

## ABSTRACT

NARAYANASWAMY CHANDRASEKARAN, SHRIKANTH. Taming Confusions in Software Engineering. (Under the direction of Dr. Timothy Menzies).

Software Engineering (SE) researchers and practitioners struggle to learn effective quality assurance practices from project data. A repeated result is that data is processed poorly. As a result, developers make contradictory conclusions about what most effects (e.g.) defects to development time. Also, researchers keep relying on decades-old truisms about software development, even when data does not support those truisms.

This thesis (a) documents those contradictions, (b) explains their cause and, using that explanation, (c) shows how to improve and simplify software analytics.

In PART1, this thesis documents many contradictions. The first study explores developer beliefs and compares stated developer beliefs with the empirical evidence from 37 Open Source projects. Only two of ten beliefs had strong support across all the projects. Worse yet, most widely-held beliefs studied are only sporadically supported in the data; i.e., large effects can appear in project data and then disappear in subsequent releases. In the second study, researcher beliefs are explored, and decades-old truisms about software development against industry data collected from 1,356 developers from 1995 to 2006 are checked. That analysis showed support for only one of the five beliefs titled “quality entails productivity”, but not for others.

In PART2, the disconnect between beliefs and data is elucidated. Here, a third study explores the temporal pattern of data within a project. This work finds a previously unreported effect: (a) much of the quality-relevant experience in a project happens very early in its life cycle; (b) yet developers and researchers persist in reasoning across the entire life cycle (even when the relevant experience is scarce). Hence it is hardly surprising that developers and researchers make the wrong conclusions (since they are reasoning about only a small fraction of the relevant data). Also, in part two, this thesis reports that the quality model learned very early in the life cycle (after only 150 commits) work as well as anything else. This means that models can be learned quickly (using only a small sample of data). Notably, the conclusions of that early data-lite model are stable across the rest of the life cycle. In other words, overwhelming the experience with other analytics would lead to *unstable* conclusions.

In PART3, the early data-lite method from PART2 is applied to simplify software analytics. Specifically, extensive experiments with transfer learning, classifier tuning (hyper-parameter optimization), and an ensemble method were conducted. Those methods can be very slow, especially for large data sets. For the purposes of building defect predictors, this thesis reports that the early data-lite sampling method worked better than the ensemble and hyper-parameter optimization methods (that use later or full life cycle data). Furthermore, the combination of transfer learning and early life cycle sampling actually outperformed either technique.

From the above, this thesis offers the following conclusion. Before researchers rush to reason across all available data using the most complex methods, it is prudent to first check for simpler alternatives.

And when exploring simpler methods, the results of this work strongly recommend trying early life cycle sampling.

© Copyright 2022 by Shrikanth Narayanaswamy Chandrasekaran

All Rights Reserved

Taming Confusions in Software Engineering

by  
Shrikanth Narayanaswamy Chandrasekaran

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

2022

APPROVED BY:

---

Dr. Noboru Matsuda

---

Dr. Bradley Reaves

---

Dr. Jamie Jennings

---

Dr. Timothy Menzies  
Chair of Advisory Committee

## **DEDICATION**

To my parents Swarubarani Chandrasekaran and Chandrasekaran Narayanaswamy—for everything

## BIOGRAPHY

The author was born in Tamil Nadu, India, in 1986. In 2008, he received his bachelor's degree in Electronics and Communication Engineering from Saveetha Engineering College (affiliated with Anna University, India). Then he worked in the software industry for nine years. Specifically, he played a lead Technology R&D Specialist at Accenture Labs. Before that, he was a Software Engineer and Senior Systems Engineer at ABB India Ltd and Infosys Ltd. The author holds two Java programming certifications from Sun Microsystems, published two papers at ICSE <sup>1</sup> (apart from the papers from this dissertation), and holds three granted patents. From 2018 to 2021, the author worked in the RAISE lab under the supervision of Dr. Tim Menzies at North Carolina State University, USA. His primary research interest lies in identifying shortcuts to software analytics through investigating software engineering theories. He was a research intern at Fujitsu Labs and Microsoft USA in the last two summers (2020 and 2021).

---

<sup>1</sup>IEEE/ACM International Conference on Software Engineering

## ACKNOWLEDGEMENTS

I want to wholeheartedly thank,

- ★ **Prof. Tim Menzies** <sup>2</sup> for his *trust, support, guidance, patience, and knowledge sharing*. Despite being a seasoned software engineering practitioner, attending his graduate-level classes was *pivotal* in anchoring my research. Importantly those classes *taught* me the science behind software engineering.
- All my Accenture Labs, India colleagues (specifically **Dr. Sanjoy Paul, Dr. Anurag Dwarakanath** and **Pradeep Kumar Duraisamy**) and ABB India colleagues (specifically **Bala Venkateswaralu Rayana**) for their encouragement and recommendations.
- **Dr. Kathryn T. Stolee** for attending my talk at ICSE <sup>3</sup> held in Austin, USA. Moreover, suggesting that I apply to the Ph.D. program at North Carolina State University.
- North Carolina State University and the Department of Computer Science for considering my Ph.D. application and offering me a fully-funded Ph.D. admit in 2017.
- **Dr. George N. Rouskas** (Director of Graduate Programs) for his support and guidance to excel in my Ph.D. journey.
- My sibling **Shrividya Narayanaswamy Chandrasekaran** and her family for helping me on-board back to graduate school in a foreign land at 31.
- **Dr. Rada Chirkova** to help me cope in my first academic job as a Graduate Teaching Assistant. I am grateful to have learned a lot in managing a semester-long class of quizzes, assignments, project demos, and exams.
- Many friends at North Carolina State University specifically **Kashyap Sivasubramanian, Charan Ram V C, Karthik MedidiSiva** and **Muhundhan Mohan** for proof-reading my papers and importantly being good listeners.
- All the researchers at RAISE lab <sup>4</sup> especially **Dr. Rahul Krishna** for being the go-to person for those timely clarifications and lunch buffet hangouts.
- My research collaborators **Dr. William Nichols** (Software Engineering Institute, Carnegie Mellon University) and **Suvodeep Majumder** (North Carolina State University) for their support in the work [Shr21b] and [Shr21c].
- My dissertation committee members **Dr. Bradley Reaves, Dr. Jamie Jennings, Dr. Khaled Abdel Harfoush, Dr. Noboru Matsuda, and Dr. Timothy Menzies** for their valuable feedback.
- The Department of Computer Science staff **Kathy Luca, Carol Allen**, and all the others.

---

<sup>2</sup>★ Ph.D. advisor

<sup>3</sup>International Conference on Software Engineering

<sup>4</sup><http://ai4se.net/>

# TABLE OF CONTENTS

<b>LIST OF TABLES</b> .....	<b>ix</b>
<b>LIST OF FIGURES</b> .....	<b>x</b>
<b>Part 1</b> .....	<b>1</b>
<b>Chapter 1 Introduction</b> .....	<b>2</b>
1.1 What is the Problem ? .....	2
1.1.1 Examples of Confusions .....	3
1.2 Why is it interesting and important? .....	4
1.3 What is wrong with previous proposed solutions? .....	4
1.4 What is novel about this work ? .....	4
1.5 What are the key components of my approach and results? .....	5
1.6 Statement of Thesis .....	6
1.7 Research Questions .....	6
1.8 Novel Contributions of this Work .....	7
1.9 Published Works .....	7
1.9.1 Conference Full Papers .....	7
1.9.2 Journal Full Paper .....	8
1.9.3 Poster .....	8
1.10 Under Review .....	8
1.11 Structure of this Thesis .....	8
1.12 Acknowledgement .....	8
<b>Chapter 2 Problems in Software Analytics</b> .....	<b>9</b>
2.1 Definition .....	9
2.2 Motivation .....	10
2.2.1 Software Quality Assurance .....	10
2.2.2 Trust .....	10
2.2.3 Insight .....	11
2.2.4 Tool development and Training .....	11
2.3 Where to Tame? .....	12
2.4 Challenges .....	12
2.4.1 Confusions caused by Data-hungry (late-data) Oracles .....	12
2.4.2 Confusions of Practitioner Beliefs .....	13
2.5 Checking the Conclusion Instability Effect .....	14
2.5.1 Investigating Software Engineering Theories .....	14
2.5.2 Quality .....	15
2.5.3 Productivity .....	15
2.5.4 Expertise .....	16
2.5.5 Investigating Software Quality Theories .....	16
2.5.6 Data-lite Software Analytics .....	18
2.6 Summary .....	19
<b>Chapter 3 Data, Methods, Metrics &amp; Measures</b> .....	<b>20</b>
3.1 Introduction .....	20

3.2	Data Source: Industry data to assess Software Engineering Theories . . . . .	21
3.2.1	Data Filtering . . . . .	25
3.3	Data Source: Open Source data to assess Software Quality Theories . . . . .	26
3.3.1	Data distribution . . . . .	27
3.3.2	Data Attributes . . . . .	27
3.4	Data Source: Open Source data to assess Oracles . . . . .	27
3.5	Statistical Tests . . . . .	30
3.5.1	Rank Treatments . . . . .	30
3.5.2	Correlation Analysis . . . . .	31
3.5.3	Thresholds . . . . .	31
3.5.4	Software Engineering Measures . . . . .	32
3.6	Classifiers (Oracles) . . . . .	32
3.7	Data Pre-processers . . . . .	36
3.7.1	Oracle Evaluation Criteria . . . . .	37
3.8	Summary . . . . .	39
<b>Part 2</b>	. . . . .	<b>40</b>
<b>Chapter 4</b>	<b>Documenting &amp; Understanding the Source of Human Confusions . . . . .</b>	<b>41</b>
4.1	Introduction . . . . .	41
4.2	Investigating Software Engineering Theories . . . . .	42
4.2.1	Why explore only these five beliefs ? . . . . .	42
4.2.2	Belief 1: Corbató's law . . . . .	42
4.2.3	Belief 2: Dahl-Goldberg hypothesis . . . . .	44
4.2.4	Belief 3: Mills-Jones hypothesis . . . . .	45
4.2.5	Belief 4: Sackman's second Law . . . . .	47
4.2.6	Belief 5: Apprentice's Law . . . . .	48
4.2.7	Threats to validity . . . . .	51
4.3	Investigating Software Quality Assurance Theories . . . . .	53
4.3.1	Why explore only these ten beliefs ? . . . . .	53
4.3.2	Belief 1: Complex Code Changes . . . . .	54
4.3.3	Belief 2 & Belief 10: Ownership . . . . .	54
4.3.4	Belief 3 & Belief 9 : <i>Code Churn</i> . . . . .	54
4.3.5	Belief 4 & Belief 6: <i>Temporal</i> . . . . .	54
4.3.6	Belief 5: Commit Churns . . . . .	55
4.3.7	Belief 7 & Belief 8: <i>Something More</i> . . . . .	55
4.3.8	Threats to validity . . . . .	56
4.4	RQ1: Why do beliefs diverge among practitioners? . . . . .	56
4.4.1	Diverged Software Engineering Beliefs . . . . .	56
4.4.2	Diverged Software Quality Assurance Beliefs . . . . .	57
4.4.3	Importance of Re-checking . . . . .	58
4.5	RQ2: What disconnects Beliefs and Evidence? . . . . .	58
4.5.1	Sporadic Evidence . . . . .	58
4.5.2	Evidence Decay . . . . .	60
4.6	Summary . . . . .	63
<b>Chapter 5</b>	<b>Taming Confusions with Early Data . . . . .</b>	<b>64</b>
5.1	Is More data much more useful? . . . . .	64

5.2	The ‘Early Trend’ discovery . . . . .	66
5.2.1	Chai-Break Anecdote . . . . .	66
5.2.2	Sporadic Evidence . . . . .	66
5.2.3	Evidence Decay . . . . .	67
5.2.4	Trends in 100+ Software Projects . . . . .	67
5.3	Building the Early Data-lite Method . . . . .	68
5.3.1	Revisiting the Objective . . . . .	68
5.3.2	The Data-lite Stop Early Method . . . . .	68
5.4	Assessing the Data-lite Method . . . . .	70
5.5	RQ3: Does learning from more data pacify confusion? . . . . .	73
5.6	RQ4: Is recent data more important than older data to pacify confusion? . . . . .	74
5.7	Threats to Validity . . . . .	74
5.7.1	Sampling Bias . . . . .	74
5.7.2	Learner bias . . . . .	75
5.7.3	Evaluation bias . . . . .	75
5.7.4	Input Bias . . . . .	75
5.8	Summary . . . . .	76
<b>Part 3</b>	. . . . .	<b>77</b>
<b>Chapter 6</b>	<b>Expanding the Scope of Early Methods . . . . .</b>	<b>78</b>
6.1	Introduction . . . . .	78
6.2	Defect Prediction Analytics . . . . .	80
6.2.1	Data: Transfer Learning . . . . .	80
6.2.2	Techniques : Tuning and Ensemble . . . . .	81
6.2.3	Issues with current approaches . . . . .	82
6.3	Related Work . . . . .	82
6.3.1	Literature Survey . . . . .	83
6.4	Experimental Methods . . . . .	86
6.4.1	Data . . . . .	87
6.4.2	Sampling Methods . . . . .	87
6.5	RQ5: Can we build early defect prediction models from unpopular projects? . . . . .	90
6.5.1	Motivation . . . . .	90
6.5.2	Approach . . . . .	90
6.5.3	Results . . . . .	91
6.6	RQ6: Can we build early defect prediction models with fewer features? . . . . .	92
6.6.1	Motivation . . . . .	92
6.6.2	Approach . . . . .	92
6.6.3	Results . . . . .	92
6.7	RQ7: Can build early defect prediction models from transferring early life-cycle data from other projects? . . . . .	93
6.7.1	Motivation . . . . .	93
6.7.2	Approach . . . . .	94
6.7.3	Results . . . . .	95
6.8	RQ8: Do complex methods supersede early defect prediction models? . . . . .	96
6.8.1	Motivation . . . . .	96
6.8.2	Approach . . . . .	97
6.8.3	Results . . . . .	98

6.9	Threats to Validity . . . . .	99
6.9.1	Sampling Bias . . . . .	99
6.9.2	Construct Validity . . . . .	99
6.9.3	Learner bias . . . . .	99
6.9.4	Evaluation bias . . . . .	100
6.9.5	Input Bias . . . . .	100
6.10	Summary . . . . .	101
<b>Chapter 7</b>	<b>Conclusions and Future Work . . . . .</b>	<b>102</b>
7.1	Thesis Revisited . . . . .	102
7.2	Discussion . . . . .	103
7.2.1	Management Implications . . . . .	103
7.2.2	Systems Implications . . . . .	104
7.2.3	Research Implications . . . . .	105
7.3	Future Work . . . . .	106
7.3.1	Log Analysis . . . . .	106
7.3.2	Code Reuse . . . . .	106
7.3.3	Software Fairness . . . . .	107
7.3.4	Other domains: . . . . .	107
7.4	Epilogue . . . . .	107
	<b>BIBLIOGRAPHY . . . . .</b>	<b>109</b>

## LIST OF TABLES

Table 2.1	SE Beliefs studied in this thesis. . . . .	15
Table 2.2	Defect Prediction Beliefs . . . . .	17
Table 3.1	Count of Engineers (Developers) by Domain . . . . .	21
Table 3.2	Overview of PSP Tasks . . . . .	23
Table 3.3	OS Projects List . . . . .	27
Table 3.4	Project’s commit-level features . . . . .	28
Table 3.5	Tree of Hyper-parameter Options . . . . .	34
Table 4.1	Distributions of “program size”, “production rate” and “defects” . . . . .	43
Table 4.2	Distribution of “defects (Coding + Design)” in two different groups (programming languages and task 10) . . . . .	46
Table 4.3	Novice .Vs. Expert distributions . . . . .	51
Table 4.4	Defect Prediction Beliefs ranked by empirical support . . . . .	57
Table 4.5	Support of beliefs in different release types. . . . .	62
Table 5.1	Papers discussing different sampling policies. All (papers that utilize all historical data to build defect prediction models, shaded in gray). . . . .	71
Table 5.2	Four representative sampling policies from literature and an early life cycle policy (the row shown in gray). . . . .	72
Table 5.3	Ranking of 24 defect prediction models . . . . .	72
Table 5.4	Ranking of 12 defect prediction models . . . . .	73
Table 6.1	Table of Important Acronyms . . . . .	80
Table 6.2	50 ‘highly cited’ cross-project defect prediction articles. . . . .	84
Table 6.3	44 ‘highly cited’ ensemble and tuning articles. . . . .	85
Table 6.4	Baseline Approaches . . . . .	86
Table 6.5	Ranking of 12 defect prediction models . . . . .	91
Table 6.6	Ranking of 14 defect prediction models . . . . .	94
Table 6.7	Ranking of 31 defect prediction models . . . . .	97
Table 6.8	Ranking of 7 defect prediction models . . . . .	98
Table 6.9	Ranking of 12 defect prediction models . . . . .	100

## LIST OF FIGURES

Figure 1.1	Thesis Overview . . . . .	5
Figure 1.2	Thesis landscape . . . . .	6
Figure 2.1	Replicated from [Men17] showing a partial list of sources of conclusion instability.	11
Figure 2.2	Summary of sampling types . . . . .	13
Figure 2.3	This chapter filled the missing pieces ‘?’ of part-1 in the thesis landscape. . . . .	19
Figure 3.1	Distribution of tasks completed in a specific programming language. . . . .	22
Figure 3.2	PSP Task Sample . . . . .	23
Figure 3.3	Distribution of defects in PSP tasks . . . . .	24
Figure 3.4	Proportion of developers (bar-chart) completing PSP tasks . . . . .	26
Figure 3.5	Information about the distribution of project data. . . . .	28
Figure 3.6	Information about the distribution of popular projects data. . . . .	29
Figure 3.7	Information about the distribution of unpopular project data. . . . .	29
Figure 3.8	Pseudo-code of DODGE replicated from [Agr19] . . . . .	33
Figure 3.9	Two-Layer Ensemble Learning framework . . . . .	35
Figure 4.1	Distribution of correlation scores grouped by different programming language . . . . .	47
Figure 4.2	Distribution of correlation scores grouped by different programming language . . . . .	49
Figure 4.3	Performance of <i>expert</i> and <i>novice</i> programmers . . . . .	50
Figure 4.4	A visual summary of beliefs . . . . .	59
Figure 4.5	Updated visual map of beliefs as per evidence . . . . .	60
Figure 4.6	Coverage of Beliefs in Projects . . . . .	61
Figure 4.7	Evolution of beliefs in projects . . . . .	63
Figure 4.8	A landscape of this thesis, where the first part (Sporadic evidence) of part-2 is answered. . . . .	63
Figure 5.1	Extracted from the talk ‘The Unreasonable Effectiveness of Data’ - Peter Norvig	65
Figure 5.2	Early Defect Trends in 155 Projects . . . . .	65
Figure 5.3	Project commit distributions . . . . .	68
Figure 5.4	Depicts the objective of this thesis . . . . .	69
Figure 5.5	A Visual map of Sampling Policies. . . . .	70
Figure 5.6	A landscape of this thesis, where the second part (data-hungry methods) of part-2 is answered. . . . .	75
Figure 6.1	A visual map of defect trends in projects . . . . .	79
Figure 6.2	Unpopular projects. Median= 5 releases, 7 months and 29 defective commits. . . . .	87
Figure 6.3	A visual map of Sampling Policies . . . . .	88
Figure 6.4	Frequency of features . . . . .	93
Figure 6.5	Bellwether - Venn Diagram . . . . .	95
Figure 6.6	Run-time Comparison of Bellwether Identification . . . . .	96
Figure 6.7	Sampling policy run-time comparison . . . . .	96
Figure 6.8	Training run-time comparison . . . . .	98
Figure 6.9	Early defect trends in both popular and unpopular projects . . . . .	100
Figure 6.10	A landscape of this thesis, where the final piece (early data-lite method) of part-3 is placed. . . . .	101

Figure 7.1	Work duration histograms . . . . .	104
Figure 7.2	A guide to build defect predictors . . . . .	105

# **Part 1 - Documenting Contradictions**

## CHAPTER

# 1

# INTRODUCTION

## 1.1 What is the Problem ?

Actively for over many decades, practitioners and researchers are exploring ways to predict the quality of the software before it is deployed. More generally, the “software defect prediction” research approach largely uses data miners to input static code attributes and output models that predict where the code probably contains most defects [Ost05; Men06]. Wan et al. [Wan18a] reports that there is much industrial interest in these predictors since they can guide the deployment of more expensive and time-consuming quality assurance methods (e.g., human inspection). Misirili et al. [Mis11] and Kim et al. [Kim11] report considerable cost savings when such predictors are used in guiding industrial quality assurance processes. Also, Rahman et al. [Rah14] show that such predictors are competitive with more elaborate approaches.

However, confusions (or conclusion instability) in software engineering (SE), specifically software quality assurance activities like “defect prediction,” can be caused by humans (practitioners) and oracles (machine learning models). Largely, practitioners are deluded by their beliefs because they assume lessons learned in their past projects are generalizable to all their future projects. Numerous studies show that many prevalent beliefs are not backed by evidence [Men17; Pas11; Nag15]. On the other hand, oracles that guide practitioners (like software developers and managers) confuse practitioners by providing different decisions in their day-to-day work. Furthermore, oracles are confused by the underlying methods requiring them to revise and accumulate more data every time. A presumption that supports this approach is that practitioners and researchers presume that more data is much more helpful and revise oracles continuously as new data accumulate in their software projects.

### 1.1.1 Examples of Confusions

Practitioners have limited time and resource for software quality assurance activities like code review. Therefore it is important that their decisions are stable throughout their software project life-cycle. Below are some examples of what this work refers to as confusions in software engineering specifically in software quality.

#### 1.1.1.1 Among Humans

- Developer ‘X’ observed a complex code change (a lesson learned from their past project) induced more defects. Now ‘X’ believes that effect will continue to hold in subsequent projects. Accordingly, ‘X’ will conduct the project’s quality assurance activities, prioritizing over that belief (but that effect may not hold in a different project or later within the same project).
- During the beginning of a project, developer ‘A’ believed that a file with more changes induces more defects (bugs), but after few months, ‘A’ observed that the file that underwent many bug fixes is more defect prone towards the end. It could mean that practitioners who hopped on to a long-running matured project would have a different belief than one working on a recent project.
- Developers ‘B’ and ‘C’ work for the same organization but with different teams. And ‘B’ believes that lines added to a file are more critical to investigate, whereas ‘C’ thinks lines deleted to a file should be prioritized for review.
- ...and many more confusions.

#### 1.1.1.2 Caused by Oracles

Oracles (defect predictors) can yield different decisions every time to classify a particular software change either as defective or clean. Practitioners time is limited to dedicate to code review activities and they will be demotivated if such oracles produce many false alarms (classifying a change as defective but after thorough investigation it turned out to be clean). Later this work will show much of those confusions caused by oracles are due to the following presumptions found in decades of software defect prediction literature.

In defect prediction, data-hungry researchers assume that if data is useful, then even more data is much more useful. For example:

- “..as long as it is large; the resulting prediction performance is likely to be boosted more by the size of the sample than it is hindered by any bias polarity that may exist” [Rah13b].
- “It is natural to think that a closer previous release has more similar characteristics and thus can help to train a more accurate defect prediction model. It is also natural to think that accumulating multiple releases can be beneficial because it represents the variability of a project” [Ama20].
- “Long-term JIT models should be trained using a cache of plenty of changes” [Mc117].

Not only are researchers hungry for data, but, they are also most hungry for the most recent data. For example: Hoang et al. say “We assume that older commits changes may have characteristics that no longer effects to the latest commits” [Hoa19]. Also, it is common practice in defect prediction to perform

“recent validation” where predictors are tested on the latest release after training from the prior one or two releases [Tan15; McI17; Kon20; Fu16a]. For a project with multiple releases, recent validation ignores any insights that are available from older releases.

## **1.2 Why is it interesting and important?**

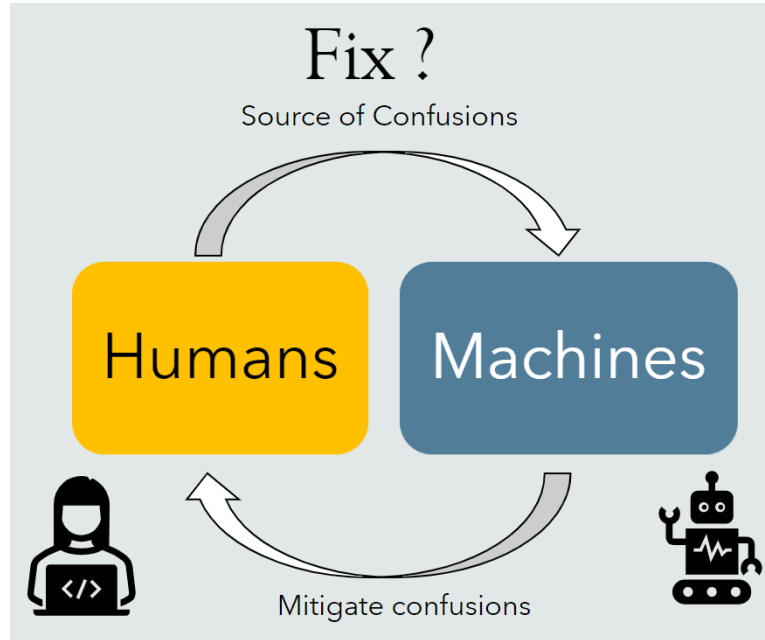
Software engineers have limited time and resources; hence fixing defects becomes an expensive activity. A study conducted by the National Institute of Standards and Technology shows that defects (due to inadequate testing infrastructure) cost industries in the U.S between \$22 to \$50 billion dollars annually [Pla02]. Further from a practitioner point of view, subject matter experts (SME) make important business decisions on a day-to-day basis. Therefore it is important that they are not premising their decisions on feeble beliefs. Practitioners lose trust if their results keep changing. It is also difficult to guide their teams with feeble decisions. Lastly, researchers will only be able to disseminate their methods to practice only if they yield accurate decisions.

## **1.3 What is wrong with previous proposed solutions?**

In addition, to the conclusion instability problem, current software analytics methods have three issues (a) they are data-hungry and (b) the oracles are revised with newer data (re-trained) on every new project release, and (c) they learn from recent project data. First, because practitioners have to wait for the project to accumulate local data, it delays learning and decision-making early in the project life cycle. Second, when oracles accumulate more data on every release, the training time increases, and their conclusions fluctuate. On the other hand, if they are only trained from a small portion of recent data, then it would miss important signals available early in the project life cycle. Some methods use recent data for training, under the assumption that older data may not have enough signals (later this thesis in Chapter 5 will show that this assumption is wrong). This thesis will find a meaningful pattern that reasons why learning from recent data could be a methodological error.

## **1.4 What is novel about this work ?**

In an attempt to find the root cause of practitioner confusion, this thesis will discover a startling trend later in Chapter 5. Then using that discovery, this work will devise a novel sampling method that mitigates the problems of current approaches by (a) using an early data-lite approach instead of a data-hungry (or late-data) approach, (b) avoiding re-training of oracles on every new project release, unlike the data-hungry approach, (c) does not reason using recent project data to avoid the methodological error and (e) portray many of the complex methods are needless, and some can be augmented with early data.



**Figure 1.1** Thesis Overview

## 1.5 What are the key components of my approach and results?

An overview of this thesis is shown in the Figure 1.1, where this thesis aims to understand the root cause of practitioner confusions by investigating their beliefs about software engineering and software defect prediction (quality). Then it uses that root cause to devise a method to build immutable machines (oracles) that guide practitioners with the rest of the software life-cycle.

A roadmap of this work is shown in Figure 1.2 with many missing pieces ‘?’. By the end of Chapter 6 all those missing pieces will be answered. The three parts (part-1, part-2, and part-3) of this thesis are portrayed in the landscape. The critical components of each part are as follows:

- In part-1, this thesis documents the disconnect between practitioner beliefs and empirical evidence.
- In part-2, this thesis explains that disconnect to devise a novel method and show how to build immutable oracles that last the entire software project life cycle.
- In part-3, this improves the scope of the early-date-lite method identifier in part-2 to simply software analytics.

The critical component of this work is part-2 that discovered a significant early trend missed in decades of SE literature. Using that discovery of ‘early trend,’ this work identifies a *systematic error* and *simplifies* data-hungry software analytic approaches to data-lite alternate. To reiterate, this work solves the conclusion instability problem by:

- Building software analytic models using only 4% of project data that perform just as well as using 100% of project data.
- There is no need to revise (re-train) the model as the project matures with more releases.

- With no-retraining, the early data-lite models will offer stable conclusions throughout the software development life-cycle without the use of complex optimizer or ensemble methods.

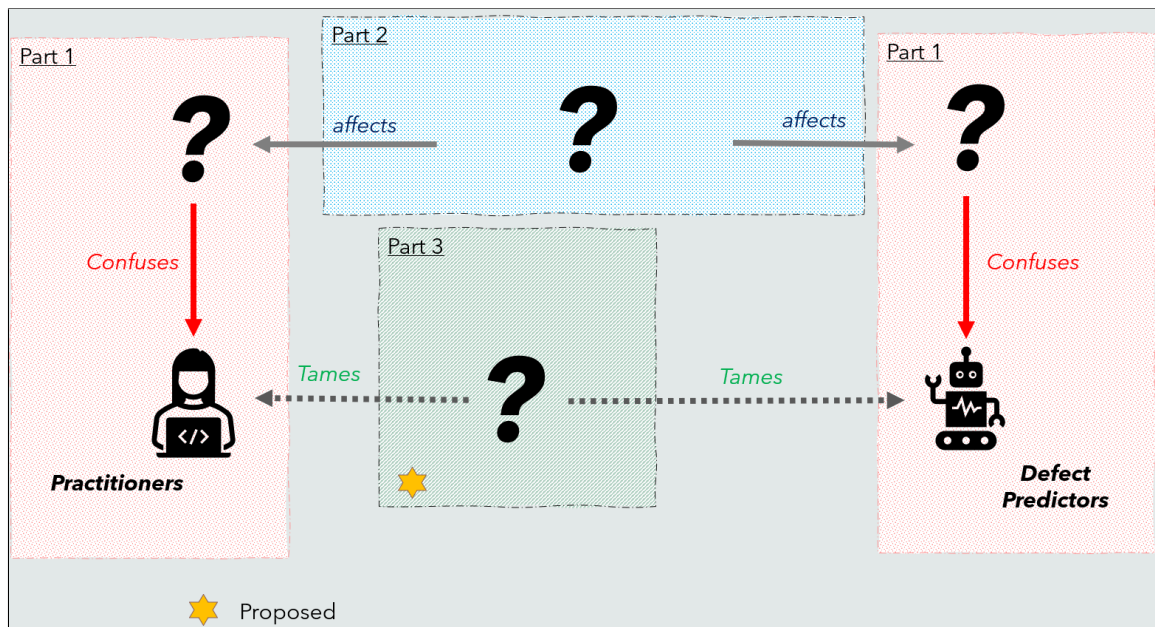
However, for many decades, SE researchers and practitioners have been reasoning using voluminous data-hungry approaches. They reason across the entire project cycle leading to a presumption that there is not much local information in their projects and transferred years of multiple project data from neighboring projects to build software analytic models. Worse, they also built models using recent project data of limited value to premise their conclusions. Therefore conclusions based on the entire project history or transferring data from other projects are *needless*. On the other hand, using only the recent history could be *wrong*.

## 1.6 Statement of Thesis

In the domain of software defect prediction, humans are (a) **confused about what influences software defects** since they tend to (b) **reason across over all project data** while the (c) **most relevant experience occurs very early in project life-cycle**. Hence (d) **better reasoning can be achieved by focusing only on earlier life cycle data** (where “better” means simpler, faster, and more effective).

## 1.7 Research Questions

This thesis explores eight research questions in three parts. Part-1 of this thesis elucidates the problem, data, and methods needed to build this work. Part-2 of this thesis focuses on three separate studies. Two



**Figure 1.2** An initial landscape of this thesis having three parts (part-1, 2 and 3). ‘?’→ denotes missing pieces to be completed in Chapters 4, 5 and 6.

studies heavily focus on assessing software engineering and defect prediction beliefs to understand the confusion in software analytics, while the third study identifies a key trend to fix the problem.

The research questions explored in this thesis are as follows:

- RQ1: Why do beliefs diverge among practitioners?
- RQ2: What disconnects Beliefs and Evidence?

The answers to these questions led to discovering a startling early trend that holds in hundreds of software engineering projects. Using that observation, this thesis explores the following questions:

- RQ3: Does learning from more data pacify confusion?
- RQ4: Is recent data more important than older data to pacify confusion?

That shows a way to tame conclusion instability in software analytic.

In the last section, of this thesis (part-3) four research questions listed below expand the scope of early-date-lite methods to unpopular SE projects but more importantly show complex software analytic methods are needless if data is sampled from early project regions.

- RQ5: Can we build early defect prediction models from unpopular projects?
- RQ6: Can we build early defect prediction models with fewer features?
- RQ7: Can build early defect prediction models from transferring early life-cycle data from other projects?
- RQ8: Do complex methods supersede early defect prediction models ?

## 1.8 Novel Contributions of this Work

In summary, the contributions of this work are:

1. Documented the confusions in software engineering among practitioners (due to their beliefs) and oracles (due to current methods).
2. Investigated the root cause of those confusions to devise a novel data-lite stop early approach to overcome needlessly complicated current data-hungry approaches that cause confusions.
3. Showed that much of the complex methods (hyper-parameter optimization, ensemble etc) are needless.
4. All the data and scripts part of this thesis are available online at these following locations:
  - **Defect Prediction Beliefs:** [https://github.com/ai-se/defect\\_perceptions](https://github.com/ai-se/defect_perceptions)
  - **Software Engineering Beliefs:** <https://zenodo.org/record/4553435>
  - **Early Analytics:** <https://zenodo.org/record/445956>
  - **Simplify Analytics:** <https://github.com/snaraya7/simplifying-software-analytics>

## 1.9 Published Works

### 1.9.1 Conference Full Papers

- [Shr20a] **Shrikanth, N. C.**, and Tim Menzies. "Assessing practitioner beliefs about software defect prediction." 2020 IEEE/ACM 42nd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP). IEEE, 2020.

- [Shr21c] **Shrikanth, N. C.**, Suvodeep Majumder, and Tim Menzies. "Early Life Cycle Software Defect Prediction. Why? How?." ACM/IEEE International Conference on Software Engineering (2021).

### 1.9.2 Journal Full Paper

- [Shr21b] **Shrikanth, N. C.**, William Nichols, Fahmid Morshed Fahid, and Tim Menzies. "Assessing practitioner beliefs about software engineering." Empirical Software Engineering (2021).

### 1.9.3 Poster

- [Shr20b] **Shrikanth, N. C.**, and Tim Menzies. "What disconnects practitioner belief and empirical evidence?." In Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings, pp. 286-287. 2020.

## 1.10 Under Review

- [Shr21a] **Shrikanth, N. C.**, and Tim Menzies. "Simplifying Software Defect Prediction (via the "early bird" Heuristic)" (EMSE, 2021)

## 1.11 Structure of this Thesis

The Figure 1.2 presents a high-level flow where both humans (practitioners) and oracles (defect predictors) can be confused in software engineering environments.

The rest of this thesis is structured as follows:

- Chapter 2 elucidates the definition of the conclusion instability alongside the challenges this work aims to overcome.
- Chapter 3 presents the data, methods, and tests required to analyze different theories in identifying the source of the conclusion instability problem.
- Chapter 4 assesses many SE and software quality theories to lists the various avenues for the conclusion instability problem.
- Chapter 5 introduces a novel method to tame conclusion instabilities in software analytics derived from the lessons obtained previously in Chapter 4.
- Chapter 6 checks the scope of the novel method among additional projects and analytics methods to determine simplicity in software analytics.
- Chapter 7 Lists future directions and presents a summary of this thesis.

This chapter is partially structured using the advice provided here <sup>1</sup>.

## 1.12 Acknowledgement

This work was partially supported by an NSF grant 1908762.

---

<sup>1</sup><https://cs.stanford.edu/people/widom/paper-writing.html#intro>

## CHAPTER

# 2

# PROBLEMS IN SOFTWARE ANALYTICS

This chapter will elucidate the ‘conclusion instability’ (confusions) problem among SE practitioners, researchers due to their beliefs (presumptions) and software analytics in general due to the prevalent oracle construction methods. Furthermore, this chapter will answer the missing pieces ‘?’ in part-1 of the roadmap shown in Figure 1.2 earlier. Also, this chapter will present the challenges of the ‘conclusion instability’ problem and debates why we need to handle this problem differently.

## 2.1 Definition

Conclusion instability (confusions) is the problem of changing decisions frequently when analysts inspect more data or engineers presuming specific effects to hold in their project that is not backed by evidence. *Solutions* to the conclusion instability problem in SE aims to minimize the need for revising software analytic oracles (that guide practitioners to make decisions during the software development activities) while maximizing the capability to provide accurate decisions. Formally, the conclusion instability problem can be described as follows:



### **Antidote to Confusion Instability**

To devise a method to build static oracles (no re-training) for a software project with better long-lasting decision-making capability than dynamic oracles (frequent re-training).

