyOverHard
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; Wan13b].
- 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 [Wan13b]. Much of that CPU time can be saved if there is a faster way.

### 5.2.2 What is Deep Learning?

Deep learning is a branch of machine learning built on multiple layers of neural networks that attempt to model high level abstractions in data. According to LeCun et al. [LeC15], deep learning methods are representation-learning methods with multiple levels of representation, obtained by composing simple but non-linear modules that each transforms the representation at one level (starting with the raw input) into a representation at a higher, slightly more abstract level. Compared to the conventional machine learning algorithms, deep learning methods are very good at exploring high-dimensional data.

By utilizing extensive computational power, deep learning has been proven to be a very powerful method by researchers in many fields [LeC15], like computer vision and natural language
processing [Kri12; Mik13; Sut14; Sch15; Are10]. In 2012, Convolution neural networks method won the ImageNet competition [Kri12], which achieves half of the error rates of the best competing approaches. After that, CNN became the dominant approach for almost all recognition and detection tasks in computer vision community. CNNs are designed to process the data in the form of multiple arrays, e.g., image data. According to LeCun et al. [LeC15], recent CNN methods are usually a huge network composed of 10 to 20 layers, hundreds of millions of weights and billions of connections between units. With advanced hardware and algorithm parallelization, training such model still need a few hours [LeC15]. For the tasks that deal with sequential data, like text and speech, recurrent neural networks (RNNs) have been shown to work well. RNNs are found to be good at predicting the next character or word given the context. For example, Graves et al. [Gra13] proposed to use long short-term memory (LSTM) RNNs to perform speech recognition, which achieves a test set error of 17.7% on the benchmark testing data. Sutskever et al. [Sut14] used two multiplelayered LSTM RNNs to translate sentences in English to French.

5.2.3 Deep Learning in SE

We study deep learning since, recently, it has attracted much attentions from researchers and practitioners in software community [Wan16; Gu16; Xu16; Whi16; Whi15; Lam15; Cho16; Yua14; Mou16]. These researchers applied deep learning techniques to solve various problems, including defect prediction, bug localization, clone code detection, malware detection, API recommendation, effort estimation and linkable knowledge prediction.

We find that this work can be divided into two categories:

- Treat deep learning as a feature extractor, and then apply other machine learning algorithms to do further work [Lam15; Wan16; Cho16].

- Solve problems directly with deep learning [Gu16; Xu16; Whi16; Whi15; Yua14; Mou16].

5.2.3.1 Deep Learning as Pre-Processor

Lam et al. [Lam15] proposed an approach to apply deep neural network in combination with rVSM to automatically locate the potential buggy files for a given bug report. By comparing it to baseline methods (Naive Bayes [Kim13], learn-to-rank [Ye14], BugLocator [Zho12]), Lam et al. reported, 16.2-46.4%, 8-20.8% and 2.7-20.7% higher top-1 accuracy than baseline methods, respectively [Lam15]. The training time for deep neural network was reported from 70 to 122 minutes for 6 projects on a computer with 32 cores 2.00GHz CPU, 126 GB memory. However, the runtime information of the baseline methods was not reported.

Wang et al. [Wan16] applied deep belief network to automatically learn semantic features from token vectors extracted from the studied software program. After applying deep belief network to
generate features from software code, Naive Bayes, ADTree and Logistic Regression methods are used to evaluate the effectiveness of feature generation, which is compared to the same learners using traditional static code features (e.g. McCabe metrics, Halstead's effort metrics and CK object-oriented code metrics [KR87; CK94; McC76; Hal77]). In terms of runtime, Wang et al. only report time for generating semantics features with deep belief network, which ranged from 8 seconds to 32 seconds [Wan16]. However, the time for training and tuning deep belief network is missing. Furthermore, to compare the effectiveness of deep belief network for generating features with methods that extract traditional static code features in terms of time cost, it would be favorable to include all the time spent on feature extraction, including paring source code, token generation and token mapping for both deep belief network and traditional methods (i.e., an end-to-end comparison).

Choetkiertikul et al. [Cho16] proposed to apply deep learning techniques to solve effort estimation problems on user story level. Specifically, Choetkiertikul et al. [Cho16] proposed to leverage long short-term memory (LSTM) to learn feature vectors from the title, description and comments associated with an issue report and after that, regular machine learning techniques, like CART, Random Forests, Linear Regression and Case-Based Reasoning are applied to build the effort estimation models. Experimental results show that LSTM has a significant improvement over the baseline method bag-of-words. However, no further information regarding runtime as well as experimental hardware is reported for both methods and there is no cost of this deep learning method at all.

5.2.3.2 Deep Learning as a Problem Solver

White et al. [Whi15; Whi16] applied recurrent neural networks, a type of deep learning techniques, to address code clone detection and code suggestion. They reported, the average training time for 8 projects were ranging from 34 seconds to 2977 seconds for each epoch on a computer with two 3.3 GHz CPUs and each project required at least 30 epochs [Whi16]. Specifically, for the JDK project in their experiment, it would take 25 hours on the same computer to train the models before getting prediction. For the time cost for code suggestions, authors did not mention any related information [Whi15].

Gu et al. [Gu16] proposed a recurrent neural network (RNN) based method, DEEPAPI, to generate API usage sequences for a given natural language query. Compared with the baseline method SWIM [Rag16] and Lucene + UP-Miner [Wan13a], DEEPAPI improved the performance significantly. However, that improvement came at a cost: that model was trained with a Nivdia K20 GPU for 240 hours [Gu16].

XU [Xu16] utilized neural language model and convolution neural network (CNN) to learn word-level and document-level features to predict semantically linkable knowledge units on Stack Overflow. In terms of performance metrics, like precision, recall and F1-score, CNN method was
evaluated much better than the baseline method support vector machine (SVM). However, once again, that performance improvement came at a cost: their deep learner required 14 hours to train CNN model on a 2.5GHz PC with 16 GB RAM [Xu16].

Yuan et al. [Yua14] proposed a deep belief network based method for malware detection on Android apps. By training and testing the deep learning model with 200 features extracted from static analysis and dynamic analysis from 500 sampled Android app, they got 96.5% accuracy for deep learning method and 80% for one baseline method, SVM [Yua14]. However, they did not report any runtime comparison between the deep learning method and other classic machine learning methods.

Mou et al. [Mou16] proposed a tree-based convolutional neural network for programming language processing, in which a convolution kernel is designed over programs’ abstract syntax trees to capture structural information. Results show that their method achieved 94% accuracy, which is better than the baseline method RBF SVM 88.2% on program classification problem [Mou16]. However, Mou et al. [Mou16] did not discuss any runtime comparison between the proposed method and baseline methods.

5.2.3.3 Issues with Deep Learning

In summary, deep learning is used extensively in software engineering community. A common pattern in that research is to:

- Report deep learning’s benefits, but not its CPU/GPU cost [Whi15; Cho16; Yua14; Mou16];
- Or simply show the cost, without further analysis [Wan16; Lam15; Gu16; Xu16; Whi16].

Since deep learning techniques cost large amount of time and computational resources to train its model, one might question whether the improvements from deep learning is worth the costs. Are there any simple techniques that achieve similar improvements with less resource costs? To investigate how simple methods could improve baseline methods, we select XU [Xu16] study as a case study. The reasons are as follows:

- Most deep learning paper’s baseline methods in SE are either not publicly available or too complex to implement [Whi16; Lam15]. XU define their baseline methods precisely enough so others can confidently reproduce it locally. XU’s baseline method is SVM learner, which is available in many machine learning toolboxes.
- Further, it is not yet common practice for deep learning researchers in SE community to share their implementations and data [Whi16; Whi15; Lam15; Wan16; Cho16; Gu16], where a tiny difference may lead to a huge difference in the results. Even though XU do not share their CNN tool, their training and testing data are available online, which can be used for our proposed
method. Since the same training and testing data are used for XU’s CNN and our proposed method, we can compare results of our method to their CNN results.

- Some studies do not report their runtime and experimental environment, which makes it harder for us to systematically compare our results with theirs in terms of computational costs [Cho16; Yua14; Whi15; Mou16]. XU clearly report their experimental hardware and runtime, which will be easier for us compare our computational costs to theirs.

5.2.4 Parameter Tuning in SE

In this study, we use DE as an optimizer to do parameter tuning for SVM, which achieves results that are competitive with deep learning. This section discusses related work on parameter tuning in SE community.

Machine learning algorithms are designed to explore the instances to learn the bias. However, most of these algorithms are controlled by parameters such as:

- The maximum allowed depth of decision tree built by CART;
- The number of trees to be built within a Random Forest.

Adjusting these parameters is called hyperparameter optimization. It is a well explored approach in other communities [BB12; Li16]. However, in SE, such parameter optimization is not a common task (as shown in the following examples).

In the field of defect prediction, Fu et al. [Fu16a] surveyed hundreds of highly cited software engineering paper about defect prediction. Their observation is that most software engineering researchers did not acknowledge the impact of tunings (exceptions: [Les08; Tan16]) and use the “off-the-shelf” data miners. For example, Elish et al. [EE08] compared support vector machines to other data miners for the purposes of defect prediction. However, the Elish et al. paper makes no mention of any SVM tuning study [EE08]. More details about their survey refer to [Fu16a].

In the field of topic modeling, Agrawal et al. [Agr18] investigated the impact of parameter tuning on Latent Dirichlet Allocation (LDA). LDA is a widely used technique in software engineering field to find related topics within unstructured text, like topic analytics on Stack Overflow [Bar14] and source code analysis [Bin14]. Agrawal et al. found that LDA suffers from conclusion instability (different input orderings can lead to very different results) that is a result of poor choice of the LDA control parameters [Agr18]. Yet, in their survey of LDA use in SE, they found that very few researchers (4 out of 57 papers) explored the benefits of parameter tuning for LDA.

One troubling trend is that, in the few SE papers that perform tuning, they do so using methods heavily deprecated in the machine learning community. For example, two SE papers that use tuning [Les08; Tan16], apply a simple grid search to explore the potential parameter space for
optimal tunings (such grid searchers run one for-loop for each parameter being optimized). However, Bergstra et al. [BB12] and Fu et al. [Fu16b] argue that random search methods (e.g. the differential evolution algorithm used here) are better than grid search in terms of efficiency and performance.

5.3 Method

5.3.1 Research Problem

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Default</th>
<th>Xue et al.</th>
<th>Tuning Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>1.0</td>
<td>unknown</td>
<td>[1, 50]</td>
<td>Penalty parameter C of the error term.</td>
</tr>
<tr>
<td>kernel</td>
<td>'rbf'</td>
<td>'rbf'</td>
<td>['linear', 'poly', 'rbf', 'sigmoid']</td>
<td>Specify the kernel type to be used in the algorithms.</td>
</tr>
<tr>
<td>gamma</td>
<td>1/\sqrt{n}_{features}</td>
<td>1/200</td>
<td>[0, 1]</td>
<td>Kernel coefficient for 'rbf', 'poly' and 'sigmoid'.</td>
</tr>
<tr>
<td>coef0</td>
<td>0</td>
<td>unknown</td>
<td>[0, 1]</td>
<td>Independent term in kernel function. It is only used in 'poly' and 'sigmoid'.</td>
</tr>
</tbody>
</table>

This section is an overview of the the task and methods used by XU. Their task was to predict relationships between two knowledge units (questions with answers) on Stack Overflow. Specifically, XU divided linkable knowledge unit pairs into 4 difference categories namely, duplicate, direct link, indirect link and isolated, based on its relatedness. The definition of these four categories are shown in Table 5.1 [Xu16]:

In that paper, XU provided the following two methods as baselines [Xu16]:

- TF-IDF + SVM: a multi-class SVM classifier with 36 textual features generated based on the TF and IDF values of the words in a pair of knowledge units.
- Word Embedding + SVM: a multi-class SVM classifier with word embedding generated by the word2vec model [Mik13].

Both of these two baseline methods are compared against their proposed method, Word Embedding
+ CNN. In this study, we select Word Embedding + SVM as the baseline because it uses word embedding as the input, which is the same as the Word Embedding + CNN method by XU.

5.3.2 Learners and Their Parameters

SVM has been proven to be a very successful method to solve text classification problem. A SVM seeks to minimize misclassification errors by selecting a boundary or hyperplane that leaves the maximum margin between positive and negative classes (where the margin is defined as the sum of the distances of the hyperplane from the closest point of the two classes [Joa98]).

Like most machine learning algorithms, there are some parameters associated with SVM to control how it learns. In XU’s experiment, they used a radial-bias function (RBF) for their SVM kernel and set \( \gamma \) to \( 1/k \), where \( k \) is 36 for TF-IDF + SVM method and 200 for Word Embedding + SVM method. For other parameters, XU mentioned that grid search was applied to optimize the SVM parameters, but no further information was disclosed.

For our work, we used the SVM module from Scikit-learn [Ped11], a Python package for machine learning, where the parameters shown in Table 5.2 are selected for tuning. Parameter \( C \) is to set the amount of regularization, which controls the tradeoff between the errors on training data and the model complexity. A small value for \( C \) will generate a simple model with more training errors, while a large value will lead to a complicated model with fewer errors. \( Kernel \) is to introduce different nonlinearities into the SVM model by applying kernel functions on the input data. \( Gamma \) defines how far the influence of a single training example reaches, with low values meaning ‘far’ and high values meaning ‘close’. \( coef0 \) is an independent parameter used in sigmod and polynomial kernel function.

As to why we used the “Tuning Range” shown in Table 5.2, and not some other ranges, we note that (a) those ranges include the defaults and also XU’s values; (b) the results presented below show that by exploring those ranges, we achieved large gains in the performance of our baseline method. This is not to say that larger tuning ranges might not result in greater improvements. However, for the goals of this study (to show that tuning baseline method does matter), exploring just these ranges shown in Table 5.2 will suffice.

5.3.3 Learning Word Embedding

Learning word embeddings refers to find vector representations of words such that the similarities between words can be captured by cosine similarity of corresponding vector representations. It is been shown that the words with similar semantic and syntactic are found closed to each other in the embedding space [Mik13].

Several methods have been proposed to generate word embeddings, like skip-gram [Mik13], GloVe [Pen14] and PCA on the word co-occurrence matrix [LC13]. To replicate XU work, we used
the continuous skip-gram model (word2vec), which is an unsupervised word representation learning method based on neural networks and also used by Xu [Xu16]. The skip-gram model learns vector representations of words by predicting the surrounding words in a context window. Given a sentence of words \( W = w_1, w_2, \ldots, w_n \), the objective of skip-gram model is to maximize the average log probability of the surrounding words:

\[
\frac{1}{n} \sum_{i=1}^{n} \sum_{-c \leq j \leq c, j \neq 0} \log p(w_{i+j} | w_i)
\]

where \( c \) is the context window size and \( w_{i+j} \) and \( w_i \) represent surrounding words and center word, respectively. The probability of \( p(w_{i+j} | w_i) \) is computed according to the softmax function:

\[
p(w_O | w_I) = \frac{\exp(v_{w_O}^T v_{w_I})}{\sum_{w=1}^{||W||} \exp(v_{w}^T v_{w_I})}
\]

where \( v_{w_O} \) and \( v_{w_I} \) are the vector representations of the input and output vectors of \( w \), respectively. \( \sum_{w=1}^{||W||} \exp(v_{w}^T v_{w_I}) \) normalizes the inner product results across all the words. To improve the computation efficiency, Mikolove et al. [Mik13] proposed hierarchical softmax and negative sampling techniques. More details can be found in Mikolove et al.’s study [Mik13].

Skip-gram’s parameters control how that algorithm learns word embeddings. Those parameters include window size and dimensionality of embedding space, etc. Zuccon et al. [Zuc15] found that embedding dimensionality and context window size have no consistent impact on retrieval model performance. However, Yang et al. [Yan16a] showed that large context window and dimension sizes are preferable to improve the performance when using CNN to solve classification tasks for Twitter. Since this work is to compare performance of tuning SVM with CNN, where skip-gram model is used to generate word vector representations for both of these methods, tuning parameter of skip-gram model is beyond the scope of this study (but we will explore it in future work).

To train our word2vec model, 100,000 knowledge units tagged with “java” from Stack Overflow posts table (include titles, questions and answers) are randomly selected as a word corpus\(^2\). After applying proper data processing techniques proposed by Xu, like remove the unnecessary HTML tags and keep short code snippets in code tag, then fit the corpus into gensim word2vec module [RS10], which is a python wrapper over original word2vec package.

When converting knowledge units into vector representations, for each word \( w_i \) in the post processed knowledge unit (including title, question and answers), we query the trained word2vec model to get the corresponding word vector representation \( v_i \). Then the whole knowledge unit with \( s \) words is converted to vector representation by element-wise addition, \( U v = v_1 \oplus v_2 \oplus \ldots \oplus v_s \). This

\(^2\)Without further explanation, all the experiment settings, including learner algorithms, training/testing data split, etc, strictly follow Xu’s work.
vector representation is used as the input data to SVM.