## 2.2 Motivation

Why should practitioners and researchers care about ‘conclusion instability’ at all? This work argues that taming confusion is helpful since that would help software engineers worldwide make better decisions during the software development activity. Therefore this fits into a broad class of SE problems that is worth addressing. The rest of this section will elucidate the importance of this problem.

### 2.2.1 Software Quality Assurance

Conclusion instability is well documented. Zimmermann et al. [Zim09] learned defect predictors from 622 pairs of projects (project1, project2). In only 4% of pairs, predictors from project1 worked on project2. Also, Menzies et al. [Men12] studied defect prediction results from 28 recent studies, most of which offered widely differing conclusions about what most influences software defects. Menzies et al. [Men11] reported experiments where data for software projects are clustered, and data mining is applied to each cluster. They report that very different models are learned from different parts of the data, even from the same projects.

The RAISE<sup>1</sup> lab found conclusion instability in one of their past work, meaning they had to throw years of data. In one sample of GitHub data, they sought to learn everything they could from 700,000+ commits. Unfortunately, the web slurping required for that process took nearly 500 days of CPU (using five machines with 16 cores, over seven days). In addition, they found significant differences in the models learned from different parts of the data within that data space. So even after all that work, they could not offer the business users a stable predictor for their domain.

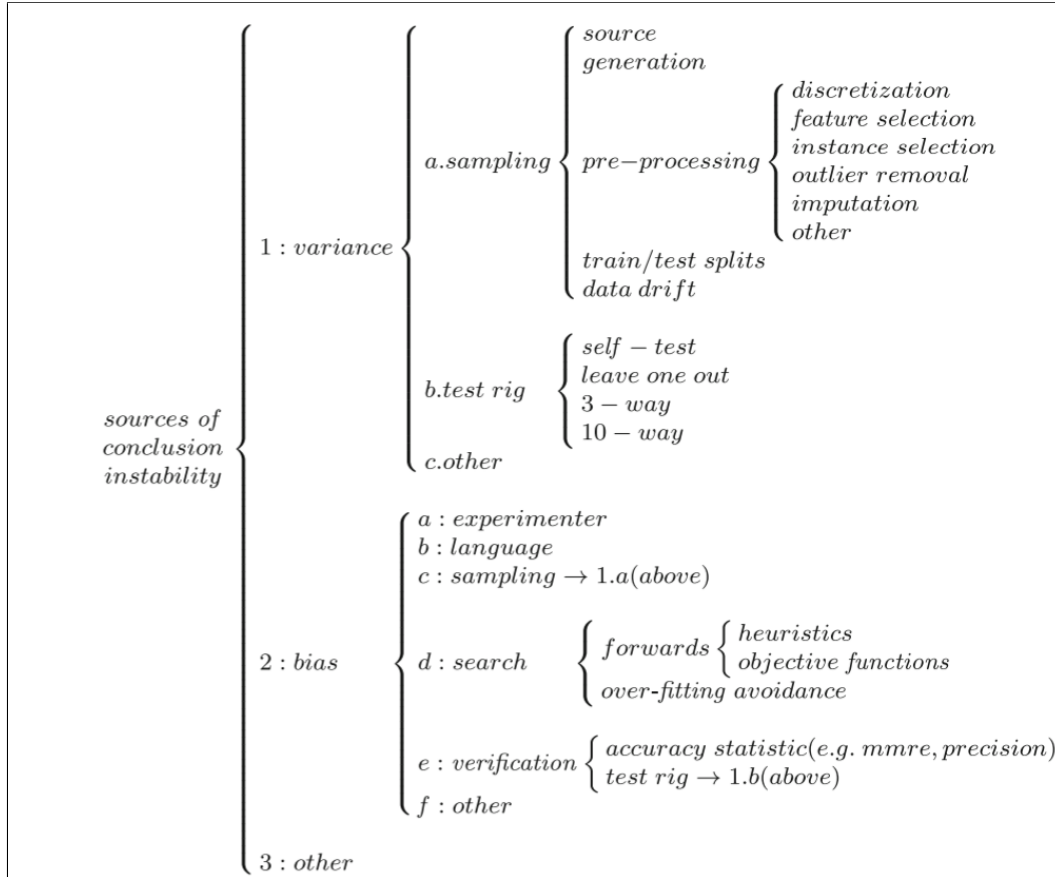
Is that the best we can do? Are there general defect prediction principles we can use to guide project management, software standards, education, tool development, and legislation about software? Or is SE some “patchwork quilt” of ideas and methods where it only makes sense to reason about specific, specialized, and small sets of related projects? Note that if the software were a “patchwork” of ideas, then there would be no stable conclusions about what constitutes best practice for software engineering (since those best practices would keep changing as we move from project to project). Such conclusion instability would have detrimental implications for *trust*, *insight*, *training*, and *tool development*.

### 2.2.2 Trust

Conclusion instability is unsettling for project managers. Hassan [Has17] warns that managers lose trust in software analytics if its results keep changing. Such instability prevents project managers from offering clear guidelines on many issues, including (a) when a certain module should be inspected; (b) when modules should be refactored; Moreover, (c) deciding where to focus on expensive testing procedures.

---

<sup>1</sup><http://ai4se.net/>



**Figure 2.1** Replicated from [Men17] showing a partial list of sources of conclusion instability.

### 2.2.3 Insight

Sawyer et al. assert that insights are essential to catalyzing business initiative [Saw13]. From Kim et al. [Kim16] perspective, software analytics is a way to obtain fruitful insights that guide practitioners to accomplish software development goals, whereas for Tan et al. [Tan16a] such insights are a central goal. From a practitioner's perspective, Bird et al. [Bir15] report, insights occur when users respond to software analytics models. However, frequent model generation could exhaust users' ability for confident conclusions from new data.

### 2.2.4 Tool development and Training

It hard to onboard novice software engineers, without knowing what factors most influence the local project, it is hard to design and build appropriate tools for quality assurance activities.

## 2.3 Where to Tame?

The purpose of empirical SE could be undermined if the discoveries do not hold in multiple contexts (or at least different projects of similar context). Given the diversity in software engineers and software projects, different effects can occur. However, researchers should continue to work towards generality. Positively recent works have shown it is possible to discover universal lessons in SE [Zha14].

In order to tame conclusion instability, it is essential to track its source. Figure 2.1 shows only a partial list of avenues that causes conclusion instability in SE. While Menzies and Shepperd agree that the list is partial, it acknowledges that conclusion instability can occur in many ways. Further, that list lacks theoretical rigor and does not show instances of controlling conclusion instabilities using the items listed in that tree of Figure 2.1. This thesis responds to their call to show substantial improvements by portraying, investigating, and mitigating confusion via software analytics.

While reporting their findings, at least in the space of defect prediction, highly cited literature does take care of most of the issues shown in Figure 2.1. However, the findings only hold for the scope of the findings specific to that work. More importantly, Chapter 5 will show a significant trend that decades of SE literature missed.

Nevertheless, this work will investigate the root cause of practitioners' delusion to bring a fundamental change to tame confusion. Therefore, chapter 4 will model and thoroughly assess various beliefs. To narrow the root cause of 'conclusion instability' beyond what is shown in Figure 2.1.

## 2.4 Challenges

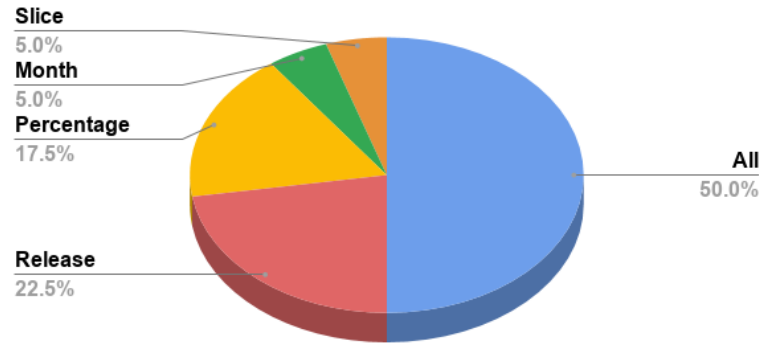
This section will discuss the two avenues of the conclusion instability problem caused by oracles (machines) and humans (practitioners).

### 2.4.1 Confusions caused by Data-hungry (late-data) Oracles

We record more SE data than ever before, but the challenge will always be to find the right insights that lead to building strong software oracles or, in fact, strong theories. Data-hungry methods are often cited as the key to success for data mining applications. For example, in his famous talk, "The Unreasonable Effectiveness of Data," Google's Chief Scientist Peter Norvig argues that "billions of trivial data points can lead to understanding" [Nor11] (a claim he supports with numerous examples from vision research).

In the literature, there are a bewildering array of different ways to sample project data. Figure 2.2 is only an approximation of the diverse number sampling policies in the literature. A more comprehensive picture is shown later in Figure 5.5 and Figure 6.3. Figure 2.2 shows a high-level view of the sampling policies in hundreds of software quality assurance papers this work investigated:

- **All:** When the historical data (commits/files/modules etc.) is used for evaluation within some cross-validation study (where the data is divided randomly into  $N$  bins, and the data from bin ' $i \in N$ ' is used to test a model trained from all other data) [D2].



**Figure 2.2** Summary of sampling types from decades of SE literature on software quality assurance.

- **Percentage:** The historical data is stratified by some percentage, like 80-20%. The minimum % this work found was 67% [Tan18].
- **Release:** The models are trained on the immediate or more past releases in order to predict defects on the current release [Ben19].
- **Month:** When 3 or 6 months of historical data is used to predict defects in future files, commits, or release [McI17].
- **Slice:** An arbitrary stratification is used to divide the data based on a specific number of days (like 180 days or six months in [Zha14]).

The challenge is not only in the amount of data, but the underlying software analytic models were updated all the time (say for every future release) and reasoning from the shallow end of the project life-cycle (which will be detailed later in Chapters 5 and 6). Such an approach made decision-making very hard for practitioners and made them lose trust in these oracles (machine learning models).

#### 2.4.2 Confusions of Practitioner Beliefs

This section argues that just because software developers say they believe in “X”, that does not necessarily mean that “X” is true. Jørgensen & Gruschke [Jør09] note that in Software Engineering (SE) domain, seldom lessons from past projects are used to improve future reasoning (to the detriment of new projects). Passos et al. note that developers often assume that lessons learned from a few past projects are general to all future projects [Pas11]. Devanbu et al. record opinions about software development from 564 Microsoft software developers from around the world [Dev16]. They comment that programmer beliefs can (a) vary with each project and (b) may not necessarily correspond with actual evidence in their current projects. Menzies and Nagappan et al. offer specific examples of this effect. Nagappan et al. [Nag15] have shown that the much-feared *goto* statement is usually benign and sometimes even useful. Menzies et al. recently found no evidence for the *delayed issue effect* [Men17] (“*The longer a bug remains in the system, the exponentially more costly it becomes to fix*”) in 171 industry projects. That study shows that a widely held belief by both practitioners and academics cannot always be assumed to hold.

This work asserts that it is important to quantitatively assess SE beliefs, as such beliefs are used by

- *Practitioners* when they justify design or process decisions; e.g. “better not use goto statements in our code”;
- *Managers* to justify purchases or training programs or hiring decisions; e.g. “test-driven development processes are best”;
- and *Researchers* as they select what issues they should explore next; e.g. “it is better to remove more bugs, earlier in the life-cycle, since the longer they stay in the code, the more expensive it becomes to remove them.”

However, the justification for such beliefs may be weak. Nagappan et al. recently rechecked and rejected Dijkstra’s famous comment that goto is necessarily considered harmful [Nag15]. As to early bug removal, Menzies et al. looked for evidence about whether or not “the longer a bug remains in the system, the exponentially more costly it becomes to fix.” Unfortunately, an extensive literature survey found only ten papers that experimented with this issue, of which five did, and five did not support this belief [Men17]. Further, Fucci et al. reviewed numerous studies on test-driven development and found no evidence of an advantage from writing tests before writing code [Fuc17]. To say the least, this result is very different from numerous prior claims [Fra03],

More generally, Devanbu et al. reported at ICSE’16 just how widely practitioner beliefs at Microsoft diverged from each other and from the existing empirical evidence [Dev16]. Also, this thesis in Chapter 4 reasons the discrepancy between practitioners and empirical evidence by documenting the poverty of evidence for numerous defect prediction beliefs in dozens of software projects.

## 2.5 Checking the Conclusion Instability Effect

This whole thesis is predicated on the practitioner (and researcher) confusion effects documented above. Accordingly, before going any further, it is prudent to check that effect using new data not explored above. Therefore, the challenge here is how to analyze decades-old beliefs and beliefs specific to software quality. Some of the challenges are,

- How should the beliefs be modeled for experimentation?
- What type of data do we need?
- What are the metrics required to assess?
- How to make reasonable modeling assumptions that do not deviate from SE literature?
- How to use the understanding to devise a method to construct stable oracles?

### 2.5.1 Investigating Software Engineering Theories

This section discusses SE beliefs about software quality, developer productivity, and programming expertise in the literature. Later in Chapter 3 this work assesses a group of five most prevalent SE beliefs listed in Table 2.1 using the Personnel Software Process databases from the Software Engineering Institute [Wil19].

**Table 2.1** SE Beliefs studied in this thesis.

#	Belief	Conceived
1	<i>Productivity and reliability depend on the length of a program's text, independent of language level used.</i>	Corbató's law [Cor69] (1969)
2	<i>Object-oriented programming reduces errors and encourages reuse.</i>	Dahl-Goldberg Hypothesis [Dah01; Gol83] (1967 & 1989)
3	<i>Quality entails productivity.</i>	Mills-Jones Hypothesis [Mil83; Cob90] (1983 & 1990)
4	<i>Individual developer performance varies considerably.</i>	Sackman's Second law [Sac66] (1968)
5	<i>It takes 5000 hours to turn a novice into an expert.</i>	Apprentice's law [Nor93] (1993)

### 2.5.2 Quality

Belief 2 claims that “*Object-oriented programming reduces errors and encourages reuse*”; i.e., some groups of programming languages induce more defects than others. In the literature, there is some support for this claim:

- Ray et al. analyzed Open Source (OS) projects and found a modest but significant effect of programming languages affecting software quality [Ray14].
- Kochhar et al. [Koc16; Bis13] showed some languages used together (interoperability) with other languages induced defects.
- Bhattacharya and Neamtiu [Bha11] argue that C++ is a better choice than C for both software quality and developer productivity.
- Mondel et al. empirically assessed four beliefs related to systems testing. They found evidence for an old belief based on a more reused code to be harmful [Tho97] in one of the two organizations they assessed [Mon17].

Belief 3 claims that “*Quality entails productivity*”; i.e., this belief implies a relationship between quality and productivity. Mills by applying Cleanroom Software Engineering, showed the possibility of simultaneous productivity and software quality improvements in both commercial and research projects [Mil83].

### 2.5.3 Productivity

Much prior work [Sac66; Ker06; LaT20] in the past decades studied developer productivity. Belief 1 titled “*Productivity and reliability depend on the length of a program's text, independent of language level used*” implies that Lines of Code (LOC) is a better indicator of software quality and productivity than some programming languages. To the best of this work's knowledge, this 1969 belief is not well explored in the past. Compared to the late '60s, practitioners now write code in numerous programming languages using tools (like Integrated Development Environments) to catalyze software development. Thus it is essential to revisit the claimed effect.

Interestingly, some researchers acknowledge the widely held belief that some good developers are

much better (almost 10X) than many poor developers [Sac66]. Belief 4 is centered around the belief titled “*Individual developer performance varies considerably.*” On related lines of thought, using the same data set, Nichols pointed out that a developer who is productive in one task is not necessarily productive in another [Nic19]. That result warns us that even if one does find a *hero* [Agr18b] developer, they may not remain *heroes* consistently. Thus the focus should be to answer whether this productivity variance also impacts software quality? If it does not, then practitioners can confidently withdraw their large appeal around these moderate productivity variances in practice.

While exploring literature on developer productivity, I observed a joint debate on universal productivity metrics,

- In one study, Vasilescu et al. measured productivity as the number of pull requests to show productivity improvements through Continuous Integration practice in the GitHub arena [Vas15].
- In another recent study, Murphy et al. showed non-technical factors (self-rated metric) were good predictors for productivity [MH19].
- Suggestions about how to augment traditional measures such as incorporating rework time were also discussed in the past [Pau06].

Since all the above productivity measures discussed have their limitations, let us lean towards the most prevalent measure, ‘production rate’ (program size over time) used in the literature. Therefore, the list of measures used in this study refers to §3.7.1.

#### **2.5.4 Expertise**


Two common beliefs are that experts perform the same task better (higher quality and meet deadlines) than novices and that expertise is built over time. The differences between experts and novices are discussed in various domains [Eri04; Eri93]. In SE in 1985, Wiedenbeck considered 20 developers in two equal groups of 10 and found the expert group significantly better in specific programming sentence identification tasks than the other novice group. The expert group had 20,000 hours (mean) experience in their programming languages, whereas the novice population had as little as 500 hours (mean).

Although some studies have highlighted there is more than just years of experience to expertise [Bal18], in this work it is essential to revisit prevalent beliefs. Primarily belief five titled “*It takes 5000 hours to turn a novice into an expert*”; is known to influence software quality and developer productivity. For example, a 2014 TSE article by Bergersen et al. claimed that the first few years of experience correlated with developer performance. However, later, a 2017 EMSE article by Dieste et al. found years of experience to be a poor predictor of developer productivity and quality [Die17].

#### **2.5.5 Investigating Software Quality Theories**

To start the research of this work, this work found five defect prediction papers [Lo15; Xia17; Xia19; Zou18; Wan18a] which list practitioner beliefs, but do not test those beliefs with respect to empirical evidence. From this, a recent qualitative study from IEEE TSE (see Table 2.2) work by Wan et al. [Wan18a] was chosen. This work performed a comprehensive study by gathering practitioners’ opinions (395 responses) on various defect prediction research hypotheses discussed in the literature between

2012-2017. More importantly, they highlight practitioners’ agreement % on defect prediction metrics, prioritization strategy, etc. In support of that work, this work finds that many of the beliefs reported in [Wan18a] by Wan et al. were also reported previously in [D’A10; Kam12]. In this work, this work revisits these beliefs independently to look for current evidence and reasoning disconnect between beliefs and evidence using the prevalence of their support.

 **Objective:**  
 To export  $P_{BX}$  which is a population of significant  $\rho$  scores for a belief  $BX$ , where each score is computed as a correlation between,  $F_{BX}$  &  $F_D$  and collected overall releases  $R$  in a project  $P$ . This is repeated for all 37 projects and ten beliefs.

To support this objective:

$r$  : is a release in project  $p$ . Ignore first release of all projects  $r > 1$ .

$F$  : is a file created or modified during the pre-release  $r$  period.

$BX$  : denotes metric associated with beliefs 1 to 10,  $X \in [1..10]$ .

$F_{BX}$  : denotes metric captured on file  $F$

$D$  : defects fixed 6 months after  $r$  (post-release bugs)

$F_D$  : defects fixed 6 months after  $r$  (post-release bugs) on  $F$

This work employs the traditional release-based approach to assess the beliefs. First, capture metrics in the pre-release period to find associations with post-release bugs six months after the release. Second, assess the beliefs “ $BX$ ” for all releases in a project. Lastly, build a population with the effect scores collected in all releases and projects to be analyzed in the §5.4. In each release, the belief metric for each distinct file “ $F$ ” modified in a release  $r$  is computed; where  $r \in R$  (total number of releases). And defect proneness “ $D$ ” is the number of post-release bugs. Then “ $F_D$ ” is the number of post-release ( $r + 6$ months) bugs on file “ $F$ ”. Hence, “ $F_D$ ” computation is common across all the beliefs whereas “ $F_{BX}$ ” (belief metric on a file) captured during the pre-release period  $r$  is elucidated in the following sections.

In the following, this work will discuss the specifics of how to assess the beliefs of Table 2.1.

**Table 2.2** Practitioners’ agreement % on defect prediction metrics (beliefs) reported in a recent IEEE Transactions on Software Engineering (TSE) paper by Wan et al. [Wan18a].

#	Belief	%
<b>Belief 1</b>	A file with a complex code change process tends to be buggy.	76
<b>Belief 2</b>	A file that is changed by more developers is more bug-prone.	64
<b>Belief 3</b>	A file with more added lines is more bug-prone.	61
<b>Belief 4</b>	Recently changed files tend to be buggy.	58
<b>Belief 5</b>	A commit that involves more added and removed lines is more bug-prone.	57
<b>Belief 6</b>	Recently bug-fixed files tend to be buggy	49
<b>Belief 7</b>	A file with more fixed bugs tends to be more bugprone.	48
<b>Belief 8</b>	A file with more commits is more bug-prone.	46
<b>Belief 9</b>	A file with more removed lines is more bug-prone.	35
<b>Belief 10</b>	Files with fewer lines contributed by their owners (who contribute most changes) are bug-prone.	30

To start the research of this thesis, this work found five papers [Lo15; Xia17; Xia19; Zou18; Wan18a] that list practitioner beliefs about defect prediction but do not test those defect prediction beliefs with respect to empirical evidence. From this, a recent qualitative study was chosen from IEEE TSE (see Table 2.2) paper by Wan et al. [Wan18a].

That work performed a comprehensive study by gathering practitioners' opinions (395 responses) on various defect prediction research hypotheses discussed in the literature between 2012-2017. More importantly, they highlight practitioners' agreement % on defect prediction metrics, prioritization strategy, etc. In support of that work, many of the beliefs reported in [Wan18a] by Wan et al. were also reported previously in [D'A10; Kam12]. This thesis revisits these beliefs independently to look for current evidence and reasoning disconnect between beliefs and evidence using the prevalence of their support.

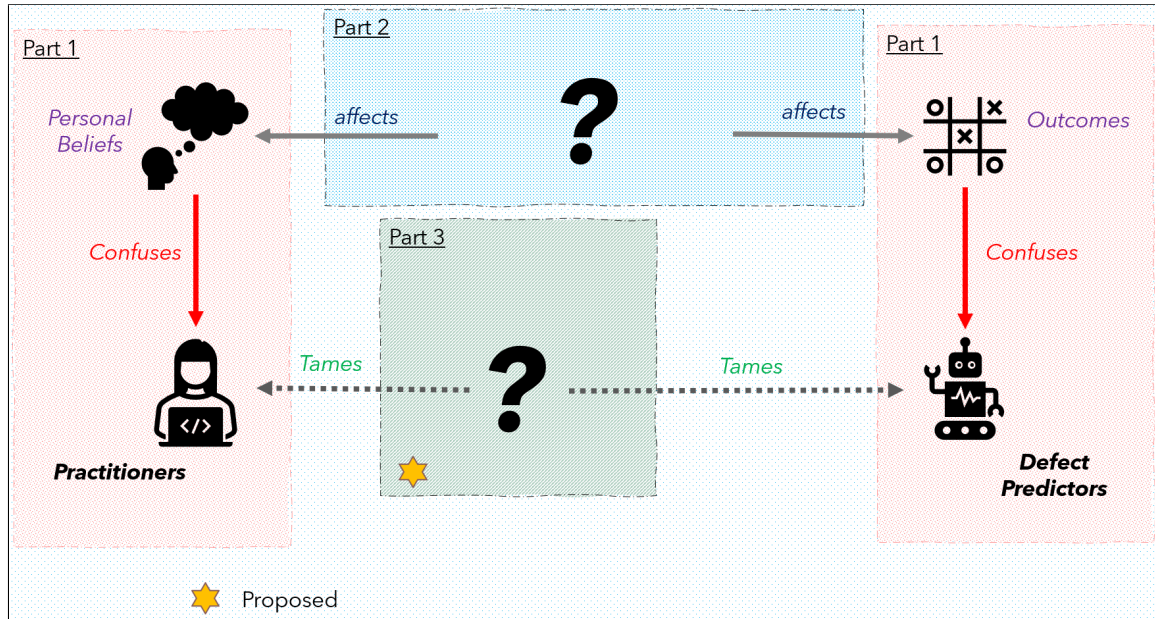
### **2.5.6 Data-lite Software Analytics**

Before moving on, let us first look into the related work on early life cycle defect prediction. In 2008, Fenton et al. [Fen08] explored the use of human judgment (rather than data collected from the domain) to handcraft a causal model to predict residual defects (defects caught during independent testing or operational usage) [Fen08]. Fenton needed two years of expert interaction to build models that compete with defect predictors learned by data miners from domain data. Hence this thesis does not explore those methods here since they were very labor-intensive.

In 2010, Zhang and Wu showed that it is possible to estimate the project quality with fewer programs sampled from an entire space of programs (covering the entire project life-cycle) [Zha10a]. Although the proposed approach also draws fewer samples (commits), they are sampled 'early' in the project life-cycle to build defect prediction models. In another 2013 study about sample size, Rahman et al. stress the importance of using a large sample size to overcome bias in defect prediction models [Rah13b]. This work finds the proposed 'data-lite' approach performs similar to 'data-hungry' approaches while this work does not deny bias in defect prediction data sets. The proposed approach and recent defect prediction work handle bias by balancing defective and non-defective samples [Ben19; Agr18a] (class-imbalance).

Recently (2020), Arokiam and Jeremy [Aro20] explored bug severity prediction. They showed it is possible to predict bug severity early in the project development by using data transferred from other projects [Aro20]. Their analysis was on the cross-projects, but unlike this work, they did not explore just how early in the life cycle did within project data became effective. In similar work to Arokiam and Jeremy, in 2020, Sousuke [Ama20] explored Cross-Version defect prediction (CVDP) using Cross-Project Defect Prediction (CPDP) data (approaches). After much experimentation, Sousuke recommends training oracles using recent project release data. But Sousuke's study was not as extensive (only 41 project releases) as this work. However, unlike Sousuke, this work offers contrary evidence in this work. The endorsed policy based on earlier commits works similar to all other prevalent policies (including the most recent release) reported in the literature. Notably, this work assesses the approach on 1000+ project releases and evaluates that on seven performance measures.

Even in the context of advanced software analytics that part-3 of this thesis will explore is data-hungry. For example, the following state-of-the-art software analytics methods in,



**Figure 2.3** This chapter filled the missing pieces ‘?’ of part-1 in the thesis landscape.

- DODGE (Hyper-parameter-optimization) by Agrawal et al. [Agr19]
- TLEL (ensemble learning) by Yang et al. [Yan17] and
- Bellwether (transfer learning) by Krishna et al. [Kri18].

These advanced methods ingest the entire project life-cycle data (or many projects) to improve the predictive performance of defect prediction models. However, later in chapter 6 using the early data-lite methods, these methods will be simplified (either augmented or deprecated).

In summary, as far as this work is concerned, this is the first study to perform an extensive comparison of prevalent sampling policies practiced in the defect prediction space.

## 2.6 Summary

This chapter answered the missing pieces ‘?’ of part-1 of the roadmap shown in Figure 2.3. The next chapter will discuss all the data and methods required to investigate and tame confusions in software engineering.

## CHAPTER

# 3

# DATA, METHODS, METRICS & MEASURES

This chapter offers details about the data used in experiments, various algorithms, features used by models and evaluation measures used throughout this thesis.

### 3.1 Introduction

In order to document, understand the root cause, and tame conclusion instability (discussed next in Chapter 4 and 5), considerable effort is spent in collecting data and in modeling appropriate experimental setups.

This thesis collected three data sources that use different metric sets, specifically:

- Industry data from Personal Software Process (PSP) to assess software engineering theories.
- 37 Open Source projects from GitHub to assess Defect Prediction Beliefs.
- 240 Open Source projects (155 popular and 85 unpopular) from GitHub to portray the efficacy of the novel data-lite method proposed in this work.

The three sets of metrics collected in each of those datasets are:

- In the PSP data, program size, production rate, and quality metrics are captured.
- In 37 Open Source projects, metrics related to defect prediction beliefs are inferred along with project release information.
- 14 process metrics are collected from the 155 OS GitHub projects along with project release information.

This thesis repeatedly uses two statistical tests to validate the findings, they are:

**Table 3.1** Count of Engineers (Developers) by Domain

Product Domain	Number	Product Domain	Number
Software Services	378	Telecom	92
Business IT	351	Financial	68
Automation&Control	112	Government	66
Accounting Software	99	Embedded	55
Consumer Electronics	99	Aerospace	51
Automotive	97	Other	319

- The Scott-Knott test to rank different treatments (populations)
- Spearman’s Correlation

Then, the oracles are built using six classification algorithms, they are:

- Logistic Regression (LR);
- Nearest neighbor (KNN) (minimum five neighbors);
- Decision Tree (DT);
- Random Forrest (RF);
- Naïve Bayes (NB);
- Support Vector Machines (SVM)

Additionally, both advanced methods like optimizers, ensemble learners, transfer learning, and trivial methods are employed in building oracles (later in chapter 6). Finally, the conclusion instability is measured using eight performance measures: Brier, Initial number of false alarms, recall, false-positive rate, the area under the receiving operating characteristic curve, distance to heaven, g-measure, and mathew’s correlation coefficient.

### **3.2 Data Source: Industry data to assess Software Engineering Theories**

To assess researcher beliefs about SE, this specific work needs sufficient data. The data comes from a decades-long training program. The consultants from the Software Engineering Institute (SEI, based in Pittsburgh, USA) traveled around the world to train developers in personal data collection. The “Personal Software Process” (or PSP) [Hum95] is based on a belief that a disciplined process can improve productivity and quality [Pau10]. Specifically, if a practitioner uses PSP, they are encouraged to guess how long some tasks will take and then explain any differences between the predicted and actual effort.

There are several reasons to use this data. Firstly, it is a minimal intrusion into the actual development work of practitioners. With the support of the right tools (e.g., with the tools from the SEI), practitioners spend less than 20 minutes per day on the PSP data collection. Hence, PSP can generate accurate and insightful records of actual developer activity [Pau10; Val16; Pau06; Pau05; Nic19].

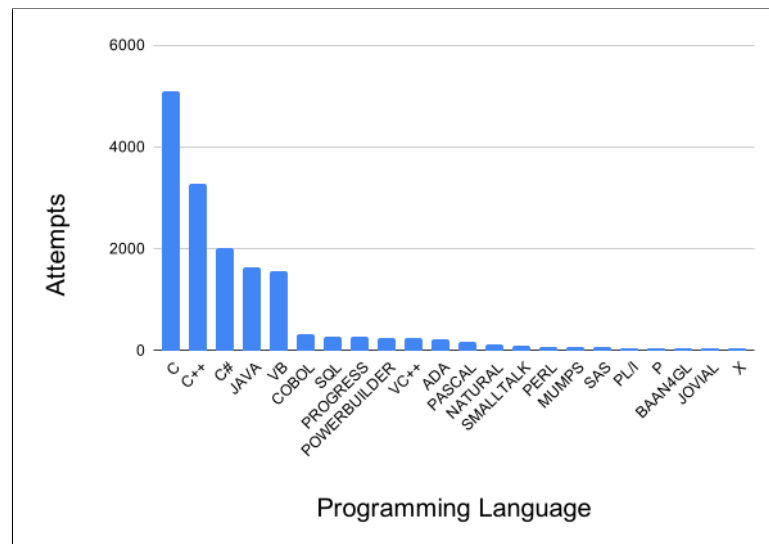
Secondly, when SEI consultants train practitioners in PSP, they use a standard set of ten tasks. The course is taught over 10 class days, with one week focused on measurement and estimation and the second week focused on reviews, design, and quality (and there was typically a minimum two-week gap

between weeks one and two). Hence, this work has data on thousands of developers doing the same set of tasks, using a wide variety of programming methods and tools. For an overview of that data:

- Table 3.1 lists the thousands of developers who have had this PSP training, along with the kind of software they usually develop.
- Figure 3.1 lists the languages used by attendees as they tried to complete the ten programming tasks.
- Table 3.2 sorts the ten tasks (labeled from 1 to 10) from simplest (at level “0”) to hardest (at level “2”). Small dice of a 20-page task 10 specification are presented in Figure 3.2. Concise requirements for task 10 include writing programs to
  - Read a table of historical data using the linked list from task 1
  - Write a multiple regression solver to estimate the regression parameters
  - From user-supplied values of estimates for new LOC, reused LOC, and modified LOC compute the expected effort and prediction interval
  - Print out the results
- Figure 3.3 lists the tens of thousands of defects recorded during the PSP training tasks.

Thirdly, this PSP data comes from industrial practitioners from around the world. The PSP classes were taught in the US, Japan, Korea, Australia, Mexico, Sweden, Germany, the United Kingdom, the Netherlands, and India. Class size ranged from 1 to 20 developers, with a mean of 10.4 and an inter-quartile range of 7 to 14. Only about 3.2% of subjects (123 of 3,832) were from a university setting. While most of the classes, 361 of 373 were taught in the industry to practicing software developers. Early adopters included Air Force, ABB, Honeywell, Allied Signal, Boeing, and Microsoft.

Fourthly, this is high-fidelity data. For example, a study of PSP data collection by Disney and Johnson



**Figure 3.1** Distribution of all the tasks attempted & completed by developers using a specific programming language. For a fair sample size comparison, this work only considers tasks completed using C, C++, C#, Java, and VB programming languages in this study. This bar-chart ignores 15 other languages that less than 30 developers have completed.

**Table 3.2** Overview of the data set composed of 10 tasks at various levels completed in one of the five programming languages this work considered. Level 2 (the rows are shown in gray) task have the highest complexity than its earlier levels.

Level	Task	Developer Attempts	Programming Languages
0	1	1,356	C, C++, C#, Java and VB
	2	1,356	
	3	1,356	
1	4	1,356	
	5	1,356	
	6	1,356	
2	7	1,356	
	8	1,356	
	9	1,356	
	10	1,356	

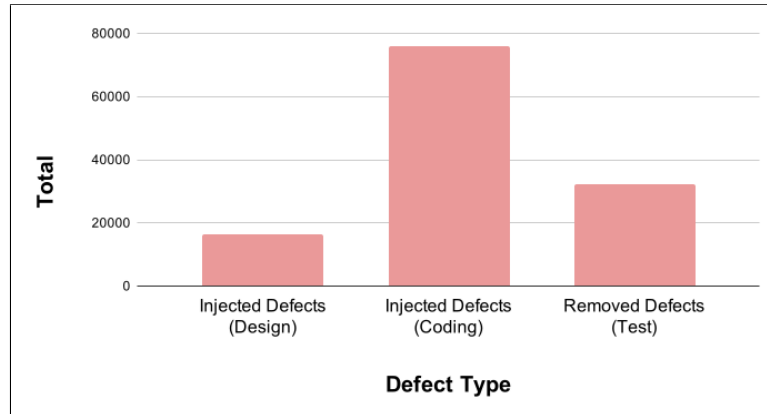
Program Requirements for Assignment 10A	
<b>Workshop Objectives</b>	After this workshop, you will <ul style="list-style-type: none"> <li>• understand program 10A requirements</li> <li>• have completed program 10A planning               <ul style="list-style-type: none"> <li>- conceptual design</li> <li>- size estimate</li> <li>- resource estimate</li> <li>- defect estimate</li> </ul> </li> </ul>
<b>Program 10A Requirements</b>	Using PSP2.1, develop program 10A to calculate the three-variable multiple-regression estimating parameters ( $\beta_0$ , $\beta_1$ , $\beta_2$ , $\beta_3$ ).  Make an estimate from user-supplied inputs, and determine the 70% and 90% prediction intervals for the estimate.  Use a linked list to store the data and the Simpson's integration routine from 5A to calculate the $t$ distribution.  Test the program using the data in table D16, page 763, as the historical data. For the user inputs, use 185 LOC of new code, 150 LOC of reused code, and 45 LOC of modified code.

**Figure 3.2** The task 10 (or assignment 10A) listed in Table 3.2 asks the developers to extend program 6 to calculate the three-parameter multiple-regression factors from a historical data set, then estimate the development effort with a 70% to 90% prediction intervals.

[Joh99] using ten developers who wrote 89 programs found that manual collection and calculations on paper led to a 5% error rate, mostly in derived calculations or transcription errors. One explanation for this low error rate is the way the data was collected. The SEI authorized instructors to review each developer's (student) PSP data as a required criterion for completing the task. Grading rubrics included self-consistency checks, checks to ensure that estimates and actuals are consistent with historical data, and comparisons with peers for data from each sub-process. Developers are also shown class summaries for comparison to their peers. Hence, various studies [Rom08; Val12; Gra13] have found the data to be very accurate.

As to the nature of the ten programming tasks:

- They varied slightly in size, difficulty, and complexity.



**Figure 3.3** Distribution of 124,521 defects recorded by developers while completing the 10 tasks using 5 programming languages C, C++, C#, Java and VB. Specifically 16,437 design and 75,927 coding defects were injected and 32,157 test defects were removed by 1,356 developers.

- They were chosen to be sufficiently challenging to generate useful data on estimation, effort, size, and defects. They could typically be completed in an afternoon with 100 to 200 Lines of Code in a 3rd Generation language.
- Two programs were dedicated to counting program size; the remainder were primarily statistical, including regression, multiple regression, a Simpson's rule integral, Chi function, Student's T function, and prediction intervals.
- Developers were not expected to be domain experts and were provided a specification package that included descriptions of necessary formulas, algorithms, required test cases, and numeric examples suitable for a developer with no specific statistical expertise.

The developers were instructed to bring their own devices to the class to understand that they should be familiar with the development environment and use the programming language with which they were most comfortable. That work made it clear that this was a process course, not a programming course, and that their results depended upon not introducing confounders. Students were also instructed that the purpose of the exercises was to produce a measurable amount of code, effort, and defects; therefore, they should not use library procedures. However, primitive language functions such as square root, logs, and trig functions were expected.

The developers collected their personal data for effort, size, and defects using the PSP data framework, which measures direct time in minutes and program size in new and changed lines of code. Developers were instructed to build solutions with incremental cycles of design, code, and test, selecting their own increment size, typically a component or feature of 25 to 50 lines of code. However, some developers could produce working programs in a single cycle; most used 3 to 5 cycles, depending on their solution size and complexity. For effort accounting, each increment was initially designed and coded (creation), reviewed (appraisal), followed by the compile and test (failure). All-time required to achieve a clean compile was attributed to compile. All rework necessary to get the tests to pass was attributed to the test. The accounting highlighted rework so that rework could be minimized.

The developers counted all defects that escaped a development phase. A defect was defined as any change needed to correct the program that was discovered after a phase was considered to be complete. Detection of defects was primarily during a personal review, a compile, or a test. For example, a coding defect would typically be discovered during code review or compile but might escape into testing. Defect data included the fixed time, the type, the phase origin, and the discovery phase.

To help highlight rework, the phase was defined as a logical step, where a time of activity was the primary phase rather than the literal activity performed. For example, coding must be followed by compile, then test. A defect in the test did not trigger a new accounting cycle. Any changes resulting from code and re-compile were attributed to the test. IDE and static analysis tools required additional instruction for consistent accounting.

The use of static code analysis was uncommon and prohibited until after the compile, and then it must be considered part of the compile phase. Compile was complete when all discovered defects were resolved. IDE real-time syntax checking presented another unique condition, in which students chose to use IDE was not recorded. Those using an IDE were instructed to disable real-time syntax checking to have the maximum number of defects available for finding in the review or compile. That work later abandoned the guidance to disable IDE checking because the individual baselines were sufficient for the course objectives. Nonetheless, the instruction to disable IDE checks affected the courses from which this data was taken.

For accounting purposes, PSP categorizes the activities as Creation (Design, Code), Appraisal (Design review, code review), and failure (compile, test). These were logical phases rather than strict activities (i.e., A logical sequence is to design, code, compile, and test a piece of code).

For small programs, a waterfall was practicable, but most chose to proceed through the phases using incremental development. If incremental, each increment proceeded through the phases proceeded without reentry. For example, fixing a bug in the test required some coding and re-compile, but effort and defects were assigned to the test. This accounting choice made rework more visible and allowed additional auditing of the data quality.

Defects were mostly injected during a creation phase, design, or code, with a few, injected while fixing another defect. Defects were discovered during the compilation, execution of test cases, or by a personal review. The course design was to baseline the defects levels in compile and test then demonstrate the ability of review to find at least 60% of those defects before escaping into a failure phase.

### **3.2.1 Data Filtering**

This work filtered the PSP data as follows:

- Although developers used numerous programming languages to complete the tasks, predominately 85% of the developers used C, C++, C#, Java, and Visual Basic (VB) programming languages as shown in Figure 3.4. This work focuses on these programming languages since they are very prevalent in the industry.
- For simplicity in presentation belief, 1 and 4 use data only from level 2 tasks (labeled 7,8,9, and 10) listed in Table 3.2. For all other beliefs (2, 3, and 5), this work consider all the ten tasks.

- Suiting to the nature of the beliefs, this work use data from the appropriate type of defect in the analysis. The three types of defects this work examines are shown in Figure 3.3.

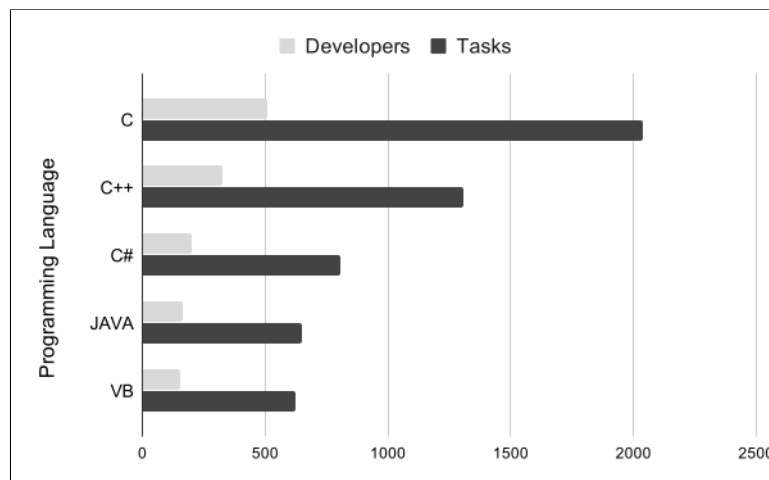
### 3.3 Data Source: Open Source data to assess Software Quality Theories

Turning then to GitHub<sup>1</sup>, this part chose 50 GitHub repository links from a recent defect prediction paper [Wal18] that uses Munaiah et al. GitHub project database [Mun17]. Their work simplified choosing substantive Open Source (OS) samples from GitHub, attributing several metrics such as maturity, active developers, license, etc. This work mined file-level commit histories and project-release information using GitHub API. To minimize bias in the modeling, this work applied the following sanity checks and discarded projects that have less than,

- 1000 commits
- 10% Bug Fixes
- 5 releases
- 30 developers
- 3 years of activity

Furthermore, this resulted in 37 projects shown in Table 3.3. The selected projects use many current languages such as Python, Ruby, Java, JavaScript, PHP, etc. To remove noise, this work ignores test cases, configuration files, and static resources such as text, readme, images, etc. To achieve this, first, this work identified source code extensions from the samples they are *py, java, rb, c, cpp, h, php, sh, cs, scss, html, scala, js, css, clj, ctp, erb, go, haml, hs and sql*. Essentially this work consider only the file paths the ends with these extensions, with an additional check that the file/path names do not contain “test”, to eliminate test cases.

<sup>1</sup><https://github.com>



**Figure 3.4** This bar-chart portrays the proportion of 1,356 developers completing 5,424 level 2 (labeled 7, 8, 9, and 10) tasks grouped by a specific programming language.

**Table 3.3** 37 OS projects developed in popular programming languages. Some projects are developed using multiple programming languages. (Rb - Ruby, Py - Python, JS - JavaScript and SCSS - Sassy-CSS)

URL (github.com/)	Language	URL (github.com/)	Language
<i>activemerchant/active_merchant</i>	Rb	<i>xetorthio/jedis</i>	Java,HTML
<i>activeadmin/activeadmin</i>	Rb,SCSS	<i>spotify/luigi</i>	Py,JS
<i>puppetlabs/beaker</i>	Rb,Shell	<i>lra/mackup</i>	Py
<i>boto/boto3</i>	Py	<i>mikel/mail</i>	Rb
<i>thoughtbot/bourbon</i>	SCSS,HTML	<i>Seldaek/monolog</i>	PHP
<i>bundler/bundler</i>	Rb,HTML	<i>omniauth/omniauth</i>	Rb,HTML
<i>teamcapybara/capybara</i>	Rb	<i>aws/opsworks-cookbooks</i>	Rb
<i>Codeception/Codeception</i>	PHP,HTML	<i>thoughtbot/paperclip</i>	Rb
<i>oocici/coi-services</i>	Py,C/C++	<i>getpelican/pelican</i>	HTML,Py
<i>pentaho/data-access</i>	Java,JS	<i>cakephp/phinx</i>	PHP
<i>pennersr/django-allauth</i>	Py,HTML	<i>propelorm/Propel2</i>	PHP,HTML
<i>encode/django-rest-framework</i>	Py,HTML	<i>puppetlabs/puppetlabs-apache</i>	Rb
<i>django-tastypie/django-tastypie</i>	Py	<i>sferik/rails_admin</i>	JS,Rb
<i>doorkeeper-gem/doorkeeper</i>	Rb	<i>reactjs/react-rails</i>	JS,Rb
<i>drapergem/draper</i>	Rb,HTML	<i>resque/resque</i>	Rb
<i>errbit/errbit</i>	Rb,HTML	<i>restsharp/RestSharp</i>	C#
<i>jordansissel/fpm</i>	Rb	<i>mperham/sidekiq</i>	Rb,JS
<i>ros-simulation/gazebo_ros_pkgs</i>	C&C++	<i>plataformatec/simple_form</i>	Rb
<i>ruby-grape/grape</i>	Rb,HTML		

### 3.3.1 Data distribution

Overall this sample contains data modified from 2005 to 2019 by 12,361 developers in over 524,000 file entries. These modifications were made to 127,950 active branch commits (in all, 19,617,483 line insertions and 13,642,341 line deletions). On average, the chosen set of projects has been active for over seven years (see the distributions of Figure 3.5).

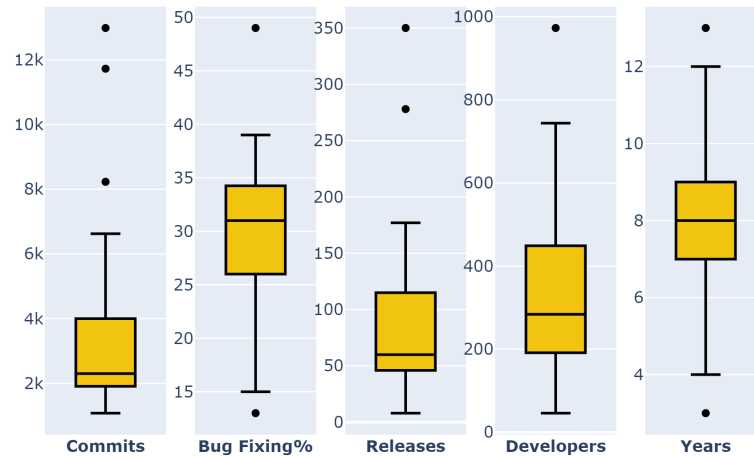
### 3.3.2 Data Attributes

To orchestrate this dataset, the entire commit history of a project was mined. A tuple of the main dataset contains the following attributes unique *commit\_id*, *commit\_time*, the author who pushed the changes *commit\_author*, affected *file\_path*, number of lines inserted and deleted (*insertions*, *deletions*). Importantly for every commit, a label Bug Fixing Commit (*BFC*) is set as 1, if it was pushed in an attempt to fix the bug (s) or 0 if otherwise. This is achieved this by scanning the commit message for any derivatives of the following stemmed words like *bug*, *fix*, *issu*, *error*, *correct*, *proper*, *deprecat*, *broke*, *optimize*, *patch*, *solve*, *slow*, *obsolete*, *vulnerab*, *debug*, *perf*, *memory*, *minor*, *wart*, *better*, *complex*, *break*, *investigat*, *compile*, *defect*, *inconsist*, *crash*, *problem* or *resol*. Lastly, the releases of each project are extracted using Git releases/tags.

## 3.4 Data Source: Open Source data to assess Oracles

This section describes the data used in chapters 5 and 6 and what “clean” and “defective” (buggy) commits means.

All the data comes from open source projects hosted on GitHub that other SE researchers can replicate. For details on the data features, see Table 3.4 and distribution see Figure 3.6 and Figure 3.7.



**Figure 3.5** Distribution of Commits (2304), Bug Fixing (31%) Releases (60), Developers (284) and Years (8) active among 37 projects.  $\bar{x}$  (median)

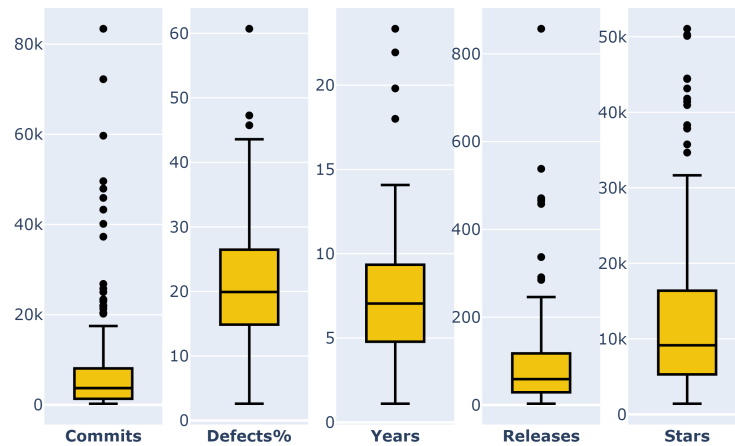
**Table 3.4** 14 Commit level features that Commit Guru tool [Kam12; Ros15] mines from GitHub repositories

Dimension	Feature	Definition
Diffusion	NS	Number of modified subsystems
	ND	Number of modified directories
	NF	Number of modified Files
	ENTROPY	Distribution of modified code across each file
Size	LA	Lines of code added
	LD	Lines of code deleted
	LT	Lines of code in a file before the change
Purpose	FIX	Whether the change is defect Changes that fixing the defect are more likely to introduce more defects fixing ?
History	NDEV	Number of developers that changed the modified files
	AGE	The average time interval from the last to the current change
	NUC	Number of unique changes to the modified files before
Experience	EXP	Developer experience
	REXP	Recent developer experience
	SEXP	Developer experience on a subsystem

The dataset curated by Munaiah et al. [Mun17] has numerous criteria to differentiate an engineering project from a trivial one (such as homework assignments). Using that list, a large random sample of projects were chosen and mined using the Commit-Guru [Ros15] tool. Projects were rejected according to the standard sanity checks by SE GitHub miners [Mun17; Yan20]:

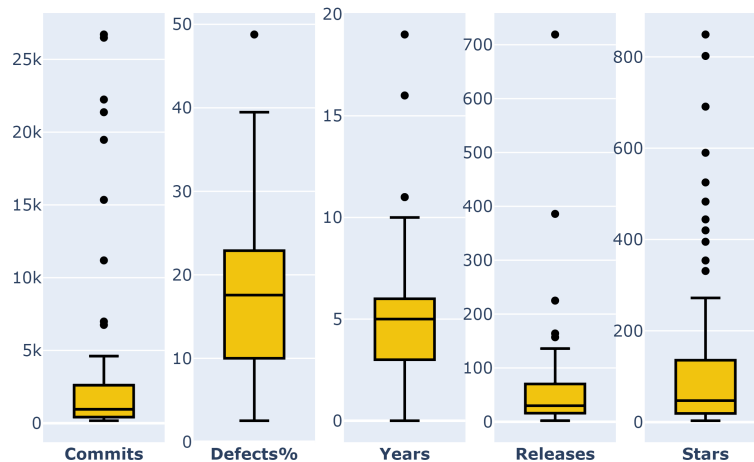
- Less than 1% defective commits;
- Less than two releases;
- Less than one year of activity;
- No license information;
- Less than five defective and five clean commits.

All these project data that comes from open source (OS) GitHub projects [Cit] were mined randomly



**Figure 3.6** Distributions seen in all 1.2 millions commits of all 155 *popular* (stars > 1000 projects: median values of commits (3,728), percent of defective commits (20%), life span in years (7), releases (59) and stars (9,149).

using *Commit Guru* [Ros15]. *Commit Guru* is a publicly available tool based on a 2015 ESEC/FSE paper used in numerous prior works [Xia16b; Kon20]. *Commit Guru* provides a portal where it takes a request (URL) to process a GitHub repository. It extracts all commits and their features to be exported to a file. Commits are categorized (based on the occurrence of certain keywords) similar to the approach in SZZ algorithm [Š05]. The “defective” (bug-inducing) commits are traced using the git diff (show changes between commits) feature from bug fixing commits; the rest are labeled “clean”. Then the codes associated with those changes were then summarized by *Commit Guru* using the attributes of Table 3.4. Those attributes became the independent attributes used in the analysis. Note that the use of these particular attributes has been endorsed by prior studies [Kam12; Rah13a].



**Figure 3.7** Distributions seen in all 258,000+ commits of all 89 *unpopular* (stars < 1000) projects: median values of commits (957), percent of defective commits (18%), life span in years (5), releases (30) and stars (47).

But, *CommitGuru* does not provide project release information. Therefore we cloned each GitHub project to our local machine and extracted GitHub releases/tags information by executing the following command shown below:

```
git log --tags --simplify-by-decoration --pretty="format:%ai %d"
```

Our sampling was done twice:

- Once for *popular* project; i.e., those with more than 1000 “star” rankings in GitHub;
- Once for *unpopular* projects; i.e., those with less than 1000 “star” rankings in GitHub.

This sampling yield 155 popular projects and 85 unpopular projects. 1000 stars were used as the cut-off since that what was used in Munaiah et al. [Mun17]. Also, looking at Figures 3.6 and 3.7, we see that above and below 1000 stars, the distributions are very different. Specifically, above and below 1000 stars, the median number of stars is 9,149 and 47 (respectively).

**Note:** Chapter 5 will explore only *popular* projects, whereas chapter 6 will extend the scope of early methods endorsed in chapter 5 and run all experiments using both popular and unpopular projects.

## 3.5 Statistical Tests

- *Rank*: Clusters a list of populations to report significant differences.
- *Correlation*: Reports significant associations between two variables.

### 3.5.1 Rank Treatments

Later in the experiments in chapters 4, 5 and 6, this thesis compares populations of various SE and oracle evaluation measures such as defects, production rate, recall etc.,. Note that populations may have the same median. Still, their distribution could be very different, hence to identify significant differences or rank among many populations, this work uses the ScottKnott test recommended by Mittas et al. in TSE’13 [Mit13].

ScottKnott is a top-down bi-clustering approach used to rank different treatments; the treatment could be program size, production rate, defects, etc. This method sorts a list of  $l$  treatments with  $l$ s measurements by their median score. However, before this step sorts a list of  $l$  treatments, this work normalizes <sup>2</sup> the data between  $[0, 1]$ , because the SE measures, like program size, defects, etc., do not typically fall between a fixed range to fit the quartile plots (later in 4). Thus to overcome this issue, this test transforms the list of  $l$  treatments by applying min-max normalization, as shown below. Note that this transformation does not impact the rank of the  $l$  treatments in any way.

$$x' = \frac{x - \min(x)}{\max(x) - \min(x)}$$

Where,

- $\max(x)$  is the global maximum ie., (largest value among the list of  $l$  treatments)

---

<sup>2</sup>Normalization is only applied while assessing SE beliefs

- $\min(x)$  is the global minimum i.e., (least value among the list of  $l$  treatments)

The Scott-Knott approach then splits the normalized  $l$  into sub-lists  $m, n$  in order to maximize the expected value of differences in the observed performances before and after divisions. For lists  $l, m, n$  of size  $l_s, m_s, n_s$  where  $l = m \cup n$ , the “best” division maximizes  $E(\Delta)$ ; i.e. the difference in the expected mean value before and after the spit:

$$E(\Delta) = \frac{m_s}{l_s} a b s(m.\mu - l.\mu)^2 + \frac{n_s}{l_s} a b s(n.\mu - l.\mu)^2$$

Notably, these techniques are preferred since they do not make Gaussian assumptions (non-parametric). To avoid “small effects” with statistically significant results, this work employs the conjunction of bootstrapping and A12 effect size test by Vargha and Delaney [Var00] for the hypothesis test H to check if  $m, n$  are truly significantly different.

### 3.5.2 Correlation Analysis

Spearman’s rank correlation (a non-parametric test) assesses associations between two measures discussed earlier, for example, a correlation between production rate and software quality. This work chose Spearman like some SE quality study [D’A10] recommended to handle skewed data; further, it is unaffected by transformations (such as log, reciprocal, square-root, etc.) on variables.

The Spearman’s rank correlation,  $\rho = \frac{\text{cov}(X, Y)}{\sigma_x \sigma_y}$  between two samples  $X, Y$  (with means  $\bar{x}$  and  $\bar{y}$ ), as estimated using  $x_i \in X$  and  $y_i \in Y$  via

$$\rho = \frac{\sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^n (x_i - \bar{x})^2 (y_i - \bar{y})^2}}$$

This work concludes using both the correlation coefficient ( $\rho$ ) and its associated  $p\_value$  in all the experiments. The correlation coefficient ( $\rho$ ) varies from +1, i.e., ranks are identical, to -1, i.e., ranks are the opposite, where 0 indicates no correlation.

- Higher  $|\rho|$  value indicates strong evidence.
- Lower  $p\_value$  indicates the evidence is statistically significant.

### 3.5.3 Thresholds

This work applies Spearman’s rank correlation ( $\rho$ ) ranges in chapter 4 shown below to discuss strength of the evidence. From the usage of Spearman’s  $\rho$  in this defect prediction literature [Zim07] this work derive the following ranges for  $|\rho|$  :

**Correlation:**

- \* 0.0 to 0.39 as *no support*
- \* 0.4 to 0.49 as *minimum/weak support*
- \* 0.5 to 0.59 as *support*
- \* 0.6 to 0.69 as *strong support*
- \* 0.7 to 1.00 as *very strong support*

*This work acknowledges that these ranges are debatable. All the correlations this work report are at the 99% confidence level (ie.,  $p\_value < 0.01$ ).*

**Rank:**

- \* *Lower Scott-Knott rank indicates better production rate, better quality (fewer defects) and large program size (LOC) ie., Population distribution in Rank 1 is better than Rank 2.*
- \* *Distributions placed in different ranks indicate significantly different population.*

### 3.5.4 Software Engineering Measures

This work uses the three SE measurements below to derive the conclusions while assessing the five beliefs in the next chapter.

**Program size (LOC)** = Lines of Code

**Production Rate** = LOC/Hour (*Coding time*)

**Quality (defects)** = Number of defects

*(Unless specified this work consider defects injected in the coding phase. Other type of defects this work analyze are defects injected in design phase and defects removed in testing phase.)*

As mentioned earlier, information per programming task, such as the number of defects, program size, coding time, etc., is captured in practice by developers. To recollect, developers completed the ten programming tasks of increasing complexity listed in Table 3.2. They used various programming languages, as shown in Figure 3.1, but mainly using C, C++, C#, Java, and VB, are considered in this study. These traditional SE measures are used in these related studies [Woh02; Pau06].

## 3.6 Classifiers (Oracles)

After an extensive analysis, Ghotra et al. [Gho15] rank over 30 defect prediction algorithms into four ranks. This work takes six of their learners that are widely used in the literature. Those learners can be found at all four ranks of the Ghtora et al. study. Those learners were:

- Logistic Regression (LR);
- Nearest neighbour (KNN) (minimum 5 neighbors);

<p>INPUT:</p> <ul style="list-style-type: none"> <li>• A dataset</li> <li>• <math>\mathcal{E} \in \{0.05, 0.1, 0.2\}</math></li> <li>• A goal predicate <math>p</math>; e.g., <math>P_{opt}</math> or <math>d2h</math>;</li> <li>• Objective, either to maximize or minimize <math>p</math>.</li> </ul> <p>OUTPUT:</p> <ul style="list-style-type: none"> <li>• Optimal choices of preprocessor and learner with corresponding parameter settings.</li> </ul> <p>PROCEDURE:</p> <ul style="list-style-type: none"> <li>• Separate the data into train and test</li> <li>• Choose set of preprocessors, data miners with different parameter settings from Table 3.5.</li> <li>• Build a tree of options for preprocessing and learning. Initialize all nodes with a weight of 0.</li> <li>• Sample at random from the tree to create random combinations of preprocessors and learners.</li> <li>• Evaluate <math>N_1</math> (in our case <math>N_1 = 12</math>) random samples on training set and reweigh the choices as follows: <ul style="list-style-type: none"> <li>– Deprecate (<math>w = w - 1</math>) those options that result in the similar region of the performance score <math>\alpha</math> (<math>\alpha \pm \mathcal{E}</math>)</li> <li>– Otherwise endorse those choices (<math>w = w + 1</math>)</li> </ul> </li> <li>• Now, for <math>N_2</math> (<math>N_2 \in \{30, 100, 1000\}</math>) evaluations <ul style="list-style-type: none"> <li>– Pick the learner and preprocessor choices with the highest weight and mutate its parameter settings. Mutation is done, using some basic rules, for numeric ranges of attribute (look for a random value between <math>(best, (best + worst)/2)</math> seen so far in <math>N_1 + N_2</math>). For categorical values, look for the highest weight.</li> </ul> </li> <li>• For <math>N_1 + N_2</math> evaluations, track optimal settings (those that lead to best results on training data).</li> <li>• Return the optimal setting and apply these to test data.</li> </ul>
---

**Figure 3.8** Pseudo-code of DODGE replicated from [Agr19]

- Decision Tree (DT);
- Random Forrest (RF);
- Naïve Bayes (NB);
- Support Vector Machines (SVM)

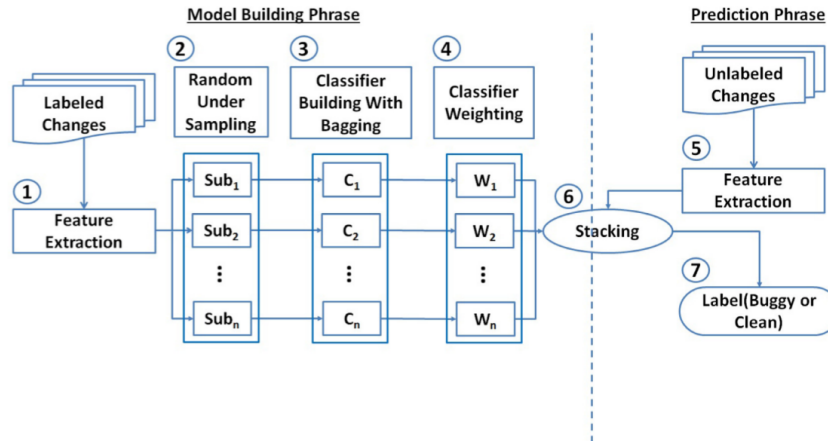
**DODGE:** Agrawal et al.’s DODGE is a state-of-the-art hyper-parameter optimization method extensively assessed for defect prediction [Agr19]. One critical problem in hyper-parameter optimization (such as grid-search or brute force) is the run-time overhead. DODGE overcomes this by terminating much faster by skipping redundant options.

DODGE is an ensemble tree of classifiers and pre-processors as shown in Table 3.5. DODGE is broadly a two-step process as shown in Figure 3.8. First, DODGE iteratively shrinks (prunes the tree) the tuning search space by ignoring redundant options sampled from the tree. Then in the next set of iterations, it finds near-optimal options by looking between the best and worst options seen so far.

DODGE is shown to perform much better than building models with classifiers or pre-processors directly with off-the-shelf default options [Agr19]. Notably, Agrawal et al. have highlighted that DODGE fails on complex data sets (having intrinsic dimensional  $\mu > 8$  ). We found the intrinsic dimensionality of 240 projects explored in this study to be  $\mu < 2$ . Moreover, this being a defect prediction study, we explore DODGE in this study.

**Table 3.5** Hyperparameter tuning options explored in this paper. Note that we make no claim that this is a complete list of options. Rather, we merely claim that a reader of the recent SE literature on hyperparameter optimization might be tempted to try some subset of the following.

<p><b>DATA PRE-PROCESSING</b></p> <p>Software defect prediction:</p> <ul style="list-style-type: none"> <li>• <b>Transformations</b> <ul style="list-style-type: none"> <li>- StandardScaler</li> <li>- MinMaxScaler</li> <li>- MaxAbsScaler</li> <li>- RobustScaler(quantile_range=(a, b)) <ul style="list-style-type: none"> <li>* a,b= randint(0,50), randint(51,100)</li> </ul> </li> <li>- KernelCenterer</li> <li>- QuantileTransformer(n_quantiles=a, output_distribution=c, subsample=b) <ul style="list-style-type: none"> <li>* a, b = randint(100, 1000), randint(1000, 1e5)</li> <li>* c=randchoice(['normal', 'uniform'])</li> </ul> </li> <li>- Normalizer(norm=a) <ul style="list-style-type: none"> <li>* a = randchoice(['l1', 'l2', 'max'])</li> </ul> </li> <li>- Binarizer(threshold=a) <ul style="list-style-type: none"> <li>* a= randuniform(0,100)</li> </ul> </li> </ul> </li> </ul>
<p><b>LEARNERS</b></p> <p>Defect prediction and text mining:</p> <ul style="list-style-type: none"> <li>• DecisionTreeClassifier(criterion=b, splitter=c, min_samples_split=a) <ul style="list-style-type: none"> <li>- a= randuniform(0.0,1.0)</li> <li>- b, c= randchoice(['gini', 'entropy']), randchoice(['best', 'random'])</li> </ul> </li> <li>• RandomForestClassifier(n_estimators=a,criterion=b, min_samples_split=c) <ul style="list-style-type: none"> <li>- a,b = randint(50, 150), randchoice(['gini', 'entropy'])</li> <li>- c = randuniform(0.0, 1.0)</li> </ul> </li> <li>• LogisticRegression(penalty=a, tol=b, C=float(c)) <ul style="list-style-type: none"> <li>- a=randchoice(['l1', 'l2'])</li> <li>- b,c = randuniform(0.0,0.1), randint(1,500)</li> </ul> </li> <li>• MultinomialNB(alpha=a) <ul style="list-style-type: none"> <li>- a= randuniform(0.0,0.1)</li> </ul> </li> <li>• KNeighborsClassifier(n_neighbors=a, weights=b, p=d, metric=c) <ul style="list-style-type: none"> <li>- a, b = randint(2, 25), randchoice(['uniform', 'distance'])</li> <li>- c = randchoice(['minkowski', 'chebyshev'])</li> <li>- if c=='minkowski': d= randint(1,15) else: d=2</li> </ul> </li> </ul>



**Figure 3.9** Two-Layer Ensemble Learning framework replicated from the work by Yang et al. [Yan17]

**Hyperopt:** A prevalent hyper-parameter optimizer proposed by Bergstra et al. in [Ber11]. Agrawal et al. in a recent defect prediction work [Agr19] showed DODGE to outperform hyperopt for software defect prediction. Yet, this thesis includes Hyperopt because its a state of the art optimizer in the machine learning community that may augument early methods and works differently when compared to DODGE. For the experiments the Tree-structured Parzen Estimator (TPE) wrapped in the Hyperopt toolkit publicly available here [Ber13] is used.

Hyperopt is stochastic method that produces random hyper-parameter settings to TPE. The TPE groups the evaluations of various hyper-parameter settings to best and rest. Essentially, TPE reflects over the history of evaluations seen up to date to jump to the next best setting to explore.

The TPE explores the best hyper-parameter setting to from the history of evaluations seen to date by order. TPE selects the best hyper-parameter options is then modelled as a Gaussian with its own mean and standard deviation. The groups are modelled as a Gaussian computed with its own mean and standard deviation.

**Two-layer ensemble learning (TLEL):** Yang et al. in 2017 proposed an ensemble approach that leverages decision tree with bagging similar to a Random Forest model [Yan17] depicted in Figure 3.9. A difference here is that Random Forest uses many decision trees, but in TLEL, the training data is under-sampled randomly to train many yet different Random Forest models. A commit is classified by TLEL as defective when most of the Random Forest models trained with different samples of data being stacked agree (second layer ensemble).

**Manual Up/Down:** In the past, Menzies et al. showed trivial approaches that use no training information like Manual Up/Down outperformed in identifying defects compared to complex methods [Men10]. Recently (2018) by Zhou et al. reported that simple size based models show a promising predictive performance, therefore advised researchers to include Manual Up/Down methods as a baseline

while proposing any new technique [Zho18].

Koru et al. related a module's defect proneness with their size. In one case with two commercial projects they found smaller modules were defective [Kor08b] whereas in another they found larger classes were more defect prone [Kor08a]. This thesis includes both their methods to classify our commits:

- **ManualDown** We classify all test commits as defective if its size ('la' in Table 3.4) is greater than the median size among the test commits [Kor08b]. The philosophy here is more enormous changes should be inspected first and therefore penalized.
- **ManualUp** We classify all test commits as defective if its size ('la' in Table 3.4) is less than or equal to the median size among the test commits [Kor08a]. The philosophy here is more minor changes should be inspected first and therefore penalized.

To reiterate, these two methods do not require any training commits.

### 3.7 Data Pre-processers

Following some advice from the literature, this step is applied for some feature engineering to the Table 3.4 data. First, LA and LD are normalized by dividing by LT and normalized LT and NUC by dividing by NF; then, this step drops ND and REXP since they reported that NF and ND are highly correlated with REXP and EXP [Nag05; Kam12; Kon20]. Lastly, this step applies the logarithmic transform to the remaining process measures (except the boolean variable 'FIX') to alleviate skewness [Shi10].

In other pre-processing steps, this work applied Correlation-based Feature Selection (CFS). The initial experiments with this data set lead to unpromising results (recalls less than 40%, high false alarms). However, those results improved after applying *feature subset selection* to remove spurious attributes. CFS is a widely applied feature subset selection method proposed by Hall [Hal03] and is recommended in building supervised defect prediction models [Kon19]. CFS is a heuristic-based method to find (evaluate) a subset of features incrementally. CFS performs a best-first search to find influential sets of features that are not correlated with each other, however, correlated with the classification. Each subset is computed as follows:  $merits = k rcf / \sqrt{k + k(k-1)r_{ff}}$  where:

- $merits$  is the value of subset  $s$  with  $k$  features;
- $rcf$  is a score that explains the connection of that feature set to the class;
- $r_{ff}$  is the feature to feature mean and connection between the items in  $s$ , where  $rcf$  should be large and  $r_{ff}$ .

Another pre-processor that was applied to some sampling policies was *Synthetic Minority Over-Sampling*, or SMOTE. When the proportion of defective and clean commits (or modules, files, etc.) is not equal, learners can struggle to find the target class. SMOTE, proposed by Chawla et al. [Cha02] is often applied in defect prediction literature to overcome this problem [Agr18a; Tan18]. To achieve balance, SMOTE artificially synthesizes examples (commits), extrapolating using K-nearest neighbors (minimum five commits required) in the data set (training commits in the case) [Cha02]. Note that:

- This work does not apply SMOTE to policies that already guarantee class balancing. For example, the preferred early life-cycle method selects at random 25 defective, and 25 non-defective (clean) commits

from the first 150 commits.

- Also, just to document that this work avoided a potential methodological error [Agr18a], this work record here that this work applied SMOTE to the training data, but never the test data.

### 3.7.1 Oracle Evaluation Criteria

This section elucidates all the evaluation measures used in chapters 5 and 6 to gauge the predictors (oracles) built using various sampling methods endorsed by consulting from widely-used measures [Men08; Wan13; Tan18; Ben19; Zha14; McI17; Tan18; Kon20; Yat19; Yan19a; Zha16b; D2; Kam12] in the defect prediction literature.

Note for the following eight predictive performance measures:

- Initial number of False Alarms range from 0 to # (maximum number of commits inspected);
- MCC range from -1 to 1;
- *D2H*, *IFA*, *Brier*, *PF* of these criteria need to be minimized, i.e., for these criteria *less is better*.
- For four of these *AUC*, *Recall*, *G-Measure* and *MCC* criteria need to be maximized, i.e., for these criteria *more is better*.

Prior work has shown that precision has significant issues for unbalanced data; hence precision is not included in the evaluation [Men08]. Prior reviewers of this work have noted that this might mean we miss some effects regarding that section of the evaluation space. To address that point, we have added an evaluation metric that draws from multiple “corners” of the evaluation space (see the MCC measure of §3.7.1.8). Nevertheless, MCC did not affect any conclusion made in this thesis and therefore is only used in chapter 6 aimed to expand the scope of the results found in chapter 5.

Before listing these criteria, we note that in practice, at least for the data explored here, many of them turned out to be uninformative. For example, IFA turned out to have statistically indistinguishable results across all our treatments. Similarly, all our better methods will achieve similar G-Measures (so that measure will not be so critical to determining what treatment is “best”).

#### 3.7.1.1 Brier

Brier is the absolute predictive accuracy measure. Numerous defect prediction papers [McI17; Tan18; Tan18; Kon20] endorse this measure. Let  $C$  be the total number of the test commits. Let  $y_i$  be 1 (for defective commits) or 0 otherwise. Let  $y_{\hat{y}}$  be the probability of commit being defective (computed from the loss function in scikit-learn library [Ped11]).

Then:

$$Brier = \frac{1}{C} \sum_{i=1}^C (y_i - \hat{y}_i)^2 \quad (3.1)$$

#### 3.7.1.2 Initial number of False Alarms (IFA)

Based on the observation by Parnin and Orso [Par11] that developers lose their trust in such analytics if they encounter many initial false alarms. Thus by simply counting the number of false alarms encountered

after sorting the commits in the order of probability of being defect-prone. IFA is simply the number of false alarms before finding the first actual alarm.

### 3.7.1.3 Recall

Recall is the number of inspected defective commits divided by all the defective commits.

$$Recall = \frac{True\ Positives}{True\ Positives + False\ Negatives} \quad (3.2)$$

### 3.7.1.4 False Positive Rate (PF)

PF is the ratio between the number of clean commits predicted as defective to all the defective commits (irrespective of classification).

$$PF = \frac{False\ Positives}{False\ Positives + True\ Negatives} \quad (3.3)$$

### 3.7.1.5 Area Under the Receiver Operating Characteristic curve (AUC)

It is simply the area under the curve between the false-positive rate and true positive rate.