### 5.3.4 Tuning Algorithm

A tuning algorithm is an optimizer that drives the learner to explore the optimal parameter in a given searching space. According to our literature review, there are several searching algorithms used in SE community: simulated annealing [FM02; Men07b]; various genetic algorithms [Jon96; Har07; AF11] augmented by techniques such as differential evolution [SP97; Fu16a; Fu16b; CGPT15; Agr18], tabu search and scatter search [Bea06b; Mol07; Cor13]; particle swarm optimization [Win07]; numerous decomposition approaches that use heuristics to decompose the total space into small problems, then apply a response surface methods [Kra15b]; NSGA-II [Zha07] and NSGA-III [Mka14].

![Figure 5.1 Procedure TUNER: strives to find “good” tunings which maximizes the objective score of the model on training and tuning data. TUNER is based on Storn's differential evolution optimizer [SP97].](image)

1. Given a model (e.g., SVM) with \( n \) decisions (e.g., \( n = 4 \)), TUNER calls SAMPLE \( N = 10 \times n \) times. Each call generates one member of the population \( pop_{\in \mathbb{N}} \).
2. TUNER scores each \( pop_i \) according to various objective scores \( o \). In the case of our tuning SVM, the objective \( o \) is to maximize F1-score.
3. TUNER tries to each replace \( pop_i \) with a mutant \( m \) built using Storn’s differential evolution method [SP97]. DE extrapolates between three other members of population \( a, b, c \). At probability \( p_1 \), for each decision \( a_k \in a \), then \( m_k = a_k \lor (p_1 < \text{rand}() \land (b_k \lor c_k)) \).
4. Each mutant \( m \) is assessed by calling \( \text{EVALUATE}(\text{model, prior}=m) \); i.e. by seeing what can be achieved within a goal after first assuming that \( \text{prior} = m \).
5. To test if the mutant \( m \) is preferred to \( pop_i \), TUNER simply compare \( \text{SCORE}(m) \) with \( \text{SCORE}(pop_i) \).
6. TUNER repeatedly loops over the population, trying to replace items with mutants, until new better mutants stop being found.
7. Return the best one in the population as the optimal tunings.

Of all the mentioned algorithms, the simplest are simulated annealing (SA) and differential evolution (DE), each of which can be coded in less than a page of some high-level scripting language. Our reading of the current literature is that there are more advocates for differential evolution than SA. For example, Vesterstrom and Thomsen [VT04] found DE to be competitive with particle swarm optimization and other GAs. DEs have already been applied before for parameter tuning in SE community to do parameter tuning (e.g. see [Omr05; Chi12; Fu16a; Fu16b; Agr18]). Therefore, in this work, we adopt DE as our tuning algorithm and the main steps in DE is described in Figure 5.1.
5.4 Experimental Setup

5.4.1 Research Questions

To systematically investigate whether tuning can improve the performance of baseline methods compared with deep learning method, we set the following three research questions:

- **RQ1**: Can we reproduce XU’s baseline results (Word Embedding + SVM)?
- **RQ2**: Can DE tune a standard learner such that it outperforms XU’s deep learning method?
- **RQ3**: Is tuning SVM with DE faster than XU’s deep learning method?

RQ1 is to investigate whether our implementation of Word Embedding + SVM method has the similar performance with XU’s baseline, which makes sure that our following analysis can be generalized to XU’s conclusion. RQ2 and RQ3 lead us to investigate whether tuning SVM comparable with XU’s deep learning from both performance and cost aspects.

5.4.2 Dataset and Experimental Design

Our experimental data comes from Stack Overflow data dump of September 2016\(^3\), where the *posts* table includes all the questions and answers posted on Stack Overflow up to date and the *postlinks* table describes the relationships between posts, e.g., *duplicate* and *linked*. As mentioned in Section 5.3.1, we have four different types of relationships in knowledge unit pairs. Therefore, *linked* type is further divided into *indirectly linked* and *directly linked*. Overall, four different types of data are generated according the following rules [Xu16]:

- Randomly select a pair of posts from the *postlinks* table, if the value in *PostLinkTypeId* field for this pair of posts is 3, then this pair of posts is *duplicate* posts. Otherwise they’re *directly linked* posts.

- Randomly select a pair of posts from the *posts* table, if this pair of posts is linkable from each other according to *postlinks* table and the distance between them are greater than 2 (which means they are not duplicate or directly linked posts), then this pair of posts is indirectly linked. If they’re not linkable, then this pair of posts is isolated.

In this work, we use the same training and testing knowledge unit pairs as XU [Xu16]\(^4\), where 6,400 pairs of knowledge units for training and 1,600 pairs for testing. And each type of linked knowledge units accounts for 1/4 in both training and testing data. The reasons that we used the same training and testing data as XU are:

\(^3\)https://archive.org/details/stackexchange

\(^4\)https://github.com/XBWer/ASEDataset
• It is to ensure that performance of our baseline method is as closed to XU’s as possible.

• Since deep learning method is way complicated compared to SVM and a little difference in implementations might lead to different results. To fairly compare with XU’s result, we can use the performance scores of CNN method from XU’s study [Xu16] without any implementation bias introduced.

For training word2vec model, we randomly select 100,000 knowledge units (title, question body and all the answers) from posts table that are related to “java”. After that, all the training/tuning/testing knowledge units used in this study are converted into word embedding representations by looking up each word in word2vec model as described in Section 5.3.3.

As seen in Figure 5.2, instead of using all the 6,400 knowledge units as training data, we split the original training data into new training data and tuning data, which are used during parameter tuning procedure for training SVM and evaluating candidate parameters offered by DE. Afterwards, the new training data is again fitted into the SVM with the optimal parameters found by DE and finally the performance of the tuned SVM will be evaluated on the testing data.

To reduce the potential variance caused by how the original training data is divided, 10-fold cross-validation is performed. Specifically, each time one fold with 640 knowledge units pairs is used as the tuning data, and the remaining folds with 5760 knowledge units are used as the new training data, then the output SVM model will be evaluated on the testing data. Therefore, all the
performance scores reported below are averaged values over 10 runs.

In this study, we use Wilcoxon single ranked test to statistically compare the differences between tuned SVM and untuned SVM. Specifically, the Benjamini-Hochberg (BH) adjusted p-value is used to test whether a difference is statistically significant at the level of 0.05 [BH95]. To measure the effect size of performance scores between tuned SVM and untuned SVM, we compute Cliff’s $\delta$ that is a non-parametric effect size measure [Rom06]. As Romano et al. suggested, we evaluate the magnitude of the effect size as follows: negligible ($|\delta| < 0.147$), small ($0.147 < |\delta| < 0.33$), medium ($0.33 < |\delta| < 0.474$), and large ($0.474 \leq |\delta|$) [Rom06].

5.4.3 Evaluation Metrics

When evaluating the performance of tuning SVM on the multi-class linkable knowledge units prediction problem, consistent with XU [Xu16], we use accuracy, precision, recall and F1-score as the evaluation metrics.

Table 5.3 Confusion matrix.

<table>
<thead>
<tr>
<th>Actual</th>
<th>Classified as</th>
<th>$C_1$</th>
<th>$C_2$</th>
<th>$C_3$</th>
<th>$C_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C_1$</td>
<td>$c_{11}$</td>
<td>$c_{12}$</td>
<td>$c_{13}$</td>
<td>$c_{14}$</td>
<td></td>
</tr>
<tr>
<td>$C_2$</td>
<td>$c_{21}$</td>
<td>$c_{22}$</td>
<td>$c_{23}$</td>
<td>$c_{24}$</td>
<td></td>
</tr>
<tr>
<td>$C_3$</td>
<td>$c_{31}$</td>
<td>$c_{32}$</td>
<td>$c_{33}$</td>
<td>$c_{34}$</td>
<td></td>
</tr>
<tr>
<td>$C_4$</td>
<td>$c_{41}$</td>
<td>$c_{42}$</td>
<td>$c_{43}$</td>
<td>$c_{44}$</td>
<td></td>
</tr>
</tbody>
</table>

Given a multi-classification problem with true labels $C_1$, $C_2$, $C_3$ and $C_4$, we can generate a confusion matrix like Table 5.3, where the value of $c_{ii}$ represents the number of instances that are correctly classified by the learner for class $C_i$.

**Accuracy** is defined as the number of correctly classified knowledge units over the total number of knowledge units, i.e.,

$$\text{accuracy} = \frac{\sum_i c_{ii}}{\sum_j \sum_i c_{ij}}$$

where $\sum_j \sum_i c_{ij}$ is the total number of knowledge units. For a given type of knowledge units, $C_j$, the precision is defined as probability of knowledge units pairs correctly classified as $C_j$ over the number of knowledge unit pairs classified as $C_j$ and recall is defined as the percentage of all $C_j$ knowledge unit pairs correctly classified. F1-score is the harmonic mean of recall and precision. Mathematically, precision, recall and F1-score of the learner for class $C_j$ can be denoted as follows:
\[ \text{prec}_j = \text{precision}_j = \frac{c_{jj}}{\sum_i c_{ij}} \]

\[ \text{pd}_j = \text{recall}_j = \frac{c_{jj}}{\sum_j c_{ij}} \]

\[ F1_j = 2 * \text{pd}_j * \text{prec}_j / (\text{pd}_j + \text{prec}_j) \]

Where \( \sum_i c_{ij} \) is the predicted number of knowledge units in class \( C_j \) and \( \sum_j c_{ji} \) is the actual number of knowledge units in class \( C_j \).

Recall from Algorithm 1 that we call differential evolution once for each optimization goal. Generally, this goal depends on which metric is most important for the business case. In this work, we use \( F1 \) to score the candidate parameters because it controls the trade-off between precision and recall, which is also consistent with XU [Xu16] and is also widely used in software engineering community to evaluate classification results [Wan16; Men07b; Fu16a; Kim08].

5.5 Results

In this section, we present our experimental results. To answer research questions raised in Section 5.4.1, we conducted two experiments:

- Compare performance of Word Embedding + SVM method in XU [Xu16] and our implementation;

- Compare performance of our tuning SVM with DE method with XU’s CNN deep learning method.

Since we used the same training and testing data sets provided by XU [Xu16] and conducted our experiment in the same procedure and evaluated methods using the performance measures, we simply used the results reported in the work by XU [Xu16] for the performance comparison.

RQ1: Can we reproduce XU’s baseline results (Word Embedding + SVM)?

This first question is important to our work since, without the original tool released by XU, we need to insure that our reimplementation of their baseline method (WordEmbedding + SVM) has a similar performance to their work. Accordingly, we carefully follow XU’s procedure [Xu16]. We use the SVM learner from scikit-learn with the setting \( \gamma = \frac{1}{200} \) and kernel = “rbf”, which are used by XU. After that, the same training and testing knowledge unit pairs are applied to SVM.

Table 5.4 and Figure 5.3 show the performance scores and corresponding score delta between our implementation of WordEmbedding + SVM with XU’s in terms of accuracy \( ^5 \), precision, recall and F1-score. As we can see, when predicting these four different types of relatedness between knowledge unit pairs, our Word Embedding + SVM method has very similar performance scores to

\(^5\)XU just report overall accuracy, not for each class, hence it is missing in this table.
Figure 5.3 Score delta between our SVM with XU’s SVM in [Xu16] in terms of precision, recall and F1-score. Positive values mean our SVM is better than XU’s SVM in terms of different measures; Otherwise, XU’s SVM is better.

Table 5.4 Comparison of our baseline method with XU’s. The best scores are marked in **bold**.

<table>
<thead>
<tr>
<th>Metrics</th>
<th>Methods</th>
<th>Duplicate</th>
<th>Direct Link</th>
<th>Indirect Link</th>
<th>Isolated</th>
<th>Overall</th>
</tr>
</thead>
<tbody>
<tr>
<td>Precision</td>
<td>Our SVM</td>
<td><strong>0.724</strong></td>
<td>0.514</td>
<td>0.779</td>
<td>0.601</td>
<td>0.655</td>
</tr>
<tr>
<td></td>
<td>XU’s SVM</td>
<td>0.611</td>
<td><strong>0.560</strong></td>
<td><strong>0.787</strong></td>
<td><strong>0.676</strong></td>
<td><strong>0.659</strong></td>
</tr>
<tr>
<td>Recall</td>
<td>Our SVM</td>
<td>0.525</td>
<td><strong>0.492</strong></td>
<td>0.970</td>
<td><strong>0.645</strong></td>
<td>0.658</td>
</tr>
<tr>
<td></td>
<td>XU’s SVM</td>
<td><strong>0.725</strong></td>
<td>0.433</td>
<td><strong>0.980</strong></td>
<td>0.538</td>
<td><strong>0.669</strong></td>
</tr>
<tr>
<td>F1-score</td>
<td>Our SVM</td>
<td>0.609</td>
<td><strong>0.503</strong></td>
<td>0.864</td>
<td><strong>0.622</strong></td>
<td>0.650</td>
</tr>
<tr>
<td></td>
<td>XU’s SVM</td>
<td><strong>0.663</strong></td>
<td>0.488</td>
<td><strong>0.873</strong></td>
<td>0.600</td>
<td><strong>0.656</strong></td>
</tr>
<tr>
<td>Accuracy</td>
<td>Our SVM</td>
<td>0.525</td>
<td>0.493</td>
<td>0.970</td>
<td>0.645</td>
<td>0.658</td>
</tr>
<tr>
<td></td>
<td>XU’s SVM</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td><strong>0.669</strong></td>
</tr>
</tbody>
</table>
the baseline method reported by XU in [Xu16], with the maximum difference less than 0.2. Except for Duplicate class, where our baseline has a higher precision (i.e., 0.724 v.s. 0.611) but a lower recall (i.e., 0.525 v.s. 0.725).

Figure 5.3 presents the same results in a graphical format. Any bar above zero means that our implementation has a better performance score than XU’s on predicting that specific knowledge unit relatedness class. As we can see, most of the differences (\( \frac{8}{12} \)) are within 0.05 and the score delta of overall performance shows that our implementation is a little worse than XU’s implementation. For this chart we conclude that:

Overall, our reimplementation of WordEmbedding + SVM has very similar performance in all the evaluated metrics compared to the baseline method reported in XU’s study [Xu16].

The significance of this conclusion is that, moving forward, we are confident that we can use our reimplementation of WordEmbedding+SVM as a valid surrogate for the baseline method of XU.

**RQ2: Can DE tune a standard learner such that it outperforms XU’s deep learning method?**

To answer this question, we run the workflow of Figure 5.2, where DE is applied to find the optimal parameters of SVM based on the training and tuning data. The optimal tunings are then applied on the SVM model and the built learner is evaluated on testing data. Note that, in this study, since we mainly focus on precision, recall and F1-score measures where F1-score is the harmonic mean of precision and recall, we use F1-score as the tuning goal for DE. In other words, when tuning parameters, DE expects to find a pair of candidate parameters that maximize F1-score.

Table 5.5 presents the performance scores of XU’s baseline, XU’s CNN method and Tuned SVM for all metrics. The highest score for each relatedness class is marked in bold. Note that: Without tuning, XU’s CNN method outperforms the baseline SVM in \( \frac{10}{12} \) evaluation metrics across all four classes. The largest performance improvement is 0.47 for recall on Direct Link class. Note that this result is consistent with XU’s conclusion that their CNN method is superior to standard SVM. After tuning SVM, the deep learning method has no such advantage. Specifically, CNN has advantage over tuned SVM in \( \frac{4}{12} \) evaluation metrics across all four classes. Even when CNN performs better that our tuning SVM method, the largest difference is 0.065 for Recall on Direct Link class, which is less than 0.1.

Figure 5.4 presents the same results in a graphical format. Any bar above zero indicates that tuned SVM has a better performance score than CNN. In this figure: CNN has a slightly better performance on Duplicate class for precision, recall and F1-score and a higher recall on Direct Link class. Across all of Figure 5.4, in \( \frac{8}{12} \) evaluation scores, Tuned SVM has better performance scores than CNN, with the largest delta of 0.222.

Figure 5.5 compares the performance delta of tuned SVM with XU’s untuned SVM. We note that DE-based parameter tuning never degrades SVM’s performance (since there are no negative values
Figure 5.4 Score delta between tuned SVM and CNN method [Xu16] in terms of precision, recall and F1-score. Positive values mean tuned SVM is better than CNN in terms of different measures; Otherwise, CNN is better.

Table 5.5 Comparison of the tuned SVM with XU's CNN method. The best scores are marked in bold.

<table>
<thead>
<tr>
<th>Metrics</th>
<th>Methods</th>
<th>Duplicate</th>
<th>Direct Link</th>
<th>Indirect Link</th>
<th>Isolated</th>
<th>Overall</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>XU's SVM</td>
<td>0.611</td>
<td>0.560</td>
<td>0.787</td>
<td>0.676</td>
<td>0.658</td>
</tr>
<tr>
<td></td>
<td>XU's CNN</td>
<td>0.898</td>
<td>0.758</td>
<td>0.840</td>
<td>0.890</td>
<td>0.847</td>
</tr>
<tr>
<td></td>
<td>Tuned SVM</td>
<td>0.885</td>
<td>0.851</td>
<td><strong>0.944</strong></td>
<td><strong>0.903</strong></td>
<td><strong>0.896</strong></td>
</tr>
<tr>
<td></td>
<td>XU's SVM</td>
<td>0.725</td>
<td>0.433</td>
<td>0.980</td>
<td>0.538</td>
<td>0.669</td>
</tr>
<tr>
<td></td>
<td>XU's CNN</td>
<td><strong>0.898</strong></td>
<td><strong>0.903</strong></td>
<td>0.773</td>
<td>0.793</td>
<td>0.842</td>
</tr>
<tr>
<td></td>
<td>Tuned SVM</td>
<td>0.860</td>
<td>0.828</td>
<td><strong>0.995</strong></td>
<td><strong>0.905</strong></td>
<td><strong>0.897</strong></td>
</tr>
<tr>
<td></td>
<td>XU's SVM</td>
<td>0.663</td>
<td>0.488</td>
<td>0.873</td>
<td>0.600</td>
<td>0.656</td>
</tr>
<tr>
<td></td>
<td>XU's CNN</td>
<td><strong>0.898</strong></td>
<td>0.824</td>
<td>0.805</td>
<td>0.849</td>
<td>0.841</td>
</tr>
<tr>
<td></td>
<td>Tuned SVM</td>
<td>0.878</td>
<td><strong>0.841</strong></td>
<td><strong>0.969</strong></td>
<td><strong>0.909</strong></td>
<td><strong>0.899</strong></td>
</tr>
</tbody>
</table>
Tuning dramatically improves scores on predicting some classes of KU relatedness. For example, the recall of predicting Direct_Link is increased from 0.433 to 0.903, which is 108% improvement over XU’s untuned SVM (To be fair for XU, it is still 84% improvement over our untuned SVM). At the same time, the corresponding precision and F1 scores of predicting Direct_Link are increased from 0.560 to 0.851 and 0.488 to 0.841, which are 52% and 72% improvement over XU’s original report [Xu16], respectively. A similar pattern can also be observed in Isolated class. On
average, tuning helps improve the performance of XU’s SVM by 0.238, 0.228 and 0.227 in terms of precision, recall and F1-score for all four KU relatedness classes. Figure 5.6 compares the tuned SVM with our untuned SVM. We note that we get the similar patterns that observed in Figure 5.5. All the bars are above zero, etc.

Based on the performance scores in Table 5.5 and score delta in Figure 5.4, Figure 5.5 and Figure 5.6, we can see that:

- Parameter tuning can dramatically improve the performance of Word Embedding + SVM (the baseline method) for the multi-class KU relatedness prediction task;

- With the optimal tunings, the traditional machine learning method, SVM, if not better, is at least comparable with deep learning methods (CNN).

When discussing this result with colleagues, we are sometimes asked for a statistical analysis that confirms the above finding. However, due the lack of evaluation score distributions of the CNN method in [Xu16], we cannot compare their single value with our results from 10 repeated runs. However, according to Wilcoxon singed rank test over 10 runs results, tuned SVM performs statistically better than our untuned SVM in terms of all evaluation measures on all four classes ($p < 0.05$). According to Cliff $\delta$ values, the magnitude of difference between tuned SVM and our untuned SVM is not trivial ($|\delta| > 0.147$) for all evaluation measures.

Overall, the experimental results and our analysis indicate that:

In the evaluation conducted here, the deep learning method, CNN, does not have any performance advantage over our tuning approach.

RQ3: Is tuning SVM with DE faster than XU’s deep learning method?

When comparing the runtime of two learning methods, it obviously should be conducted under the same hardware settings. Since we adopt the CNN evaluation scores from [Xu16], we can not run on our tuning SVM experiment under the exactly same system settings. To allow readers to have a objective comparison, we provide the experimental environment as shown in Table 5.6. To obtain the runtime of tuning SVM, we recorded the start time and end time of the program execution, including parameter tuning, training model and testing model.

<table>
<thead>
<tr>
<th>Methods</th>
<th>OS</th>
<th>CPU</th>
<th>RAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tuning SVM</td>
<td>MacOS 10.12</td>
<td>Intel Core i5 2.7 GHz</td>
<td>8 GB</td>
</tr>
<tr>
<td>CNN</td>
<td>Windows 7</td>
<td>Intel Core i7 2.5 GHz</td>
<td>16 GB</td>
</tr>
</tbody>
</table>
According to XU, it took 14 hours to train their CNN model into a low loss convergence ($< e^{-3}$) [Xu16]. Our work, on the other hand only takes 10 minutes to run SVM with parameter tuning by DE on a similar environment. That is, the simple parameter tuning method on SVM is $84X$ faster than XU’s deep learning method.

Compared to CNN method, tuning SVM is about $84X$ faster in terms of model building.

The significance of this finding is that, in this case study, CNN was neither better in performance scores (see RQ2) nor runtimes. CNN’s extra runtimes are a particular concern since (a) they are very long; and (b) these would be incurred anytime researchers wants to update the CNN model with new data or wanted to validate the XU result.

5.6 Discussion

5.6.1 Why DE+SVM works?

Parameter tuning matters. As mentioned in Section 5.2.4, the default parameter values set by the algorithm designers could generate a good performance on average but may not guarantee the best performance for the local data [BB12; Fu16a]. Given that, it is most strange to report that most SE researchers ignore the impacts of parameter tuning when they utilize various machine learning methods to conduct software analytic (evidence: see our reviews in [Fu16a; Fu16b; Agr18]). The conclusion of this work must be to stress the importance of this kind of tuning, using local data, for any future software analytics study.

Better explore the searching space. It turns out that one exception to our statement that “most researchers do not tune” is the XU study. In that work, they unsuccessfully perform parameter tuning, but with with grid search. In such a grid search, for $N$ parameters to be tuned, $N$ for loops are created to run over a range of settings for each parameter. While a widely used method, it is often deprecated. For example, Bergstra et al.[BB12] note that grid search jumps through different parameter settings between some $min$ and $max$ values of pre-defined tuning range. They warn that such jumps may actually skip over the critical tuning values. On the other hand, DE tuning values are adjusted based on better candidates from previous generations. Hence DE is more likely than grid search to “fill in the gaps” between the initialized values.

That said, although DE +SVM works in this study, it does not mean DE is the best parameter tuner for all SE tasks. We encourage more researchers to explore faster and more effective parameter tuners in this direction.
5.6.2 Implication

Beyond the specifics of this case study, what general principles can we take from the above work?

**Understand the task.** One reason to try different tools for the same task is to better understand the task. The more we understand a task, the better we can match tools to that task. Tools that are poorly matched to task are usually complex and/or slow to execute. In this study, we would say that

- Deep learning is a poor match to the task of predicting whether two questions posted on Stack Overflow are semantically linkable since it is so slow;
- Differential evolution tuning SVM is a much better match since it is so fast and obtain competitive performance.

That said, it is important to stress that the point of this study is not to deprecate deep learning. There are many scenarios were we believe deep learning would be a natural choice (e.g. when analyzing complex speech or visual data). In SE, it is still an open research question that in which scenario deep learning is the best choice. Results from this study show that, at least for classification tasks like knowledge unit relatedness classification on Stack Overflow, deep learning does not have much advantage over well tuned conventional machine learning methods. However, as we better understand SE tasks, deep learning could be used to address more SE problems, which require more advanced artificial intelligence.

**Treat resource constraints as design challenges.** As a general engineering principle, we think it insightful to consider the resource cost of a tool before applying it. It turns out that this is a design pattern used in contemporary industry. According to Calero and Pattini [CP15], many current commercial redesigns are motivated (at least in part) by arguments based on sustainability (i.e. using fewer resources to achieve results). In fact, they say that managers used sustainability-based redesigns to motivate extensive cost-cutting opportunities.

5.6.3 Threads to Validity

Threats to internal validity concern the consistency of the results obtained from the result. In our study, to investigate how tuning can improve the performance of baseline methods and how well it perform compared with deep learning method. We select XU’s Word Embedding + SVM baseline method as a case study. Since the original implementation of Word Embedding + SVM (baseline 2 method in [Xu16]) is not publicly available, we have to reimplement our version of Word Embedding + SVM as the baseline method in this study. As shown in RQ1, our implementation similar results to XU’s on the same data sets. Hence, we believe that our implementation reflect the original baseline method in Xu’s study [Xu16].

Threats to external validity represent if the results are of relevance for other cases, or the ability to generalize the observations in a study. In this study, we compare our tuning baseline method with
The experimental results are quite consistent for this knowledge units relatedness prediction task. Nonetheless, we do not claim that our findings can be generalized to all software analytics tasks. However, those other software analytics tasks often apply deep learning methods on classification tasks [Cho16; Wan16] and so it is quite possible that the methods of this study (i.e., DE-based parameter tuning) would be widely applicable, elsewhere.

5.7 Conclusion

In this study, we perform a comparative study to investigate how tuning can improve the baseline method compared with state-of-the-art deep learning method for predicting knowledge units relatedness on Stack Overflow. Our experimental results show that:

- Tuning improves the performance of baseline methods. At least for Word Embedding + SVM (baseline in [Xu16]) method, if not better, it performs as well as the proposed CNN method in [Xu16].

- The baseline method with parameter tuning runs much faster than complicated deep learning. In this study, tuning SVM runs 84X faster than CNN method.
Part III

However, Oversimplification Can Be Harmful
This chapter firstly appeared at *2017 11th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2017), pp. 72-83. ACM.* titled “Revisiting Unsupervised Learning for Defect Prediction”. In this chapter, we show that sometimes, software analytics can be too easy.

Software analytics can be easier as shown in previous chapters. However, if we don't take account of actual limitations during the model deployment, we might easily get things worse. At FSE’16, Yang et al. reported startling results where unsupervised defect predictors outperformed supervised predictors for effort-aware just-in-time defect prediction. If confirmed, these results would lead to a dramatic simplification of a seemingly complex task (data mining) that is widely explored in the software engineering literature. In this chapter, we repeat and refute those results as follows.

(1) There is much variability in the efficacy of the Yang et al. predictors so even with their approach, some supervised data is required to prune weaker predictors away. (2) Their findings were grouped across $N$ projects. When we repeat their analysis on a project-by-project basis, supervised predictors are seen to work better. (3) If we plan to deploy such model in practice, we might have difficulties to get a consistent prediction among those unsupervised learners. Therefore, the case study in this chapter serves to defend out claim that *oversimplification can be harmful.*
6.1 Introduction

This study repeats and refutes recent results from Yang et al. [Yan16b] published at FSE’16. The task explored by Yang et al. was effort-ware just-in-time (JIT) software defect predictors. JIT defect predictors are built on code change level and could be used to conduct defect prediction right before developers commit the current change. They report an unsupervised software quality prediction method that achieved better results than standard supervised methods. We repeated their study since, if their results were confirmed, this would imply that decades of research into defect prediction [Les08; Has09; Fu16b; Fu16a; Men07b; Hal12; NB05b; Jia13; Kim07; Jia13; Kim08; Jia13; Jin14; Lee11; Mos08; Nam13; Wan16; Yan16a; Kam13] had needlessly complicated an inherently simple task.

The standard method for software defect prediction is learning from labeled data. In this approach, the historical log of known defects is learned by a data miner. Note that this approach requires waiting until a historical log of defects is available; i.e. until after the code has been used for a while. Another approach, explored by Yang et al., uses general background knowledge to sort the code, then inspect the code in that sorted order. In their study, they assumed that more defects can be found faster by first looking over all the “smaller” modules (an idea initially proposed by Koru et al. [Kor09]). After exploring various methods of defining “smaller”, they report their approach finds more defects, sooner, than supervised methods. These results are highly remarkable:

- This approach does not require access to labeled data; i.e. it can be applied just as soon as the code is written.
- It is extremely simple: no data pre-processing, no data mining, just simple sorting.

Because of the remarkable nature of these results, this study takes a second look at the Yang et al. results. We ask three questions:

**RQ1: Do all unsupervised predictors perform better than supervised predictors?**

The reason we ask this question is that if the answer is “yes”, then we can simply select any unsupervised predictor built from the the change metrics as Yang et al suggested without using any supervised data; if the answer is “no”, then we must apply some techniques to select best predictors and remove the worst ones. However, our results show that, when projects are explored separately, the majority of the unsupervised predictors learned by Yang et al. perform worse than supervised predictors.

Results of RQ1 suggest that after building multiple predictors using unsupervised methods, it is required to prune the worst predictors and only better ones should be used for future prediction. However, with Yang et al. approach, there is no way to tell which unsupervised predictors will perform better without access to the labels of testing data. To test that speculation, we built a new learner, OneWay, that uses supervised training data to remove all but one of the Yang et al. predictors. Using this learner, we asked:
RQ2: Is it beneficial to use supervised data to prune away all but one of the Yang et al. predictors?

Our results showed that OneWay nearly always outperforms the unsupervised predictors found by Yang et al. The success of OneWay leads to one last question:

RQ3: Does OneWay perform better than more complex standard supervised learners?

Such standard supervised learners include Random Forests, Linear Regression, J48 and IBk (these learners were selected based on prior results by [Men07b; Les08; Kam13; Hal12]). We find that in terms of Recall and \( P_{opt} \) (the metric preferred by Yang et al.), OneWay performed better than standard supervised predictors. Yet measured in terms of Precision, there was no advantage to OneWay.

From the above, we make an opposite conclusion to Yang et al.; i.e., there are clear advantages to use supervised approaches over unsupervised ones. We explain the difference between our results and their results as follows:

- Yang et al. reported averaged results across all projects;
- We offer a more detailed analysis on a project-by-project basis.

The rest of this chapter is organized as follows. Section 6.2 is a commentary on Yang et al. study and the implication of this study. Section 6.3 describes the background and related work on defect prediction. Section 6.4 explains the effort-aware JIT defect prediction methods investigated in this study. Section 6.5 describes the experimental settings of our study, including research questions that motivate our study, data sets and experimental design. Section 6.6 presents the results. Section 6.7 discusses the threats to the validity of our study. Section 6.8 presents the conclusion and future work.

Note one terminological convention: in the following, we treat “predictors” and “learners” as synonyms.

6.2 Science in the 21st Century

While this study is specific about effort-aware JIT defect prediction and the Yang et al. result, at another level this study is also about science in the 21st century.

In 2017, the software analytics community now has the tools, data sets, experience to explore a bold wider range of options. There are practical problems in exploring all those possibilities specifically, too many options. For example, in section 2.5 of [Koc12a], Kocaguneli et al. listed 12,000+ different ways of estimation by analogy. We have had some recent successes with exploring this space of options [Fu16a] but only after the total space of options is reduced by some initial study to a manageable set of possibilities. Hence, what is needed are initial studies to rule out methods that are
generally unpromising (e.g. this study) before we apply second level hyper-parameter optimization study that takes the reduced set of options.

Another aspect of 21st century science that is highlighted by this study is the nature of repeatability. While this study disagrees the conclusions of Yang et al., it is important to stress that their paper is an excellent example of good science that should be emulated in future work.

Firstly, they tried something new. There are many papers in the SE literature about defect prediction. However, compared to most of those, the Yang et al. paper is bold and stunningly original.

Secondly, they made all their work freely available. Using the “R” code they placed online, we could reproduce their result, including all their graphical output, in a matter of days. Further, using that code as a starting point, we could rapidly conduct the extensive experimentation that leads to this study. This is an excellent example of the value of open science.

Thirdly, while we assert their answers were wrong, the question they asked is important and should be treated as an open and urgent issue by the software analytics community. In our experiments, supervised predictors performed better than unsupervised— but not outstandingly better than unsupervised. Hence, they may indeed be some combination of unsupervised learners to achieve comparable performance to supervised. Therefore, even though we reject the specific conclusions of Yang et al., we still endorse the question they asked strongly and encourage others to work in this area.

6.3 Background and Related Work

6.3.1 Defect Prediction

As soon as people started programming, it became apparent that programming was an inherently buggy process. As recalled by Maurice Wilkes [Wil85], speaking of his programming experiences from the early 1950s: “It was on one of my journeys between the EDSAC room and the punching equipment that ‘hesitating at the angles of stairs’ the realization came over me with full force that a good part of the remainder of my life was going to be spent in finding errors in my own programs.”

It took decades to gather the experience required to quantify the size/defect relationship. In 1971, Fumio Akiyama [Aki71] described the first known “size” law, saying the number of defects $D$ was a function of the number of LOC where $D = 4.86 + 0.018 \times LOC$.

In 1976, Thomas McCabe argued that the number of LOC was less important than the complexity of that code [McC76]. He argued that code is more likely to be defective when his “cyclomatic complexity” measure was over 10. Later work used data miners to build defect predictors that proposed thresholds on multiple measures [Men07b].

Subsequent research showed that software bugs are not distributed evenly across a system.
Rather, they seem to clump in small corners of the code. For example, Hamill et al. [HGP09] report studies with (a) the GNU C++ compiler where half of the files were never implicated in issue reports while 10% of the files were mentioned in half of the issues. Also, Ostrand et al. [Ost04] studied (b) AT&T data and reported that 80% of the bugs reside in 20% of the files. Similar “80-20” results have been observed in (c) NASA systems [HGP09] as well as (d) open-source software [Kor09] and (e) software from Turkey [Mis11].