### 3.7.1.6 Distance to Heaven (D2H)

D2H or “distance to heaven” is computed as an aggregation on two metrics Recall and False Positive Rate (PF). Where “heaven” is a place with  $Recall = 1$  &  $PF = 0$  [Che18a].

$$D2H = \frac{\sqrt{(1 - Recall)^2 + (0 - PF)^2}}{\sqrt{2}} \quad (3.4)$$

### 3.7.1.7 G-measure (GM)

GM is computed as a harmonic mean between the compliment of PF and Recall. It is measured as shown below:

$$G\text{-Measure} = \frac{2 * Recall * (1 - PF)}{Recall + (1 - PF)} \quad (3.5)$$

GM and D2H essentially combine the same two measures, *Recall* and *PF*. Nevertheless, this work still employ those as they have been used endorsed separately in the literature. Note, it is not necessary that achieving good results on GM would also associate with good D2H (or vice-versa).

Due to the nature of the classification process, some criteria will always offer contradictory results:

- A classifier may simply achieve 100% *Recall* just by labeling all the test commits as defective. But as a side-effect, that method will incur a high *PF*.
- Secondly, a classifier may classify all test commits as clean to show 0% *PF*, but that method will incur a very low *Recall*.
- Lastly, Brier and Recall are also antithetical since reducing the loss function implies missing some conclusions lowering *Recall*.

### 3.7.1.8 Mathew's Correlation Coefficient (MCC)

MCC utilizes all the four computations namely True Positives (TP), True Negatives (TN), False Positives (FP) and False Negatives (FN) of the confusion matrix, such that:

$$\text{MCC} = \frac{TP \times TN - FP \times FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}} \quad (3.6)$$

Thus, many researchers endorse the use of MCC, especially in the space of software defect prediction [Yao20; Kon20]. It returns a score between  $-1$  and  $+1$ , where  $+1$  indicates higher predictive performance,  $-1$  contrary predictions and  $0$  indicates most of the predictions poor predictive performance (random).

## 3.8 Summary

This chapter elucidated all the required data and methods for all the experiments in the subsequent Chapters 4, 5 and 6.

## **Part 2 - Explaining the disconnect**

## CHAPTER

# 4

# DOCUMENTING & UNDERSTANDING THE SOURCE OF HUMAN CONFUSIONS

This chapter is based on two materials first published as:

- Shrikanth, N. C., and Tim Menzies. "Assessing practitioner beliefs about software defect prediction." *2020 IEEE/ACM 42nd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 2020. and
- Shrikanth, N. C., et al. "Assessing practitioner beliefs about software engineering." *Empirical Software Engineering* 26.4 (2021): 1-32.

Essentially this chapter will portray the discrepancies between practitioner beliefs and empirical evidence. More importantly it will show support for practitioner beliefs (factors believed to affect quality) tend to decay as the project matures.

## 4.1 Introduction

Typically the conclusion instability problem may be attempted by directly jumping into the pragmatic software analytics perspective. In other words, one can look at ways to improve the data miners, optimizers, pre-processors or even cascade them with ensemble approaches. However, though some of those methods would improve the predictive performance, none of them have mitigated (at least so far in literature) the need to revise the oracles or stop accumulating more data. In order to change decades of data-hungry

SE practice, this work should bring a fundamental change. Therefore this chapter invests more effort to reason the source of all confusions by assessing numerous practitioner beliefs, finds coherence between them but more importantly, answers ‘what disconnects decisions and evidence?’

“ *Though deeply learned, unflecked by fault, ’tis rare  
to see when closely scanned, a man from all unwisdom free.*”

– Valluvar’s sacred couplet (translated, 1886, G.U. Pope [Pop99; Thi])

## 4.2 Investigating Software Engineering Theories

In this section, for each belief listed in Table 2.1, the rationale, experiment setup, and belief strengths will be discussed.

### 4.2.1 Why explore only these five beliefs ?

This thesis only explores the five SE beliefs listed in Table 2.1, because:

- No single article could explore all the beliefs recorded by Endres and Rombach.
- The data used in this study provided by Nichols et al. [Wil19] could only comment on a subset of the Endres and Rombach beliefs. As to beliefs such as “*Prototyping (significantly) reduces requirement and design errors, especially for user interfaces*” (Boehm’s second Law) or “*Screen pointing-time is a function of distance and width*” (Fitts–Shneiderman law) that would require a different data source to assess.
- Of the remaining beliefs, five were most widely cited. For example, one of the original SMALLTALK papers [Gol83] (cited 7,430 times) motivates its work using the “Dahl-Goldberg” hypothesis listed in Table 2.1. Also the paper propose the “Apprentice’s Law” [Nor93] has been cited 4,390 times. The remaining three beliefs are all referenced in the *The Mythical Man-Month* [BJ95] and the famous 1987 article *No silver bullet* [Bro87]. These two words are cited 8,649 and 5,085 times, respectively<sup>1</sup>.

### 4.2.2 Belief 1: Corbató’s law

This section discusses an effect reported in a 1969 paper by Corbató [Cor69] that

*Productivity and defects depend on the length of a program’s text, independent of the language level used.*

That is to say, (a) longer programs tend to get more defects; (b) and this effect are not mitigated by newer generation languages. Note that, if true, Corbató’s rule warns us that, by merely switching to a newer language:

- Defects cannot be reduced
- And developers cannot be made more productive

To check this rule, the experiment is constructed as follows:

---

<sup>1</sup>All the citation counts in this bullet item were collected from Google Scholar, December 2019.

**Table 4.1** This table shows *normalized* distributions of “program size”, “production rate” and “defects” of level 2 task(s) ranked using Scott-Knott test (elucidated in §3.5.1). Group 1 shows task 10 completed independently using C and C# share similar LOC (the rows are shown in gray), whereas in the subsequent groups 2 and 3 the distributions of 10, C and 10, C# to be significantly different.

Program Size (LOC)					
Group 1	Rank	Task, Language	Median	IQR	
Level 2 tasks	1	10, C#	18	11	●—
	1	10, C	17	10	●—
	2	9, C#	13	9	●
	3	9, C	11	7	●
	4	7, C#	7	6	●
	5	8, C#	6	5	●
	5	7, C	6	4	●
	5	8, C	6	5	●
Production rate (LOC/hour)					
Group 2	Rank	Task, Language	Median	IQR	
Task 10 completed using C and C#	1	10, C#	13	10	●—
	2	10, C	8	7	●
Defects					
Group 3	Rank	Task, Language	Median	IQR	
Task 10 completed using C and C#	1	10, C#	5	7	●
	2	10, C	9	9	●—

- (a) Group similar tasks written in both non-oo and oo programming languages.
- (b) In that group, select the same tasks that share similar LOC <sup>2</sup>.
- (c) Investigate production rate and defect distribution in those two groups.

Note, for this belief to be widely accepted, this rule should hold in “every” oo-vs-non-oo programming language pair (such as C, C# or C, C++, or C, Java or C, VB) that satisfy the above two experiment constraints (a and b). On the other hand, this belief cannot be endorsed if it does not show support even in any one of the oo-vs-non-oo language pairs.

To find the two groups that satisfy the experiment constraints, first, this assessment will consider the most complex level 2 tasks. If the assessment does not find distributions that meet criterion (b), then the assessment will repeat with lesser complex tasks; those are at levels 0 and 1. Lastly, in that group, assessment chose C and C# because Hejlsberg and Li et al. [Hej06; Li17b] assert that C and C# are two programming languages at different “levels,” but more importantly, they satisfy the experiment constraints (a and b).

High-level features help developers to write less code. For example, automatic memory management (garbage collection) is one of the numerous high-level features available in C#. Automatic memory management can help developers to focus more on the assigned task’s functional requirements rather than writing additional code to manage memory.

<sup>2</sup>Following the belief statement, this section uses LOC (length of the program text) and not function points that share identical distribution. Then compute production-rate (productivity) using LOC just as defined in this book [End03] (source of all the beliefs in this study) but also in this prominent studies [Ngu11; Dev96].

*Prediction:* If Corbató was wrong, then one should see either

- Production rates differ by programming language and/or
- Defects differ by programming language.

#### 4.2.2.1 Result

Table 4.1 shows the results in three groups program size, production rate, and defects. From this table, several observations are made.

- Program size distributions in group 1 reveal that tasks 8 and 10 completed using C and C# share similar (same rank) LOC distribution.
- Subsequently, in groups 2 and 3 (“production rate” and “defects”), this part only focuses on tasks 10, C, and 10, C# results (the rows are shown in gray). This step does that because (a) this step can remove the conflating factor of different LOCs (tasks 7 & 9), and (b) task 10 has higher LOC ranges than task 8, making it naturally a better choice for to carry further analysis.
- The focus of groups 2 and 3 (“production rate” and “defects”) on task 10 (chosen in the previous step) reveals developers who completed the task using C# were more productive and induced fewer defects than those completed using C.
- Thus, as per Corbató’s Law, if only LOC matters and language level does not, then task 10 irrespective of whichever language (C or C#) used should also portray similar production rate and defects distribution. However, in Table 4.1, this section observes a significant difference in the production rate and defects of these groups. Thus one cannot ignore the level of a language as it impacts both developer production rate and defects.
- Lastly, as mentioned earlier in §4.2.2, given that this belief weakened with a C and C# group, there is no need to assess this belief on remaining oo-vs-non-oo language pairs.

Accordingly, this section finds:

Belief 1: *The results contradicted Corbató’s law as with similar program size (LOC), tasks completed using C# is significantly “better” (higher production rate and fewer defects) than using C.*

#### 4.2.3 Belief 2: Dahl-Goldberg hypothesis

This section discusses an effect reported in a 1983 paper by Dahl and Goldberg [Dah01; Gol83] that.

*Programs written using non-OO languages naturally induce more defects.*

If true, then programs written in OO languages like Java should get fewer defects than written in C (non-OO).

To check this effect, tasks completed by developers in 5 programming languages are studied. Among those five languages VB, C#, and Java support OO, whereas C does not support OO. C++, often termed

as an extension to C, does support OO; however, programmers may still write C like coding in C++. Hence, this part does not premise the conclusion considering only C++ in the experiment.

The rationale behind this belief, as discussed by Endres & Rombach is that OO basically restricts the developer's freedom to prevent them from introducing unwanted defects. For example, information hiding (encapsulation), a concept in OO, is performed by developers while writing code to pacify software complexity and improve robustness. Hands-on, developers make use of access-modifiers such as *private*, *protected* (in Java) to encapsulate certain complex parts of code. Further, modern OO languages such as C# and Java do not easily expose low-level control or memory management for developers to manipulate them, but those features are readily available in C.

To check if OO effect designs and the prevalence of defects, this section considers two types of defects from all the 10 tasks to assess this belief, they are:

- “defects injected in design” (design defects) and
- “defects injected in code” (coding defects).

Note this belief is not about examining the OO design paradigm, rather certain OO language features. As discussed by Endres and Rombach, OO languages offer certain features (such as automatic memory management, in-built libraries, etc.) that may prevent developers from injecting unwanted defects. In other words, one may still write a non-OO code using a OO language but take advantage of in-built features that OO languages offer.

*Prediction:* If Dahl & Goldberg were wrong, then programming similar tasks using OO languages such as C#, Java, and VB programs should have more or about the same range of defects compared to C.

#### 4.2.3.1 Result:

Table 4.2 presents the “defects (Code + Design)” in two groups (programming languages and task 10). From this table, several observations are made.

- The defect distributions in group 1 of developers using C#, and VB (the rows are shown in gray) have fewer defects compared to those completed using Java, C, and C++.
- Notably, tasks completed using Java that support OO show more defects similar to those written in C.
- A focused analysis of defects in group 2 shows task 10 completed in C#, and VB also shares the least range of defects.
- Defects are lower only in two of four languages that have some support for OO (C# and VB), whereas Java and C++ (that support OO) portray significantly more defects similar to those written in C. Thus, this work cannot endorse the Dahl-Goldberg hypothesis.

Accordingly, this section finds:

Belief 2: *Programs written in OO are not necessarily less defect prone*

#### 4.2.4 Belief 3: Mills-Jones hypothesis

This section discusses an effect from two papers by Mill & Jones [Mil83; Cob90] in 1983 and 1990:

**Table 4.2** This table shows *normalized* distribution of “defects (Coding + Design)” in two different groups (programming languages and task 10). Using the Scott-Knott test (elucidated in §3.5.1) tasks completed using C# and VB in group 1 share least range of defects (the rows are shown in gray) compared to C, Java, and C++. The subsequent group 2 shows that pattern of similar low defects among C# and VB for the most advanced task 10.

Defects (Coding + Design)					
Group 1	Rank	Language	Median	IQR	
Programming Language	1	VB	5	5	•
	1	C#	5	5	•
	2	C	8	7	•
	2	Java	8	8	•
	3	C++	11	10	•
Group 2	Rank	Task, Language	Median	IQR	
Task 10	1	10, C#	5	5	•
	1	10, VB	6	6	•
	2	10, Java	8	11	•
	2	10, C	8	8	•
	3	10, C++	11	13	•

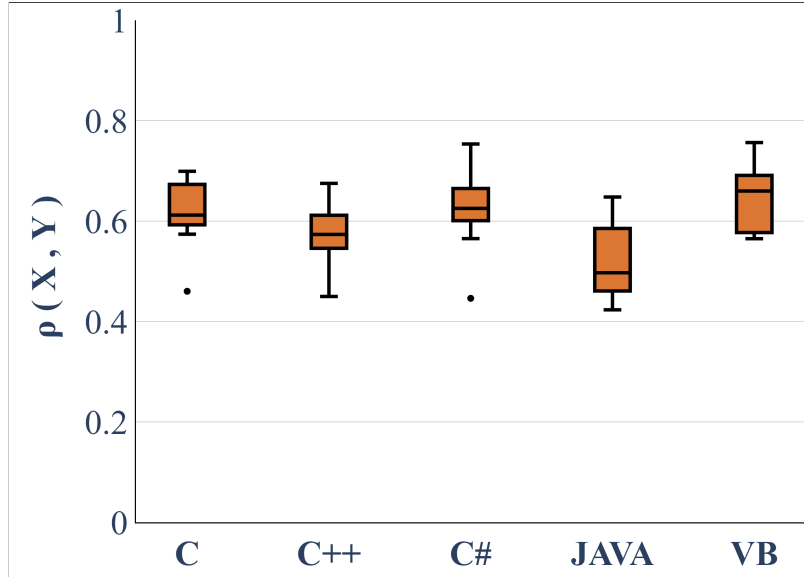
*Quality entails productivity.*

That is to say, a lack of early emphasis on quality in the project life-cycle will lead to a lot of rework (unproductive) and defective software. Mills showed that highly reliable software could be produced through cleanroom software engineering, which employs statistical-based independent testing [Mil83; Mil93]. Disciplined processes such as cleanroom software engineering focus on quality right from the early stages of the project. Having such a focus minimizes unnecessary effort in the later stages of the project, unnecessary efforts like fixing defects in the final testing phase, which were undetected early in the project life cycle (such as coding or unit test).

To study this effect, this step checks for a linear trend between “code and design” (defects injected during coding and design phases) and “test defects” (defects escaped to testing phase) using the correlation test elucidated in 4. This work considers all ten tasks (labeled 1 to 10) in Table 3.2 to gain more data points for the independent and dependant variables. Lastly, this step exports the significant correlation scores visually into a box-plot and discusses the strength of the observed trend based on the median. To achieve that, the following steps are performed:

- Capture the number of “code and design defects,” and the number of “test defects” for each task completed using a specific programming language.
- Then, correlate between the captured list of “code and design defects” and “test defects” across all the 10 tasks and export the correlation coefficient ( $\rho$ ) values.
- The above step results in 50  $\rho$  scores (10 tasks x 5 programming languages). Then plot the exported scores (distributions) in Figure 4.1.

*Prediction:* If Mills & Jones were wrong, then it could mean that managers need not invest in quality assurance activities early in their project life cycle.



**Figure 4.1** The box-plots in this chart show the distribution of correlation scores grouped by different programming language. The correlation is computed between “code and design defects” (X) and “test defects” (Y) (computed as described in §3.5.2) for each task (listed in Table 3.2) completed using a specific programming language.

#### 4.2.4.1 Result:

Figure 4.1 presents a box-plot of all the exported correlation ( $\rho$ ) scores grouped by programming language. All tasks (labeled 1 to 10) from Table 3.2 are used. From this figure, the following observations are made.

- This part finds a median of +0.5 ( $\rho$ ) between “code and design defects” and “test defects” in Java, and in the remaining four programming languages this part analyzed, the correlation is above +0.6 ( $\rho$ ).
- An overall median correlation of +0.6 ( $\rho$ ) considering all the five programming languages confirm rework will increase (more test defects) if there is a lack of emphasis on quality in the early stages.

Accordingly, this section finds:

Belief 3: *Emphasis on early quality does minimize rework.*

That said, the strength of this support is not very strong  $\rho$  (+0.7).

#### 4.2.5 Belief 4: Sackman’s second Law

This section discusses an effect reported in a 1966 paper by Sackman et al. [Sac66]:

*Individual developer performance varies considerably.*

That is to say; developer ‘X’ is considerably “better” in completing a task than developer ‘Y.’ By “better”, this section implies developer ‘X’ writes more lines of code in less time than developer ‘Y,’ and

developer X's deliverable gets fewer defects than developer Y's deliverable.

Also, note that, if true, Sackman's second Law warns us that:

- Only some developers are productive and write quality code.

A variation between developers was rather a surprising finding by Sackman in 1966 as the objective of the original study was to compare productivity between online programming and offline programming [Sac66]. Endres & Rombach also note that this effect is not been extensively studied in the past few decades. They also offer some doubts concerning the small sample size and the statistical approach used in the original (Sackman's) study. Sackman's study considered only 12 developers, and their conclusion is based on extremes and not on the entire distribution. Note that this section compares large distributions of production rate and defect scores captured from over 1000 developers.

Naturally, managers would prefer few high performers over many low performers, but recently (2019), Nichols using the same data, showed that a developer X who is productive in one task is not necessarily productive in another [Nic19]. Thus this section addresses the quality aspect of this belief. To check whether large production rate variance among developers associate with more defects (low quality), this section construct the experiment as follows:

- Capture the number of "code defects," and the "production rate" for each task completed using a specific programming language.
- Then, correlate between the captured list of "code defects" and "production rate" across all the ten tasks and export the correlation coefficient ( $\rho$ ) values. To expose any effect of the programming language in this analysis, this workgroups the distributions by programming language, similar to the analysis earlier in §4.2.4 and later in §4.2.6.
- The above step results in 50  $\rho$  scores (10 tasks x 5 programming languages); finally plot the exported scores (distributions) in Figure 4.2.

*Prediction:* If Sackman was wrong, then practitioners may ease their large appeal towards some high-performing developers.

#### 4.2.5.1 Result:

Figure 4.2 presents a box-plot of all the exported correlation ( $\rho$ ) scores grouped by programming language. That figure reports a '0' ( $\rho$ ) correlation score considering all the tasks (labeled 1 to 10 from Table 3.2) and all the five programming languages independently. This result confirms the absence of a linear association between 'production rate' and 'code defects.'

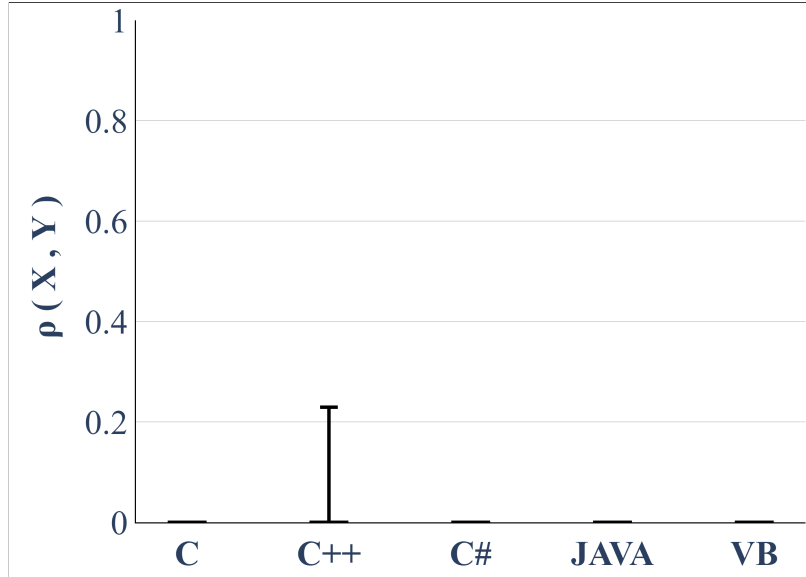
Accordingly, this section finds:

Belief 4: *Software Quality is not impacted by the variance in developer production rate.*

#### 4.2.6 Belief 5: Apprentice's Law

This section discusses an effect reported in a 1993 paper by Norman et al. [Nor93]; specifically:

*It takes 5000 hours to turn a novice into an expert.*



**Figure 4.2** The box-plots in this chart show the distribution of correlation scores grouped by different programming language. The correlation is computed between “code defects” (X) and “production rate” (Y) (computed as described in §3.5.2) for each task (listed in Table 3.2) completed using a specific programming language.

To assess the effect of prolonged programming experience this part analyzes “production rate” and “defects” among the *expert* and *novice* groups. An expert is someone who is both knowledgeable and skilled in their field of work. An expert in this study is a developer who can complete the task on time (productive) with no defects (quality).

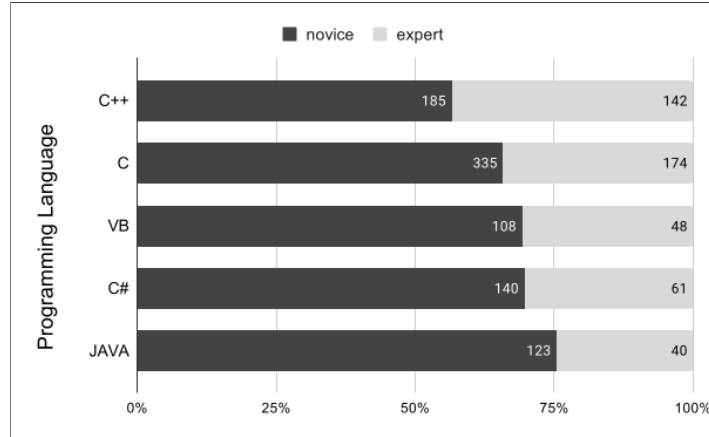
Adopting from [End03] this section maps the 5000-hour threshold as follows:

**expert** :  $\geq 3$  years of experience (or  $\geq 5000$  hours of programming experience)

**novice** :  $< 3$  years of experience (or  $< 5000$  hours of programming experience)

That is to say, (a) *expert* developers induce *less* defects than novices; (b) *expert* developers are more productive in completing tasks than *novice* developers. Note that, if true, the Apprentice’s Law warns us that, one should mistrust *novices* due to their lower quality code. To check this, this section will analyze the distributions of “production rate” and “defects” among *experts* and *novices*.

All the specific tasks labeled 1 to 10 in Table 3.2 are new to the developers. Nonetheless, while on real-world projects, developers will not get precisely similar task assignments, they may use similar skills including as follows: applying the same language features; use iterations, conditionals, and subroutine interfaces; use of data types and manipulation of data structures; developing test cases and debugging; and so forth. Test whether a developer, ‘X,’ with four years of prior Java development experience, is better in completing the task, ‘T,’ than another Java developer, ‘Y,’ with less than a year of experience. Both developers ‘X’ and ‘Y’ are new to task ‘T’ but have differing levels of experience using the underlying skills.



**Figure 4.3** Proportion of *expert* and *novice* developers who completed all the 10 tasks listed in Table 3.2 using a specific programming language.

#### 4.2.6.1 Result:

The ratio of *expert* to *novice* developers in the data is shown in Figure 4.3.

Table 4.3 presents the results on production rate and defects in 4 groups. From this Table, the following observations are made:

- Despite numerous studies in the past that endorsed this effect, groups 1 and 2 reveal no effect on developers with years of prolonged programming experience. In other words, *Novice* developers were as productive and induced the same amount of defects as *expert* developers.
- The earlier results confirm some programming languages to have an effect on “production rate” and “quality” (defects). Thus to check whether “years of experience” also influence developers using different programming languages, this step segregates the *expert* and *novice* population by programming languages to find the following:
  - “Years of experience” has less influence on “defects” among developers using different programming languages. Like in the earlier results seen in Table 4.2, overall C# and VB *novices* portray better quality (fewer defects) than developers of three other languages. This also implies that strangely C# and VB *novices* portray better quality (fewer defects) than *experts*.

*Apprentice Law* is only supported on the lines of production rate, only for Java and C++ developers (2 of 5 groups of developers), and has no influence in mitigating defects. The evidence supports the counterclaim that practical industrial experience has little to do with expertise. There is no noticeable performance difference among *experts* and *novices* (Groups 1 and 2). This work believes the conditions for deliberate practice [Eri04] are not achieved in normal work; thus, *years of experience* has limited benefit.

Hence, overall, this section says that.

**Table 4.3** This table shows “production rate” and “defects” distributions of *expert* and *novice* developers in four groups. Using Scott-Knott test (elucidated in §3.5.1) groups 1 and 2 indicate no difference in either production rate nor defects among *expert* and *novice* developers. On the other hand groups 3 and 4 portray significantly different distributions (as seen from different Scott-Knott ranks) among developers using different programming languages.

Production rate (LOC/hour)					
Group 1	Rank	Experience	Median	IQR	
All tasks	1	expert	6	4	•
	1	novice	5	5	•
Defects					
Group 2	Rank	Experience	Median	IQR	
All tasks	1	expert	6	6	•
	1	novice	5	6	•
Production rate (LOC/hour)					
Group 3	Rank	Language, Experience	Median	IQR	
Programming Language expertise	1	C#, novice	8	6	•
	1	C#, expert	7	6	•
	2	C++, expert	6	5	•
	2	Java, expert	6	5	•
	2	VB, novice	6	5	•
	3	Java, novice	5	4	•
	3	C++, novice	5	5	•
	3	C, expert	5	3	•
	3	VB, expert	5	4	•
	3	C, novice	5	4	•
Defects					
Group 4	Rank	Language, Experience	Median	IQR	
Programming Language expertise	1	C#, novice	3	3	•
	1	VB, novice	3	3	•
	2	C#, expert	5	6	•
	2	C, novice	5	5	•
	2	VB, expert	5	5	•
	3	Java, expert	6	6	•
	3	C, expert	6	6	•
	3	C++, novice	6	7	•
	3	Java, novice	6	7	•
	3	C++, expert	7	8	•

Belief 5: *Experienced developers did not necessarily write better (fewer defects) programs on time.*

#### 4.2.7 Threats to validity

This work draw the following subsections from Wohlin et al. [Woh12] (first conceived by Cook and Campbell [Cam79])

#### **4.2.7.1 Conclusion Validity**

Construct validity checks whether the findings could be incorrect because of the operationalizations of the concepts, incorrect modeling, or misleading data. While Johnson and Disney found that 5% of the data was incorrect [Joh99], their study used manual recording, transcription, and computation. Only the data prior to 1996 in this study were manually recorded. The vast majority used an Excel spreadsheet for initial data entry and all derived computations. Authorized PSP instructors also verified the data. A concern may arise due to the nature of the data set composed of only ten tasks (assignments). But, using similar data set, certain useful observations have been made in the past, and those are reported in these SE articles [Men17; Nic19]. Another concern may arise from the sample size of the data set and the fact that OS can offer much more data at scale. But to directly check whether the conclusions apply to practice, a data corpus of industry nature is needed. More than 90% of the tasks that happened in the industry across various geographies and this work only report statistically significant results.

Lastly, some studies operationalize quality as defect density. But defect density could be sensitive to the verbosity of either the programming language or the programmer. That is, the same assignments could contain the same number of defects yet differ because one program has more lines of code than the other. Defect density is often used because different programs cannot be directly compared. Because this data replicates the same task across multiple developers, quality is best measured by the total number of defects in code, design, and test accordingly.

#### **4.2.7.2 Internal Validity**

Threats to internal validity concern the causal relationship due to the artifacts of the study design and execution. It may also include factors that have not been controlled or measured or study execution introducing some unintended factors.

PSP course's emphasis on measuring production, estimation, and quality could have influenced the developer's performance. The mitigation was that the developers were not in any sort of competition with each other; instead, they were instructed to take consistent data to measure their performance trend. Also, there are no overlaps, i.e., the same developer completed the ten tasks only once using a programming language. Other factors that were uncontrolled include experience with a specific programming language or aspects of the development environment in which the class was taken.

Analyst bias in conducting the research is always a potential threat. This is minimized because the data was collected for an entirely different purpose over an extended time by several independent individuals. This work further minimizes this threat by relying on quantitative data and fully revealing that data. While the tasks are unique, the underlying skills are somewhat consistent. The problems require reading input, writing output, performing basic data manipulation with lists, sorts, modular decomposition into subroutines, employing iterations, and conditional logic. There is some difference in that the two exercises place more emphasis on text manipulation, and a couple of others require nested floating-point iteration structures.

Lastly, this work does not consider PSP as a treatment, but used that data to observe evidence in

the prevalent beliefs it evaluated. This work does not question the authenticity of these beliefs in the past, given the notable increase in the number of programming languages, supporting tools, memory, computation power, and online workforce.

#### **4.2.7.3 External validity**

The domain of the programs is not representative of all software development. The tasks were principally numeric and statistical. Nonetheless, they included the standard elements of modular design, input, output, and control structures common to many professional programs. The numeric specifications were provided; therefore, no special domain knowledge was required. Production rates and defect rates will likely differ across specific domains. The programs were not intended to be of production quality; therefore, the test cases were not extensive. The goal of this work was to show that it is both possible and important to revisit old beliefs (and to advise practitioners to regularly monitor and discard effects that are not backed by evidence). The above results show that this is indeed possible.

As to specifics of the conclusions, the set of programming languages this work explores may not cover some of the recent trending web development languages like PHP, Ruby, etc. Thus this work does not claim the results to generalize to all projects. On the other hand, the languages this work analyzed are in existence for decades in long-living proprietary software systems (in banks, healthcare, etc.) and will remain prominent in the future (if only for maintenance reasons).

### **4.3 Investigating Software Quality Assurance Theories**

This section evaluates all the beliefs listed in Table 2.2 by Wan et al. [Wan18a]. Wan et al.'s study collected 395 responses from practitioners to document developer beliefs about willingness to adopt technologies, challenges, defect prediction metrics, etc. This thesis looks for evidence for the Table 2.2 beliefs in 300,000+ changes seen in dozens of open-source projects.

While assessing, this section will describe the construction of the correlation experiments using the underlying rationale and relevant literature attached to each of the beliefs in Table 2.2. Then, the role of relevant attributes is scraped from the sample projects to be used as an independent variable to compute associations as required.

#### **4.3.1 Why explore only these ten beliefs ?**

This section only explores 10 of the 15 metric-related beliefs documented by Wan et al. [Wan18a] in Table 2.2. Because some of the modeling decisions about how to map data into Table 2.2 required extensive, possibly even arcane, explanations. Objectively, this section was able to answer the central question using the ten beliefs, and it would remain the same with or without exploring additional beliefs.

### 4.3.2 Belief 1: Complex Code Changes

**Description:** “A file with a complex code change process tends to be buggy”. In 2009, Hassan [Has09] adopted Shannon Entropy from information theory to devise few code change models that weigh scattered modifications (complexity) of a file using its change history. The intuition behind this belief is that the change scattering of a file over some periods makes it cumbersome for developers to maintain, thus making it defect prone. This belief uses Hassan’s compute History Complexity Metric ( $HCM^{1d}$ ) with the decay factor ( $d$ ) to assess this belief. The decay factor is used to undermine earlier modifications.

**Procedure:** Divide the pre-release period  $r$  into bi-weekly (14 days) periods to compute entropy  $HCM$  for a file  $F$  and this will be  $F_{Belief1}$ . In cases where releases have less than 14 days,  $r$  will have just two equal size periods. Then export the correlation between  $F_{Belief1}$  &  $F_D$ .

### 4.3.3 Belief 2 & Belief 10: Ownership

**Description:** Belief 2: “Files changed by more developers are more buggy”. In 2010, Matsumoto et al. [Mat10] showed that human factors could be used to forecast defects. Specifically, they saw that files touched by more developers make the file prone to defects.

Belief 10: “Files with fewer lines contributed by their owners (who contribute most changes) are bug-prone”. Bird et al., in their work [Bir11] explore various ownership-related metrics. To model, this employs the idea of minor & major contributor, where many minor contributors modifying a file makes it defect prone. They define a minor contributor as someone who has made less than 5% changes and a major contributor as someone who has made at least 5% or more. Thus if a file in a release has changed by a large proportion of minor contributors, it is prone to more defects.

**Procedure:**  $F_{Belief2}$  here is the count of distinct *commit authors* who made some changes to the file in the pre-release.  $F_{Belief10}$  is the % of minor contributors (who contributed less than 5% code churn) for a file  $F$  in the pre-release. Then export the correlation between  $F_{Belief2}$  &  $F_D$  and  $F_{Belief10}$  &  $F_D$ .

### 4.3.4 Belief 3 & Belief 9 : Code Churn

**Description:** Belief 3: “A file with more added lines is more bug-prone” and Belief 9: “A file with more removed lines is more bug-prone”. In 2005 Nagappan & Ball in [Nag05] showed that relative code churns of files using number of added or deleted lines between subsequent versions are good indicators to forecast defects.

**Procedure:** Measure  $F_{Belief3}$  or  $F_{Belief9}$  for a file  $F$  by aggregating on the *number of lines added* (Belief 3) or *number of lines deleted* (Belief 9) only during the pre-release period  $r$ . Then export the correlation between  $F_{Belief3}$  &  $F_D$  and  $F_{Belief9}$  &  $F_D$ .

### 4.3.5 Belief 4 & Belief 6: Temporal

**Description:** Two of these temporal heuristics are introduced and explored by Hassan et al. [Has05] in their popular work, “Top 10 list for dynamic defect prediction”. Belief 4: “Recently changed files tend to be buggy” and Belief 6: “Recently bug-fixed files tend to be buggy”. The rationale here is that files

modified closer to release periods may not be tested effectively and as a result, recently changed files would tend to introduce more bugs (also see [Gra00]) in the near future (Belief 4). Another intuition is that faults tend to arise at the same spot is a good indicator to measure defect proneness (Belief 6).

**Procedure:** Attributes *commit\_time* (longer value indicates more recent) and *BFC* (1 indicates bug fixing, 0 otherwise) are used to model these two beliefs. But a file can be modified (committed) multiple times in the pre-release period. Hence, assign the maximum *commit\_time* for  $F_{Belief4}$ , similarly assign the maximum *commit\_time* for  $F_{Belief6}$ ; provided  $BFC = 1$  in the pre-release period. Further in  $F_{Belief6}$  if a file is never modified for the purpose of fixing a defect in the pre-release period, its ignored for analysis. then export the correlation between  $F_{Belief4}$  &  $F_D$  and  $F_{Belief6}$  &  $F_D$ .

#### 4.3.6 Belief 5: Commit Churns

**Description:** “A commit that involves more added and removed lines is more bug-prone.”. A single commit affects one or more files with some line additions and/or deletions. Hindle et al. [Hin08] and Hattori et al. [Hat08] studied the nature and distribution of large commits in terms of the number of files it changed and highlighted that although large commits are rare, they are unsafe to ignore. A commit can push the same (or more) amount of line changes with fewer files. Thus rather than the number of files a commit changes, this section weighs this belief based on the amount of added and removed lines a commit pushes into the system.

**Procedure:** Unlike other beliefs, that can be measured at file-level, a commit is a collection of files. It’s immutable, meaning it cannot be directly traced in the post-release period. Hence, this method assesses the impact of a commit using the files it affected. Thus, the independent variable is a commit in the pre-release period  $F_{Belief5}$  (represents a commit), which is an aggregate of code churn (insertions + deletions) for all the files part of the commit. Similarly, commit defect proneness is the aggregation of file defect proneness  $F_D$  in the post-release period. In other words, this method collectively correlates a file  $F$  that is part of a large commit in the pre-release period with the number of bugs introduced by  $F$  in the post-release period. Lastly, export the correlation between  $F_{Belief5}$  &  $F_D$ .

#### 4.3.7 Belief 7 & Belief 8: Something More

**Description:** Graves et al. in [Gra00] found an effect among the two process-related metric Belief 7 & Belief 8 ie., “A file with more fixed bugs tends to be more bug-prone” and “A file with more commits is more bug-prone” through analyzing code from a telephone switching system.

**Procedure:**  $F_{Belief7}$  is the count of bug fixes on file  $F$  and  $F_{Belief8}$  is the count of modifications (commits) made on file  $F$ , computed during their corresponding pre-release periods. Then export the correlation between  $F_{Belief7}$  &  $F_D$  and  $F_{Belief8}$  &  $F_D$ .

## 4.3.8 Threats to validity

### 4.3.8.1 Threats to internal validity

Like this prominent [Ray14] and a recent [Wal18] large-scale analysis, this work relies on labeling commit messages for bug fixes as the independent variable. Due to limitations in the heuristics used to generate those labels [Š05; Fan19], such labels might be misleading. To partially mitigate this issue of false negatives, this work expanded the set of keywords for better coverage. To validate if false positives impacted the results, this work cross-checked whether projects with higher *Bug Fix %* were likely to show support for more beliefs.