Given this skewed distribution of bugs, a cost-effective quality assurance approach is to sample across a software system, then focus on regions reporting some bugs. Software defect predictors built from data miners are one way to implement such a sampling policy. While their conclusions are never 100% correct, they can be used to suggest where to focus more expensive methods such as elaborate manual review of source code [Shu01]; symbolic execution checking [Pas08], etc. For example, Misirli et al. [Mis11] report studies where the guidance offered by defect predictors:

- Reduced the effort required for software inspections in some Turkish software companies by 72%;
- While, at the same time, still being able to find the 25% of the files that contain 88% of the defects.

Not only do static code defect predictors perform well compared to manual methods, they also are competitive with certain automatic methods. A recent study at ICSE’14, Rahman et al. [Rah14] compared (a) static code analysis tools FindBugs, Jlint, and Pmd and (b) static code defect predictors (which they called “statistical defect prediction”) built using logistic regression. They found no significant differences in the cost-effectiveness of these approaches. Given this equivalence, it is significant to note that static code defect prediction can be quickly adapted to new languages by building lightweight parsers that extract static code metrics. The same is not true for static code analyzers– these need extensive modification before they can be used on new languages.

To build such defect predictors, we measure the complexity of software projects using McCabe metrics, Halstead’s effort metrics and CK object-oriented code metrics [KR87; CK94; McC76; Hal77] at a coarse granularity, like file or package level. With the collected data instances along with the corresponding labels (defective or non-defective), we can build defect prediction models using supervised learners such as Decision Tree, Random Forests, SVM, Naive Bayes and Logistic Regression [KA01; KS03; Kho00; Men07b; Les08; Hal12]. After that, such trained defect predictor can be applied to predict the defects of future projects.

6.3.2 Just-In-Time Defect Prediction

Traditional defect prediction has some drawbacks such as prediction at a coarse granularity and started at very late stage of software development circle [Kam13], whereas in JIT defect prediction
paradigm, the defect predictors are built on code change level, which could easily help developers narrow down the code for inspection and JIT defect prediction could be conducted right before developers commit the current change. JIT defect prediction becomes a more practical method for practitioners to carry out.

Mockus et al. [MW00] conducted the first study to predict software failures on a telecommunication software project, 5ESS, by using logistic regression on data sets consisted of change metrics of the project. Kim et al. [Kim08] further evaluated the effectiveness of change metrics on open source projects. In their study, they proposed to apply support vector machine to build a defect predictor based on software change metrics, where on average they achieved 78% accuracy and 60% recall. Since training data might not be available when building the defect predictor, Fukushima et al. [Fuk14] introduced cross-project paradigm into JIT defect prediction. Their results showed that using data from other projects to build JIT defect predictor is feasible.

Most of the research into defect prediction does not consider the effort\textsuperscript{1} required to inspect the code predicted to be defective. Exceptions to this rule include the work of Arishom and Briand [AB06], Koru et al. [Kor09] and Kamei et al. [Kam13]. Kamei et al. [Kam13] conducted a large-scale study on the effectiveness of JIT defect prediction, where they claimed that using 20% of efforts required to inspect all changes, their modified linear regression model (EALR) could detect 35% defect-introducing changes. Inspired by Menzies et al.’s ManualUp model (i.e., small size of modules inspected first) [Men10], Yang et al. [Yan16b] proposed to build 12 unsupervised defect predictors by sorting the reciprocal values of 12 different change metrics on each testing data set in descending order. They reported that with 20% efforts, many unsupervised predictors perform better than state-of-the-art supervised predictors.

6.4 Method

6.4.1 Unsupervised Predictors

In this section, we describe the effort-aware just-in-time unsupervised defect predictors proposed by Yang et al. [Yan16b], which serves as a baseline method in this study. As described by Yang et al. [Yan16b], their simple unsupervised defect predictor is built on change metrics as shown in Table 6.1. These 14 different change metrics can be divided into 5 dimensions [Kam13]:

- **Diffusion**: NS, ND, NF and Entropy.
- **Size**: LA, LD and LT.
- **Purpose**: FIX.

\textsuperscript{1}Effort means time/labor required to inspect total number of files/code predicted as defective.
• History: NDEV, AGE and NUC.

• Experience: EXP, REXP and SEXP.

**Table 6.1** Change metrics used in our data sets.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NS</td>
<td>Number of modified subsystems [MW00].</td>
</tr>
<tr>
<td>ND</td>
<td>Number of modified directories [MW00].</td>
</tr>
<tr>
<td>NF</td>
<td>Number of modified files [Nag06].</td>
</tr>
<tr>
<td>Entropy</td>
<td>Distribution of the modified code across each file [D’A10; Has09].</td>
</tr>
<tr>
<td>LA</td>
<td>Lines of code added [NB05b].</td>
</tr>
<tr>
<td>LD</td>
<td>Lines of code deleted [NB05b].</td>
</tr>
<tr>
<td>LT</td>
<td>Lines of code in a file before the current change [Kor09].</td>
</tr>
<tr>
<td>FIX</td>
<td>Whether or not the change is a defect fix [Guo10; Yin11].</td>
</tr>
<tr>
<td>NDEV</td>
<td>Number of developers that changed the modified files [Mat10].</td>
</tr>
<tr>
<td>AGE</td>
<td>The average time interval between the last and the current change [Gra00].</td>
</tr>
<tr>
<td>NUC</td>
<td>The number of unique changes to the modified files [D’A10; Has09].</td>
</tr>
<tr>
<td>EXP</td>
<td>The developer experience in terms of number of changes [MW00].</td>
</tr>
<tr>
<td>REXP</td>
<td>Recent developer experience [MW00].</td>
</tr>
<tr>
<td>SEXP</td>
<td>Developer experience on a subsystem [MW00].</td>
</tr>
</tbody>
</table>

The diffusion dimension characterizes how a change is distributed at different levels of granularity. As discussed by Kamei et al. [Kam13], a highly distributed change is harder to keep track and more likely to introduce defects. The size dimension characterizes the size of a change and it is believed that the software size is related to defect proneness [NB05b; Kor09]. Yin et al. [Yin11] report that the bug-fixing process can also introduce new bugs. Therefore, the Fix metric could be used as a defect evaluation metric. The History dimension includes some historical information about the change, which has been proven to be a good defect indicator [Mat10]. For example, Matsumoto et al. [Mat10] find that the files previously touched by many developers are likely to contain more defects. The Experience dimension describes the experience of software programmers for the current change because Mockus et al. [MW00] show that more experienced developers are less likely to introduce a defect. More details about these metrics can be found in Kamei et al's study [Kam13].

In Yang et al.'s study, for each change metric $M$ of testing data, they build an unsupervised predictor that ranks all the changes based on the corresponding value of $\frac{1}{M(c)}$ in descending order, where $M(c)$ is the value of the selected change metric for each change $c$. Therefore, the changes with smaller change metric values will ranked higher. For each project, Yang et al. define 12 unsupervised predictors (LA and LD are excluded as Yang et al. [Yan16b]).
6.4.2 Supervised Predictors

To further evaluate the unsupervised predictor, we selected some supervised predictors that already used in Yang et al.'s work.

As reported in both Yang et al.'s [Yan16b] and Kamei et al.'s [Kam13] work, EALR outperforms all other supervised predictors for effort-aware JIT defect prediction. EALR is a modified linear regression model [Kam13] and it predicts \( \frac{Y(x)}{\text{Effort}(x)} \) instead of predicting \( Y(x) \), where \( Y(x) \) indicates whether this change is a defect or not (1 or 0) and \( \text{Effort}(x) \) represents the effort required to inspect this change. Note that this is the same method to build EALR as Kamei et al. [Kam13].

In defect prediction literature, IBk (KNN), J48 and Random Forests methods are simple yet widely used as defect learners and have been proven to perform, if not best, quite well for defect prediction [Les08; Men07b; Hal12; Tur09]. These three learners are also used in Yang et al.'s study. For these supervised predictors, \( Y(x) \) was used as the dependant variable. For KNN method, we set \( K = 8 \) according to Yang et al. [Yan16b].

6.4.3 OneWay Learner

Based on our preliminary experiment results shown in the following section, for the six projects investigated by Yang et al, some of 12 unsupervised predictors do perform worse than supervised predictors and there is no one predictor constantly working best on all project data. This means we can not simply say which unsupervised predictor works for the new project before predicting on the testing data. In this case, we need a technique to select the proper metrics to build defect predictors.

We propose OneWay learner, which is a supervised predictor built on the implication of Yang et al.'s simple unsupervised predictors. The pseudocode for OneWay is shown in Algorithm 2. In the following description, we use the superscript numbers to denote the line number in pseudocode.

The general idea of OneWay is to use supervised training data to remove all but one of the Yang et al. predictors and then apply this trained learner on the testing data. Specifically, OneWay firstly builds simple unsupervised predictors from each metric on training data\(^{14}\), then evaluates each of those learners in terms of evaluation metrics\(^{15}\), like \( P_{opt} \), \( Recall \), \( Precision \) and \( F1 \). After that, if the desirable evaluation goal is set, the metric which performs best on the corresponding evaluation goal is returned as the best metric; otherwise, the metric which gets the highest mean score over all evaluation metrics is returned\(^{19}\) (In this study, we use the latter one). Finally, a simple predictor is built only on such best metric\(^{19}+10\) with the help of training data. Therefore, OneWay builds only one supervised predictor for each project using the local data instead of 12 predictors directly on testing data as Yang et al [Yan16b].
Algorithm 2 Pseudocode for OneWay

**Input:** data\_train, data\_test, eval\_goal ∈ {F1, P\_opt, Recall, Precision,...}

**Output:** result

1: function ONEWAY(data\_train, data\_test, eval\_goal)
2: all\_scores ← NULL
3: for metric in data\_train do
4: learner ← buildUnsupervisedLearner(data\_train, metric)
5: scores ← evaluate(learner)
6: // scores include all evaluation goals, e.g., P\_opt, F1,...
7: all\_scores.append(scores)
8: end for
9: best\_metric ← pruneFeature(all\_scores, eval\_goal)
10: result ← buildUnsupervisedLearner(data\_test, best\_metric)
11: return result
12: end function

13: function PRUNEFEATURE(all\_scores, eval\_goal)
14: if eval\_goal == NULL then
15: mean\_scores ← getMeanScoresForEachMetric(all\_scores)
16: best\_metric ← getMetric(max(mean\_scores))
17: return best\_metric
18: else
19: best\_metric ← getMetric(max(all\_scores["eval\_goal"]))
20: return best\_metric
21: end if
22: end function

### 6.5 Experimental Settings

#### 6.5.1 Research Questions

Using the above methods, we explore three questions:

- Do all unsupervised predictors perform better than supervised predictors?
- Is it beneficial to use supervised data to prune all but one of the Yang et al. unsupervised predictors?
- Does OneWay perform better than more complex standard supervised predictors?

When reading the results from Yang et al. [Yan16b], we find that they aggregate performance scores of each learner on six projects, which might miss some information about how learners perform on each project. Are these unsupervised predictors working consistently across all the project data? If not, how would it look like? Therefore, in RQ1, we report results for each project separately.
Another observation is that even though Yang et al. [Yan16b] propose that simple unsupervised predictors could work better than supervised predictors for effort-aware JIT defect prediction, one missing aspect of their report is how to select the most promising metric to build a defect predictor. This is not an issue when all unsupervised predictors perform well but, as we shall see, this is not the case. As demonstrated below, given $M$ unsupervised predictors, only a small subset can be recommended. Therefore it is vital to have some mechanism by which we can down select from $M$ models to the $L \ll M$ that are useful. Based on this fact, we propose a new method, OneWay, which is the missing link in Yang et al.’s study [Yan16b] and the missing final step they do not explore. Therefore, in RQ2 and RQ3, we want to evaluate how well our proposed OneWay method performs compared to the unsupervised predictors and supervised predictors.

Considering our goals and questions, we reproduce Yang et al’s results and report for each project to answer RQ1. For RQ2 and RQ3, we implement our OneWay method, and compare it with unsupervised predictors and supervised predictors on different projects in terms of various evaluation metrics.

### Table 6.2 Statistics of the studied data sets

<table>
<thead>
<tr>
<th>Project</th>
<th>Period</th>
<th>Total Change</th>
<th>% of Defects</th>
<th>Avg LOC per Change</th>
<th># Modified Files per Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bugzilla</td>
<td>08/1998 - 12/2006</td>
<td>4620</td>
<td>36%</td>
<td>37.5</td>
<td>2.3</td>
</tr>
<tr>
<td>Platform</td>
<td>05/2001 - 12/2007</td>
<td>64250</td>
<td>14%</td>
<td>72.2</td>
<td>4.3</td>
</tr>
<tr>
<td>Mozilla</td>
<td>01/2000 - 12/2006</td>
<td>98275</td>
<td>5%</td>
<td>106.5</td>
<td>5.3</td>
</tr>
<tr>
<td>JDT</td>
<td>05/2001 - 12/2007</td>
<td>35386</td>
<td>14%</td>
<td>71.4</td>
<td>4.3</td>
</tr>
<tr>
<td>Columba</td>
<td>11/2002 - 07/2006</td>
<td>4455</td>
<td>31%</td>
<td>149.4</td>
<td>6.2</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>07/1996 - 05/2010</td>
<td>20431</td>
<td>25%</td>
<td>101.3</td>
<td>4.5</td>
</tr>
</tbody>
</table>

### 6.5.2 Data Sets

In this study, we conduct our experiment using the same data sets as Yang et al. [Yan16b], which are six well-known open source projects, Bugzilla, Columba, Eclipse JDT, Eclipse Platform, Mozilla and PostgreSQL. These data sets are shared by Kamei et al. [Kam13]. The statistics of the data sets are listed in Table 6.2. From Table 6.2, we know that all these six data sets cover at least 4 years historical information, and the longest one is PostgreSQL, which includes 15 years of data. The total changes for these six data sets are from 4450 to 98275, which are sufficient for us to conduct an empirical study. In this study, if a change introduces one or more defects then this change is considered as defect-introducing change. The percentage of defect-introducing changes ranges from 5% to 36%.
All the data and code used in this study is available online\(^2\).

### 6.5.3 Experimental Design

The following principle guides the design of these experiments:

*Whenever there is a choice between methods, data, etc., we will always prefer the techniques used in Yang et al. [Yan16b].*

By applying this principle, we can ensure that our experimental setup is the same as Yang et al. [Yan16b]. This will increase the validity of our comparisons with that prior work.

When applying data mining algorithms to build predictive models, one important principle is not to test on the data used in training. To avoid that, we used time-wise-cross-validation method which is also used by Yang et al. [Yan16b]. The important aspect of the following experiment is that it ensures that all testing data was created after training data. Firstly, we sort all the changes in each project based on the commit date. Then all the changes that were submitted in the same month are grouped together. For a given project data set that covers totally \(N\) months history, when building a defect predictor, consider a sliding window size of 6,

- The first two consecutive months data in the sliding window, \(i\)th and \((i+1)\)th, are used as the training data to build supervised predictors and OneWay learner.

- The last two months data in the sliding window, \((i+4)\)th and \((i+5)\)th, which are two months later than the training data, are used as the testing data to test the supervised predictors, OneWay learner and unsupervised predictors.

After one experiment, the window slides by “one month” data. By using this method, each training and testing data set has two months data, which will include sufficient positive and negative instances for the supervised predictors to learn. For any project that includes \(N\) months data, we can perform \(N - 5\) different experiments to evaluate our learners when \(N\) is greater than 5. For all the unsupervised predictors, only the testing data is used to build the model and evaluate the performance.

To statistically compare the differences between OneWay with supervised and unsupervised predictors, we use Wilcoxon single ranked test to compare the performance scores of the learners in this study the same as Yang et al. [Yan16b]. To control the false discover rate, the Benjamini-Hochberg (BH) adjusted p-value is used to test whether two distributions are statistically significant at the level of 0.05 [BH95; Yan16b]. To measure the effect size of performance scores among OneWay and supervised/unsupervised predictors, we compute Cliff’s \(\delta\) that is a non-parametric effect size measure [Rom06]. As Romano et al. suggested, we evaluate the magnitude of the effect size as follows:

\(^2\)https://github.com/WeiFoo/RevisitUnsupervised
negligible ($|\delta| < 0.147$), small ($0.147 \leq |\delta| < 0.33$), medium ($0.33 \leq |\delta| < 0.474$), and large ($0.474 \leq |\delta|$) [Rom06].

### 6.5.4 Evaluation Measures

For effort-aware JIT defect prediction, in addition to evaluate how learners correctly predict a defect-introducing change, we have to take account the efforts that are required to inspect prediction. Ostrand et al. [Ost05] report that given a project, 20% of the files contain on average 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 [Yan16b; Kam13; MK10; Mon13]. That is, given 20% effort, how many defects can be detected by the learner. To be consistent with Yang et al, in this study, we restrict our efforts to 20% of total efforts.