### 4.3.8.2 Threats to external validity

The findings could be biased by the projects and data found in the sample. This means that the conclusions could differ from other publications. For example, many of those publications discussed C and C++ projects, while the sample contained many Ruby projects. Having said this, the sample sufficiently covers many popular programming languages like Python, Java, etc. All the projects are OS, but Agrawal et al. in [Agr18b] showed that many open-source lessons hold in-house. Releases were identified using *git tags* which mark a boundary for readiness in the commit. Some of these changes were too small (just a few hours) even for rapid releases [Män15]. Hence, this work only considers releases that had at least three distinct files changed.

### 4.3.8.3 Threats to statistical conclusion validity

The conclusions are based on correlation which means the strength (or weakness) of this analysis is the same as the strength or weakness of correlation. To increase the validity of those conclusions, this work only reported significance at the 0.01 level. Also, correlations with less than 4 observations were ignored (even at a significant level). As some releases have less than 3 files changed and technically, it is possible to get a high correlation with a zero *p\_value* with just two observations, but this work considered this unwise (so such conclusions were discarded).

## 4.4 RQ1: Why do beliefs diverge among practitioners?

### 4.4.1 Diverged Software Engineering Beliefs

Given that the five beliefs are decades-old prevalent beliefs, one naturally expects strong support, but surprisingly, the analysis showed none of these beliefs is *strongly* supported *presently*. It is important to note that such beliefs naturally hold in practice [End03; Pas11; Dev16], and this is not to say that these beliefs *were* not true.

A reason below that a probable source of divergence of beliefs among practitioners [Dev16] could arise from misinterpreting effects by observing partial evidence. For example, recall the effect reported in belief two that,

*Programs written using non-OO languages naturally induce more defects.*

Although the results of belief two from §4.2.3 show that programs written in OO languages C# and VB showed better quality (fewer defects), this work did not endorse this effect because programs written in the two other OO programming languages C++ and Java shared defects similar or higher to those written in C (non-OO). While this analysis *can not* say why defects were lower in C# and VB, but this analysis *can* say that it is not due to the OO paradigm. To that end, it is reasonable to imagine that practitioners with a narrow scope who work only with C# and C-based projects will hold on to this belief. Note that similar examples of misinterpretations backed by partial evidence can be weaved from the other beliefs that this work did not endorse. Thus this work conjectures that practitioners could believe some effect to hold in their work due to the lack of a broader perspective.

Lastly, looking at Figure 4.5 amidst overall negative results, support for only one of three beliefs titled “*Quality entails productivity*” is found. Notably, the strength of the belief is unaffected among 4 of 5 programming languages.

Accordingly, this section finds:

*Apart from belief 3 titled “Quality entails productivity”, none of the other beliefs are supported.*

#### 4.4.2 Diverged Software Quality Assurance Beliefs

**Motivation:** Similar to the work by Devanbu et al. in [Dev16] this work compare practitioners’ agreement % with empirical evidence. The objective here is to find beliefs that have strong support.

**Approach:** Compute  $P_{BX}$  for each belief  $BX$  for all the 37 projects. This results in 10 independent  $P_{BX}$  populations one for each belief. Next, rank these populations by their median and effect difference using the Scott-Knott-test of §3.5.1. That results in Table 4.4 to be compared with practitioners’ agreement % in Table 2.2.


**Table 4.4** Scott-Knott test applied to all the 10 beliefs, that contains support  $\rho$  scores of 3,198 releases in all the 37 projects. Each row represents a population of  $P_{BX}$  scores across all 37 projects. (Higher rank indicates stronger support). IQR - Interquartile Range.

Rank	Belief	Median	IQR	
1	Belief 9 (35%)	43	27	—●—
1	Belief 4 (58%)	44	28	—●—
2	Belief 3 (61%)	45	21	—●—
2	Belief 1 (76%)	46	21	—●—
2	Belief 6 (49%)	49	47	—●—
2	Belief 7 (48%)	49	22	—●—
2	Belief 2 (64%)	50	21	—●—
2	Belief 8 (46%)	51	22	—●—
3	Belief 10 (30%)	63	23	—●—
4	Belief 5 (57%)	71	26	—●—

**Findings:** Using the results in Table 4.4 this part reports *popular* belief Belief 10 and *unpopular* belief Belief 5 have significantly *strong* support. Surprisingly, these *unpopular* beliefs Belief 6, Belief 7, Belief 8 and Belief 10 have better support than some *popular* beliefs Belief 1, Belief 3 & Belief 4. For example:

- The ownership-based belief Belief 10 with only 30% practitioner agreement shows *strong support*.
- The temporal belief Belief 4 though with 58% practitioner agreement, has relatively weaker support 0.4. Belief 1 with the largest practitioner agreement 76% shows weak support.

Overall, only a few beliefs, Belief 2, Belief 5 & Belief 9, are in agreement (%) with practitioners' beliefs. Of those, Belief 5:*Large Commits*, with 57% practitioner agreement, has the highest effect (0.7) among all the beliefs.

 **Result:**  
Only beliefs labeled Belief 10 & Belief 5 in Table 2.2, which is believed by (30% & 57%) of practitioners has strong support (i.e., large correlations), whereas the more popular belief, Belief 1, which is believed by 76% of practitioners, has relatively weaker support.

### 4.4.3 Importance of Re-checking

Figure 4.4 and Figure 4.5 confirm that what *was* true before, may not necessarily true *now*. Note that the beliefs shown in those figures are decades-old SE theories that are deeply rooted in SE research and practice. But after a careful investigation, this section found that only one of the five beliefs is supported.

## 4.5 RQ2: What disconnects Beliefs and Evidence?

### 4.5.1 Sporadic Evidence

**Motivation:** The above results show that, overall, many commonly held beliefs do not always hold across all the data. But this is not to say that *sometimes* it may be true that *some* of the beliefs of Table 2.2 are not true.

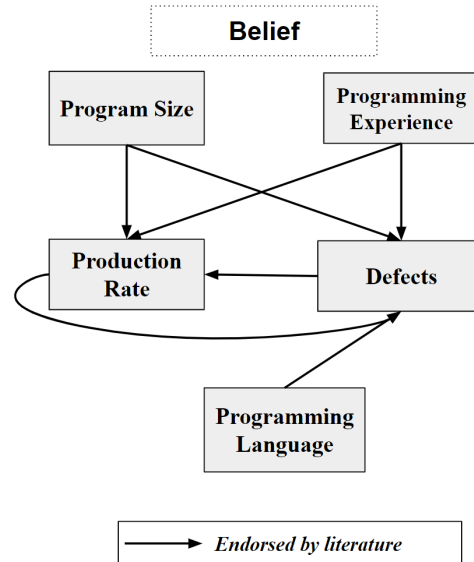
Devanbu et al. [Dev16] found that practitioners had different opinions working for different projects within the same organization. To explore the source of that disconnect, this section will investigate irregularities of evidence among projects, releases, and over time to ask:

Do projects show evidence for all the beliefs?

Does the size of a release affect belief support?

#### 4.5.1.1 Does the size of a release affect belief support?

**Approach:** This work defines size of a release based on the number of files created or modified in a release. To measure fluctuations, first group releases into three non-overlapping categories namely *small*, *medium* & *large* based on the number of distinct files  $D_F$  in a release. Using the distribution of  $D_F$  among



**Figure 4.4** A summary of beliefs in Table 2.1 is shown here. Beliefs are broken into their entities and edges are drawn between two entities to acknowledge the presence of an effect as reported in SE literature. Strength of that effect using the data this work collected is assessed in 4 and this figure is updated (re-drawn) as per the current evidence later in Figure 4.5.

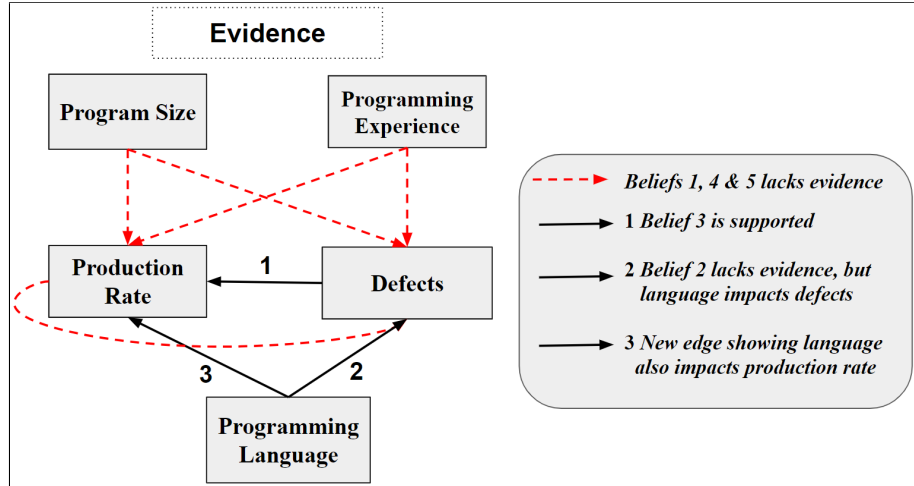
releases of all the 37 projects, this step finds the  $\tilde{x}(D_F)$  to be 18 (files). And using the Inter-Quartile range of this distribution, this method places a release  $r$  of a project  $P$  in one of the following categories:

- *small*: If,  $3 < D_F < 18$  ( $\tilde{x}$ )
- *medium*: If,  $18 \leq D_F < Q3$  (Third quartile)
- *large*: If,  $D_F \geq Q3$  (Third quartile)

Then, for each belief  $BX$ , this step segregates the support score population  $P_{BX}$  (computed in RQ1) into three different populations based on the size of the release. After grouping, this results in 30 populations (treatments) which this step clusters using Scott-Knott-test §3.5.1, resulting in Table 4.5. Note that this step groups releases based on the number of files  $D_F$  rather than the median duration (21 days) since this method model the experiments based on files  $F$ . This decision should not affect the experiment as this step tested  $D_F$  and it's corresponding release duration to be positively correlated with *strong support* of 0.6 (median), considering all releases in all the 37 projects.

**Findings:** This part of the result observes a clear drift in effect distribution among the three release sizes in Table 4.5, indicating the size of a release affects belief support. For example, temporal belief Belief 4 shows strong support in *small* releases but not in *large* releases. Notably, smaller releases have better support than larger releases. Thus, in summary, Table 4.5 tells us that one can reason better about smaller releases than larger releases.

That said, it is important to add that while 47% of releases in the projects are *small*, only 18% (718 releases) of them were qualified for the treatments above as the rest were insignificant ( $p\_value \geq 0.01$ ). So while the conclusions about small releases are a promising result, overall, across all the data, it holds



**Figure 4.5** A copy of Figure 4.4 but, edges are redrawn based on the results obtained from assessing the five beliefs in 4. Here a line edge (black) between two entities denote an effect backed by empirical evidence.

for a very small group.

#### 4.5.1.2 Do projects show evidence for all the beliefs ?

**Approach:** For each project, this method measures coverage of beliefs  $P_C$ , which is simply the number of beliefs a project  $P$  shows at least a *minimum support*. In other words a project  $P$  covers a belief  $BX$  if  $\tilde{x}(P_{BX}) \geq 0.4$ . Then, this step measure  $P_r\%$  (release prevalence), which is the proportion of the aggregated correlation experiments that show at least a *minimum support*. To compute this, aggregate  $P_{BX}$  for all 10 beliefs for a project  $P$ . Then using this aggregated population, compute the % of  $\rho \geq 0.4$ .

**Findings:** The results in Figure 4.6 show that:

- Only 24% of the projects show support for all the 10 beliefs.
- Only 24% of projects show support for less than 5 beliefs (but projects that covered all 10 beliefs had their evidence in only (10 - 36%) of its releases).

#### 4.5.2 Evidence Decay

**Approach:** To observe whether belief support tends to strengthen or decay as a project matures with more releases, this method tests if there is a linear relationship between belief support and its maturity. To measure this, this step correlates between all release dates ( $\{r_{date}\}$ ) of a project with its corresponding support scores  $P_{BX}$  for a belief  $BX$ .

The *Growth*(%) for a belief is the proportion of projects that shows a correlation  $\rho \geq 0.4$  (indicating a positive relationship between release dates and belief support). Similarly, *Decay*(%) for a belief is the proportion of projects that shows a correlation  $\rho \leq -0.4$  (indicating a negative relationship between release dates and belief support).

**Findings:** Figure 4.7 show some support for beliefs both decaying and growing. Beliefs Belief 6 &

Project	$P_c$	$P_r\%$
Propel2	2	21
activeadmin	2	10
mackup	2	3
errbit	2	19
opsworks-cookbooks	3	4
bourbon	3	6
mail	4	13
rails_admin	4	14
doorkeeper	4	8
data-access	5	4
boto3	5	2
omniauth	6	7
bundler	6	21
capybara	6	14
active_merchant	6	4
django-allauth	6	14
monolog	7	5
Codeception	7	20
paperclip	8	9
resque	8	8
puppetlabs-apache	8	13
react-rails	8	8
RestSharp	8	26
fpm	9	14
gazebo_ros_pkgs	9	11
luigi	9	23
django-rest-framework	9	29
sidekiq	9	11
grape	10	30
pelican	10	35
jedis	10	36
draper	10	21
django-tastypie	10	30
coi-services	10	22
beaker	10	12
phinx	10	15
simple_form	10	10

**Figure 4.6** Coverage of beliefs  $P_c$  and its prevalence among releases  $P_r$  is shown here. Projects are sorted based on  $P_c$ .  $P_c$  is the count of beliefs that show at least a *minimum support* in the project.  $P_r\%$  is the proportion of aggregated experiments that show at least a *minimum support* in the project for all the beliefs.

Belief 9 show growth in 21% of the projects. Four of the beliefs (Belief 2, Belief 4, Belief 9 & Belief 10) are decaying in more than 25% of the projects with the highest decay of 51% is observed with ownership-based belief *Belief 10* and notably the growth of just 2%.

That said, the major trend in Figure 4.7 is that beliefs tend to decay, not strengthen as a project matures. Further, to highlight that this step only assessed for linearity and does not delve into the magnitude of this effect which would remain a future work.

**Table 4.5** Small (S) Medium (M) Large (L) Scott-Knott-test placed 30 treatments (10 beliefs \* 3 release size) into various ranks using their support  $P_{BX}$  population. Beliefs with high support  $\rho$  are found in the bottom and less support in the top. (Higher rank indicates stronger support). IQR - Interquartile Range. Treatment labels are sub-scripted with practitioners' agreement from Table 2.2.

Rank	Treatment	Median	IQR	
1	<span style="background-color: #ffd700;">L<sub>Belief</sub>9(35%)</span>	33	21	—●—
1	<span style="background-color: #ffd700;">L<sub>Belief</sub>6(49%)</span>	34	23	—●—
1	<span style="background-color: #ffd700;">L<sub>Belief</sub>4(58%)</span>	37	22	—●—
2	<span style="background-color: #ffd700;">L<sub>Belief</sub>1(76%)</span>	38	17	—●—
2	<span style="background-color: #ffd700;">L<sub>Belief</sub>3(61%)</span>	38	14	—●—
2	<span style="background-color: #ffd700;">L<sub>Belief</sub>8(46%)</span>	41	19	—●—
2	<span style="background-color: #ffd700;">L<sub>Belief</sub>2(64%)</span>	42	18	—●—
2	<span style="background-color: #ffd700;">L<sub>Belief</sub>7(48%)</span>	42	19	—●—
3	<span style="background-color: #ffd700;">L<sub>Belief</sub>10(30%)</span>	52	22	—●—
3	<span style="background-color: #add8e6;">M<sub>Belief</sub>9(35%)</span>	52	14	—●—
3	<span style="background-color: #add8e6;">M<sub>Belief</sub>4(58%)</span>	54	16	—●—
3	<span style="background-color: #add8e6;">M<sub>Belief</sub>7(48%)</span>	54	14	—●—
3	<span style="background-color: #add8e6;">M<sub>Belief</sub>3(61%)</span>	54	12	—●—
3	<span style="background-color: #add8e6;">M<sub>Belief</sub>2(64%)</span>	55	14	—●—
3	<span style="background-color: #add8e6;">M<sub>Belief</sub>8(46%)</span>	55	13	—●—
3	<span style="background-color: #add8e6;">M<sub>Belief</sub>1(76%)</span>	55	17	—●—
3	<span style="background-color: #add8e6;">M<sub>Belief</sub>6(49%)</span>	55	24	—●—
4	<span style="background-color: #ffd700;">L<sub>Belief</sub>5(57%)</span>	63	25	—●—
5	<span style="background-color: #add8e6;">M<sub>Belief</sub>10(30%)</span>	65	15	—●—
5	<span style="background-color: #d3d3d3;">S<sub>Belief</sub>1(76%)</span>	66	4	●
5	<span style="background-color: #add8e6;">M<sub>Belief</sub>5(57%)</span>	69	23	—●—
5	<span style="background-color: #d3d3d3;">S<sub>Belief</sub>2(64%)</span>	74	13	—●—
5	<span style="background-color: #d3d3d3;">S<sub>Belief</sub>6(49%)</span>	75	22	—●—
5	<span style="background-color: #d3d3d3;">S<sub>Belief</sub>9(35%)</span>	74	15	—●—
5	<span style="background-color: #d3d3d3;">S<sub>Belief</sub>3(61%)</span>	76	19	—●—
5	<span style="background-color: #d3d3d3;">S<sub>Belief</sub>7(48%)</span>	78	16	—●—
6	<span style="background-color: #d3d3d3;">S<sub>Belief</sub>10(30%)</span>	80	15	—●—
6	<span style="background-color: #d3d3d3;">S<sub>Belief</sub>4(58%)</span>	82	20	—●—
6	<span style="background-color: #d3d3d3;">S<sub>Belief</sub>8(46%)</span>	80	16	—●—
6	<span style="background-color: #d3d3d3;">S<sub>Belief</sub>5(57%)</span>	87	18	—●—

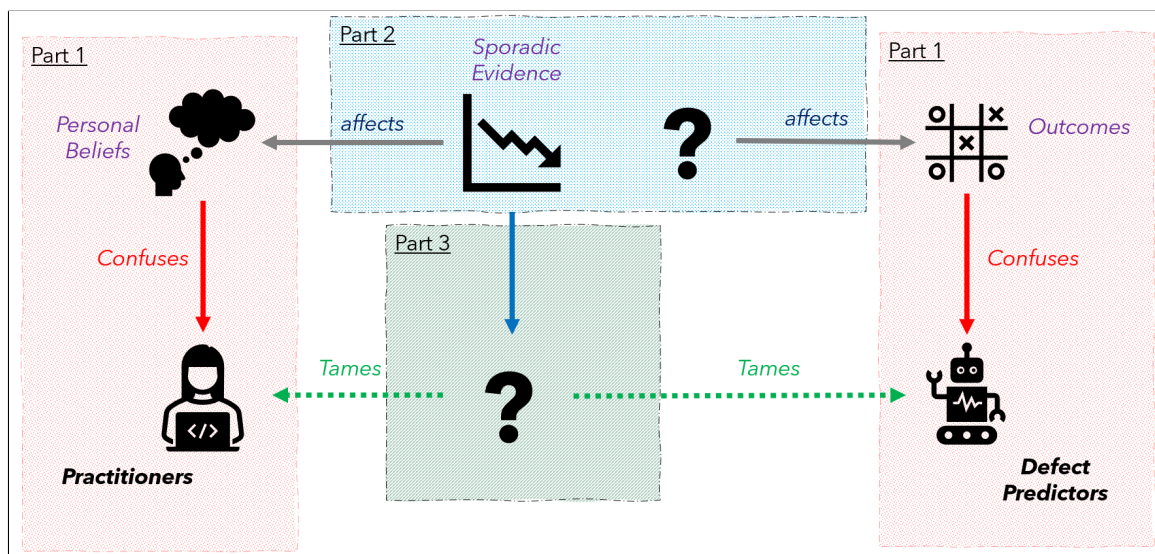
**Result:**  
 No, the same evidence does not appear everywhere. Only 24% of the projects show support for all the 10 beliefs. And those projects showed support for beliefs among 10% - 36% of their releases. Beliefs appear stronger in smaller than larger releases, with fluctuations like Belief 4 (weaker in *large* releases but stronger in *small* releases). Support for beliefs tended to decay than strengthen as the project matured.

## 4.6 Summary

This chapter argues that the sporadicity of evidence prevalence is the leading cause of the disconnect (confusions) between stated practitioner beliefs and empirical evidence. Accordingly, one part (left) of the Figure 4.8 is updated. The next chapter will use this reasoning (sporadic evidence and evidence decay) identified herein in this chapter to devise an early data-lite method and show a way to tame confusions in SE.

Belief ( $B_X$ )	Growth%	Decay%
Belief 5	10%	10%
Belief 7	18%	16%
Belief 8	18%	16%
Belief 6	21%	18%
Belief 3	13%	21%
Belief 1	21%	24%
Belief 2	18%	27%
Belief 4	16%	27%
Belief 9	21%	35%
Belief 10	2%	51%

**Figure 4.7** Evolution of beliefs among all 37 projects. Growth % is the proportion of projects where a corresponding belief correlated with  $\rho \geq 0.4$  between project release dates and empirical evidence ( $P_{B_X}$ ). And, Decay % is the proportion of projects where a corresponding belief correlated with  $\rho \leq -0.4$  between project release dates and empirical evidence ( $P_{B_X}$ ). Beliefs in the plot are sorted based on Decay %. Growth & Decay



**Figure 4.8** A landscape of this thesis, where the first part (Sporadic evidence) of part-2 is answered.

## CHAPTER

# 5

# TAMING CONFUSIONS WITH EARLY DATA

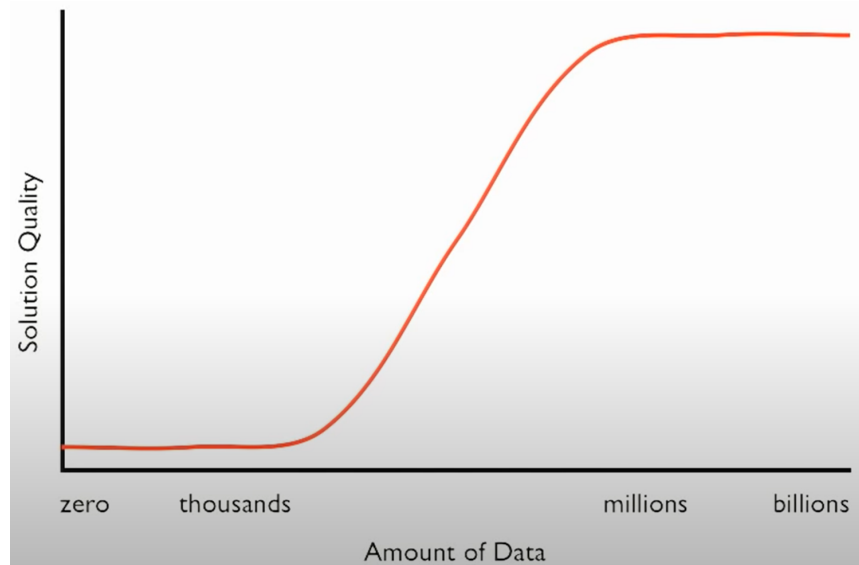
This chapter is based on the material first published as: *Shrikanth, N. C., Suvodeep Majumder, and Tim Menzies. "Early Life Cycle Software Defect Prediction. Why? How?." 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE). IEEE, 2021.*

In an overview, this chapter investigates the project evidence of defect prediction beliefs decaying trend identified in the previous chapter using 155 popular engineering projects. On exploring that effect, this chapter finds and reports a substantial defect decaying trend across projects. Using that trend, it finds defect predictors need no more than the first few months of project data.

### 5.1 Is More data much more useful?

This thesis proposes a *early data-lite* method that finds effective software defect predictors using data just from the first 4% of a project's lifetime. This new method is recommended since one need not always revise defect prediction models, even if new data arrives. This is important since managers, educators, vendors, and researchers lose faith in methods that are constantly changing their conclusions.

This method is somewhat unusual since it takes an opposite approach to *data-hungry (late-data)* methods that (e.g.) use data collected across many years of a software project [D2; Kam12]. Such data-hungry (late-data) methods are often cited as the key to success for data mining applications. For example, in his famous talk, "The Unreasonable Effectiveness of Data," Google's Chief Scientist Peter Norvig argues that "billions of trivial data points can lead to understanding" [Nor11] (a claim he supports with numerous examples from vision research). While Norvig agrees that solution quality would attain a

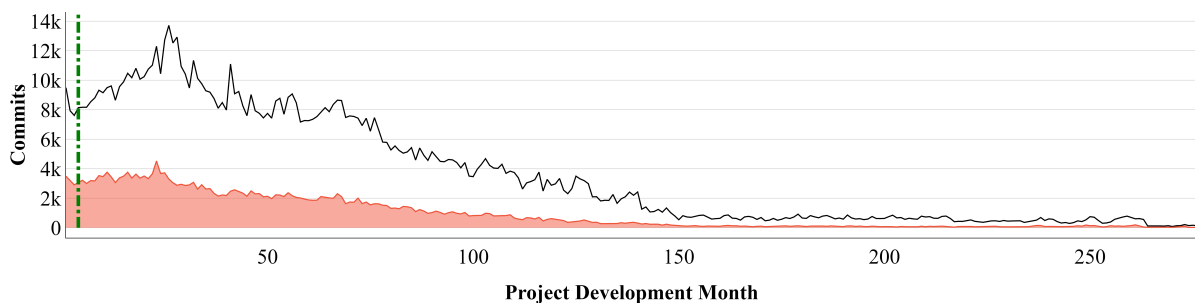


**Figure 5.1** Extracted from the talk ‘The Unreasonable Effectiveness of Data’- Peter Norvig

saturation point beyond a certain amount of data in Figure 5.1, this work questions that amount for defect prediction (x-axis) of that figure.

But what if some Software Engineering (SE) data was not like Norvig’s data? What if SE needs its own AI methods, based on what one learned about the specifics of software projects? If that were true, then data-hungry (late-data) methods might be needless over-elaborations of a fundamentally more straightforward process.

Therefore, one does not need a data-hungry (late-data) approach for one specific software analytics task (learning defect predictors). An observation in Figure 5.2 that while the median lifetime of many projects is 84 months, most of the defects from those projects occur much earlier than that. That is, very little of the defect experience occurs later in the life cycle. Hence, predictors learned after four months (the vertical green dotted line in Figure 5.2) do just as well as anything else; i.e., learning can stop after just 4% of the life cycle (i.e., 4/84 months). That is to say when learning defect predictors:



**Figure 5.2** 1.2 million commits for 155 GitHub projects. Black:Red (shaded) = Clean:Defective commits. This chapter compares (a) models learned up to the vertical green (dotted) line to (b) models learned using more data.

*96% of the time, we do not want and  
we do not need data-hungry (late-data) methods.*

So far, this work has only shown an “early data is enough” effect in the particular case of (a) defect prediction for (b) long-running non-trivial engineering GitHub projects studied here (what Munaiah et al.[Mun17] would call “organizational projects”). Such projects can be readily identified by how many “stars” (approval marks) they have accumulated from GitHub users. Like other researchers (e.g., see the TSE’20 article by Yan et al. [Yan20]), this work explores projects with at least 1000 stars.

That said, even within these restrictions, this work is exploring an interesting range of projects. This sample includes numerous widely used applications developed by Elastic (search-engine<sup>1</sup>), Google (core libraries<sup>2</sup>), Numpy (Scientific computing<sup>3</sup>), etc. Also, the sample of projects is written in widely used programming languages, including C, C++, Java, C#, Ruby, Python, JavaScript, and PHP.

## **5.2 The ‘Early Trend’ discovery**

This section reports an unobserved novel pattern missed by many researchers and practitioners in the software defect prediction space. That pattern is conceived as a result of the conclusions derived in the previous chapter. In other words, the subsequent sub-sections will use the source of disconnect between practitioner beliefs and empirical evidence to show the early trends in hundreds of long-running (7 years median) software projects.

### **5.2.1 Chai-Break Anecdote**

During my tenure with Accenture Labs, India, between 2014 and 2017, I made some friends (colleagues) who accompany me for tea breaks during the later part of the day. Most of my friends who accompany me for the chai-break belonged to a particular project. Although I was part of a different group, we shared the same office space. There were times when they could not accompany me for breaks. Their excuse was that they are closer to the project’s release (a reasonable excuse, in my view). It was interesting to me that this was only the case with the first few releases of that project. In other words, chai-breaks resumed after the project matured (in few months). Notably, in all the organizations I worked with, I experienced similar high software engineering activities in teams during the early stages of the project. On the lines of this thesis, the takeaway here is simple. That much of the project experience (chaos/entropy/high-activity) occur during the early stages of the project life-cycle; than at the later stage.

### **5.2.2 Sporadic Evidence**

The previous chapter showed support for practitioner beliefs were not consistent neither among different projects nor within projects (i.e., releases). Such a sporadicity confuses practitioners’ beliefs in different

---

<sup>1</sup><https://github.com/elastic/elasticsearch>

<sup>2</sup><https://github.com/google/guava>

<sup>3</sup><https://github.com/numpy/numpy>

things while working with different projects at different times (life-cycle). As mentioned earlier, such an effect was apparent at Microsoft, where Devanbu et al. showed beliefs among Microsoft employees were different, though they were part of the same organization. This reasoning is vital because it explains why practitioners are confused and hint that experience (knowledge) may be isolated to a specific region in the software project life-cycle.

### 5.2.3 Evidence Decay

Previously in Chapter 4 showed support for defect prediction beliefs decayed as the project matured with more releases. In other words, the project became stable with fewer defects reported. When fewer defects are reported over time, it naturally affects the set of defect prediction beliefs. This result questions the decades-old methods of software defect prediction that perform recent validation. In other words, the oracle performance is trained and tested on recent changes, whereas much of the defects are recorded earlier in the life cycle.

### 5.2.4 Trends in 100+ Software Projects

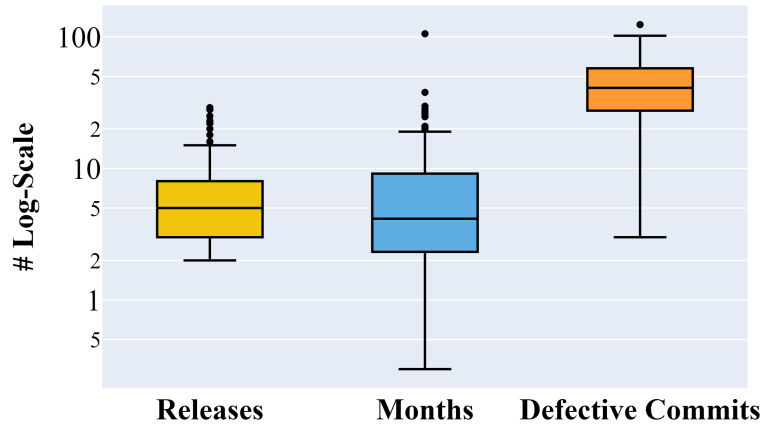
The findings on sporadic evidence and decay of evidence motivated this work to study the pattern of defects in the software project life-cycle. As part of this work, 155 software engineering projects (elucidate earlier in §3.4) were mined. On looking for why such confusions exist – lead to the discovery of that pattern in Figure 5.2. In that figure, the project data changes dramatically over the life-cycle. Figure 5.2 shows data from 1.2m GitHub commits from 155 popular GitHub projects (the criteria for selecting those particular projects is detailed below). Note how the frequency defect data (shown in red/shaded) starts collapsing early in the life cycle. This observation suggests that it might be relatively uninformative to learn from later life cycle data. This was an exciting finding since, as mentioned in the introduction, it is common practice in defect prediction to perform “recent validation” where predictors are tested on the latest release after training from the prior one or two releases [Tan15; McI17; Kon20]. In terms of Figure 5.2, that strategy would train on red dots (shaded) taken near the right-hand side, then test on the most right-hand-side dot. Given the shallowness of the defect data in that region, such recent validation could lead to results that are not representative of the whole life cycle.

Accordingly, this chapter determined how different training and testing sampling policies across the life cycle of Figure 5.2 affected the results. After much experimentation (described below), this chapter asserts that if data is collected up until the vertical green line of Figure 5.2, then that generates a model as good as anything else.

Lastly, note Figure 3.5 shows information on the selected projects. As shown in that figure, the projects have:

- Median life spans of 84 months with 59 releases;
- The projects have (265, 3,728, 83,409) commits (min, median, max) with data up to December 2019;
- 20% (median) of project commits introduce bugs.

Figure 5.3 focuses on just the data used in the early life cycle “**E**” sampler described in Figure 5.5. By the time the project accumulates 150 commits in the median case, projects have had five releases in 4




**Figure 5.3** Distributions seen in the first 150 commits of all 155 projects; median values of project releases (5), project development months (4) and defective commits (41)

months (median value).

## 5.3 Building the Early Data-lite Method

### 5.3.1 Revisiting the Objective

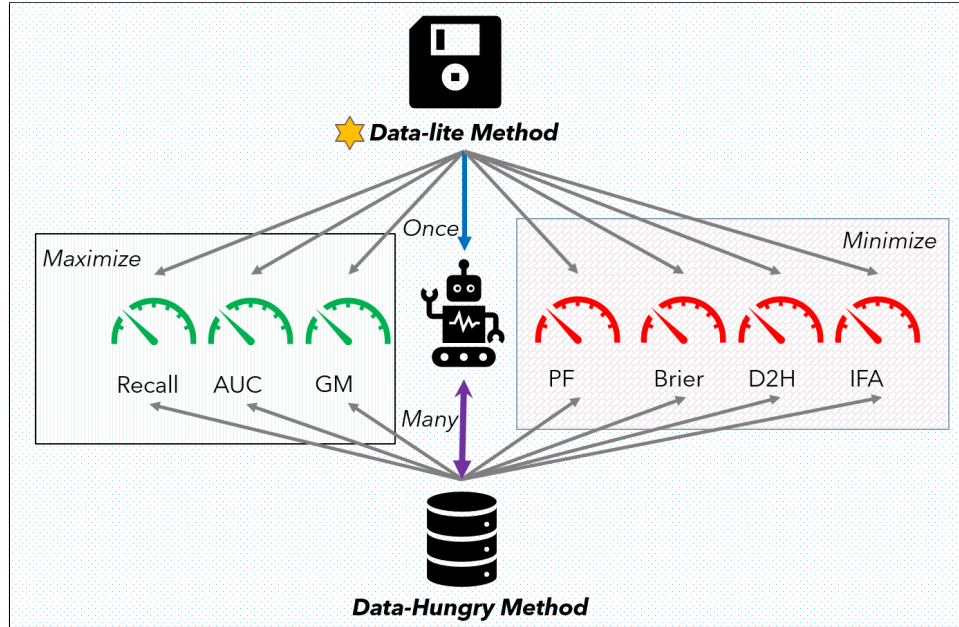
As shown in Figure 5.4 the central goal of this thesis is to devise an antidote to conclusion instability problem in SE. That is defined as follows:

 **Antidote to Conclusion Instability**  
 To devise a method to build static oracles (no re-training) for a software project with better long-lasting decision-making capability than dynamic oracles (frequent re-training).

It turns out Figure 2.2 is only an approximation of the diverse number sampling policies this work sees in the literature. A more comprehensive picture is shown in Figure 5.5 where this work divides software releases  $R_i$  that occur over many months  $M_j$  into some *train* and *test* set.

### 5.3.2 The Data-lite Stop Early Method

This work sat out to determine how different training and testing sampling policies across the life cycle of Figure 5.2 affected the results. After much experimentation (described below), the results asserts that if data is collected up until the vertical green line of Figure 5.2, then that generates a model as good as anything else. One way to summarize this chapter is to evaluate a novel “stop early” sampling policy for collecting the data needed for defect prediction. This section describes a survey of sampling policies in defect prediction. Each sampling policy has its way of extracting training and test data from a project. As shown below, there is a remarkably diverse number of policies in the literature that have not been systematically and comparatively evaluated prior to this work.



**Figure 5.4** Depicts the objective of this thesis

In April 2020, this work found 737 articles in Google Scholar using the query (“software” AND “defect prediction” AND “just in time”, “software” AND “defect prediction” AND “sampling policy”). “Just in time (JIT)” defect prediction is a widely-used approach where the code is seen in each commit is assessed for its defect proneness [Kam12; Fuk14; Kon20; Tan15].

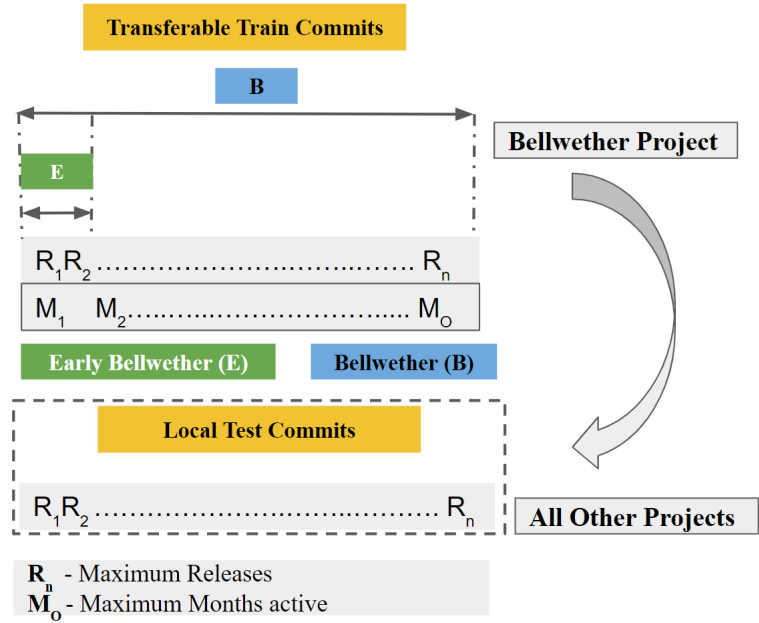
From the results of that query, this work applied some temporal filtering: (1) all articles more recent than 2017 were examined; (ii) for older articles, this work examined all papers from the last 15 years with more than Ten citations per year. After reading the title, abstracts, and methodology sections, this work found the 39 articles of Table 5.1 that argued for particular sampling policies.

Using a little engineering judgment and guided by the frequency of different policies (from Figure 2.2), this chapter elected to focus on four sampling policies from the literature and one ‘early stopping’ policy (see Table 5.2). The % share in Figure 2.2 show ‘ALL and RR’ are prevalent practices whereas ‘M3 and M6’ though not prevalent are used in related literature [McI17; Hoa19]. This work did not consider separate policies for ‘Percentage’ and ‘Slice’ as the former is similar to ‘ALL’ (100% of historical data), and the latter is least prevalent and similar to M6 (180 days or six months).

Note the “magic numbers” in Table 5.2:

- *3 months, 6 months*: these are thresholds often seen in the literature.
- *25 clean + 25 defective commits*: This work arrived at these numbers based on the work of Nam et al. built defect prediction models for using just 50 samples [Nam17].
- *Sampling at random from the first 150 commits*. Here, some experiments were performed recursively dividing the data in half until defect prediction stopped working.

This work will show below that early sampling (shown in gray in Table 5.2) works just as well as the



**Figure 5.5** A visual map of sampling. Project time-line divided into ‘Train commits’ and ‘Test commits’. Learners learn from ‘Train’ to classify defective commits in the ‘Test’.

other policies.

## 5.4 Assessing the Data-lite Method

Tables 5.3 and 5.4 show results when the six learners applied the five sampling policies. The results were plotted into two tables since the policies lead to results with different samples sizes: the recent release, or “RR”, the policy uses data from just two releases while “ALL” uses everything.

In the first row of those tables, “+” and “-” denote criteria that need to be maximized or minimized, respectively. Within the tables, **gray cells** show statistical test results (conducted separately on each criterion). Anything ranked “best” is colored gray, while all else have white backgrounds.

Columns one and two show the policy/learners that lead to these results. Rows are sorted by how often policy/learners “win”; i.e., achieve best ranks across all criteria. In Tables 5.3 and 5.4, no policy+learner wins 7 out of 7 times on all criteria, so some judgment will be required to make an overall conclusion. Specifically, based on results from the multi-objective optimization literature, this step will first remove the policies+learners that score worse on most criteria, then debate trade-offs between the rest.

The results offers two notes. Firstly, across all the learners, the median value for IFA is very small—zero or one; i.e., developers using these tools only need to suffer one false alarm or less before finding something they need to fix. Second, since these observed IFA scores are so small, this work says that “losing” on IFA is hardly a reason to dismiss a learner/sampler combination. Secondly, D2H and GM combine multiple criteria. For example, “winning” on D2H and GM means performing well on both Recall and PF; i.e., these two criteria are somewhat more informative than the others.

**Table 5.1** Papers discussing different sampling policies. All (papers that utilize all historical data to build defect prediction models, shaded in gray).

Paper	Year	Citations	Sampling	Projects
[Men08]	2008	172	All	12
[Zha10b]	2010	21	All	5
[Men10]	2010	347	Percentage	10
[Rom11]	2011	94	All	8
[Kam12]	2012	264	All	11
[D2]	2012	387	All	5
[Wan13]	2013	322	All	10
[Zha14]	2014	105	All	1,403
[Lu14]	2014	44	Release	1
[Rah14]	2014	93	Release	5
[Tan15]	2015	129	All	7
[Ryu16b]	2016	87	All	10
[Kri16]	2016	42	Release	23
[Fu16a]	2016	111	Release	17
[Fu16b]	2016	28	Release	10
[Ray16]	2016	130	Release	10
[Hua17]	2017	44	All	6
[McI17]	2017	36	Month	6
[Nam17]	2017	220	All	34
[Zha17]	2017	44	Percentage	255
[Özt17]	2017	8	All	10
[Dam19]	2018	36	All	11
[Tan18]	2018	66	Percentage	
[Wu18]	2018	33	All	18
[Wan18b]	2018	25	All	16
[Che18b]	2018	40	All	6
[Agr18a]	2018	61	All	9
[Tan18]	2018	43	Percentage	101
[Pas18]	2018	15	Release	13
[Che19b]	2019	16	Percentage	7
[Pas19]	2019	14	Month	10
[Kon19]	2019	11	Percentage	26
[Hua19]	2019	14	Slice	6
[Yan19a]	2019	1	All	10
[Yan19b]	2019	0	Percentage	6
[Yat19]	2019	4	Release	9
[Ben19]	2019	10	Release	20
[Hoa19]	2019	8	All, Slice	2
[Kon20]	2020	0	All	6

Turning now to those results, this work explores two issues. For defect prediction:

**RQ3:** Does learning from more data pacify confusion?

**RQ4:** Is recent data more important than older data to pacify confusion?

**Table 5.2** Four representative sampling policies from literature and an early life cycle policy (the row shown in gray).

Policy	Method
<b>ALL</b>	Train using all past software commits ( $[0, R_i)$ ) in the project before the first commit in the release under test $R_i$ .
<b>M6</b>	Train using the recent six months of software commits ( $[R_i - 6months)$ ) made before the first commit in the release under test $R_i$ .
<b>M3</b>	Train using the recent three months of software commits ( $[R_i - 3months)$ ) made before the first commit in the release under test $R_i$ .
<b>RR</b>	Train using the software commits in the previous release $R_{i-1}$ before the first commit in the release under test $R_i$ .
<b>E</b>	Train using early 50 commits (25 clean and 25 defective) randomly sampled within the first 150 commits before the first commit in the release under test $R_i$ .

**Table 5.3** 24 defect prediction models tested in all 4,876 applicable project releases. In the first row, “+” and “-” denote the criteria that need to be maximized or minimized, respectively. ‘Wins’ is the frequency of the policy found in the top #1 Scott-Knott rank in each of the seven evaluation measures (the cells shaded in gray).

Policy	Classifier	Wins	D2H-	AUC+	IFA-	Brier-	Recall+	PF-	GM+
M6	NB	4	0.37	0.67	1.0	0.32	0.78	0.33	0.67
M3	NB		0.37	0.67	1.0	0.31	0.76	0.32	0.68
E	LR		0.36	0.68	1.0	0.32	0.71	0.31	0.68
M6	SVM		0.43	0.65	0.0	0.21	0.44	0.1	0.48
M3	SVM		0.43	0.65	0.0	0.21	0.43	0.1	0.48
ALL	NB	3	0.4	0.65	1.0	0.36	0.83	0.40	0.67
E	KNN	2	0.39	0.65	1.0	0.33	0.65	0.32	0.62
ALL	LR		0.38	0.66	1.0	0.3	0.65	0.25	0.62
M6	LR		0.36	0.68	1.0	0.25	0.59	0.19	0.60
M3	LR		0.36	0.68	1.0	0.24	0.58	0.17	0.60
M6	KNN		0.4	0.65	0.0	0.23	0.50	0.14	0.53
ALL	SVM		0.4	0.66	0.0	0.25	0.50	0.14	0.54
M3	KNN		0.41	0.65	0.0	0.23	0.47	0.13	0.51
M6	RF		0.44	0.63	0.0	0.24	0.43	0.12	0.47
E	SVM	1	0.4	0.64	1.0	0.31	0.6	0.26	0.59
ALL	KNN		0.38	0.66	1.0	0.25	0.55	0.17	0.57
ALL	DT		0.42	0.62	1.0	0.32	0.52	0.25	0.54
M6	DT		0.43	0.62	1.0	0.29	0.5	0.2	0.51
ALL	RF		0.42	0.64	1.0	0.26	0.49	0.15	0.51
M3	DT		0.43	0.62	1.0	0.28	0.48	0.19	0.5
M3	RF		0.44	0.63	1.0	0.24	0.42	0.11	0.46
E	DT		0	0.46	0.58	1.0	0.38	0.57	0.35
E	NB	0.54		0.54	1.0	0.37	0.55	0.29	0.41
E	RF	0.44		0.61	1.0	0.33	0.52	0.26	0.52

KEY: ■ More data (ALL, M6 and M3) ■ Early (E)

Note that this work does explore a third research issue: are different learners better at learning from a little, a lot, or all the available data. Based on the results, this work has nothing definitive to offer on that issue. That said, if one were pressed to recommend a particular learning algorithm, then one could say

**Table 5.4** 12 defect prediction models tested on 3,704 project releases. In the first row, “+” and “-” denote criteria that need to be maximized or minimized, respectively. ‘Wins’ is the frequency of the policy found in the top #1 Scott-Knott rank in each of the seven evaluation measures (the cells shaded in gray).

Policy	Classifier	Wins	D2H-	AUC+	IFA-	Brier-	Recall+	PF-	GM+
E	LR	4	0.36	0.68	1.0	0.32	0.71	0.31	0.68
RR	NB	3	0.38	0.66	1.0	0.32	0.71	0.30	0.65
RR	LR		0.35	0.68	1.0	0.24	0.59	0.18	0.61
RR	SVM		0.42	0.64	0.0	0.23	0.47	0.12	0.5
E	KNN	2	0.39	0.64	1.0	0.34	0.64	0.32	0.62
RR	KNN		0.41	0.64	1.0	0.25	0.5	0.15	0.53
RR	RF		0.43	0.63	1.0	0.24	0.43	0.13	0.48
E	SVM	1	0.4	0.64	1.0	0.31	0.6	0.26	0.59
E	DT	0	0.46	0.58	1.0	0.39	0.56	0.35	0.54
E	NB		0.54	0.54	1.0	0.37	0.54	0.29	0.42
E	RF		0.44	0.61	1.0	0.33	0.52	0.26	0.53
RR	DT		0.42	0.62	1.0	0.28	0.50	0.20	0.51

KEY: ■ Recency (RR) ■ Early (E)

there are no counterexamples to the claim that “it is useful to apply CFS+LR”.

## 5.5 RQ3: Does learning from more data pacify confusion?

*Belief1:* Earlier, this chapter discussed examples where proponents of data-hungry (late-data) methods advocated that if data is useful, then even more data is much more useful.

*Prediction:1* If that belief was the case, then in Table 5.3, data-hungry (late-data) sampling policies that used more data should defeat “early data-lite” sampling policies.

*Observation1a:* In Table 5.3, The “data hungriest” sampling policy (ALL) loses on most criteria. While it achieves the highest Recall (83%), it also has the highest false alarm range (40%). As to which other policy is preferred in the best *wins=4* zone of Table 5.3, there is no clear winner. What this work would say here is that the preferred “early data-lite” method called “E” (that uses 25 defective and 25 non-defective commits selected at random from the first 150 commits) is competitive with the rest. Hence:

**Answer3a:** For defect prediction, it is not clear that more data is inherently better.

*Observations1b:* Figure 3.5 of this work showed that within the sample of projects, this work has data lasting a median of 84 months. Figure 5.3 noted that by the time this work gets to 150 commits, most projects are 4 months old (median value). The “E” results of Table 5.3 showed that defect models learned from that 4 months of data are competitive with all the other policies studied here. Hence this work says,

**Answer3b:** 96% of the time, we do not want and we do not need data-hungry (late-data) methods

## 5.6 RQ4: Is recent data more important than older data to pacify confusion?

*Belief2*: As discussed earlier in this chapter, many researchers prefer using recent data over data from earlier periods. For example, it is common practice in defect prediction to perform “recent validation” where predictors are tested on the latest release after training from the prior one or two releases [Tan15; McI17; Kon20; Fu16a]. For a project with multiple releases, recent validation ignores all the insights that are available from older releases.

*Prediction2*: If recent data is comparatively more informative than older data, then defect predictors built on recent data should out-perform predictors built on much older data.

*Observations2*: This work observes that:

- Figure 3.5 of this work showed that within the sample of projects, this work has data lasting a median of 84 months.
- Figure 5.3 noted that by the time this gets to 150 commits, most projects are 4 months old (median value).
- Table 5.4 says that “E” wins over “RR” since it falls in the best *wins=4* section.
- Hence this work could conclude that older data is *more* effective than recent data.

That said, this thesis feels somewhat more the circumspect conclusion is in order. When this work compare E+LR to the next learner in that table (RR+NB) this work only finds a minimal difference in their performance scores. Hence this work makes a somewhat humbler conclusion:

**Answer4**: Recency-based methods perform no better than results from early life cycle defect predictors.

This is a startling result for two reasons. Firstly, compared to the “RR” training data, the “E” training data is very old indeed. For projects lasting 84 months long, “RR” is trained on information from recent few months, with “E” data comes from years before that. Secondly, this result calls into question any conclusion made in a paper that used recent validation to assess their approach; e.g. [Tan15; McI17; Kon20; Fu16a].

## 5.7 Threats to Validity

### 5.7.1 Sampling Bias

The conclusion’s generalizability will depend upon the samples considered; i.e., what matters here may not be true everywhere. To improve the conclusion’s generalizability, this work mined 155 long-running OS projects that are developed for disparate domains and written in numerous programming languages. Sampling trivial projects (like homework assignments) is a potential threat to the analysis. To mitigate that, this work adhered to the advice from prior researchers. This work finds the sample of projects have 20% (median) defects as shown in Figure 3.5 nearly the same as data used by Tantithamthavorn et

al. [Tan18] who report 30% (median) defects.

### 5.7.2 Learner bias

Any single study can only explore a handful of classification algorithms. For building the defect predictors this work elected six learners (Logistic Regression, Nearest neighbor, Decision Tree, Support Vector Machines, Random Forrest, and Naïve Bayes). These six learners represent a plethora of classification algorithms [Gho15].

### 5.7.3 Evaluation bias

This work uses seven evaluation measures (Recall, PF, IFA, Brier, GM, D2H, and AUC). Other prevalent measures in this defect prediction space include precision. However, as mentioned earlier, precision has issues with unbalanced data [Men08].

### 5.7.4 Input Bias

The proposed sampling policy ‘E’ randomly samples 50 commits from the early 150 commits. Thus it may be true that different executions could yield different results. However, this is not a threat because each time, the early policy ‘E’ randomly samples 50 commits from the early 150 commits to test sizeable 8,490 releases (from Table 5.3 and Table 5.4) across all the six learners. In other words, the conclusions about ‘E’ hold on a large sample size of numerous releases.

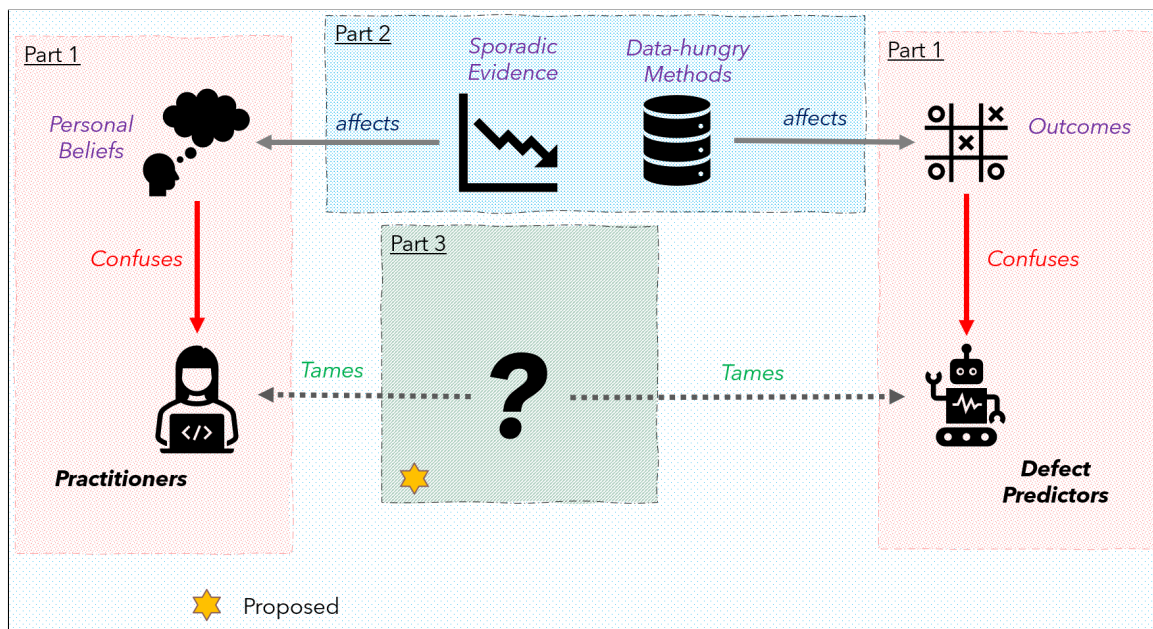


Figure 5.6 A landscape of this thesis, where the second part (data-hungry methods) of part-2 is answered.

## 5.8 Summary

When data keep changing, the models we can learn from that data may also change. If conclusions become too fluid (i.e., change too often), then no one has a stable basis for making decisions or communicating insights. Issues with conclusion instability disappear if we can learn a predictive model that is effective for the rest of the project early in the life cycle. The Figure 5.6 is updated to reflect the results endorsed in this chapter. This chapter reported a key trend in software projects and tested an early data-lite sampling method to tame confusions caused by oracles during SQA activities. The next chapter will explore this early data-lite method to simplify software analytics.

## **Part 3 - Simplifying Software Analytics**

## CHAPTER

# 6

# EXPANDING THE SCOPE OF EARLY METHODS

This chapter is based on the material titled "*Simplifying Software Defect Prediction (via the "early bird" Heuristic)*" by Shrikanth, N. C., and Tim Menzies [Shr21a] which is under review.

To repeat the main message of this thesis: in the domain of software defect prediction, humans are (a) confused about what influences software defects since they tend to (b) reason across over *all* project data while the (c) most relevant experience occurs very early in the project life-cycle. Hence (d) better reasoning can be achieved by focusing only on earlier life cycle data

So far, the current work has only demonstrated points a,b,c. The actual test of this thesis is point (d); i.e., that the effect reported in (c) can be used to improve software analytics (where "improved" means simpler, faster, and more effective).

## 6.1 Introduction

In defect prediction, researchers often fail to try simpler methods before trying complex analytics [Shr21c; Zho18]. For example:

- "For many new projects may not have enough historical data to train prediction models" [Rah12]
- "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." [Fu16a]
- "Many research studies have shown that ensemble learning can achieve much better classification performance than a single classifier" [Yan17]

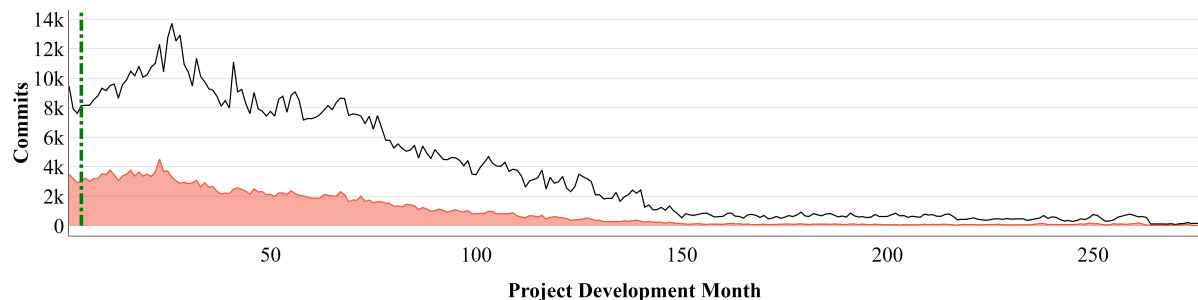
While this work does not doubt those results in that context, show in this work that can simplify software defect prediction if one focuses more on the quality of the data than on applying various software analytic techniques. Furthermore, this chapter shows that much of the prevalent methods used to build defect predictors are needless and can be ‘simplified’ to a large extent with knowledge-rich early project data.

By simplified mean:

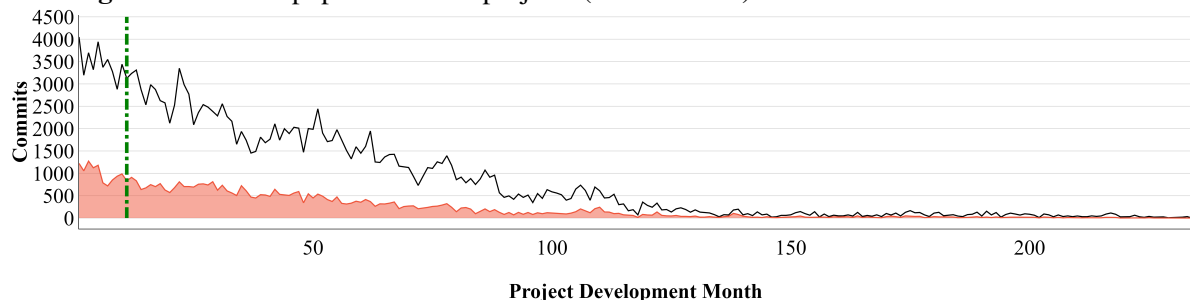
- Stable conclusions (no need to update defect predictors for every new project release).
- Less run-time (With fewer project data can find and build relevant predictors faster).
- Better explanations (With fewer features, one can communicate the oracles (predictors) decisions to stakeholders effectively).

If “too much data” can be too much, when is enough data just enough to build effective defect predictors? The answer to this question comes from a previously unreported effect is shown in Figure 6.1. As shown in this figure, when looking at the percent of buggy commits in GitHub projects, a remarkable pattern emerges. Specifically, most of the buggy commits occur earlier in the life cycle.

This observation prompted an investigation of a “early data-lite” approach that just uses early life cycle data to predict for defects. The literature review of §6.3 shows that, surprisingly, this approach to defect prediction has been overlooked by prior works. This is a significant oversight since, in 240 GitHub projects, can show that defect predictors learned from the first 150 commits work as well, or better, than state-of-the-art alternatives.



**Figure 6.1.A:** 155 popular GitHub projects (#stars> 1000). Data from 1.2 million commits.



**Figure 6.1.B:** 85 unpopular GitHub projects (#stars< 1000). Data from 253,289 commits.

**Figure 6.1** Most defective commits occur early in the life cycle. Black:Red = Clean:Defective commits. In this chapter, compare (a) models learned up to the vertical green (dotted) line to (b) models learned using more data.

Overall, this chapter highlights:

- The information within projects may not be evenly distributed across the life cycle. For such data, it can be very useful to adopt a “early data-lite” approach.
- For example, using early life cycle data, found simple models (with only two features) that generalize across hundreds of projects. Such models can be built much faster than traditional methods (weeks versus months of CPU time).
- So before researchers use all available data, they need to first check that their (e.g.) buggy commit data occurs at equal frequency across the life cycle. say this since much prior work on methods for learning from multiple projects [Men11; Li12; Ma12; He12; Men12; Bet12; Rah12; Tur13; Nam13; Pet13b; Can13; Pet13a; Her13; He13; Rah13a; Fuk14; Pan14; Zha14; Ryu15; Che15; Zha15; Pet15; Can15; Jin15; He15; Zha16a; Kam16; Yan16; Ryu16b; Xia16a; Jin16; Ryu16a; Zha16c; Kri16; Wan16a; Ryu17; Nam17; Ni17; Zho18; Che18b; Hos18; Kri18; Li18; Wan18b; Wu18; Dam18; Hua19; Che19a; Che19b; Liu19] needlessly complicated an inherently simple process.

The rest of this chapter assesses the efficacy of the early data-lite method proposed in the previous chapter and offers experimental evidence that this early life cycle method works better than sophisticated methods like classifier tuning, ensemble and notably made transfer learning algorithms work simpler, faster, and more stable. This chapter uses the abbreviations of Table 6.1.

## 6.2 Defect Prediction Analytics

### 6.2.1 Data: Transfer Learning

Defect predictors are learned from project data. What happens if there is not enough data to learn those models? This is an especially acute problem for newer projects (and in the absence of historical data in

**Table 6.1** Table of Important Acronyms

Acronym	Abbreviation
AUC	Area under the receiver operating characteristic curve
CFS	Correlation-based Feature Selection
DODGE	Optimizer proposed by Agrawal et al. in [Agr19]
DT	Decision Tree
HPO	Hyperparameter optimization
HYPEROPT	Optimizer proposed by Bergstra et al. [Ber11]
IFA	Initial Number of False Alarms
KNN	k-nearest neighbors algorithm
LA and LT	Refer to features list in Table 3.4
LR	Logistic Regression
MCC	Matthews correlation coefficient
NB	Naive Bayes classifier
PF	False Alarm Rate
RF	Random forest
SMOTE	Synthetic Minority Over-sampling Technique [Cha02]
SVM	Support vector machines
SZZ	Sliwerski Zimmerman Zellar Algorithm [S05]
TCA	Transfer Component Analysis [Nam13]
TLEL	Two-layer Ensemble Learning Algorithm [Yan17]
TPE	Tree-structured Parzen Estimator [Ber11]

some legacy projects [Bri02]).

In such scenarios, practitioners and researchers might identify matured projects that share some similarities to their local projects. Once found, then lessons learned could be *transferred* from the older to the new project. There are kinds of transfer:

- *Cross: cross-project defect prediction.* Lessons learned from *other* projects and applied to *this* project.
- *Within: within-project defect prediction.* Using data from *this* project, lessons learned from prior experience are used to make predictions about later life cycle development.

To say the least, transfer learning is a very active research area in software engineering (SE). Can find more than 1,000 articles in the last five years alone (found using the query "cross-project defect prediction,"<sup>1</sup>). By count, within that corpus, there have been at least two dozen transfer learning methods [Ama20]. Interesting methods evolved in that research include:

- Heterogeneous transfer that lets data expressed indifferent formats transferred from project to projects [Nam17];
- Temporal transfer learning, which is a within-project defect prediction (*Within*) tool where earlier life cycle data is used to make predictions later in the life cycle [Koc15].

The reading of the literature is that, apart from this research, the prior state-of-the-art in the SE literature is Nam et al.'s TCA+ (Transfer Component Analysis) method [Nam13]. For a list of important abbreviations used in this chapter, see Table 6.1. Given data from some source and target project, TCA strives to "align" the source and target data via dimensionality rotation, expansion, and contraction. TCA+ is an extension to basic TCA that uses automatic methods to find normalization options for TCA.

## 6.2.2 Techniques : Tuning and Ensemble

Classification algorithms can be trained to classify a project commit as defective. Studies have shown their predictive performance can depend upon the set of hyper-parameter they are initially configured [Fu16a; Agr19].

For example, the k-nearest neighbors (*KNN*) classifier explored in this study is widely used and available in the scikit-learn [Ped11] machine learning library. Notably, by default *KNN* is set to the following default parameters based on standard machine learning literature as follows:

*n\_neighbors = 5, weights = uniform, algorithm = auto, leaf\_size = 30, p = 2, metric = minkowski, metric\_params = None, n\_jobs = None*

However, as seen from the numerous parameters available for *KNN*, there is a huge parameter space of options the *KNN* can be tried and tested (tuned). Therefore most machine learning algorithm's hyper-parameters can be tuned using training data before testing on project releases in this case.

There are numerous ways to find the right set of parameters for a classifier given the training data. But the decision predominantly comes with the run-time cost. For example, searching through all available options 'Manual Search' but it may not terminate. One may try 'Random Search' with a terminating condition, but the probability of finding the near best parameter may be low. Fu et al. [Fu16a] explored 'grid-search' a baseline hyper-parameter approach in the field of machine learning [Fu16a] but found it to be very slow for defect prediction. Recently (2019), Agrawal et al. proposed a novel optimizer

---

<sup>1</sup>Queried <https://scholar.google.co.in/> in 2020

that terminates quickly after finding near-optimal hyper-parameters for defect predictors [Agr19]. More details on DODGE will be presented in §3.6. Therefore this chapter will explore DODGE with early methods from the previous chapter.

Another avenue of complex approach is the use of ensemble method, where the philosophy is why use just ‘one’ when ‘many’ is better. Numerous studies had shown defect predictors had shown significant improvement when an ensemble of classifiers was used rather than just one [Yan17; Wan13; He09]. In 2017, Yang et al. proposed a two-layer ensemble approach for defect prediction called TLEL (Two-layer Ensemble Learner) that showed promising improvements when compared to a predictor with a single classifier. §3.6 elucidates its operation, and this chapter will explore TLEL as part of the complex methods to test the efficacy of early methods.

### 6.2.3 Issues with current approaches

Nevertheless, just because a technique is popular does not mean that it should be recommended. The reading of the literature is most recent SE analytics chapters have taken a complex approach (e.g., see the quotes in the previous chapter) and use state-of-the-art analytics without testing with simpler alternatives [Zho18] argue there that it can be useful to try early data-lite before adopting complex techniques that demand more data, delay analytic, does not offer explanations and utilize system resources. Because, for one thing, all that data might not be available. The availability of more data in an industrial setting is not assured. Also, it may not be useful to learn from more data. Proprietary data may not be readily available for practitioners to build predictors for their local projects. Zimmermann et al. showed that transferring predictors from the same domain does not guarantee quality predictions [Zim09]. Finally, due to privacy concerns, teams even within the same organization may not readily make their matured project available for others to use [Pet13a].

For another thing, several researchers have reported that a complex approach can be problematic:

- The introduction chapter discussed an issue seen when learning from 700,000+ GitHub commits.
- At her ESEM’11 keynote address, Elaine Weyuker questioned that she will ever have the option to make the AT&T information public [Wey08].
- In over 30 years of COCOMO effort, Boehm could share cost estimation data from only 200 projects.

Next, applying techniques discussed in §6.4.2 comes with a CPU run-time cost. From time to time, researchers have strived and endorsed to look for simpler alternatives; hence it is important that explore them.

## 6.3 Related Work

In the previous chapter it was shown that historical data within the project were used in large quantities to build defect predictors were needless. In this chapter, the efficacy of early methods in various contexts to be tested are as follows:

- On Unpopular SE projects
- Transfer learning scenarios

- Hyper-parameter optimization and Ensemble approaches

One reason to explore early life cycle data reasoning is that it has not been done before. Much of the SE transfer learning methods [Tur13; Nam13; Pet13b; Can13; Pet13a; Fuk14; Pan14; Ryu15; Che15; Zha15; Pet15; Can15; Jin15; Kam16; Yan16; Ryu16b; Xia16a; Jin16; Ryu16a; Zha16c; Kri16; Ryu17; Nam17; Ni17; Zho18; Che18b; Hos18; Kri18; Li18; Wan18b; Hua19] and Tuning or ensemble methods [Bow18; Tan15; Xia16a; Lar15; Tan16b; Wan18b; Ryu16b; Zho19; Tan18; Rhm20; Kam16; Yan17; Che20; Son18; Bir11; Gho15; Wan13; Ryu15; Pan14; Jin16; Ryu17; Pet13b; Wan16b; Sun12; Sie15; Pan20; Hud18; Xu19; Ton18; Pas19; Rat17; Kam16; Lam17; Ye14; Sca14; Agr18a; Agr18c; Zha16c; Agr19; Jia13; Fu16a; Li17a; Wan16a; Liu19] could be characterized as “late-data” (data-hungry) since they transferred all available data from one or more projects to build their predictors.

### 6.3.1 Literature Survey

In the previous chapter, we looked at 100’s of articles on defect prediction but scoped to within-project learning to understand the prevalent sampling policies. However, predictors can be built using data transferred from other projects and employing complex methods like ensemble or optimizers.

Therefore to assess the scope of the early data-lite method in other areas we perform additional literature reviews in this chapter for two reasons, (a) to understand more about late-data reasoning in SE and (b) to chose representative transfer learning, ensemble and optimizer methods to gauge the value of early project data.

Hence, in early 2021 we queried Google Scholar with:

- Query “cross-project defect prediction” and that returned a total of 982 articles in the last ten years.
- Query “defect prediction” AND “tuning OR hyper-paramater optimization” and that returned a total of 1,740 in the last five years.
- Query “defect prediction” AND “defect prediction” AND “ensemble” and that returned a total of 3,050 in the last five years.

Following the advice of Agrawal et al. [Agr18a], and Mathews et al. [Mat18] only “highly cited” papers were chose, i.e., those with more than ten citations per year. In that list, survey papers and papers that do not build defect predictors were discarded. As a result only papers that performed some transfer learning experiments or a complex method ( ensemble or tuning) were retained. The final list of papers are presented in Table 6.2 (Transfer learning) and Table 6.3 (Tuning and Ensemble methods).

Within those three sets of papers, three categories of sampling policies emerged:

- ‘All’ if the methods of that chapter use all rows from one or more projects to build defect predictors.
- ‘Part’ if the methods of that chapter explore a large search space of one or more projects to find a small set of rows that are worthy of transfer data.
- In one case, in 2016, an unsupervised approach used no training data and just clustered the test data to find outliers, which were then labeled as bugs [Zha16a].

Note that regardless of being “All” or “Part”, the analysis looks at the data across the entire life cycle before returning some or all of it.

As to other kinds of sampling policies, for all these papers, counted:

**Table 6.2** 50 ‘highly cited’ (more than 10 citations per year) chapters that ran one or more *Cross* experiment(s) since 2010.

Year	#Data	#Features	Projects	Cites/Year	chapter
2011	■	20	2	16.44	[Men11]
		198	6	19.25	[Li12]
2012	■	17	10	39.13	[Ma12]
		20	10	30.13	[He12]
		20	7	23.38	[Men12]
		20	2	13.88	[Bet12]
		8	9	25.38	[Rah12]
2013	■	16	41	14	[Tur13]
		17	8	48.43	[Nam13]
		20	41	22.43	[Pet13b]
		20	10	19.57	[Can13]
		20	10	13.29	[Pet13a]
		20	14	13.29	[Her13]
		20	10	11.43	[He13]
2014	■	54	12	32	[Rah13a]
		14	11	16.5	[Fuk14]
		20	10	20.5	[Pan14]
2015	■	26	1398	18.5	[Zha14]
		18	7	13.4	[Ryu15]
		20	11	19.8	[Che15]
		20	10	14.4	[Zha15]
		20	17	12.4	[Pet15]
		20	10	11.2	[Can15]
		28	11	24.4	[Jin15]
2016	None	20	10	33	[He15]
		X	26	35.5	[Zha16a]
		14	11	26.5	[Kam16]
		14	6	19.75	[Yan16]
		17	10	23.5	[Ryu16b]
		20	10	35.5	[Xia16a]
		20	21	20.75	[Jin16]
		20	30	11.75	[Ryu16a]
		26	1390	16.25	[Zha16c]
61	23	10.5	[Kri16]		
2017	■	X	10	70.25	[Wan16a]
		20	15	21.33	[Ryu17]
		61	34	80	[Nam17]
2018	■	61	8	10.67	[Ni17]
		6	58	21.5	[Zho18]
		14	6	20.5	[Che18b]
		20	11	23	[Hos18]
		61	18	18.5	[Kri18]
		61	28	14.5	[Li18]
		20	10	14	[Wan18b]
		61	16	19.5	[Wu18]
2019	■	X	10	20	[Dam18]
		14	6	17	[Hua19]
		61	8	14	[Che19a]
		20	7	20	[Che19b]
2019	■	20	14	19	[Liu19]

KEY: ■ Part, □ All, None - No training data,  
X - Features automated

- The number projects used in those studies;
- The number of features used by their predictors.
- The applied any ensemble or tuning or both techniques.

On trying to place the early methods explored in this chapter into Table 6.2 and Table 6.3, one can

**Table 6.3** 44 ‘highly cited’ (more than 10 citations per year) chapters that built defect prediction models applying methods based on ensemble/tuning/both in the past five years (2016 to 2021).