![Effort-based cumulative lift chart](image)

**Figure 6.1** Example of an effort-based cumulative lift chart [Yan16b].

\[
\text{Precision} = \frac{\text{True Positive}}{\text{True Positive} + \text{False Positive}}
\]

\[
\text{Recall} = \frac{\text{True Positive}}{\text{True Positive} + \text{False Negative}}
\]

\[
F1 = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Recall} + \text{Precision}}
\]

where Precision denotes the percentage of actual defective changes to all the predicted changes and Recall is the percentage of predicted defective changes to all actual defective changes. $F1$ is
a measure that combines both Precision and Recall which is the harmonic mean of Precision and Recall.

To evaluate the performance of effort-aware JIT defect prediction learners in our study, we used the following 4 metrics: Precision, Recall, F1 and \( P_{opt} \), which are widely used in defect prediction literature [Men07b; Men10; Zim07; Kam13; Yan16b; Mon13].

The last evaluation metric used in this study is \( P_{opt} \), which is defined as \( 1 - \Delta_{opt} \), where \( \Delta_{opt} \) is the area between the effort (code-churn-based) cumulative lift charts of the optimal model and the prediction model (as shown in Figure 6.1). In this chart, the x-axis is considered as the percentage of required effort to inspect the change and the y-axis is the percentage of defect-introducing change found in the selected change. 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. [Yan16b; Kam13; Mon13], \( P_{opt} \) can be normalized as follows:

\[
P_{opt}(m) = 1 - \frac{S(\text{optimal}) - S(m)}{S(\text{optimal}) - S(\text{worst})}
\]

where \( S(\text{optimal}) \), \( S(m) \) and \( S(\text{worst}) \) represent the area of curve under the optimal model, predicted model, and worst model, respectively. Note that the worst model is built by sorting all the changes according to the actual defect density in ascending order. For any learner, it performs better than random predictor only if the \( P_{opt} \) is greater than 0.5.

Note that, following the practices of Yang et al. [Yan16b], we measure Precision, Recall, F1 and \( P_{opt} \) at the effort = 20% point. In this study, in addition to \( P_{opt} \) and ACC (i.e., Recall) that is used in Yang et al.’s work [Yan16b], we include Precision and F1 measures and they provide more insights about all the learners evaluated in the study from very different perspectives, which will be shown in the next section.

6.6 Empirical Results

In this section, we present the experimental results to investigate how simple unsupervised predictors work in practice and evaluate the performance of the proposed method, OneWay, compared with supervised and unsupervised predictors.

Before we start off, we need a sanity check to see if we can fully reproduce Yang et al.’s results. Yang et al. [Yan16b] provide the median values of \( P_{opt} \) and Recall for the EALR model and the best two unsupervised models, LT and AGE, from the time-wise cross evaluation experiment. Therefore, we use those numbers to check our results.

**RQ1: Do all unsupervised predictors perform better than supervised predictors?**
Table 6.3 Comparison in $P_{opt}$: Yang’s method (A) vs. our implementation (B)

<table>
<thead>
<tr>
<th>Project</th>
<th>EALR A</th>
<th>EALR B</th>
<th>LT A</th>
<th>LT B</th>
<th>AGE A</th>
<th>AGE B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bugzilla</td>
<td>0.59</td>
<td>0.59</td>
<td>0.72</td>
<td>0.72</td>
<td>0.67</td>
<td>0.67</td>
</tr>
<tr>
<td>Platform</td>
<td>0.58</td>
<td>0.58</td>
<td>0.72</td>
<td>0.72</td>
<td>0.71</td>
<td>0.71</td>
</tr>
<tr>
<td>Mozilla</td>
<td>0.50</td>
<td>0.50</td>
<td>0.65</td>
<td>0.65</td>
<td>0.64</td>
<td>0.64</td>
</tr>
<tr>
<td>JDT</td>
<td>0.59</td>
<td>0.59</td>
<td>0.71</td>
<td>0.71</td>
<td>0.68</td>
<td>0.69</td>
</tr>
<tr>
<td>Columba</td>
<td>0.62</td>
<td>0.62</td>
<td>0.73</td>
<td>0.73</td>
<td>0.79</td>
<td>0.79</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>0.60</td>
<td>0.60</td>
<td>0.74</td>
<td>0.74</td>
<td>0.73</td>
<td>0.73</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td>0.58</td>
<td>0.58</td>
<td>0.71</td>
<td>0.71</td>
<td>0.70</td>
<td>0.70</td>
</tr>
</tbody>
</table>

Table 6.4 Comparison in Recall: Yang’s method (A) vs. our implementation (B)

<table>
<thead>
<tr>
<th>Project</th>
<th>EALR A</th>
<th>EALR B</th>
<th>LT A</th>
<th>LT B</th>
<th>AGE A</th>
<th>AGE B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bugzilla</td>
<td>0.29</td>
<td>0.30</td>
<td>0.45</td>
<td>0.45</td>
<td>0.38</td>
<td>0.38</td>
</tr>
<tr>
<td>Platform</td>
<td>0.31</td>
<td>0.30</td>
<td>0.43</td>
<td>0.43</td>
<td>0.43</td>
<td>0.43</td>
</tr>
<tr>
<td>Mozilla</td>
<td>0.18</td>
<td>0.18</td>
<td>0.36</td>
<td>0.36</td>
<td>0.28</td>
<td>0.28</td>
</tr>
<tr>
<td>JDT</td>
<td>0.32</td>
<td>0.34</td>
<td>0.45</td>
<td>0.45</td>
<td>0.41</td>
<td>0.41</td>
</tr>
<tr>
<td>Columba</td>
<td>0.40</td>
<td>0.42</td>
<td>0.44</td>
<td>0.44</td>
<td>0.57</td>
<td>0.57</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>0.36</td>
<td>0.36</td>
<td>0.43</td>
<td>0.43</td>
<td>0.43</td>
<td>0.43</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td>0.31</td>
<td>0.32</td>
<td>0.43</td>
<td>0.43</td>
<td>0.41</td>
<td>0.41</td>
</tr>
</tbody>
</table>
As shown in Table 6.3 and Table 6.4, for unsupervised predictors, LT and AGE, we get the exact same performance scores on all projects in terms of Recall and \( P_{opt} \). This is reasonable because unsupervised predictors are very straightforward and easy to implement. For the supervised predictor, EALR, these two implementations do not have differences in \( P_{opt} \), while the maximum difference in Recall is only 0.02. Since the differences are quite small, then we believe that our implementation reflects the details about EALR and unsupervised learners in Yang et al. [Yan16b].

For other supervised predictors used in this study, like J48, IBk, and Random Forests, we use the same algorithms from Weka package [Hal09] and set the same parameters as used in Yang et al. [Yan16b].

To answer this question, we build four supervised predictors and twelve unsupervised predictors on the six project data sets using incremental learning method as described in Section 6.5.3. Figure 6.2 shows the boxplot of Recall, \( P_{opt} \), \( F1 \) and Precision for supervised predictors and unsupervised predictors on all data sets. For each predictor, the boxplot shows the 25th percentile, median and 75 percentile values for one data set. The horizontal dashed lines indicate the median of the best supervised predictor, which is to help visualize the median differences between unsupervised predictors and supervised predictors.

The colors of the boxes within Figure 6.2 indicate the significant difference between learners:

- The **blue color** represents that the corresponding unsupervised predictor is significantly better than the best supervised predictor according to Wilcoxon signed-rank, where the BH corrected p-value is less than 0.05 and the magnitude of the difference between these two learners is NOT trivial according to Cliff’s delta, where \(|\delta| \geq 0.147\).

- The **black color** represents that the corresponding unsupervised predictor is not significantly better than the best supervised predictor or the magnitude of the difference between these two learners is trivial, where \(|\delta| \leq 0.147\).

- The **red color** represents that the corresponding unsupervised predictor is significantly worse than the best supervised predictor and the magnitude of the difference between these two learners is NOT trivial.

From Figure 6.2, we can clearly see that not all unsupervised predictors perform statistically better than the best supervised predictor across all different evaluation metrics. Specifically, for Recall, on one hand, there are only \( \frac{2}{12}, \frac{3}{12}, \frac{6}{12}, \frac{2}{12}, \frac{3}{12} \) and \( \frac{2}{12} \) of all unsupervised predictors that perform statistically better than the best supervised predictor on six data sets, respectively. On the other hand, there are \( \frac{6}{12}, \frac{6}{12}, \frac{4}{12}, \frac{6}{12}, \frac{5}{12} \) and \( \frac{6}{12} \) of all unsupervised predictors perform statistically worse than the best supervised predictor on the six data sets, respectively. This indicates that:

- About 50% of the unsupervised predictors perform worse than the best supervised predictor on any data set;
Figure 6.2 Performance comparisons between supervised and unsupervised predictors over six projects (from top to bottom are Bugzilla, Platform, Mozilla, JDT, Columba, PostgreSQL).
Table 6.5 Best unsupervised predictor (A) vs. OneWay (B). The colorful cell indicates the size effect: green for large; yellow for medium; gray for small.

<table>
<thead>
<tr>
<th>Project</th>
<th>Recall</th>
<th>$P_{opt}$</th>
<th>F1</th>
<th>Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A (LT)</td>
<td>B</td>
<td>A (LT)</td>
<td>B</td>
</tr>
<tr>
<td>Bugzilla</td>
<td>0.45</td>
<td>0.36</td>
<td>0.72</td>
<td>0.65</td>
</tr>
<tr>
<td>Platform</td>
<td>0.43</td>
<td>0.41</td>
<td>0.72</td>
<td>0.69</td>
</tr>
<tr>
<td>Mozilla</td>
<td>0.36</td>
<td>0.33</td>
<td>0.65</td>
<td>0.62</td>
</tr>
<tr>
<td>JDT</td>
<td>0.45</td>
<td>0.42</td>
<td>0.71</td>
<td>0.70</td>
</tr>
<tr>
<td>Columba</td>
<td>0.44</td>
<td>0.56</td>
<td>0.73</td>
<td>0.76</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>0.43</td>
<td>0.44</td>
<td>0.74</td>
<td>0.74</td>
</tr>
<tr>
<td>Average</td>
<td>0.43</td>
<td>0.42</td>
<td>0.71</td>
<td>0.69</td>
</tr>
</tbody>
</table>

- Without any prior knowledge, we can not know which unsupervised predictor(s) works adequately on the testing data.

Note that the above two points from Recall also hold for $P_{opt}$.

For $F1$, we see that only LT on Bugzilla and AGE on PostgreSQL perform statistically better than the best supervised predictor. Other than that, no unsupervised predictor performs better on any data set. Furthermore, surprisingly, no unsupervised predictor works significantly better than the best supervised predictor on any data sets in terms of Precision. As we can see, Random Forests performs well on all six data sets. This suggests that unsupervised predictors have very low precision for effort-aware defect prediction and can not be deployed to any business situation where precision is critical.

Overall, for a given data set, no one specific unsupervised predictor works better than the best supervised predictor across all evaluation metrics. For a given measure, most unsupervised predictors did not perform better across all data sets. In summary:

Not all unsupervised predictors perform better than supervised predictors for each project and for different evaluation measures.

Note the implications of this finding: some extra knowledge is required to prune the worse unsupervised models, such as the knowledge that can come from labeled data. Hence, we must conclude the opposite to Yang et al.; i.e. some supervised labeled data must be applied before we can reliably deploy unsupervised defect predictors on testing data.

RQ2: Is it beneficial to use supervised data to prune away all but one of the Yang et al. predictors?
Figure 6.3 Performance comparisons between the proposed OneWay learner and unsupervised predictors over six projects (from top to bottom are Bugzilla, Platform, Mozilla, JDT, Columba, PostgreSQL).
To answer this question, we compare the OneWay learner with all twelve unsupervised predictors. All these predictors are tested on the six project data sets using the same experiment scheme as we did in RQ1.

Figure 6.3 shows the boxplot for the performance distribution of unsupervised predictors and the proposed OneWay learner on six data sets across four evaluation measures. The horizontal dashed line denotes the median value of OneWay. Note that in Figure 6.4, blue means this learner is statistically better than OneWay, red means worse, and black means no difference. As we can see, in Recall, only one unsupervised predictor, LT, outperforms OneWay in $\frac{4}{6}$ data sets. However, OneWay significantly outperforms $\frac{9}{12}$, $\frac{9}{12}$, $\frac{9}{12}$, $\frac{9}{12}$, $\frac{9}{12}$, and $\frac{10}{12}$ of total unsupervised predictors on six data sets, respectively. This observation indicates that OneWay works significantly better than almost all learners on all 6 data sets in terms of Recall.

Similarly, we observe that only LT predictor works better than OneWay in $\frac{3}{6}$ data sets in terms of $P_{opt}$ and AGE outperforms OneWay only on the platform data set. For the remaining experiments, OneWay performs better than all the other predictors (on average, 9 out of 12 predictors).

In addition, according to $F1$, only three unsupervised predictors EXP/REXP/SEXP perform better than OneWay on the Mozilla data set and LT predictor just performs as well as OneWay (and has no advantage over OneWay). We note that similar findings can be observed in Precision measure.

Table 6.5 provides the median values of the best unsupervised predictor compared with OneWay for each evaluation measure on all data sets. Note that, in practice, we can not know which unsupervised predictor is the best out of the 12 unsupervised predictors by Yang et al.’s method before we access to the labels of testing data. In other words, to aid our analysis, the best unsupervised ones in Table 6.5 are selected when referring to the true labels of testing data, which are not available in practice. In that table, for each evaluation measure, the number in green cell indicates that the best unsupervised predictor has a large advantage over OneWay according to the Cliff’s $\delta$; Similarly, the yellow cell means medium advantage and the gray cell means small advantage.

From Table 6.5, we observe that out of 24 experiments on all evaluation measures, none of these best unsupervised predictors outperform OneWay with a large advantage according to the Cliff’s $\delta$. Specifically, according to Recall and $P_{opt}$, even though the best unsupervised predictor, LT, outperforms OneWay on four and three data sets, all of these advantage are small. Meanwhile, REXP and EXP have a medium improvement over OneWay on one and two data sets for $F1$ and Precision, respectively. In terms of the average scores, the maximum magnitude of the difference between the best unsupervised learner and OneWay is 0.02. In other words, OneWay is comparable with the best unsupervised predictors on all data sets for all evaluation measures even though the best unsupervised predictors might not be known before testing.

Overall, we find that (1) no one unsupervised predictor significantly outperforms OneWay on all data sets for a given evaluation measure; (2) mostly, OneWay works as well as the best unsupervised predictor and has significant better performance than almost all unsupervised predictors on all
Table 6.6 Best supervised predictor (A) vs. OneWay (B). The colorful cell indicates the size effect: green for large; yellow for medium; gray for small.

<table>
<thead>
<tr>
<th>Project</th>
<th>Recall</th>
<th>$P_{opt}$</th>
<th>$F_1$</th>
<th>Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A (EALR)</td>
<td>B</td>
<td>A (EALR)</td>
<td>B</td>
</tr>
<tr>
<td>Bugzilla</td>
<td>0.30</td>
<td>0.36</td>
<td>0.59</td>
<td>0.65</td>
</tr>
<tr>
<td>Platform</td>
<td>0.30</td>
<td>0.41</td>
<td>0.58</td>
<td>0.69</td>
</tr>
<tr>
<td>Mozilla</td>
<td>0.18</td>
<td>0.33</td>
<td>0.50</td>
<td>0.62</td>
</tr>
<tr>
<td>JDT</td>
<td>0.34</td>
<td>0.42</td>
<td>0.59</td>
<td>0.70</td>
</tr>
<tr>
<td>Columba</td>
<td>0.42</td>
<td>0.56</td>
<td>0.62</td>
<td>0.76</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>0.36</td>
<td>0.44</td>
<td>0.60</td>
<td>0.74</td>
</tr>
<tr>
<td>Average</td>
<td>0.32</td>
<td>0.42</td>
<td>0.58</td>
<td>0.69</td>
</tr>
</tbody>
</table>

As a simple supervised predictor, OneWay has competitive performance and it performs better than most unsupervised predictors for effort-aware JIT defect prediction.

Note the implications of this finding: the supervised learning utilized in OneWay can significantly outperform the unsupervised models.

Figure 6.4 Performance comparisons between the proposed OneWay learner and supervised predictors over six projects (from top to bottom are Bugzilla, Platform, Mozilla, JDT, Columba, PostgreSQL).
RQ3: Does OneWay perform better than more complex standard supervised predictors?

To answer this question, we compare OneWay learner with four supervised predictors, including EALR, Random Forests, J48 and IBk. EALR is considered to be state-of-the-art learner for effort-aware JIT defect prediction [Yan16b; Kam13] and all the other three learners are widely used in defect prediction literature over past years [Les08; Fu16a; Kam13; Fuk14; Tur09; Hal12]. We evaluate all these learners on the six project data sets using the same experiment scheme as we did in RQ1.

From Figure 6.4, we have the following observations. Firstly, the performance of OneWay is significantly better than all these four supervised predictors in terms of Recall and \( P_{opt} \) on all six data sets. Also, EALR works better than Random Forests, J48 and IBk, which is consistent with Kamei et al’s finding [Kam13].