Year	Cites/Year	Type	Data	Projects	chapter	Year	Cites/Year	Type	Data	Projects	chapter
2011	37.9	ENSEMBLE	All	2	[Bir11]	2013	25	TUNING	All	6	[Jia13]
2012	17.33		All	14	[Sun12]	2014	33		All	6	[Ye14]
2013	54.13		All	10	[Wan13]	2014	36.86		All	20	[Sca14]
2013	24		All	41	[Pet13b]	2016	15.8		All	1,390	[Zha16c]
2014	22		All	10	[Pan14]	2016	33.4		Part	17	[Fu16a]
2015	53.33		All	29	[Gho15]	2016	79.8		Part	10	[Wan16a]
2015	13.33		All	7	[Ryu15]	2017	29.25		All	6	[Lam17]
2015	18.17		All	6	[Sie15]	2017	47.25		Part	7	[Li17a]
2016	29.2		All	11	[Kam16]	2018	37		All	9	[Agr18a]
2016	22.4		All	21	[Jin16]	2018	44		All	8	[Agr18c]
2016	15.8		All	12	[Wan16b]	2019	14		All	10	[Agr19]
2017	31		All	6	[Yan17]	2019	18.5		Part	14	[Liu19]
2017	22.5		All	15	[Ryu17]	2020	28		All	32	[Kam16]
2017	16.25		Part	11	[Rat17]	2015	32.67		All	7	[Tan15]
2018	32.67		All	27	[Son18]	2015	44	All	6	[Lar15]	
2018	21.33		All	15	[Hud18]	2016	37.2	All	10	[Xia16a]	
2018	27.67		All	12	[Ton18]	2016	50.8	All	18	[Tan16b]	
2019	26		All	44	[Xu19]	2016	24.4	BOTH	All	10	[Ryu16b]
2019	21.5		Part	10	[Pas19]	2018	31		All	18	[Bow18]
2020	28		All	2	[Rhm20]	2018	19.67		All	6	[Wan18b]
2020	31		All	3	[Che20]	2018	47.67		Part	18	[Tan18]
2020	20		All	12	[Pan20]	2019	13.5		All	25	[Zho19]

KEY: ■ Part, □ All

find that the approach will be quite literally off the charts. The methods advocated by this chapter are neither “whole” nor “part” since learn from early data, then stop collecting (so unlike all the research in Table 6.2 and Table 6.3, never look at all the data). Further, the “number of projects”=1 (which does not even appear in Table 6.2) and only use a handful of features (far less than the features used by other work in Table 6.2).

The “cross-project defect prediction” query was restricted to the last ten years. In contrast, the other two queries were restricted to only the last five years because there were 1000’s more articles to investigate manually. However, this is not a threat as the baseline methods in this thesis were devised in the last five years.

Hence assert, with some confidence, that the methods of this chapter have not been previously explored.

### 6.3.1.1 Representative Techniques

The following guidelines are adhered to design an appropriate experiment:

Firstly, comparing the early sample to sampling over a larger space of project data. Hence, in the following, will show *early* versus *all* experiments.

Secondly, there are two ways to find data: *within-project* and *cross-project*. Therefore, will divide the *early* early-data and *all* late-data experiments into:

- early-data: early-within and early-cross
- late-data: all-within and all-cross

Nature	Type	Method	Pre-processing	# Features (columns)	# of commits (rows)
late-data	<i>Cross</i>	Bellwether [Kri16] TCA+[Nam13]	CFS, SMOTE and steps illustrated in §6.4.1 SMOTE	Selected by CFS.  5 components with linear kernel (data supplied with all features)	All commits from the identified cross project.  Pick first 150 and last 150 commits from the bellwether project.
early-data	<i>Cross</i>	$*E_{size}(Bellwether)$  $*E_{size}(TCA+)$	Steps illustrated in §6.4.1 in [Nag05; Kam12; Kon20]  Steps illustrated in §6.4.1	LA, LT = lines added, lines of code in file before change  2 components with linear kernel (data supplied only with LA and LT)	Sample equal number of defective and clean commits as available in the first 150 commits (not exceeding 25 each).
early-data	<i>Within</i>	$E$  $*E_{size}$	CFS and steps illustrated in §6.4.1 Steps illustrated in §6.4.1	Selected by CFS  LA and LT	(same as above)
late-data	<i>Within</i>	$ALL$	CFS, SMOTE and steps illustrated in §6.4.1	Selected by CFS.	All the commits before the release under test.

KEY: All early-data sampling methods are shaded (■), and policies proposed in this chapter are indicated by \*.

**Table 6.4** This table lists three baselines approaches (where two of them are *Cross* and the remaining one is a *Within*) along with three early-data variants to support this chapter. Note many of these methods used the steps advised in [Nag05; Kam12; Kon20]. Those steps are discussed in §6.4.1. Also, for a discussion on CFS, see §3.7.

- Complex Methods: Optimizers (DODGE and HyperOpt) and ensemble method (TLEL).

Lastly, looking into the literature can see some clear state-of-the-art algorithms that should be represented in this chapter (specifically, the *TCA+* and *Bellwether* cross-project learning methods [Kri16; Nam13]). Accordingly, when exploring *cross-project* learning, those methods will be employed.

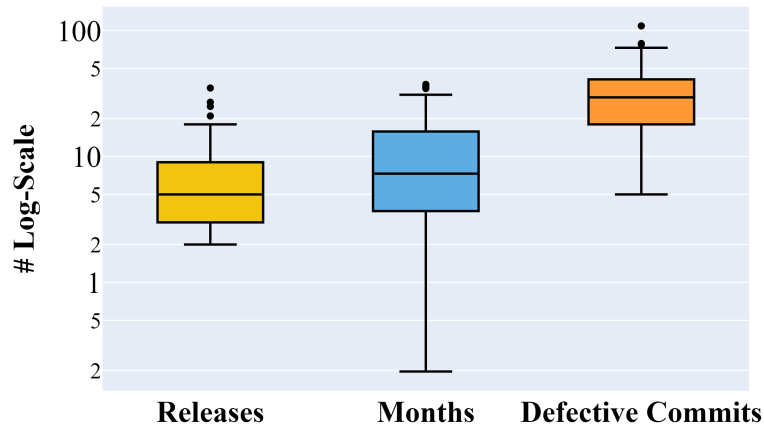
To explain the exact methods used in this chapter requires some further details on those algorithms. Please see §3.6 for the algorithm details and for a discussion of what algorithms selected. But, from the analysis of the literature survey the early-data variants of these techniques have not been explored before in SE.

The above areas do not cover all of the defect prediction literature, but it is assured that it covers a range of active research areas within defect prediction to test the scope of the conclusion about early methods.

From the above we offer the methods of the next section as representative sampling methods in SE (see list in Table 6.4).

## 6.4 Experimental Methods

The rest of this chapter offers an experimental evaluation of *early life cycle* methods versus other learning policies for the data of Figure 6.1.



**Figure 6.2** Unpopular projects. Median= 5 releases, 7 months and 29 defective commits.

### 6.4.1 Data

Please refer to chapter 3 for source and collection of data and for details about the data refer to Table 3.4, Figure 3.6 and Figure 3.7. Unlike previous chapter this chapter uses 240 projects (both popular and unpopular GitHub projects) to increase the scope of our conclusion. Further, Figure 5.3 (popular projects) and Figure 6.2 (unpopular projects) show the distribution of releases, months and defective commits when those projects made 150 commits.

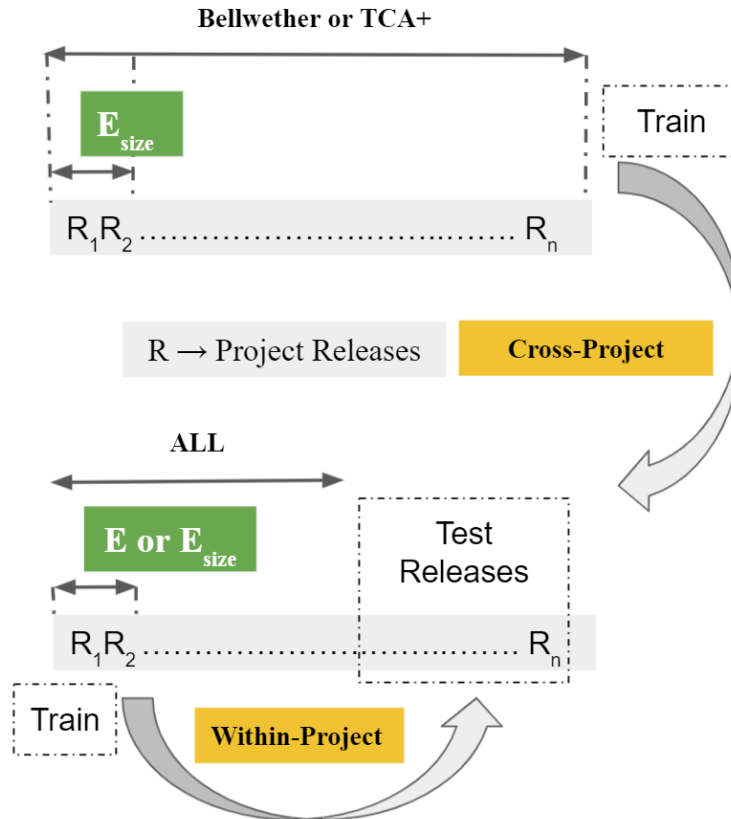
This chapter uses three sets of algorithms:

- The 11 algorithms described in §3.6;
- Pre-processing algorithms (for some sampling policies) described in §3.7;
- The seven sampling methods are described in §6.4.2.

### 6.4.2 Sampling Methods

Figure 6.3 shows a set of options from which can generate a large number of sampling options. For example, training data come from *within* or be drawn *cross* from another project. Also, in the bellwether method, most of the training data (only using the project that yields a model that outperforms all the other project models) are explored and pruned.

To find representative sampling techniques, check the papers listed in Table 6.2 for standard sampling policies, *Cross* techniques, projects, and features. Notably, numerous papers about *Cross* techniques reported in the past decades; for example, Sousuke’s work recently (2020) explored 24 *Cross* approaches [Ama20] in within-project context. In the context of *Within*, the previous chapter identified many sampling strategies (based on recent releases, recent months of data, and all past data) to build predictors. However, in the context of *Cross*, found only two (‘whole’ or ‘part’). Nevertheless, both ‘whole’ or ‘part’ are late-data. The focus of this chapter is to check the efficacy of using early-data approaches in prevalent *Cross* approaches and not rank numerous *Cross* techniques. Therefore these two representative *Cross* techniques, specifically *TCA+* and *Bellwether*, were chosen because:



**Figure 6.3** A visual map of sampling. Project time-line divided into ‘Train commits’ and ‘Test commits’. Learners learn from ‘Train’ to classify defective commits in the ‘Test’.

- *TCA+* is an active *Cross* technique in this space based on the seminal work by Nam et al. [Nam13].
- *Bellwether*, a recent (2018) baseline approach by Krishna et al. that performed better than *TCA+* [Kri18]

Further, the results of these two techniques that are tested on 1000+ project releases will be discussed later in §6.7. Let us look into each of these late-data techniques below.

**Bellwether Project (*Bellwether*):** All the commits from a *bellwether* project are used as the training data [Kri18]. The bellwether project is identified by comparing the predictive performance scores of every project with every other project within the population. The bellwether transfer learning method assumes “*When a community works on software, then there exists one exemplary project, called the bellwether, which can define predictors for the others.* [Kri18]”. According to Krishna et al. [Kri18] that finding the bellwether requires an  $O(N^2)$  study— an approach that can be difficult to scale to 100s or 1000s of projects.

**Transfer Component Analysis (TCA+):** Pan et al. proposed a domain adaptation technique that enables the transition of information between source and target domains [Pan10]. Extending that, Nam et al. stacked many normalization rules on top of TCA to form TCA+ [Nam13]. A specific normalization

rule is applied based on the similarity of the data set characteristics between the source and the target project. Rahul and Menzies [Kri18] report TCA+ to perform better than other transfer methods, namely Transfer Naive Bayes [Ma12] and Value Cognitive Boosting Learner [Ryu16b].

An approach to transfer learning using TCA+ is as follows:

- Choose a bellwether project randomly from all the bellwether projects.
- Later in this chapter, specifically §6.7 will show for TCA+ that as the number of training commits increases, the run-time cost of TCA+ increases drastically. It is practically not feasible to test in all 12,000+ releases repeated for all ten classifiers. In the case of *Bellwether* based policies can at least cache the predictor as it uses a fixed set of all the commits. Caching is impossible with TCA+ as the approach decides transformation rules based on both train and test commits.
- To manage the run-time complexity without having to hide the data-hungriness of TCA+, do the following:
  - Defect predictors are built with varying sizes in the previous chapter 5 and found that predictors needed a minimum of 150 training commits. As mentioned above, unlike other methods, *TCA+* is CPU intensive; therefore, one could only increase training commits up to 300 (2x 150) for this experiment. This is not a sampling threat because later in §5.4 find the early data-lite *TCA+* variant performs better than *TCA+* trained with 300 commits supporting the early conjecture.
  - But picking the first 300 commits would ignore the recent project data and may not represent the project life cycle. Thus of those 300 commits, sample the earliest 150 commits and the latest (recent) 150 commits of the project data. Nevertheless, note in the context of *Within* showed (previous chapter) that using all the project commits is needless for predicting defects.

For TCA+ related policies listed in Table 6.4 reused the implementation from their replication package<sup>2</sup> by Kondo et al., which is an EMSE'19 article about feature reduction techniques on defect prediction models [Kon19].

The next four sections will discuss the two early-data variants of the above two techniques.

**Early Bellwether** ( $E_{size}(Bellwether)$ ): Instead of using all the commits in the bellwether project, a small sample of training data is curated. Specifically by randomly sampling 25 defective and 25 clean changes from the first (earliest) 150 commits (a method proposed in the previous chapter). All the data features are removed except two size-based features, 'LA' and 'LT.' The rationale for choosing two features are discussed later in §6.6.

**Early TCA+** ( $E_{size}(TCA+)$ ): This proposed technique is similar to TCA+ except for the 'sampling method'. Instead of using all the commits in the selected project, a small sample of training data is curated. Specifically by randomly sampling 25 defective and 25 clean changes from the first (earliest) 150 commits (a method proposed in the previous chapter). All the data features are removed except two size-based features, 'LA' and 'LT.' The rationale for choosing two features are discussed later in §6.6.

Then, to compare *Cross* with *Within* techniques, chose  $E$  that is endorsed in the previous chapter

---

<sup>2</sup><https://sailhome.cs.queensu.ca/replication/featred-vs-featsel-defectpred/>

and shown to outperform predictors that use all past commits or recent commits. To check if  $E$  can be sufficed with just two features as did for  $E_{size}(Bellwether)$ , created  $E_{size}$ .

**Early Sampling ( $E$ ):** The training data is curated randomly sampling 25 defective and 25 clean changes from the first 150 changes made within the project under test.

**Early Sampling with few features ( $E_{size}$ ):** Training data is curated same as  $E$  but it is restricted to two size based features ‘LA’ and ‘LT’ listed in Table 3.4. The rationale for this is discussed later in §5.4.

Sampling methods that are not super-scripted by + imply that features are selected while building predictors as required using the correlation-based Feature Selection (CFS) feature selection discussed in §3.7.

Lastly, note here that this work is ‘not’ to rank different *Cross* techniques like *TCA+* or *Bellwether* but to ‘check’ if such prevalent techniques can suffice (or do better) with early-data approaches.

An overview of the experiment is shown in Figure 6.3. Predictors using the sampling strategies listed in Table 6.4 and classifiers elucidated in §3.6 are built. Then all project releases using those predictors are tested. Similar to the method in previous chapter 5 as this is a realistic way to compare all sampling policies and methods on a similar scale.

## 6.5 RQ5: Can we build early defect prediction models from unpopular projects?

### 6.5.1 Motivation

The results from previous chapter that endorsed early methods were scoped to only popular GitHub projects. Choosing only popular projects is a sampling decision often made by many SE researchers in empirical studies to mitigate generalizability threat [Mun17]. As mentioned earlier in chapter 3, unpopular projects could be non-trivial projects (like homework assignments), and that could affect the conclusion.

On the contrary, a strong argument could be that perhaps popular projects may not realistically represent software engineering in the real world. In other words, not all projects are fully staffed with a sufficient budget. Therefore it is necessary and worthy to check the value of early methods on the unpopular sample of projects. Chapter 3 and §6.9 discuss the selection of unpopular projects sampled from GitHub is non-trivial.

### 6.5.2 Approach

To check whether the proposed early method work on unpopular projects, compare defect predictors built by sampling training commits using  $E$  with those that sampled all past data, i.e., *ALL*. Since *ALL* subsumes more data, other stratifications like release or three months are not considered. Notably, *ALL* is a prevalent (50%) sampling method in software defect prediction. Therefore to assess defect predictors on each project release this construct the experiment as follows:

- Sample training data within the project. The sampling will depend upon either *E* or *ALL*.
- The sampled training data is pre-processed, and appropriate features are selected as listed in Table 3.4.
- Classifiers listed in §3.6 are instantiated with copies of pre-processed training data.
- All the instantiated predictors are tested on all 85 unpopular project releases, and their predictive performance is gauged using seven measures elucidated in §3.7.1.
- Lastly, their scores (e.g., the population of recall scores) per classifier+policy (pair) is ranked using the Scott-Knott test elucidated in §3.5.1.

Note:

- To avoid a methodological error, do not test/consider project releases before the first 150 commits.
- Therefore, the population of evaluation measures is tested on an equal number of unpopular project releases.

### 6.5.3 Results

Table 6.5 shows the result of the predictors tested in all applicable project releases of 85 unpopular projects. Policy *E* uses fewer early data (row #1) performs just as same as predictors that were built sampled with *ALL* (all past commits) in row #3.

**Table 6.5** ‘12’ *within-project* defect prediction models tested ‘only’ in 85 *unpopular* project’s releases. In the first row, “+” and “-” denote the criteria that need to be maximized or minimized, respectively. All the other rows show combinations of sampling policies and classifiers. Green cells denote “early-data” sampling was employed. Cells marked in gray all have the same top rank as the best results (and those ranks were determined by the Scott-Knott algorithm described in §3.5.1). The ‘wins’ column (see column #3) counts how often a particular sampling policy/classifier achieves a top score. Inter-quartile ranges are indicated withing ‘( )’.

Policy	Classifier	Wins	Recall+	PF-	AUC+	D2H-	Brier-	G-Score+	IFA-	MCC+
E	LR		75 (50)	40 (34)	64 (17)	41 (22)	38 (22)	65 (27)	1 (4)	24 (28)
E	SVM		69 (50)	38 (30)	64 (18)	41 (21)	37 (21)	63 (30)	1 (4)	23 (31)
ALL	SVM	6	50 (54)	15 (18)	65 (25)	40 (32)	23 (14)	54 (51)	1 (5)	28 (45)
ALL	RF		50 (52)	16 (22)	64 (22)	42 (28)	25 (16)	53 (46)	1 (4)	28 (41)
ALL	KNN		50 (43)	20 (17)	64 (21)	40 (25)	26 (14)	54 (41)	1 (4)	26 (38)
E	KNN	5	67 (40)	42 (26)	62 (19)	42 (19)	40 (17)	61 (28)	2 (5)	19 (30)
E	RF	4	56 (52)	28 (28)	63 (20)	42 (23)	33 (18)	56 (39)	2 (5)	23 (33)
ALL	NB	2	14 (60)	9 (27)	50 (13)	64 (24)	25 (22)	14 (51)	4 (12)	1 (30)
E	NB		56 (66)	35 (38)	56 (19)	48 (25)	38 (24)	52 (47)	2 (6)	14 (32)
ALL	LR	1	50 (80)	25 (28)	60 (19)	47 (37)	30 (20)	51 (69)	2 (7)	17 (36)
E	DT		50 (53)	32 (31)	57 (19)	47 (27)	37 (23)	52 (46)	2 (6)	14 (33)
ALL	DT		50 (38)	30 (23)	57 (18)	45 (23)	34 (18)	51 (35)	2 (5)	15 (34)

*Early methods are ‘not’ scoped to only popular projects.*

## 6.6 RQ6: Can we build early defect prediction models with fewer features?

### 6.6.1 Motivation

Chapter 5 showed that it is possible to build operable defect predictors using fewer early project data. One reason in the preceding work was that much of the knowledge required (defects) to build defect predictors were reported in a few months of the project. In this work, explore that region again to understand which of those 14 features listed in Table 3.4 are essential to the early predictor  $E$ . And if the answer to that is a few, one can restrict the predictors to only learning from a fixed set of features while producing operable performance? It would be easier to explain with fewer features as to why a specific commit was classified as defective by the predictor.

### 6.6.2 Approach

To answer this RQ perform three experiments as follows:

- In the first experiment, the frequency of features chosen by CFS while building defect predictors which were sampled using  $E$  is identified.
- In the second experiment, a new sampling policy (say  $E_?$ ) that only uses the most frequently chosen feature identified in the above step is created.
- Create two predictors that sampled training commits (pre-processed as listed in Table 6.4) using  $E$  and  $E_?$  and test on all project releases.
- In the third experiment, test all project releases using the ManualUp and ManualDown approach (elucidated in §3.6) that uses no training information. This is because researchers have advised to include such trivial approaches that do not use any training information [Zho18].
- Lastly, compare the predictive performance of defect predictors that were built using  $E$ ,  $E_?$  and *ManualDown* and *ManualUp* on all eight evaluation measures listed in §3.7.1 and rank them using the Scott-Knott test elucidated in §3.5.1.

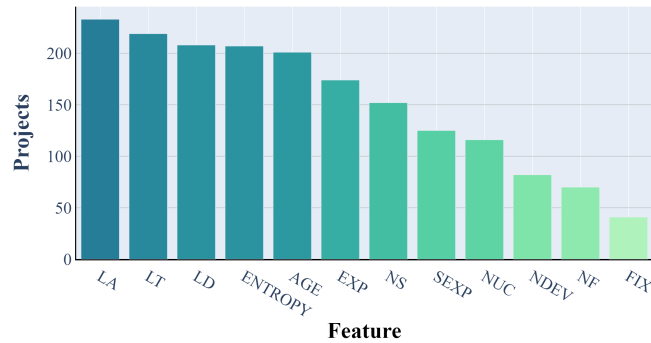
### 6.6.3 Results

The following observations are made from Figure 6.4 and Table 6.6:

#### 6.6.3.1 Finding Important Features:

The previous chapter’s early sampling rule ‘E’ shown to be adequate for *Within*. Given that building these models from such a sample, it is tempting to ask, “does that small sample produce a succinct model?.” Note that this was indeed the case; then, one could offer a clear report on what factors most influence defects.

Figure 6.4 shows the frequency of features selected by the CFS algorithm (see §3.7) across all the 240 projects (using just the first 150 commits). To select the “best” features from that space, build models



**Figure 6.4** Frequency of features chosen by the CFS feature selector (using just the first 150 commits) sampled using  $E$  in all the 240 projects

using the top  $1 \leq x \leq 12$  ranked features, stopping with  $x + 1$  features performed no better than  $x$  features (and here are testing all the releases that occurred after that first 150 commits). That procedure reported that models learned from two size-ranked features performed as well as anything else:

- LA: Lines of code added
- LT: Lines of code in a file before the change

Using the results from Figure 6.4 built early defect predictor only using the top two features ‘LA’ and ‘LT.’ Therefore  $E_?$  becomes  $E_{size}$ . The results from Table 6.6 clearly show that  $E_{size}$  have more wins than predictors that sampled training commit using  $E$ . And by extension to prior result *ALL* or other stratification like recent release or based on months. While *ManualDown* is seen at the top of the table due to higher recall, it also produced many false-alarms (see row #3 where  $PF$  is 48%).

*At-least for defect prediction, early data with two features ‘la’ and ‘lt’ is better than using recent (or more) data with many features.*

## 6.7 RQ7: Can build early defect prediction models from transferring early life-cycle data from other projects?

### 6.7.1 Motivation

§6.4.2 lists both the situations and challenges of when one might need to transfer project data to build defect predictors. Although one cannot solve all *Cross-project* challenges in one article, one could pacify some of them by checking the efficacy of early methods in this context. If early methods work, then can transfer relevant data from vast search space (many projects) in less time. Further, one can also reduce the need for data availability as one will only need a small portion of early data.

**Table 6.6** ‘14’ *within-project* defect prediction models tested in all 240 project’s releases. In the first row, “+” and “-” denote the criteria that need to be maximized or minimized, respectively. All the other rows show combinations of sampling policies and classifiers. Green cells denote “early-data” sampling was employed. Cells marked in gray all have the same top rank as the best results (and those ranks were determined by the Scott-Knott algorithm described in §3.5.1). The score highlighted in bold red text indicate undesirable PF, despite many wins on other evaluation measures. The ‘wins’ column (see column #3) counts how often a particular sampling policy/classifier achieves a top score. Inter-quartile ranges are indicated withing ‘( )’.

Policy	Classifier	Wins	Recall+	PF-	AUC+	D2H-	Brier-	G-Score+	IFA-	MCC+
$E_{size}$	LR		75 (40)	28 (25)	70 (15)	34 (17)	30 (17)	70 (25)	1 (3)	34 (26)
$E_{size}$	SVM	6	73 (44)	30 (27)	69 (15)	35 (19)	30 (18)	69 (26)	1 (3)	33 (27)
-	<i>ManualDown</i>		86 (25)	<b>48 (15)</b>	70 (12)	36 (11)	41 (17)	75 (14)	1 (4)	30 (22)
$E_{size}$	KNN	4	70 (41)	33 (26)	67 (17)	37 (19)	33 (18)	66 (26)	1 (3)	28 (28)
$E_{size}$	NB		75 (53)	38 (41)	63 (17)	44 (23)	36 (25)	61 (35)	1 (4)	24 (31)
E	LR		70 (42)	37 (33)	63 (16)	41 (22)	36 (20)	62 (30)	1 (4)	24 (28)
E	SVM		67 (42)	33 (31)	64 (17)	40 (20)	34 (19)	63 (30)	1 (4)	25 (28)
E	KNN		66 (43)	38 (30)	62 (18)	42 (20)	37 (20)	59 (28)	1 (4)	21 (31)
$E_{size}$	RF	2	64 (40)	31 (30)	64 (17)	40 (20)	33 (18)	61 (28)	1 (4)	25 (30)
$E_{size}$	DT		62 (39)	38 (30)	61 (18)	43 (19)	38 (20)	57 (28)	1 (4)	19 (32)
E	RF		60 (46)	32 (33)	61 (18)	43 (21)	35 (21)	56 (33)	1 (4)	20 (32)
E	DT		56 (46)	41 (37)	56 (17)	49 (22)	41 (24)	52 (33)	2 (5)	11 (30)
E	NB		50 (75)	27 (48)	54 (14)	55 (30)	36 (26)	38 (62)	2 (7)	8 (26)
-	<i>ManualUp</i>	0	14 (25)	52 (15)	30 (12)	73 (11)	59 (17)	16 (27)	7 (13)	-30 (22)

## 6.7.2 Approach

To check the efficacy of *Cross-project* methods against early methods will build predictors using the following sampling strategies as below:

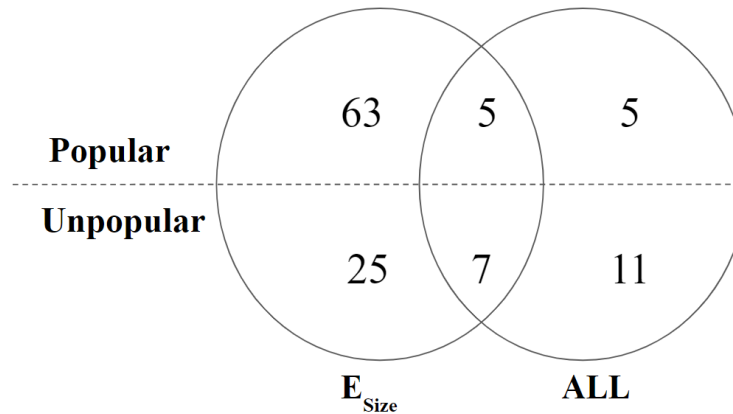
- *Cross-project* methods *Bellwether* and *TCA+*
- created early variants of *Cross-project* methods *Bellwether* and *TCA+* namely  $E_{size}(Bellwether)$  and  $E_{size}(TCA+)$ . The details of these sampling strategies are elucidated in Table 6.4 and portrayed visually in Figure 6.3.
- also include the within-project  $E_{size}$  to compare *Cross* and *Within*.
- Note: do not include *ALL* or *E* because  $E_{size}$  outperformed those sampling strategies in Table 6.6 and in the prior result.

The *Cross-projects* methods require a representative project among the 240 projects to transfer data from.

### 6.7.2.1 Finding Representative Cross-Project:

Using just LA and LT for each of the projects, a model and measured its performance on the other projects were found. If that median performance was more the 70% ‘Recall’ and less than 30% ‘PF’, then the model was deemed “satisfactory”. In a result that endorses the use of early life cycle data, Figure 6.5 shows that *more* of the models found via the  $E_{size}$  method was “satisfactory” than those found using all the data. That effect is particularly marked in the popular projects<sup>3</sup>.

<sup>3</sup>Clearly, this analysis might be overly dependent on the “magic numbers” used to select “satisfactory” bellwethers, i.e., 70% ‘Recall’ and less than 30% ‘PF.’ But the experiments using 12 ( $i \leq 12$ ) different bellwether projects with nearly equal



**Figure 6.5** Number of “satisfactory” bellwethers identified independently by two different sampling policies in all 240 projects.

Lastly, like in previous RQ’s compare the predictive performance of the various defect predictors that were built using different sampling policies on all eight evaluation measures listed in §3.7.1 and rank them using the Scott-Knott test elucidated in §3.5.1.

Note: avoid the methodological error of testing the representative *Cross-project* in this experiment.

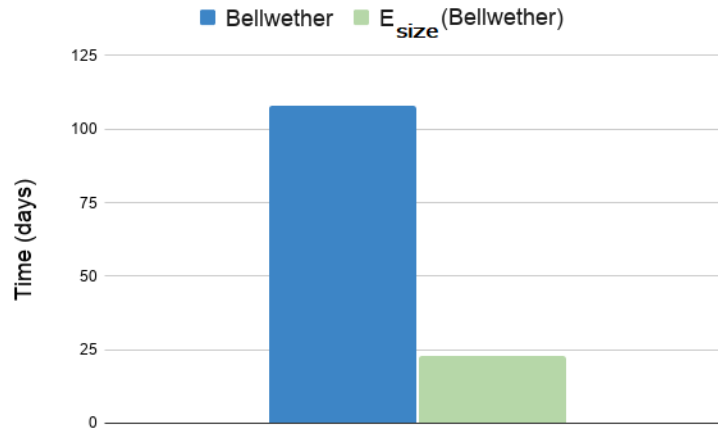
### 6.7.3 Results

The following observations are made using Figure 6.6, Figure 6.7 and Table 6.7.

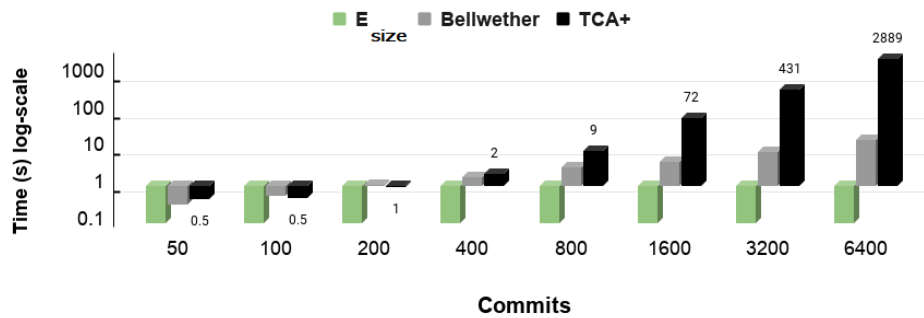
- Rows #1 and #2 top of the Table 6.7 confirm that it is better to build *Cross* based predictors using early data-lite method  $E_{size}$  than using sampling strategies that use more data (like *TCA+* or *Bellwether*).
- Figure 6.6 and Figure 6.7 shows that it is much faster to identify and build bellwethers from a large pool of projects using  $E_{size}$  than *Bellwether*. This is because  $E_{size}$  based predictors need not update (re-train) by accumulating newer project commits for every new project release.
- Row #6 of Table 6.7 is the position of the previous chapter that builds predictors using *Within E* project commits. That proves to show that one can do better than local data if one finds good bellwethers.

Transfer Learning methods perform better (faster and accurate) when they are sampled using the early method.

frequency (see the seven unpopular + 5 popular bellwethers in the middle of Figure 6.5). The statistical analysis of the 12  $E_{sizeBellwether^i}$  using §3.5.1 shows that (a) they tie with each other and (b) perform as well or better than the other transfer learning methods explored in this chapter.



**Figure 6.6** Time taken to identify qualifying bellwether projects in all 240 projects using *Bellwether* and  $E_{size}(Bellwether)$  policies. Note that these experiments ran on a multi-core cloud-based CPU farm. To collect the data here (which assumes the experiments running on a single-core machine), summed the CPU time across all the cores in the experimental rig. assert that this sum is valid since, in the experiments, there is nearly no communication between the cores (except that very end to accumulate the results).



**Figure 6.7** Time taken by different sampling policies to train a model based on the number of commits.

## 6.8 RQ8: Do complex methods supersede early defect prediction models?

### 6.8.1 Motivation

RQ's 3 to 7 have used classifiers with default parameter settings (off the shelf). However, numerous studies, especially in the space of software defect prediction, have shown considerable improvement in predictive performance when the classifiers are tuned [Fu16a; Agr19] or using an ensemble approach [Yan17]. But note the downside of tuning is the run-time overhead.

Therefore, it is essential to check if complex methods like tuning or ensemble favor sampling policies that use recent (or more) data than those confined to the early regions. It is also motivating to check if early methods can benefit from these complex methods. Perhaps the run-time overheads can be alleviated by working with fewer early data.

**Table 6.7** ‘30’ cross-project and ‘1’ within-project (denoted as  $\implies$ ) defect prediction models tested in all 240 project’s releases. In the first row, “+” and “-” denote the criteria that need to be maximized or minimized, respectively. All the other rows show combinations of sampling policies and classifiers. Green cells denote “early-data” sampling was employed. Cells marked in gray all have the same top rank as the best results (and those ranks were determined by the Scott-Knott algorithm described in §3.5.1). The ‘wins’ column (see column #3) counts how often a particular sampling policy/classifier achieves a top score. Inter-quartile ranges are indicated withing ‘()’.

Policy	Classifier	Wins	Recall+	PF-	AUC+	D2H-	Brier-	G-Score+	IFA-	MCC+
$E_{size}$ (Bellwether)	SVM		91 (26)	40 (34)	70 (16)	35 (21)	35 (23)	74 (23)	1 (4)	32 (26)
$E_{size}$ (TCA+)	SVM	6	88 (33)	38 (28)	70 (16)	35 (19)	34 (21)	74 (24)	1 (4)	32 (26)
$E_{size}$ (Bellwether)	LR		85 (30)	34 (27)	73 (15)	31 (17)	31 (18)	76 (20)	1 (4)	35 (25)
$E_{size}$ (TCA+)	LR		82 (36)	32 (25)	72 (16)	33 (19)	31 (19)	74 (22)	1 (4)	34 (26)
$E_{size}$ (TCA+)	KNN		79 (50)	32 (25)	70 (18)	35 (20)	32 (18)	72 (28)	1 (5)	30 (28)
$\implies E_{size}$ (Within)	LR	5	78 (43)	25 (25)	73 (16)	31 (18)	27 (18)	73 (28)	1 (3)	37 (27)
$E_{size}$ (Bellwether)	KNN		78 (43)	30 (26)	70 (17)	34 (19)	30 (19)	72 (27)	1 (3)	32 (27)
$E_{size}$ (TCA+)	RF		75 (46)	31 (24)	69 (17)	35 (19)	32 (18)	71 (28)	1 (4)	30 (28)
$E$ (TCA+)	LR	4	80 (40)	41 (19)	69 (16)	36 (15)	38 (17)	72 (24)	2 (5)	28 (27)
$E_{size}$ (Bellwether)	RF		71 (41)	28 (26)	68 (18)	36 (21)	30 (19)	67 (28)	1 (4)	28 (28)
Bellwether	SVM	3	21 (44)	4 (10)	57 (17)	56 (30)	19 (15)	25 (49)	1 (7)	23 (40)
Bellwether	RF		19 (36)	3 (8)	56 (15)	56 (25)	19 (15)	23 (41)	1 (7)	22 (39)
$E_{size}$ (Bellwether)	NB		88 (33)	49 (34)	64 (19)	42 (22)	42 (23)	67 (28)	2 (5)	23 (27)
$E_{size}$ (TCA+)	NB		82 (38)	43 (30)	67 (18)	39 (21)	38 (22)	70 (26)	2 (5)	25 (28)
Bellwether	LR	2	76 (51)	33 (43)	67 (16)	39 (20)	32 (24)	66 (34)	1 (4)	31 (28)
$E_{size}$ (TCA+)	DT		71 (39)	32 (25)	67 (19)	37 (19)	33 (18)	67 (27)	2 (4)	27 (28)
Bellwether	KNN		25 (41)	6 (10)	59 (17)	53 (27)	20 (16)	28 (44)	1 (5)	24 (39)
$E$ (TCA+)	KNN		71 (53)	37 (28)	64 (20)	40 (21)	36 (20)	65 (33)	2 (5)	23 (28)
$E$ (TCA+)	SVM		70 (55)	34 (31)	64 (20)	41 (24)	35 (21)	64 (40)	2 (5)	23 (32)
TCA+	LR		70 (39)	50 (22)	61 (22)	42 (21)	45 (22)	64 (28)	2 (6)	17 (35)
$E$ (TCA+)	NB		67 (56)	39 (38)	60 (20)	46 (24)	38 (24)	60 (44)	2 (6)	17 (32)
$E_{size}$ (Bellwether)	DT		67 (44)	30 (26)	65 (20)	39 (23)	33 (19)	63 (32)	1 (4)	25 (32)
$E$ (TCA+)	DT		67 (42)	37 (25)	63 (19)	41 (20)	38 (18)	62 (30)	2 (5)	20 (30)
$E$ (TCA+)	RF	1	67 (40)	34 (23)	65 (20)	38 (19)	35 (18)	65 (28)	2 (5)	24 (28)
Bellwether	NB		50 (60)	15 (21)	64 (22)	41 (32)	25 (16)	53 (55)	1 (5)	26 (38)
TCA+	SVM		50 (50)	23 (27)	61 (21)	45 (25)	31 (21)	52 (41)	2 (5)	20 (34)
TCA+	RF		50 (50)	31 (32)	57 (17)	48 (23)	36 (23)	49 (36)	2 (6)	13 (28)
TCA+	DT		50 (40)	34 (28)	56 (17)	49 (21)	39 (21)	48 (31)	2 (6)	11 (28)
TCA+	KNN		40 (42)	19 (22)	57 (18)	49 (24)	28 (21)	42 (36)	2 (5)	16 (32)
Bellwether	DT		33 (33)	13 (12)	59 (16)	48 (21)	24 (16)	38 (34)	1 (5)	19 (32)
TCA+	NB	0	23 (47)	19 (35)	50 (14)	61 (22)	34 (28)	23 (48)	3 (9)	0 (28)

## 6.8.2 Approach

The predictors are built using the state of the art methods, specifically *DODGE*, *Hyperopt*, and *TLEL* elucidated in §3.6.