Secondly, according to F1, Random Forests and IBk perform slightly better than OneWay in two out of six data sets. For most cases, OneWay has a similar performance to these supervised predictors and there is not much difference between them.

However, when reading Precision scores, we find that, in most cases, supervised learners perform significantly better than OneWay. Specifically, Random Forests, J48 and IBk outperform OneWay on all data sets and EALR is better on three data sets. This finding is consistent with the observation in RQ1 where all unsupervised predictors perform worse than supervised predictors for Precision.

From Table 6.6, we have the following observation. First of all, in terms of Recall and \( P_{opt} \), the maximum difference in median values between EALR and OneWay are 0.15 and 0.14, respectively, which are 83% and 23% improvements over 0.18 and 0.60 on Mozilla and PostgreSQL data sets. For both measures, OneWay improves the average scores by 0.1 and 0.11, which are 31% and 19% improvement over EALR. Secondly, according to F1, IBk outperforms OneWay on three data sets with a large, medium and small advantage, respectively. The largest difference in median is 0.1. Finally, as we discussed before, the best supervised predictor for Precision, Random Forests, has a very large advantage over OneWay on all data sets. The largest difference is 0.46 on PostgreSQL data set.

Overall, according to the above analysis, we conclude that:

OneWay performs significantly better than all four supervised learners in terms of Recall and \( P_{opt} \); It performs just as well as other learners for F1. As for Precision, other supervised predictors outperform OneWay.

Note the implications of this finding: simple tools like OneWay perform adequately but for all-around performance, more sophisticated learners are recommended.

As to when to use OneWay or supervised predictors like Random Forests, that is an open question. According to “No Free Lunch Theorems” [Wol02], no method is always best and we show unsupervised predictors are often worse on a project-by-project basis. So “best” predictor selection is a
matter of local assessment, requiring labelled training data (an issue ignored by Yang et al).

6.7 Threats to Validity

Internal Validity. The internal validity is related to uncontrolled aspects that may affect the experimental results. One threat to the internal validity is how well our implementation of unsupervised predictors could represent the Yang et al.’s method. To mitigate this threat, based on Yang et al “R” code, we strictly follow the approach described in Yang et al’s work and test our implementation on the same data sets as in Yang et al. [Yan16b]. By comparing the performance scores, we find that our implementation can generate the same results. Therefore, we believe we can avoid this threat.

External Validity. The external validity is related to the possibility to generalize our results. Our observations and conclusions from this study may not be generalized to other software projects. In this study, we use six widely used open source software project data as the subject. As all these software projects are written in Java, we can not guarantee that our findings can be directly generalized to other projects, specifically to the software that implemented in other programming languages. Therefore, the future work might include to verify our findings on other software project.

In this work, we used the data sets from [Yan16b; Kam13], where totally 14 change metrics were extracted from the software projects. We build and test the OneWay learner on those metrics as well. However, there might be some other metrics that not measured in these data sets that work well as indicators for defect prediction. For example, when the change was committed (e.g., morning, afternoon or evening), functionality of the the files modified in this change (e.g., core functionality or not). Those new metrics that are not explored in this study might improve the performance of our OneWay learner.

6.8 Conclusion and Future Work

This study replicated and refutes Yang et al.’s results [Yan16b] on unsupervised predictors for effort-ware just-in-time defect prediction. Not all unsupervised predictors work better than supervised predictors (on all six data sets, for different evaluation measures). This suggests that we can not randomly pick an unsupervised predictor to perform effort-ware JIT defect prediction. Rather, it is necessary to use supervised methods to pick best models before deploying them to a project. For that task, supervised predictors like OneWay are useful to automatically select the potential best model.

In the above, OneWay performed very well for Recall, P_{opt} and F1. Hence, it must be asked: “Is defect prediction inherently simple? And does it need anything other than OneWay?”. In this context, it is useful to recall that OneWay’s results for precision were not competitive. Hence we say, that if learners are to be deployed in domains where precision is critical, then OneWay is too simple.
This study opens the new research direction of applying simple supervised techniques to perform defect prediction. As shown in this study as well as Yang et al.’s work [Yan16b], instead of using traditional machine learning algorithms like J48 and Random Forests, simply sorting data according to one metric can be a good defect predictor model, at least for effort-aware just-in-time defect prediction. Therefore, we recommend the future defect prediction research should focus more on simple techniques.

For the future work, we plan to extend this study on other software projects, especially those developed by the other programming languages. After that, we plan to investigate new change metrics to see if that helps improve OneWay's performance.
Part IV

Future Work and Conclusion
This chapter is currently submitted to The ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE) 2018 titled “Building Better Quality Predictors Using ‘\( \varepsilon \)-Dominance’”. In this chapter, we show why and when software analytics can be simplified. The study and results shown in this chapter is very primary so that many further studies should be done to explore this area.

In previous chapters, we show that software analytics can be easier and we can simplify it from different perspectives for certain tasks. However, we still could not answer \textbf{when and why some simple methods work}. In this chapter, we apply Deb's principle of \( \varepsilon \)-dominance trying to understand the analytics tasks from result space. \( \varepsilon \)-dominance states that if there exists some \( \varepsilon \) value below which it is useless or impossible to distinguish results, then it is superfluous to explore anything less than \( \varepsilon \). We say that for “large \( \varepsilon \) problems”, the results space of learning effectively contains just a few regions. If many learners are then applied to such large \( \varepsilon \) problems, they would exhibit a “many roads lead to Rome” property; i.e., many different software quality prediction methods would generate a small set of very similar results.

In this chapter, we explore DART, an algorithm especially selected to succeed for large \( \varepsilon \) software quality prediction problems. DART is remarkable simple yet, on experimentation, it dramatically out-performs three sets of state-of-the-art defect prediction methods. The success of DART for defect prediction begs the questions: how many other domains in software quality predictors can also be radically simplified?
7.1 Introduction

This chapter presents DART, a novel method for simplifying supervised learning for defect prediction. DART produces tiny, easily comprehensible models (5 lines of very simple rules) and, in principle, DART could be applied to many domains in software quality predictors. When tested on software defect prediction, this method dramatically out-performs three recent state-of-the-art studies [Gho15; Fu16a; AM18].

DART was designed by working backward from known properties of software quality predictors problems. Such predictors exhibit a “many roads lead to Rome” property; i.e., many different data mining algorithms generate a small set of very similar results. For example, Lessmann et al. reported that 17 of 22 studied data mining algorithms for defect prediction had statistically indistinguishable performance [Les08]. Also, Ghotra et al. reported that the performance of 32 data mining algorithms for defect prediction clustered into just four groups [Gho15].

This chapter asks what can be learned from the above examples. In this chapter, we note that learners that have a “results space” i.e., values for various performance metrics such as recall and false alarm. Next, we ask what “shape” of result spaces leads to “many roads”? Also, given those “shapes”, do we need complex data miners? Or can we reverse engineer from that space a much simpler kind of software quality predictor?

To answer these questions we apply \( \epsilon \)-dominance [Deb05]. Deb's principle of \( \epsilon \)-dominance states that if there exists some \( \epsilon \) value below which it is useless or impossible to distinguish results, then

\[
\text{It is superfluous to explore anything less than } \epsilon .
\]

We say that for “large \( \epsilon \) problems”, the results space of learning effectively contains just a few regions. In such simple result spaces, a few DARTs thrown around the output space would sample the results just as well, or better, than more complex methods.

To test if \( \epsilon \)-dominance simplifies software quality prediction, this study compares DART-ing around the results space against three defect prediction systems:

1. The algorithms surveyed at a recent ICSE’15 paper [Gho15];

2. A hyper-parameter optimization method proposed in 2016 in the IST journal [Fu16a];

3. A search-based data pre-processing method presented at ICSE’18 [AM18].

These three were chosen since they reflect the state-of-the-art in software quality defect prediction. Also, the second and third items in this list are CPU-intensive systems that require days of computing time to execute data algorithms many times to find good configurations. Comparing something
as simple as DART to these complex systems let us critically assess the value of elaborate cloud computing environments for software quality prediction.

What we will see is that a small number of DARTs dramatically out-performs these three systems. This suggests that, at least for our data, much of the complexity associated with hyper-parameter optimization is not required. We conjecture that a few DARTs succeed so well since the results space for defect prediction exhibits the large $\epsilon$ property. We also conjecture that prior state-of-the-art algorithms fail against DART since all those models do not spread out over the results space. On the other hand, DART works so well since it knows how to spread its models across a large $\epsilon$ results space better.

The rest of this chapter is structured as follows. §7.2 introduces the SE case studies explored in this chapter (defect prediction) as well as different approaches and evaluation criteria. Secondly, it discusses the many sources of variability inherent in software quality predictors. In summary, between the raw data and the conclusions there are so many choices, some of which are stochastic (e.g., the random number generators that control test suite selection). All these choices introduce $\epsilon$, a degree of uncertainty in the conclusions. §7.3 discusses $\epsilon$-domination for software quality prediction and proposes DART, a straightforward ensemble method that can quickly sample a results space that divides into a few $\epsilon$-sized regions. §7.4 describes the experimental details in this study. §7.5 checks our conjecture. It will be seen that DART dramatically out-performs state-of-the-art defect prediction algorithms, hyper-parameter tuning algorithms, and data pre-processors.

Based on these results, we will argue in the conclusion that it is time to consider a fundamentally different approach to software quality prediction. Perhaps it is time to stop fretting about the numerous options available for selecting data pre-processing methods or machine learning algorithms, then configuring their controlling parameters. The results of this study suggest that most of those decisions are superfluous since so many methods result in the same output. Accordingly, we recommend doing something like DART; i.e. first reason about the results space before selecting an appropriate data mining technology.

One caveat on all these results is that paper has explored only one domain; i.e., software defect prediction. As to other domains, in as-yet-unpublished experiments, we have initial results suggesting that this simplification might also work elsewhere (e.g., text mining of programmer comments in Stack Overflow; for predicting Github issue close time; and for detecting programming bad smells). While those results are promising, they are still preliminary.

That said, the success of this simplification method for defect prediction begs the question: how many other domains in software quality prediction can also be radically simplified? This will be a fruitful direction for future work.
7.2 Background

7.2.1 Different Defect Prediction Approaches

Over the past decade, defect prediction has attracted many attentions from the software research community. There are many different types of defect predictors according to the metrics used for building models:

- **Module-level based defect predictors**, which use the complexity of software project, like McCabe metrics, Halstead’s effort metrics and CK object-oriented code metrics [CK94; KR87; McC76] of Table 1.1.

- **Just-in-time (JIT) defect prediction on change level**, which utilizes the change metrics collected from the software code [Kam13; Kim08; FM17c; MW00; Yan16b].

- The first two points represent much of the work in this area. For completeness, we add there are numerous other kinds of defect predictors based on many and varied other methods. For example, Ray et al. propose a defect prediction method using n-gram language models [Ray16]. Other work argues that process metrics are more important than the product metrics mentioned in the last two points [RD13].

### Table 7.1 32 defect predictors clustered by their performance rank by Ghotra et al. (using a Scott-Knot statistical test) [Gho15].

<table>
<thead>
<tr>
<th>Rank</th>
<th>Classification algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 (best)</td>
<td>Rsub+J48, SL, Rsub+SL, Bag+SL, LMT, RF+SL, Bag+LMT, Rsub+LMT, RF+LMT, RF+J48</td>
</tr>
<tr>
<td>2</td>
<td>RBFs, Bag+J48, Ad+SL, KNN, RF+NB, Ad+LMT, NB, Rsub+NB, Bag+NB</td>
</tr>
<tr>
<td>3</td>
<td>Ripper, J48, Ad+NB, Bag+SMO, EM, Ad+SMO, Ad+J48, K-means</td>
</tr>
<tr>
<td>4 (worst)</td>
<td>RF+SMO, Rsub+SMO, SMO, Ridor</td>
</tr>
</tbody>
</table>

Defect prediction models can be built via a variety of machine learning algorithms such as Decision Tree, Random Forests, SVM, Naive Bayes and Logistic Regression[KA01; KS03; Kho00; Men07b; Les08; Hal12]. Ghotra et al. [Gho15] compared various classifiers for defect prediction (for notes on a sample of those classifiers, see Table ??). According to their study, the prediction performances of classifiers group into the four clusters of Table 7.1. One advantage of this result is that, to sample across space of prior defect prediction work (including some state-of-the-art methods) researchers need only select one learner from each group.

To improve the performance of defect predictors, Fu et al. [Fu16a; Fu16b] and Tantithamthavorn et al. [Tan16] recommended improving standard typical defect predictors, like Random Forests
and CART by performing hyper-parameter tuning. Results from both research groups confirm that hyper-parameter tuning can dramatically supercharge the defect predictors. In other tuning work, Agrawal et al. [AM18] argued that better data is better than a better learner, where their results show that defect predictors can be improved a lot by changing the distribution of defective and non-defective examples seen during training.

For this study, to compare the performance of our proposed method, DART, with the state-of-the-art defect prediction methods, we picked one classification technique at random from each group of Table 7.1: SL, NB, EM, and SMO. Furthermore, we adopted techniques from Fu et al. [Fu16a; Fu16b] and Agrawal et al. [AM18] to investigate how DART performs compared to the improved (more sophisticated) defect prediction techniques. Note that Agrawal et al. also selected different classification techniques from Ghotra et al. study [Gho15; AM18]. Hence, using these classifiers, we can also compare our results to the experiments of Agrawal et al [AM18].

### 7.2.2 Evaluation Criteria

In defect prediction literature, once a learner is executed, the results must be scored. Recall measures the percentage of defective modules found by a model generated by the learner. False alarm reports how many non-defective modules the learner reports as defective. $P_{opt}$ measures how much code
some secondary quality assurance method would have to perform after the learner has terminated. $P_{opt}$ is defined as $1 - \Delta_{opt}$, where $\Delta_{opt}$ is the area between the effort (code-churn-based) cumulative lift charts of the optimal learner and the proposed learner (as shown in Figure 7.1). To calculate $P_{opt}$, we divide all the code modules into those predicted to be defective ($D$) or not ($N$). Both sets are then sorted in ascending order of lines of code. The two sorted sets are then laid out across the x-axis, with $D$ before $N$. This layout means that the x-axis extends from 0 to 100% where lower values of $x$ are predicted to be more defective than $x$ higher values.

On such a chart, the y-axis shows what percent of the defects would be recalled if we traverse the code sorted that x-axis order. According to Kamei et al. and Yang et al. [Yan16b; Kam13; Mon13], $P_{opt}$ should be normalized as follows:

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

where $S(optimal)$, $S(m)$ and $S(worst)$ represent the area of curve under the optimal learner, proposed learner, and worst learner, respectively. Note that the worst model is built by sorting all the changes according to the actual defect density in ascending order. For any learner, it performs better than random predictor only if the $P_{opt}$ is greater than 0.5.

Note that these measures are closely inter-connected. Recall appears as the dependent variable of $P_{opt}$. Also, false alarms result in flat regions of the $P_{opt}$ curve. Further, for useful learners, recall is
greater than the false alarm. Such learners have the characteristic shape of Figure 7.3.

In the following, we will assess results using $P_{opt}$ and another measure “distance to heaven” (denoted $dis2heaven$) that computes the distance of some recall, false alarm pair to the ideal “heaven” point of recall= 1 and false alarm= 0 as shown in Figure 7.2. This measure is defined in Equation 7.2:

$$dis2heaven = \frac{\sqrt{(1 - \text{recall})^2 + (\text{false alarm})^2}}{\sqrt{2}}$$  \hspace{1cm} (7.2)

The denominator of this equation means that $0 \leq dis2heaven \leq 1$. Note that:

- For $P_{opt}$, the larger values are better,
- For $dis2heaven$, the smaller values are better,

We use these measures instead of, say, precision or the F1 measure (harmonic mean of precision and recall) since Menzies et al. [Men07c] warn that precision can be very unstable for SE data (where the class distributions may be highly unbalanced).

### 7.2.3 Sources of ε Uncertainty

This section argues that there is inherent uncertainty in making conclusions via software quality predictors. The rest of this study exploits that uncertainty to simplify defect prediction.

Given that divergent nature of software projects and software developers, it is to be expected that different researchers find different effects, even from the same data sets [Hos17]. According to Menzies et al. [MS12], the conclusion uncertainty of software quality predictors come from different choices in the training data and many other factors.
**Sampling Bias:** Any data mining algorithm needs to input 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:

- The software is built by engineers with varying skills and experience.
- That construction process is performed using a wide range of languages and tools.
- The completed product is delivered on lots of platforms.
- All those languages, tools and platforms keep changing and evolving.
- 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 [Che05]. On the other hand, it can be helpful to prune data points that are very noisy or are outliers [Koc10]. The effects of pre-processing can be quite dramatic. For example, Agrawal et al. [AM18] report that their pre-processing technique (SMOTUNED) increased AUC and recall by 60% and 20%, respectively. 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 [WF11]. For all these stochastic algorithms, the conclusions are adjusted, to some extent, by the random numbers used in the processing.

Many methods have been proposed to reduce the above uncertainties such as feature selection to remove spurious outliers [Men07b], application of background knowledge to constrain model generation [FN12], optimizers to tune model parameters to reduce uncertainty [Agr18]. Despite this, some uncertainty $\epsilon$ usually remains (see an example, next section).

We conjecture that, for all the above reasons, uncertainty is an inherent property of software quality prediction. If so, the question becomes, “what to do with that uncertainty?” The starting
point for this study was the following speculation: **Instead of striving to make \( \epsilon = 0 \), use \( \epsilon > 0 \) as a tool for simplifying software quality predictors.** The next section describes such a tool.

### 7.3 \( \epsilon \)-Domination

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

Note that \( \epsilon \) effectively clusters the space of possible results. For example, consider the result space defined by recall \( r \) and false alarms \( f \). Both these measures have the range \( 0 \leq r \leq 1 \) and \( 0 \leq f \leq 1 \). If \( \epsilon = 0.2 \), then the results space of possible recalls and false alarms divides into the 5*5 grid of Figure 7.4.

Figure 7.3 showed that the results from useful learners have a characteristic shape where recall is greater than false alarms. That is, in Figure 7.4, such results avoid the red regions of that grid (where false alarms are higher than recall) and the gray regions (also called the “no-information” region where recall is the same as a false alarm).

This means that when \( \epsilon = 0.2 \), then (a) recall-vs-false alarm results space is effectively just the ten green cells of Figure 7.4; 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).

It turns out that real-world results spaces are more complicated than shown in Figure 7.4. For example, consider the results space of Figure 7.5. In this figure, 100 times, a defect predictor was
Figure 7.5 $\epsilon$ can vary across results space. 100 experiments with LUCENE results using 90% of the data, for training, and 10%, for testing. The x-axis sorts the code base, first according to predicted defective or not, then second on lines of code. The blue line shows the distribution of the 100 results across this space. For example, 70% of the results predict defect for up to 20% of the code (see the blue curve). The y-axis of this figure shows mean recall (in red); the standard deviation $\sigma$ of the recall (in yellow); and $\epsilon$ is defined as per standard t-tests that says at the 95% confidence level, two distributions differ when they are $\epsilon = 2 \times 1.96 \times \sigma$ apart. (in green).

built for LUCENE, an open-source Java text search engine. Random Forests was used to build the defect predictor using 90% of the data, then tested on the remaining 10% (Random Forests are a multi-tree classifier, widely used in defect prediction; see Table ??).

To compute $\epsilon$ in this results space, we divide the x-axis into divisions of 0.1 and report the standard deviation $\sigma$ of recall in each division. For the moment, we use a simple t-test to infer the separation required to distinguish two results within Figure 7.5 (later in this section, we will dispense with that assumption). This means that $\epsilon = 2 \times 1.96 \times \sigma$ which is the the range required to be 95% confident that two distributions are different [WF11].

The main result of Figure 7.5 is that $\epsilon$ is often very large. The blue curve shows that 70% of the results occur in the region $0 \leq LOC \leq 0.2$. At $LOC = 0.2$, $\epsilon = 0.2$; i.e., most of our models have an $\epsilon$ of 0.2 or higher. Note that learners with $\epsilon \geq 0.2$ divide into the 25 cells, or less, of Figure 7.4. More specifically, it means that most of the results of 100 learners applied to LUCENE would have statistically indistinguishable results.

From an analytic perspective, there are some limitations with the above analysis. Firstly, the threshold of $2 \times 1.96 \times \sigma$ is a simplistic measure of statistically significantly different results. It makes many assumptions that may not hold for SE data; e.g., that the data conforms to a parametric Gaussian distribution and that the variance of the two distributions is the same.

Secondly, as shown by the green curve of Figure 7.5, $\epsilon$ is not uniform across this result space. One reason for the lack of uniformity is that the results generated from 100 samples of the LUCENE data do not fall evenly across space: 70% of the 100 learned models fall far left of Figure 7.5 (up to 20% of the code—see the blue curve). This high variance means that we cannot reason about the results of space just via, e.g., some trite summary of the entire results space as a mean $\epsilon$ value.
When analytic methods fail, sampling can use instead. Rather than using $\varepsilon$ analytically, we instead use it to define a sampling method of the results space. That system, called DART is described in Figure 7.6. The algorithms work by DART-ing around results space, a couple of times. Note that if the results exhibit a large $\varepsilon$ properties, then these few samples would be enough to cover the ten green cells of Figure 7.4. Also, the results from such DART-ing around should perform as well as anything else, including the three state-of-the-art systems listed in the introduction.

To operationalize DART, we wrote some Python code based on the Fast-and-frugal tree (FFT) R-package from Phillips et al. [Phi17]. While this is not the only way to operationalize DART, it worked so well for this study; we were not motivated to try alternatives. An FFT is a binary tree where, at each level, there is one exit node predicting either for “true” for target class or “false”. Also, at the bottom of the tree, there are two leaves exiting to “true” and “false”. For example, from the Log4j dataset of Table 7.2, one tree predicting for software defects is shown in Figure. 7.7. Note that this tree has decided to exit towards the target class at lines 2, 3, 4 and otherwise on lines 1, 5.

To build one tree, our version of DART discretize numerics by dividing at median values; then scores each range using $\text{dis2heaven}$ or $P_{\text{opt}}$ according to how well they predict for “true” or “false” (this finds the extreme ranges seen in DART’s training step (2)). Next we built one level of the tree by (a) picking the exit class then (b) adding in the range that best predicts for that class. The other levels are build recursively using the data not selected by that range. Given a tree of depth $d$, there are two choices at each level about whether or not to exit to “true” or “false”. Hence, for trees of depth $d$, there are $M = 2^d$ possible trees. Each such tree is one “dart” into results space.

To throw several darts at results space, DART builds an ensemble of 16 trees, we use depth $d = 4$. This number was selected since Figure 7.4 had ten green cells. Hence: $d = 3$ would generate $2^3 = 8$ trees which would not be enough to cover results space; $d = 5$ would generate $2^5 = 32$ trees which would be excessive for results like Figure 7.4. Note that, when using this approach, the number of extreme ranges used in the models is the same as the depth of the tree $N = d = 4$. As per Figure 7.6, on the training data shown in Table 7.2, we built 16 trees, then selected the best one to be used for testing.

### 7.4 Experimental Setup

Recalling §7.2.2, the evaluation criteria used in this study was $\text{dis2heaven}$ or $P_{\text{opt}}$. Note that this criteria also echoes the criteria seen in prior work [FM17c; Kam13; Yan16b]. The rest of this section discusses our other experimental details.
INPUT:
- A dataset, such as Table 7.2;
- A goal predicate \( p \); e.g., \( P_{opt} \) or \( \text{dis2Heaven} \);
- \( M, N \) = 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 of 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 7.6 DART: an ensemble algorithm to sample results space, \( M \) number of times.

<table>
<thead>
<tr>
<th>Case</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{cob} \leq 4 )</td>
<td>false</td>
</tr>
<tr>
<td>( \text{rfc} &gt; 32 )</td>
<td>true</td>
</tr>
<tr>
<td>( \text{dam} &gt; 0 )</td>
<td>true</td>
</tr>
<tr>
<td>( \text{amc} &lt; 32.25 )</td>
<td>true</td>
</tr>
<tr>
<td>false</td>
<td></td>
</tr>
</tbody>
</table>

Figure 7.7 A simple model for software defect prediction

7.4.1 Research Questions

To compare with three established defect prediction methods, we use all machine learning implementations from Scikit-learn package and tools released by Fu et al. [Fu16a] and Agrawal et al. [Che]. In this study, we set three research questions:
RQ1: Do established learners sample results space better than a few DARTs? This questions compares DART against the sample of defect prediction algorithms surveyed by Ghotra et al. at ICSE’15 [Gho15].

RQ2: Do goal-savvy learners sample results space better than a few DARTs? This question addresses a potential problem with the RQ1 analysis. DART uses the goal function when it trains its models. Hence, this might give DART an unfair advantage compared to other learners in Table 7.3. Therefore, in RQ2, we compare DART to goal-savvy hyper-parameter optimizers [Fu16a] that make extensive use of the goal function as they tune learner parameters.

RQ3: Do data-savvy learners sample results space better than a few DARTs? Agrawal et al. [Che] argues that selecting and/or tuning data miners is less useful that repairing problems with the training data. To test that, this research question compares DART against the data-savvy methods developed by Agrawal et al.

7.4.2 Datasets

To compare the DART ensemble method against alternate approaches, we used data from SEACRAFT repository (tiny.cc/seacraft), shown in Table 7.2 (for details on the contents of those data sets, see Table 1.1). This data was selected for two reasons:

- The data is available for multiple versions of the same software. This means we can ensure that our learners are trained on past data and tested on future data.

- It is very similar, or identical, to the data used in prior work against which we will compare our new approach [Gho15; Fu16a; AM18].

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.

7.5 Results

7.5.1 RQ1: Do established learners sample results space better than a few DARTs?

In order to compare our approach to established norms in software quality predictors, we used the Ghotra et al. study from ICSE’15 [Gho15]. Recall that this study was a comparison of the the 32
Table 7.2 Statistics of the studied data sets.

<table>
<thead>
<tr>
<th>Project</th>
<th>Training Data</th>
<th>Testing Data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Versions</td>
<td>% of Defects</td>
</tr>
<tr>
<td>Poi</td>
<td>1.5, 2.0, 2.5</td>
<td>426/936 = 46%</td>
</tr>
<tr>
<td>Lucene</td>
<td>2.0, 2.2</td>
<td>235/442 = 53%</td>
</tr>
<tr>
<td>Camel</td>
<td>1.0, 1.2, 1.4</td>
<td>374/1819 = 21%</td>
</tr>
<tr>
<td>Log4j</td>
<td>1.0, 1.1</td>
<td>71/244 = 29%</td>
</tr>
<tr>
<td>Xerces</td>
<td>1.2, 1.3</td>
<td>140/893 = 16%</td>
</tr>
<tr>
<td>Velocity</td>
<td>1.4, 1.5</td>
<td>289/410 = 70%</td>
</tr>
<tr>
<td>Xalan</td>
<td>2.4, 2.5, 2.6</td>
<td>908/2411 = 38%</td>
</tr>
<tr>
<td>Ivy</td>
<td>1.1, 1.4</td>
<td>79/352 = 22%</td>
</tr>
<tr>
<td>Synapse</td>
<td>1.0, 1.1</td>
<td>76/379 = 20%</td>
</tr>
<tr>
<td>Jedit</td>
<td>3.2, 4.0, 4.1, 4.2</td>
<td>292/1257 = 23%</td>
</tr>
</tbody>
</table>

Table 7.3 DART v.s. state-of-the-art defect predictors from Ghotra et al. [Gho15] for dis2heaven and $P_{opt}$. Gray cells mark best performances on each project (so DART is most-often best).

<table>
<thead>
<tr>
<th>Goal</th>
<th>Data</th>
<th>DART</th>
<th>SL</th>
<th>NB</th>
<th>EM</th>
<th>SMO</th>
</tr>
</thead>
<tbody>
<tr>
<td>dis2heaven (less is better)</td>
<td>log4j</td>
<td>23</td>
<td>53</td>
<td>51</td>
<td>56</td>
<td>48</td>
</tr>
<tr>
<td></td>
<td>jedit</td>
<td>31</td>
<td>40</td>
<td>41</td>
<td>34</td>
<td>47</td>
</tr>
<tr>
<td></td>
<td>lucene</td>
<td>33</td>
<td>40</td>
<td>44</td>
<td>44</td>
<td>71</td>
</tr>
<tr>
<td></td>
<td>poi</td>
<td>35</td>
<td>36</td>
<td>57</td>
<td>70</td>
<td>45</td>
</tr>
<tr>
<td></td>
<td>ivy</td>
<td>35</td>
<td>50</td>
<td>40</td>
<td>71</td>
<td>43</td>
</tr>
<tr>
<td></td>
<td>velocity</td>
<td>37</td>
<td>61</td>
<td>40</td>
<td>49</td>
<td>60</td>
</tr>
<tr>
<td></td>
<td>synapse</td>
<td>38</td>
<td>51</td>
<td>39</td>
<td>34</td>
<td>62</td>
</tr>
<tr>
<td></td>
<td>xalan</td>
<td>39</td>
<td>55</td>
<td>55</td>
<td>70</td>
<td>68</td>
</tr>
<tr>
<td></td>
<td>camel</td>
<td>41</td>
<td>60</td>
<td>52</td>
<td>44</td>
<td>71</td>
</tr>
<tr>
<td></td>
<td>xerces</td>
<td>42</td>
<td>68</td>
<td>60</td>
<td>50</td>
<td>69</td>
</tr>
<tr>
<td>$P_{opt}$ (more is better)</td>
<td>ivy</td>
<td>28</td>
<td>17</td>
<td>9</td>
<td>28</td>
<td>23</td>
</tr>
<tr>
<td></td>
<td>jedit</td>
<td>39</td>
<td>10</td>
<td>9</td>
<td>16</td>
<td>17</td>
</tr>
<tr>
<td></td>
<td>synapse</td>
<td>43</td>
<td>26</td>
<td>24</td>
<td>22</td>
<td>22</td>
</tr>
<tr>
<td></td>
<td>camel</td>
<td>53</td>
<td>15</td>
<td>17</td>
<td>16</td>
<td>50</td>
</tr>
<tr>
<td></td>
<td>log4j</td>
<td>56</td>
<td>19</td>
<td>22</td>
<td>16</td>
<td>23</td>
</tr>
<tr>
<td></td>
<td>velocity</td>
<td>64</td>
<td>64</td>
<td>64</td>
<td>24</td>
<td>60</td>
</tr>
<tr>
<td></td>
<td>poi</td>
<td>73</td>
<td>51</td>
<td>19</td>
<td>33</td>
<td>64</td>
</tr>
<tr>
<td></td>
<td>lucene</td>
<td>81</td>
<td>43</td>
<td>27</td>
<td>20</td>
<td>80</td>
</tr>
<tr>
<td></td>
<td>xerces</td>
<td>90</td>
<td>4</td>
<td>9</td>
<td>15</td>
<td>48</td>
</tr>
<tr>
<td></td>
<td>xalan</td>
<td>99</td>
<td>11</td>
<td>15</td>
<td>100</td>
<td>51</td>
</tr>
</tbody>
</table>

learners shown in Table 7.1. The performance of those learners clustered into four groups, from
which we selected four representative learners (see the discussion in §7.2.1): SL, NB, EM, SMO.

For this comparison, DART and the learners from Ghotra et al. were all trained/tested on the same versions shown in Table 7.2. The resulting performance scores are shown in Table 7.3. Note that:

- DART performed as well, or better, than the sample of Ghotra et al learners in 18/20 experiments.

- When DART failed to produce best performance, it came very close to the best (e.g., for Xalan's $P_{opt}$ results, DART scored 99% while the best was 100%).

- When DART performed best, it often did so by a very wide margin. For example, for Log4j’s dis2heaven score, DART’s score was 23% and the best value of the other learners was 51%; i.e., worse by more than a factor of two. For another example, for Log4j’s $P_{opt}$ score, DART’s score was more than twice better than the scores of any other learner.

From these results, we assert that DART out-performs the established state-of-the-art defect predictors recommended by Ghotra et al. [Gho15] on the data sets of Table 7.2.

7.5.2 **RQ2: Do goal-savvy learners sample results space better than a few DARTs?**

One counter argument to the conclusions of the RQ1 is that it may not be fair to compare DART against standard data mining algorithms using their off-the-shelf parameter tunings. DART makes extensive use of the goal function $p$ (i.e., $P_{opt}$ and dis2heaven) at three points in its algorithm:

- Once when assessing individual ranges;

- Once again when assessing trees built from those ranges;

- A third time when assessing the best tree on the test set.

All the other learners in Table 7.3 use $p$ only once (when their final model was assessed) but never while they build their models. That is, DART is “goal-savvy” while all the other learners explored in RQ1 were not. Perhaps this gave DART an unfair advantage?

To address this issue, we turned to the state-of-the-art in hyper-parameter optimization for defect prediction. In 2016, Fu et al. presented in the IST journal [Fu16a] an extensive study where an optimizer tuned the control parameters of various learners applied to software quality defect prediction. That study used the goal function $p$ to guide their selection of control parameters; i.e., unlike the Table 7.3 results, this learning method is “goal-savvy” in the sense that it was allowed to reflect on the goal during model generation.
For RQ2, we compare the performance of DART with goal-savvy tuning Random Forests. We use RandomForests since they were recommended by Ghotra et al. [Gho15] and prior work hyper-parameter tuning for defect prediction by Fu et al. [Fu16a]. In this hyper-parameter tuning experiment, for each project data, we randomly split the original training data (e.g, combine poi1.5, poi2.0 and poi2.5 as the original training data) into 80% and 20% as new training data and tuning data, respectively. As recommended by Fu et al. [Fu16a], we use the TUNER algorithm of Figure 7.8 to select the parameters of Random Forests (for a list of those parameters, see Table 7.4). TUNER iterates until it runs out of tuning resources (i.e., a given tuning budget) or it cannot find any better hyper-parameters. Finally, we use the current best parameters as the best hyper-parameters to train Random Forests with the new training data.

Since different data split might have an impact on predictor's performance, we repeat tuning+testing process 30 times, each time with different random seed and return the median values of 30 runs as the result of tuning Random Forests experiment on each project data. Since we have two different goals: minimize dis2heaven and maximize P_{opt} metrics, we run two different experiments (so 60 repeats in all).

For comparison purposes, DART built its ensembles on the original training data (e.g, combine poi1.5, poi2.0 and poi2.5 as the original training data) and selected the best tree in terms of p (i.e., P_{opt} or dis2heaven). The best tree was then tested on testing data (e.g., poi3.0). Table 7.5 shows the results of DART versus Random Forests, where the latter was tuned for dis2heaven or P_{opt}. Note that in this experiment, DART was not tuned. Rather, it just used its default settings of:

- Discretizing using median splits for numeric attributes;
- Building trees of depth d = 4, which means ensembles of size 2^d = 16;
- At each level of tree, use just one extreme range.

<table>
<thead>
<tr>
<th>Tuning Object</th>
<th>Parameters</th>
<th>Default</th>
<th>Tuning Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random Forests</td>
<td>threshold</td>
<td>0.5</td>
<td>[0.01,1]</td>
<td>The value to determine defective or not.</td>
</tr>
<tr>
<td></td>
<td>max_feature</td>
<td>None</td>
<td>[0.01,1]</td>
<td>The number of features to consider when looking for the best split.</td>
</tr>
<tr>
<td></td>
<td>max_leaf_nodes</td>
<td>None</td>
<td>[1,50]</td>
<td>Grow trees with max_leaf_nodes in best-first fashion.</td>
</tr>
<tr>
<td></td>
<td>min_sample_split</td>
<td>2</td>
<td>[2,20]</td>
<td>The minimum number of samples required to split an internal node.</td>
</tr>
<tr>
<td></td>
<td>min_samples_leaf</td>
<td>1</td>
<td>[1,20]</td>
<td>The minimum number of samples required to be at a leaf node.</td>
</tr>
<tr>
<td></td>
<td>n_estimators</td>
<td>100</td>
<td>[50,150]</td>
<td>The number of trees in the forest.</td>
</tr>
<tr>
<td>SMOTE</td>
<td>k</td>
<td>5</td>
<td>[1,20]</td>
<td>Number of neighbors</td>
</tr>
<tr>
<td></td>
<td>m</td>
<td>50%</td>
<td>50, 100, 200, 400</td>
<td>Number of synthetic examples to create. Expressed as a percent of final training data.</td>
</tr>
<tr>
<td></td>
<td>r</td>
<td>2</td>
<td>[0.1, 5]</td>
<td>Power parameter for the Minkowski distance metric.</td>
</tr>
</tbody>
</table>
INPUT:
• A dataset, such as Table 7.2;
• A tuning goal G; e.g., $P_{opt}$ or dis2Heaven;
• DE parameters: $np = 10$, $f = 0.75$, $cr = 0.3$, $life = 5$

OUTPUT:
• Best tunings for learners (e.g., RF) found by DE

PROCEDURE:
• Separate the data into train and tune;
• Generate $np$ tunings as the initial population;
• Score each tuning $pop_i$ in the population with goal $G$;
• For $i = 1$ to $np$ do
  1. Generate a mutant $m$ built by extrapolating between three other members of population $a$, $b$, $c$ at probability $cr$. For each decision $m_k \in m$:
     - $m_k = a_k + f \times (b_k - c_k)$ (continuous values).
     - $m_k = a_k \lor (b_k \lor c_k)$ (discrete values).
  2. Build a learner with parameters $m$ and train data;
  3. Score $m$ on tune data using $G$;
  4. Replace $pop_i$ with $m$ if $m$ is preferred;
• Repeat the last step until run out of $life$ or could not find better tunings;
• Return the best tuning $pop_i$ of the last population as the final result.

Figure 7.8 TUNER is an evolutionary optimization algorithm based on Storn’s differential evolution algorithm [SP97; FM17b].

As shown in Table 7.5, for 13/20 experiments, untuned DART performed better than tuned RandomForests. Also, in all cases where DART performed worse, the performance delta was very small (the largest loss was 4% seen in the xerces’ $P_{opt}$ results).

From these results, we assert that DART out-performs the established state-of-the-art in parameter tuning for defect prediction. This is an interesting result since TUNER must evaluate dozens to hundreds of different models before it can select the best settings. DART, on the other hand, just had to build one ensemble, then test one tree from that ensemble.

7.5.3 RQ3: Do data-savvy learners sample results space better than a few DARTs?

There has been much recent research in hyper-parameter tuning of quality predictors. For example:

• The Fu et al. study mentioned in RQ2 tuned control parameters of the learning algorithm.
Table 7.5 DART v.s. tuning Random Forests for dis2heaven and $P_{opt}$. Gray cells mark best performances on each project. Note that even when DART does not perform best, it usually performs very close to the best.

<table>
<thead>
<tr>
<th>Data</th>
<th>dis2heaven (less is better)</th>
<th>$P_{opt}$ (more is better)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>DART</td>
<td>Tuning RF</td>
</tr>
<tr>
<td>ivy</td>
<td>35</td>
<td>56</td>
</tr>
<tr>
<td>jedit</td>
<td>31</td>
<td>35</td>
</tr>
<tr>
<td>synapse</td>
<td>38</td>
<td>57</td>
</tr>
<tr>
<td>camel</td>
<td>41</td>
<td>70</td>
</tr>
<tr>
<td>log4j</td>
<td>23</td>
<td>51</td>
</tr>
<tr>
<td>velocity</td>
<td>37</td>
<td>53</td>
</tr>
<tr>
<td>poi</td>
<td>34.8</td>
<td>27</td>
</tr>
<tr>
<td>lucene</td>
<td>33</td>
<td>35</td>
</tr>
<tr>
<td>xerces</td>
<td>42</td>
<td>70</td>
</tr>
<tr>
<td>xalan</td>
<td>38.7</td>
<td>36</td>
</tr>
</tbody>
</table>

Table 7.6 DART v.s. data-savvy learners in dis2heaven and $P_{opt}$. Gray cells mark best performances on each project.

<table>
<thead>
<tr>
<th>Goal</th>
<th>Data</th>
<th>DART</th>
<th>KNN</th>
<th>SMO</th>
<th>NB</th>
<th>RF</th>
<th>SL</th>
<th>DT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

122
• At ICSE’18, Agrawal et al. [AM18], applied tuning to the data pre-processor that was called before the learners executed.

This section compares DART to the Agrawal et al. methods. Note that the Fu et al. tuners were “goal-savvy”, the Agrawal et al. methods are “data-savvy”.

Agrawal compared the benefits of (a) selecting better learners versus (b) picking any learner but also addressing class-imbalance in the training data. Class-imbalance is a major problem in quality prediction. If the target class is very rare, 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 of Table 7.4. Agrawal et al. applied the same TUNER algorithm of Fu et al. and found that the default settings of SMOTE could be greatly improved. Agrawal et al. used the term SMOTUNED to denote their combination.

This section compares DART against SMOTUNED. As per the methods of Agrawal et al. [AM18], for each project of Table 7.2, we randomly split the original training data into 80% and 20% as new training data and tuning data, respectively. The tuning data was used to validate our parameter settings of SMOTE found by TUNER.

Similar to the RQ2 experiment, we repeated the whole process until we either run out of tuning resources (i.e., a given tuning budget) or TUNER could not find any better hyper-parameters. The best parameters found was tested against the testing set and these results are reported.

Since different data split might have an impact on predictor’s performance, we repeat tuning+testing process 30 times, each time with different random seed and return the median values of 30 runs as the result of tuning SMOTE on each project data. SMOTUNED experiment was run twice to minimize distance2heaven and maximize popt20 metrics. These two experiments were run separately.

Table 7.6 shows the results. In results that echo all the above, usually DART performs much better than the more elaborate approach of Agrawal et al:

• For the $P_{opt}$ results, DART was either the best result or no worse that 1% off the best results;

• As to dis2heaven, DART’s worst performance was for xalan, which was was 19% worse that best. Apart from that, DART either had the best result or was within 2% of the best result for $\frac{g}{10}$ of the results.

7.6 Threads to Validity

As with any large scale empirical study, biases can affect the final results. Therefore, any conclusions made from this work must be considered with the following issues in mind:
Threats to **internal validity** concern the consistency of the results obtained from the result. In our study, to investigate how DART performs compared with the state-of-the-art defect predictors, goal-savvy defect predictors, and data-savvy defect predictors, we have taken care to either clearly define our algorithms or use implementations from the public domain. All the machine learning algorithms are imported from Scikit-Learn, a machine learning package in Python [Ped11]. For example, in RQ1, DART followed the FFTs algorithm defined in [Mar03]. In RQ2 and RQ3, we adopt the original source code of DE-TUNER and SMOTUNED provided by Fu et al [Fu16a] and Agrawal et. al. [AM18], which reduce the bias introduced by implementing the rigs by ourselves. All the data used in this work is widely used open source Java system data in defect prediction field and it is also available in the SEACRAFT repository (http://tiny.cc/seacraft).

Threats to **external validity** represent if the results are of relevance for other cases, or the ability to generalize the observations in a study. In this study, we proposed that using DART as a scout to explore the results space could build better defect predictors in terms of \( \text{dis2heaven} \) and \( P_{\text{opt}} \) measures. Nonetheless, we do not claim that our findings can be generalized to all software quality predictors tasks. However, those other software quality predictors tasks often apply machine learning algorithms, like SVM and Random Forests, or other data pre-processing techniques to build predictive models. Most of those models are also exploring the results space and find the best models. Therefore, it is quite possible that FFTs method of this study would be widely applicable, elsewhere.

### 7.7 Discussion and Future Work

The thesis of this chapter is that we have been treating uncertainty incorrectly. Instead of viewing uncertainty as a problems to be solved, we instead view it as a resource that simplifies software defect prediction.

For example, Deb’s principle of \( \epsilon \)-dominance states that if there exists some \( \epsilon \) value below which it is useless or impossible to distinguish results, then it is superfluous to explore anything less than \( \epsilon \). For large \( \epsilon \) problems, the results space effectively contains just a few regions.

As shown here, there are several important benefits if we we design a learner especially for such large \( \epsilon \) problems. Firstly, the resulting learner is very simple to implement. Secondly, this learner can sample the results space very effectively. Thirdly, that very simple learner can out-perform far more elaborate systems such as three state-of-the-art defect prediction systems:

1. The algorithms surveyed at a recent ICSE’15 paper [Gho15];
2. A hyper-parameter optimization method proposed in 2016 in the IST journal [Fu16a];
3. A search-based data pre-processing method presented at ICSE’18 [AM18].
We believe that our results call for a new approach to software quality prediction. The standard approach to this problem, as shown by the top pink section of Figure 7.9 is to reason forwards from domain data, towards a model. In that approach, analysts must make many decisions about data pre-processing, feature selection, and tuning parameters for a learner. This is a very large number of decisions:

- Data can be processed by SMOTE [Cha02], SMOTUNED [AM18], or any number of other methods including normalization, discretization, outlier removal, etc [WF11];
- Feature selection can explore $2^N$ subsets of $N$ features;
- Hyper-parameter optimization explores the space of control parameters within data mining algorithms. As shown in Table 7.4, those parameters can be continuous which means the space of parameters is theoretically infinite.

Perhaps a much simpler approach is the backwards reasoning shown in the blue bottom region of Figure 7.9. In this approach, analysts do some initial data mining, perhaps at random, then reflect on what has been learned from those initial probes of the result space. Based on those results, analysts then design a software quality predictor that better understands the results space.

The DART system discussed in this chapter is an example of such backwards reasoning. We hope the success of this system inspires other researchers to explore large scale simplifications of other SE problem domains.

In this chapter, we show that a simple scout, like DART, can better explore the results space and therefore, we get a simple but effective method to build quality predictor. This study as well as all my previous studies open the door for Exploring Simple Software Analytics. Even though we might
have large amount of computing resources, those unnecessary wasted resource could be saved if we have simple and effective tools for software analytics.

As the thesis of this dissertation, **Software analytics should be easier, Software analytics can be easier, But it can be very hard to show it can be easier. Sometimes, it can be too easy.** We would like to encourage more and more software engineering researchers to focus on designing simple but effective tools for software analytics in terms of *effectiveness, economy, explainability* and *reproducibility*. However, in this direction, there are many open questions need to be addressed: what kind of software analytics can be solved by using simple methods? How to avoid over-simplify software analytics so that it works in practice? In our preliminary study about “$\epsilon$-domination”, we find that for larger “$\epsilon$” problems, we can use simple methods to sample results space. However, we didn't theoretically define larger “$\epsilon$” problems. Since software engineering data does not conform to Gaussian distribution and the variance of two distributions might change, We have to find other ways to define $\epsilon$. On the other hand, we only did experiment with “$\epsilon$-domination” on software quality predictors and we didn't know whether such idea can be generalized to other software analytics, like text mining [FM17b; Agr18]. Finally, we can make no claim that DART is the *best* scout for *all* software analytics tasks. Rather, our point is that there exists at least some software analytics problems whose results space can be easily sampled by at least one simple scout, like DART. We hope that this work inspires much future work as this community develops and debugs best practices for simple software analytics.
With a huge amount of data generated from software development process, software engineering researchers and practitioner have started to utilize the data analytics tools to get insights from the data they have about their projects. These software analytics tasks, like software quality prediction, software development effort estimation and software bug report analysis, have been actively investigated by software community. Most of these research goals are to improve the performance of models in terms of accuracy scores. However, one missing point of these research works is that researchers passionately improve the performance scores by increasing the complexity of their methods, which mostly means additional work on data preprocessing, feature selection, hyper-parameter tuning, or even use complicated methods like deep learning, but rarely consider the trade-off between cost and performance. To make it even worse, many recent researcher propose to apply deep learning on software analytics without reasoning whether the improvement brought by such resource-intensive method is worthy on the target tasks.

In this dissertation, we want to emphasize that software analytics should be simpler. It should be explainable, actionable, and less resource required and should be able to fit the fast iteration development cycle. In this sense, complex methods, like deep learning, is definitely not the best match on most software analytics because such methods require days or weeks of GPU/CPU time to train the model and the resulting models are not easily to explain to software developers when deploying such models to make predictions and recommendations. On the other hand, huge amount of computational resource should not be reason why software analytic tools could not be reproduced.
That said, we have to promote simple and effective methods for software analytics.

In Part II, we show firstly that software analytics tools, like software quality predictors, can be improved using simple search-based software engineering method, like Differential Evolution in Chapter 3. According to our experiment results, we find that tuning can quickly find tunings that improve defect detection precision from 0% to 60%. Since the improvements are so large and tuning with DE is so simple, we recommend that for future software analytics, all the data learners should be carefully tuned by using local data. Furthermore, in Chapter 4, we investigated which hyper-parameter techniques can find better parameters with less computational resources. Specifically, comparing differential evolution and grid search, we found that such evolutionary method is much faster than the seemingly more thorough grid search. In terms of computational resource, differential evolution algorithm requires 210 times less than grid search. This suggests that simple tuning method could work better and it is important to match the optimization algorithm to the problem.

In Chapter 5, to show that simpler methods can reach parity with complex methods with much lower execution costs and less understanding barriers, we introduced a study where we spent months of research time to collect data and reproduce a published work on software analytics using a complex method. In this study, the original authors proposed to apply deep learning method to text classification tasks on stack overflow. However, in stead of using complex method, we proposed to tune simple learners, like SVM, with differential evolution algorithm. By strictly following the original authors’ description, we successfully reproduced their baseline results. Furthermore, we find that simple method, like DE tuning SVM, got better performance than resource-intensive deep learning method and our method is 80 times faster. By exploring these studies, we want to defend that software analytics can be simpler. However, how many other software analytics can be simplified? that is an open research question for future work.

In Part III, we show a study where over-simplification can also introduce troubles if overlooking details about software analytics. In this study, we firstly reproduced a study about building unsupervised predictors for just-in-time defect prediction based on code change level. However, we find that there is much variability in the efficacy of simple unsupervised defect predictors proposed by the original authors. Their methods can not get a consistent prediction among all proposed unsupervised learners. To improve their method, we proposed to use local data to prune some weak predictors and only select the best one as the final predictor. Experimental results show that our proposed simple method, OneWay, performs better than the unsupervised learners as well as more complex standard supervised learners. This study opens the new research direction of applying simple supervised techniques to perform defect prediction. As shown in this study as well as Yang et al.’s work [Yan16b], instead of using traditional machine learning algorithms like J48 and Random Forests, simply sorting data according to one metric can be a good defect predictor model, at least for effort-aware just-in-time defect prediction. According to this study, we defend
that oversimplification can be harmful.

Overall, since we have shown that simpler software analytics methods work pretty well on tasks like software defect prediction and text classification tasks, we recommend software analytics researchers should focus more on simple techniques in the future.
BIBLIOGRAPHY