So far, all RQ’s explored in this thesis in either *Within* or *Cross* contexts  $E_{size}$  has performed similar or better than prevalent sampling strategies, indicating the importance of early regions of the software project. Therefore in this experiment  $E_{size}$  based variants of *DODGE*, *Hyperopt* and *TLEL* using  $E_{size}$  are created. In other words, this experiment will input fewer early data  $E_{size}$  and use those complex methods to check if they perform better or worse.

Lastly, measure the predictive performance of the above approaches listed in §3.7.1 and rank the predictive performance of defect predictors that were built using the Scott-Knott test elucidated in §3.5.1.

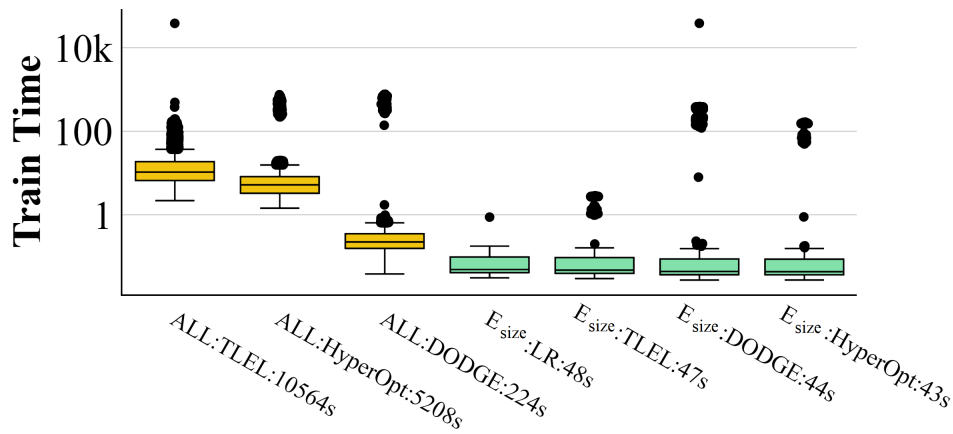
### 6.8.3 Results

The following observations are made from Table 6.8:

- $E_{size}$  (row #1) performs better than all complex methods that use either more data *ALL* or fewer early date  $E_{size}$ .
- Closest to  $E_{size}$  is the two-layer ensemble method that achieves almost similar results but with more data and more processing. But note *TLEL* (row #2) was trained on every new project release accumulating more data, whereas  $E_{size}$  or any  $E$  based method is always trained just ‘once.’

**Table 6.8** 7 *within-project* defect prediction models tested in all 240 project’s releases. In the first row, “+” and “-” denote the criteria that need to be maximized or minimized, respectively. All the other rows show combinations of sampling policies and classifiers. Green cells denote “early-data” methods (and all other rows employ optimizer or ensemble approach). Cells marked in gray all have the same top rank as the best results (and those ranks were determined by the Scott-Knott algorithm described in §3.5.1). The ‘wins’ column (see column #3) counts how often a particular sampling policy/classifier achieves a top score. Inter-quartile ranges are indicated withing ‘( )’.

Policy	Classifier	Wins	Recall+	PF-	AUC+	D2H-	Brier-	G-Score+	IFA-	MCC+
$E_{size}$	LR	7	73 (43)	28 (26)	70 (15)	34 (18)	30 (17)	69 (26)	1 (3)	35 (27)
ALL	TLEL		74 (50)	36 (56)	61 (20)	46 (31)	36 (31)	54 (46)	1 (4)	21 (38)
$E_{size}$	TLEL	2	67 (38)	33 (30)	64 (17)	39 (19)	34 (19)	62 (27)	1 (4)	26 (30)
$E_{size}$	HyperOpt		27 (60)	11 (31)	54 (14)	56 (28)	28 (22)	28 (55)	2 (11)	10 (28)
$E_{size}$	DODGE	1	100 (80)	68 (91)	50 (7)	71 (17)	49 (46)	0 (42)	3 (9)	0 (18)
ALL	DODGE		100 (80)	98 (69)	50 (0)	71 (3)	66 (36)	0 (17)	4 (9)	0 (0)
ALL	HyperOpt	0	50 (89)	40 (66)	50 (9)	63 (21)	44 (30)	27 (53)	3 (9)	0 (21)



**Figure 6.8** Time taken by different sampling policies to train a model based on the number of commits.

Notably, Figure 6.8 confirms it is far faster to build predictors sampled using  $E_{size}$  that neither needs

SMOTE or CFS and only trained once throughout the project life-cycle.

*In terms of improving defect prediction, simple early life cycle sampling does better than using optimizers.*

## 6.9 Threats to Validity

### 6.9.1 Sampling Bias

Generalizability of the conclusions will rely on the examples considered; i.e., what is essential here may not be genuine all over. Although the prevalent practice of such empirical studies is to use popular OS GitHub projects, broaden the scope by including unpopular projects. Nevertheless, all these projects are non-trivial engineering projects developed in numerous programming languages for various domains. Notably, lessons did not vary in either of these two populations of projects.

### 6.9.2 Construct Validity

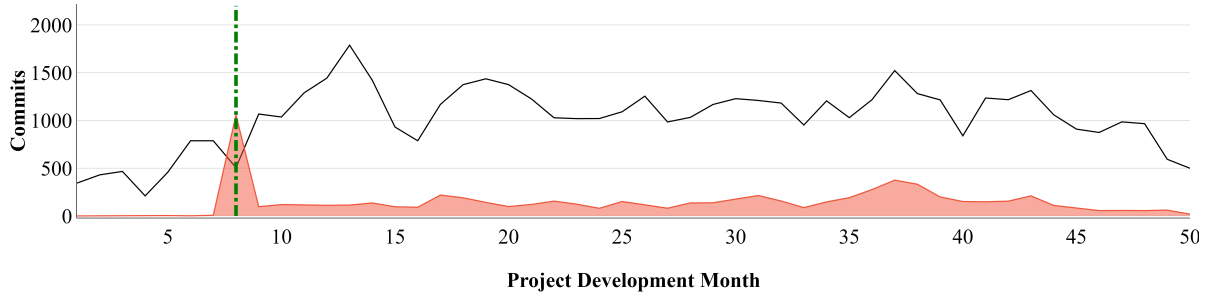
Project samples using *Commit-Guru* are used in the experiments. *Commit-Guru* determines risky Commits using a method similar to SZZ [Š05]. Such a prevalent approach is widely debated for its false positives. To minimize that threat *Commit-guru* only considers files that are source-code files and not documents like readme or pdf files. *Commit-Guru* classifies commit into different categories one of them is *merge-commit*. Studies have shown merge commits have little to no influence over making real changes to the software; therefore, in this study, all merge commits are filtered.

Lastly, to further bold, the validity of early effect is cross-checked with the two projects ‘QT,’ and ‘OPENSTACK’ used in the 2017 TSE article by McIntosh and Kamei [McI17] and found the early trend and its effect to hold. Please see the early trend of those two projects in Figure 6.9 and the results in Table 6.9 that show results of defect predictors that sampled training commits using recent six months (M6) endorsed in [McI17] compared to  $E_{size}$  endorsed in this chapter. Note that the results are compared with ten classifiers and eight evaluation measures.

### 6.9.3 Learner bias

For identifying bellwethers by all-pairs experiment (each project is tested on all 12,000+ releases), used only one classifier, ‘Logistic Regression.’ Perhaps other classifiers may qualify other projects are bellwethers. Using just one classifier may not be a threat for the following two reasons. Firstly a Logistic-Regression is recommended in the baseline in the previous chapter and widely endorsed in classifier in defect prediction literature. Secondly, in all the results in §5.4 Logistic-Regression based predictors were in-par and better than five other classifiers explored in this chapter.

Further, an empirical study can only focus on a handful of representative classifiers. The classifiers (Logistic Regression, Nearest neighbor, Decision Tree, Random Forrest, and Naïve Bayes) used in this thesis cover a broad range of classification algorithms [Gho15].



**Figure 6.9** A repeated pattern where most defects (shaded in red) occur early in the life cycle (before the green dotted line) is also observed among the two projects (QT and OPENSTACK systems) with 37,524 commits assessed recently by McIntosh and Kamei in [McI17].

**Table 6.9** ‘12’ *within-project* defect prediction models tested ‘only’ in two projects ‘QT’ and ‘OPENSTACK’. In the first row, “+” and “-” denote the criteria that need to be maximized or minimized, respectively. All the other rows show combinations of sampling policies and classifiers. Green cells denote “early-data” sampling was employed. Cells marked in gray all have the same top rank as the best results (and those ranks were determined by the Scott-Knott algorithm described in §3.5.1). The ‘wins’ column (see column #3) counts how often a particular sampling policy/classifier achieves a top score. Inter-quartile ranges are indicated withing ‘( )’.

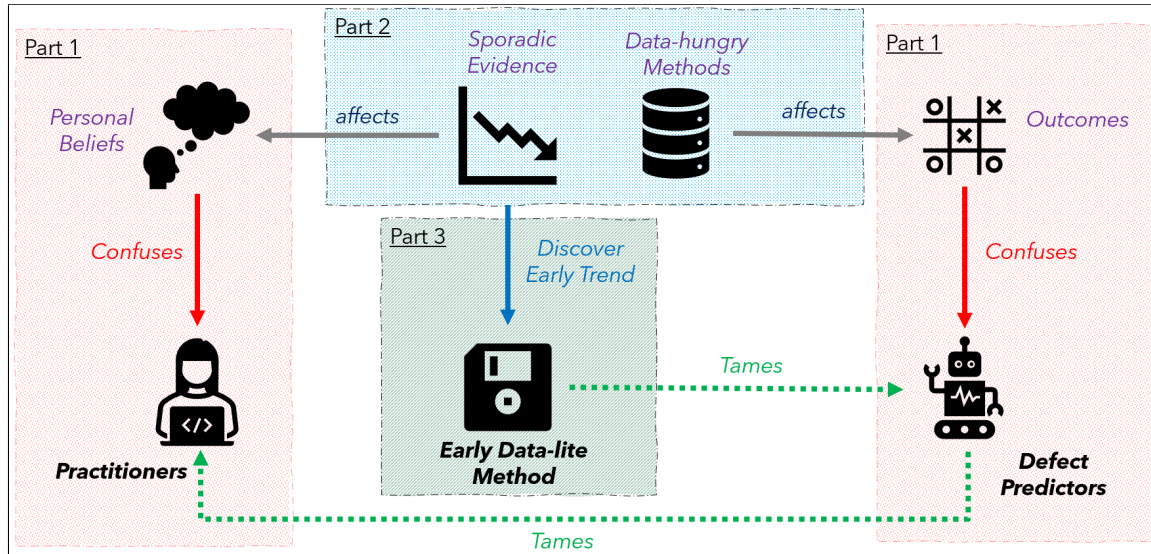
Policy	Classifier	Wins	Recall+	PF-	AUC+	D2H-	Brier-	G-Score+	IFA-	MCC+
<i>E<sub>size</sub></i>	RF	6	71 (13)	35 (8)	67 (7)	33 (6)	35 (8)	69 (10)	5 (7)	19 (9)
<i>E<sub>size</sub></i>	NB	5	75 (14)	43 (7)	67 (9)	35 (7)	42 (8)	71 (9)	6 (8)	17 (10)
<i>E<sub>size</sub></i>	SVM	4	67 (18)	33 (9)	67 (7)	34 (8)	33 (8)	66 (14)	4 (7)	18 (11)
<i>E<sub>size</sub></i>	LR		64 (16)	27 (7)	67 (8)	33 (8)	28 (6)	65 (12)	4 (8)	21 (11)
<i>M6</i>	SVM	3	14 (25)	6 (5)	54 (10)	61 (17)	13 (7)	17 (28)	3 (8)	9 (16)
<i>M6</i>	RF		13 (19)	4 (5)	55 (8)	61 (13)	9 (10)	16 (22)	3 (8)	11 (17)
<i>M6</i>	KNN	2	20 (13)	8 (6)	56 (7)	56 (9)	13 (8)	24 (14)	3 (7)	11 (13)
<i>E<sub>size</sub></i>	DT	1	67 (18)	48 (6)	59 (7)	42 (6)	47 (5)	63 (11)	7 (9)	8 (10)
<i>E<sub>size</sub></i>	KNN		65 (17)	35 (18)	65 (7)	36 (7)	35 (16)	65 (13)	5 (9)	16 (10)
<i>M6</i>	DT		25 (21)	12 (12)	55 (7)	54 (12)	15 (14)	28 (22)	4 (11)	8 (11)
<i>M6</i>	NB		14 (22)	9 (12)	52 (8)	62 (12)	17 (11)	17 (21)	6 (13)	5 (13)
<i>M6</i>	LR	0	10 (28)	10 (13)	51 (6)	64 (17)	17 (12)	12 (28)	9 (52)	1 (10)

### 6.9.4 Evaluation bias

This chapter uses both ‘Recall’ and ‘PF’ to identify bellwethers and eight evaluation measures (Recall, PF, IFA, Brier, GM, D2H, MCC, and AUC) to compare the policies extensively. Other widely used measures in defect prediction are precision and f-measure. However, as mentioned earlier, those threshold dependant metrics have issues with unbalanced data [Men08].

### 6.9.5 Input Bias

All the proposed sampling policies randomly samples 50 commits from the first 150 commits of the project. Along these lines, it could be true that different executions could yield different results. However, this is not a threat, as the conclusions hold on a large sample size of 12,000+ releases.



**Figure 6.10** A landscape of this thesis, where the final piece (early data-lite method) of part-3 is placed.

## 6.10 Summary

Before researchers rush to reason across all available data or try complex methods, perhaps it is prudent to first check for simpler alternatives. Specifically, if the historical data has the most information in some small region, a model learned from that region would suffice for the rest of the project.

Here, we have compared the defect models built via early sampling to three sets of advanced methods, they are:

- The ensemble method recommended by Yang et al. [Yan17] at IST'17;
- The hyper-parameter optimization methods recommended by Agrawal et al. [Agr19] at TSE'19 and Bergstra et al. [Ber11] at NIPS'11;
- The transfer learning methods recommended by Krishna et al. [Kri18] at TSE'19, and Nam et al. [Nam13] at ICSE'13.

The early data-lite method worked better than the data-hungry ensemble and hyperparameter optimization methods. And in the remaining case, it improved both the transfer learning methods in terms of predictive performance and run-time complexity. Additionally, this chapter reports that it is possible to early data-lite models even among unpopular SE projects.

Based on this experience with hundreds of SE projects, it is doubtful that prior work on generalizing software engineering defect prediction models may have needlessly complicated an inherently simple process. Further, prior work focused on later or full life-cycle data needs to be revisited since their conclusions were drawn from relatively uninformative regions.

Accordingly, the landscape Figure 6.10 is updated to reflect the final results (part-3) endorsed in this thesis.

## CHAPTER

# 7

# CONCLUSIONS AND FUTURE WORK

## 7.1 Thesis Revisited

In the first part, this thesis performed two empirical studies. In the first study, this thesis reports the source of the disconnect between practitioners' perception of defect prediction metrics and empirical evidence. Then the analysis of that study found that the sporadicity of evidence prevalence is the leading cause of the disconnect between stated beliefs (e.g., the Wan et al. [Wan18a] paper) and observed evidence. Despite irregularities, this thesis offers a few beliefs that stood out B5 (Large Commits) and B10 (Owner contribution) with good prevalence in that large-scale study. Thus the results of this thesis encourage developers to update their beliefs when the evidence demands it. This thesis showed fluctuations of evidence in a new dimension (size of a release) in chapter 4. In the second study, through an extensive evaluation of five old SE beliefs (originated between 1969 - 1993) in a controlled environment, this thesis finds support for one belief titled "*Quality entails productivity.*" That implies on-time delivery is achieved with a quality-driven focus. Four other beliefs this thesis assessed are not supported; uncertainties in the results of those beliefs portrayed how practitioners with a narrow scope could misinterpret specific effects to hold in their work.

In the second part, this thesis investigated the sporadicity of evidence within each of the project life-cycle. Because when data keep changing, the models this thesis can learn from that data may also vary. If conclusions become too fluid (i.e., change too often), no one has a stable basis for making decisions or communicating insights. Issues with conclusion instability disappear if this thesis can learn a predictive model that is effective for the rest of the project early in the life cycle. This thesis proposed a methodology for assessing such early life cycle predictors. On investigating, this thesis discovered a startling trend that most defects in a sample of 155 long-running software engineering projects were

recorded much earlier in the project life-cycle. This critical trend enabled building early data-lite defect prediction models that last the entire project life-cycle without retraining.

Lastly, in the third part of this thesis, the validity of the early data-lite method devised in chapter 5 is further explored. That method was only scoped in the context of within-project defect predictors and popular GitHub projects. In this chapter, this thesis checked the scope to show the efficacy of “early sampling” and highlighted that this thesis could simplify software analytics to a great extent. By showing:

- Early predictive models can be built even among unpopular GitHub projects.
- When early project data is transferred, they produce accurate and faster predictive models
- CPU intensive methods such as Hyper-parameter-optimization or non-explainable ensemble method like TLEL may not be required since early models with two features sufficed.

## 7.2 Discussion

### 7.2.1 Management Implications

The results from this thesis suggest,

- Yes, this thesis should study working software engineers to learn a list of effects that might damage software quality;
- But no, do not assume that all those effects hold at all times over the current project.

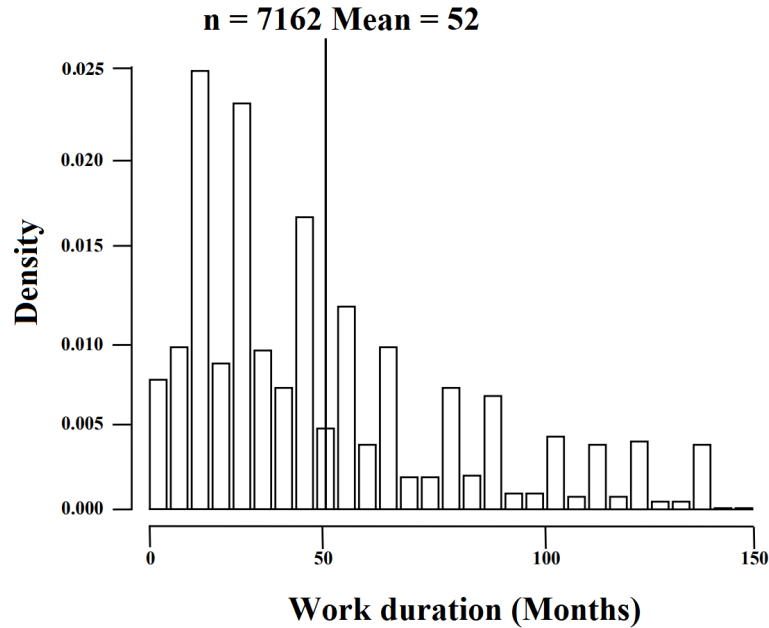
More specifically, managers and project leads should not “bet” on a small number of effects when setting policies for software projects. Rather, they should:

- Perpetually monitor for the presence or absence of a range of effects (such as those listed in Table 2.1);
- Perpetually adjust their code review and code refactoring processes such that they learn (a) not to trigger on old effects that now no longer hold or (b) trigger on new effects that have just appeared.

Much prior research has struggled to find stable conclusions that hold across multiple projects. Menzies et al. warn that many global lessons (generalizations) are not supported locally [Men11]. That is to say, the more data this thesis see, the more exceptions and special cases might appear in the models learned from that data. Such conclusion instability in SE has been often recorded in [Zim09; Men11; Shr20a].

Is that the best this thesis can do? Ideally, SE research can offer stable general defect prediction principles (such as those seen above) to guide project management, software standards, education, tool development, and legislation about software. As discussed in Chapter 2 Such conclusion stability would have benefits for *trust*, *insight*, *training*, and *tool development*.

All these problems with trust, insight, training, and tool development can be solved if, early on in the project, a defect prediction model can be learned that is effective for the rest of the life cycle (which is the main result of this thesis). Within that data, the results of the analysis show that models learned after just 150 commits, perform just as well as anything else. In terms of resolving conclusion instability, this is a very significant result since it means that for much of the life cycle, this thesis can offer stable defect predictors.



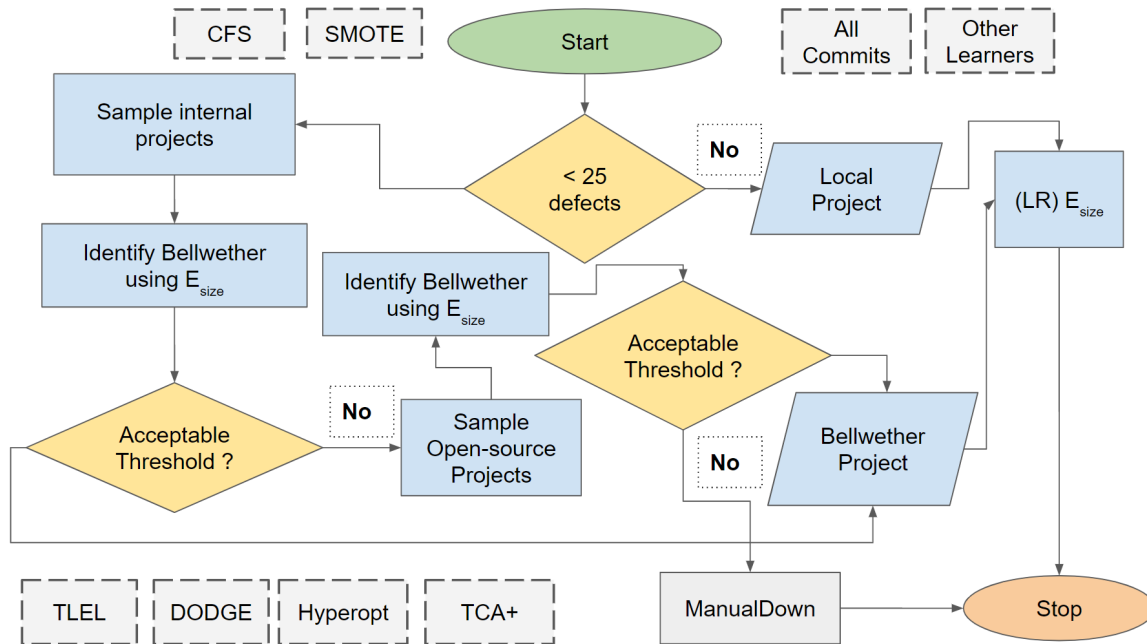
**Figure 7.1** Work duration histograms on particular projects; from [Sel18]. Data from: Facebook, eBay, Apple, 3M, Intel and Motorola.

One way to consider the impact of such early life cycle predictors is to use the data of Figure 7.1. That plot shows that software employees usually change projects every 52 months (either moving between companies or changing projects within an organization). According to Figure 7.1, in seven years (84 months), the majority of workers and managers would first appear on a job *after* the initial four months are required to learn a defect predictor. Hence, for most workers and managers, the detectors learned via the methods of this thesis would be the “established wisdom” and “the way we do things here” for their projects. This means that a detector learned in the first four months would be a suitable oracle to guide training and hiring; the development of code review practices; the automation of local “bad smell detectors”; as well as tool selection and development.

In summary, this thesis created a simple decision tree in Figure 7.2 that would help practitioners to choose the appropriate method to build a defect predictor (endorsed as per the results) given a specific scenario. That figure endorses practitioners to build predictors using the “early data-lite” method  $E_{size}$  in both *Within* and *Cross* scenarios. However, in the worst scenario, when there is no access to data or the inability to find good bellwether projects, this thesis suggests practitioners use *ManualDown* until their local or neighboring projects report more defects ( $> 24$ ). Note: Many complex methods and late-data explored in this study are unused in creating this decision tree.

## 7.2.2 Systems Implications

Figure 6.6 shows the run-time required to find bellwether projects in all 240 projects using *Bellwether* or  $E_{size}(Bellwether)$  policies. Note that the early life cycle methods use 2.5 months less CPU than the



**Figure 7.2** A decision-tree to build defect predictors drawn using the results from §5.4 that find many complex methods (shown as disconnected blocks with dashed borders) to be needless.

alternative.

Since these results showed that only the first 150 commits are needed to identify a suitable project, many projects in their early stages may be included in the pool of transferable projects. Importantly Figure 6.5 show when sampled using  $E_{size}$  30% “more projects” were qualified in much “less time”. The choice of bellwether does not matter; perhaps, in practice, practitioners need not continue to look for bellwethers and terminate early as soon as they find the first bellwether. This could also mean lesser privacy concerns and enable practitioners to share less data freely. The early-data nature of the methods can also help to gauge sophisticated techniques like TCA+ faster.

Note: This thesis does not endorse  $E_{size}(Bellwether)$  is data-lite because it would still process all the projects in the search space. But this thesis do endorse  $E_{size}$  over current methods *Bellwether* or *TCA+*. Because  $E_{size}$  is faster and accurate (see §5.5), as it uses fixed data (150 commits) and does not re-train for every new project release.

### 7.2.3 Research Implications

This thesis has shown that when defect data contains information, that information may be densest is a small part of the historical record of a project. While this thesis has *not* shown that other kinds of SE data have the same density effect, this thesis would argue that now it is at least an open question that “have this thesis been learning from the wrong parts of the data?”.

Finally, this thesis should now view it as a potential methodological error to reason across all data in a project. In the specific case of defect prediction this thesis must now revisit any conclusion based just

on later life cycle data There are many examples of such conclusions. For example:

- Hoang et al. says “We assume that older commits changes may have characteristics that no longer effects to the latest commits” [Hoa19].
- Also, it is common practice in defect prediction to perform “recent validation” where predictors are tested on the latest release after training from the prior one or two releases [Tan15; McI17; Kon20; Fu16a].

More generally, before researchers focus on later life cycle data, they must first check that their (e.g.) buggy commit data to occur at equal frequency across the life cycle. If the buggy commits data is isolated to a specific region, then predictors should be built from that region. This work finds that most buggy commits are found early in the project life-cycle.

## 7.3 Future Work

As for future work, this thesis has many suggestions. Here are a few directions.

Logically, many SE domains would benefit from early data because almost all SE activities occur during the project life-cycle. Therefore, SE research areas like log-analysis, software reuse, etc., that predominantly rely on abundant project data should perhaps be rechecked with data only from information-rich regions.

### 7.3.1 Log Analysis

In the space of software reliability engineering, log-analysis is a technique to find useful patterns in large volumes of unstructured text generated by software systems in real-time. Log-analysis has numerous challenges, like the quality of the logs written by developers to the maintenance of the logs [He21]. One key challenge is handling voluminous data from real-time software systems. Prevalent search engines such as Splunk [Car12] and elastic-search [Gor15] have capabilities to ingest and analyze stream of logs. But the cost of extracting important insights like predicting threats or defects from logs can be high (budget and time). Over time, much of the logs generated by software systems are redundant. The search space is needlessly large for the proprietary log-management systems such as Splunk to process. Perhaps by only focusing on the early project regions similar to the early-date-lite method elucidated in chapter 6, one could search for fruitful patterns in a smaller search-space, saving cost and time.

### 7.3.2 Code Reuse

Hindle et al. conjectured that code created by humans is natural and therefore likely to be repetitive and predictable [Hin12]. As the project matures over time, most of the code written by software developers could be redundant. For new features (or requirements) in the project, software developers may have to write new code that may have already accumulated in their project. But often, developers, especially newbies, may rely on external sources such as StackOverflow [Sta] or repositories like GitHub [Cit] to write new software. However, code from external sources may be defective, have licensing issues, and are not readily useable (need refactoring). Therefore, this thesis suggests testing the following conjecture.

If most of the required code is written in the first few months of the project, then perhaps much of the code can be synthesized pragmatically by ingesting early code within the project.

### 7.3.3 Software Fairness

There is currently high research traction to build fairer software. For example, recent work by Chakraborty et al. found that a small portion (10%) of labeled data is sufficient to generate pseudo-labels for the unlabeled data to tackle ethical bias in machine learning models [Cha21]. Since labeling involves considerable human effort and is also not bias-free, using fewer labeled data is very beneficial. While fewer labeled data may be perceived as data-lite, their framework uses the entire dataset to build fair software. Therefore the framework may be tested in data-lite information-rich regions to yield better performance and more straightforward explanations.

### 7.3.4 Other domains:

In a broad sense, the problems discussed in this thesis fall under the realm of better sampling for software analytics. Therefore, it would benefit domains where data is scarce, expensive to collect, or sensitive (privacy concerns) like medicine. For example, one could check the list of UCI Machine Learning Repository datasets [Dua17] that span multiple domains for the early data-lite effect explored in this thesis.

And many more avenues to test for the early data-lite efficacy such as,

- Can we estimate software effort realistically with early project data?
- Can we prioritize test cases with early project activities?
- Can we build recommend-er systems by looking only at early project activities?
- Can artificial neural networks learn faster if the pre-processors can truncate the data to a fruitful region?

## 7.4 Epilogue

From the above, this thesis can be summarized as:

*A repeated result in software engineering when reasoning about complex things, often less is better*

For decades practitioners and researchers have been reasoning across the entire project life-cycle using very complex methods. Those practitioners and researchers might defend their approach by saying that more data and complex techniques are inherently better for software analytics. However, these presumptions confuse software teams, especially when practitioners have contradictory beliefs, and oracles prescribed by researchers yield unstable conclusions as they are frequently updated with more data. This thesis questioned current methods and portrayed many contradictions among practitioner beliefs to find a startling trend within software project life-cycle. That trend shows much of the knowledge (high-activity) happens during the first few months of the project. Therefore, software analytics can be improved by focusing on the early project life cycle region without using expensive methods, more data, or re-training oracles.

Specifically, the results of this thesis find that a small early life-cycle sample can replace some methods and usefully augment others. For within project defect prediction, when trying to improve predictive performance, the extra CPU required for (a) ensemble and (b) hyper-parameter optimization is not needed since sampling from early life-cycle data performs better than both (a) or (b). As to transfer learning in cross-company learning, this thesis recommends that a small early life cycle sample on cross-project learning is significantly better than using the entire cross-project life-cycle data.

## BIBLIOGRAPHY

- [Agr18a] Agrawal, A. & Menzies, T. “Is "" Better Data"" Better Than"" Better Data Miners""?” *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE. 2018, pp. 1050–1061.
- [Agr18b] Agrawal, A. et al. “We don’t need another hero?: the impact of heroes on software development”. *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*. ACM. 2018, pp. 245–253.
- [Agr18c] Agrawal, A. et al. “What is wrong with topic modeling? And how to fix it using search-based software engineering”. *Information and Software Technology* **98** (2018), pp. 74–88.
- [Agr19] Agrawal, A. et al. “How to " DODGE" Complex Software Analytics”. *IEEE Transactions on Software Engineering* (2019).
- [Ama20] Amasaki, S. “Cross-version defect prediction: use historical data, cross-project data, or both?” *Empirical Software Engineering* (2020), pp. 1–23.
- [Aro20] Arokiam, J. & Bradbury, J. S. “Automatically predicting bug severity early in the development process”. *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: New Ideas and Emerging Results*. 2020, pp. 17–20.
- [Bal18] Baltes, S. & Diehl, S. “Towards a theory of software development expertise”. *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM. 2018, pp. 187–200.
- [Ben19] Bennin, K. E. et al. “On the relative value of data resampling approaches for software defect prediction”. *Empirical Software Engineering* **24.2** (2019), pp. 602–636.
- [Ber11] Bergstra, J. et al. “Algorithms for Hyper-parameter Optimization”. *Proceedings of the 24th International Conference on Neural Information Processing Systems*. NIPS’11. Granada, Spain: Curran Associates Inc., 2011, pp. 2546–2554.
- [Ber13] Bergstra, J. et al. “Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures”. *International conference on machine learning*. PMLR. 2013, pp. 115–123.
- [Bet12] Bettenburg, N. et al. “Think locally, act globally: Improving defect and effort prediction models”. *Mining Software Repositories (MSR), 2012 9th IEEE Working Conference on*. IEEE. 2012, pp. 60–69.
- [Bha11] Bhattacharya, P. & Neamtiu, I. “Assessing programming language impact on development and maintenance: A study on C and C++”. *Proceedings of the 33rd International Conference on Software Engineering*. ACM. 2011, pp. 171–180.
- [Bir11] Bird, C. et al. “Don’t touch my code!: examining the effects of ownership on software quality”. *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM. 2011, pp. 4–14.

- [Bir15] Bird, C. et al. *The Art and Science of Analyzing Software Data*. 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2015.
- [Bis13] Bissyandé, T. F. et al. “Popularity, interoperability, and impact of programming languages in 100,000 open source projects”. *2013 IEEE 37th Annual Computer Software and Applications Conference*. IEEE. 2013, pp. 303–312.
- [Bow18] Bowes, D. et al. “Software defect prediction: do different classifiers find the same defects?”. *Software Quality Journal* **26.2** (2018), pp. 525–552.
- [Bri02] Briand, L. C. et al. “Assessing the applicability of fault-proneness models across object-oriented software projects”. *IEEE transactions on Software Engineering* **28.7** (2002), pp. 706–720.
- [Bro87] Brooks, F & Kugler, H. *No silver bullet*. April, 1987.
- [BJ95] Brooks Jr, F. P. et al. *The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition, 2/E*. Pearson Education India, 1995.
- [Cam79] Campbell, D. T. & Cook, T. D. *Quasi-experimentation: Design & analysis issues for field settings*. Rand McNally College Publishing Company Chicago, 1979.
- [Can13] Canfora, G. et al. “Multi-objective cross-project defect prediction”. *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. IEEE. 2013, pp. 252–261.
- [Can15] Canfora, G. et al. “Defect prediction as a multiobjective optimization problem”. *Software Testing, Verification and Reliability* **25.4** (2015), pp. 426–459.
- [Car12] Carasso, D. *Exploring splunk*. CITO Research New York, USA, 2012.
- [Cha21] Chakraborty, J. et al. “Can We Achieve Fairness Using Semi-Supervised Learning?” *arXiv preprint arXiv:2111.02038* (2021).
- [Cha02] Chawla, N. V. et al. “SMOTE: synthetic minority over-sampling technique”. *Journal of artificial intelligence research* **16** (2002), pp. 321–357.
- [Che20] Chekam, T. T. et al. “Selecting fault revealing mutants”. *Empirical Software Engineering* **25.1** (2020), pp. 434–487.
- [Che18a] Chen, D. et al. “Applications of psychological science for actionable analytics”. *FSE’19* (2018).
- [Che19a] Chen, J. et al. “Multiview transfer learning for software defect prediction”. *IEEE Access* **7** (2019), pp. 8901–8916.
- [Che15] Chen, L. et al. “Negative samples reduction in cross-company software defects prediction”. *Information and Software Technology* **62** (2015), pp. 67–77.
- [Che18b] Chen, X. et al. “MULTI: Multi-objective effort-aware just-in-time software defect prediction”. *Information and Software Technology* **93** (2018), pp. 1–13.

- [Che19b] Chen, X. et al. “Software defect number prediction: Unsupervised vs supervised methods”. *Information and Software Technology* **106** (2019), pp. 161–181.
- [Cob90] Cobb, R. H. & Mills, H. D. “Engineering software under statistical quality control”. *IEEE Software* **7.6** (1990), pp. 45–54.
- [Cor69] Corbato, F. J. “PL/I as a Tool for System Programming”. *Datamation* **15.5** (1969), p. 68.
- [Dah01] Dahl, O.-J. & Nygaard, K. “Class and subclass declarations”. *Pioneers and Their Contributions to Software Engineering*. Springer, 2001, pp. 235–253.
- [Dam18] Dam, H. K. et al. “A deep tree-based model for software defect prediction”. *arXiv preprint arXiv:1802.00921* (2018).
- [Dam19] Dam, H. K. et al. “Lessons learned from using a deep tree-based model for software defect prediction in practice”. *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE. 2019, pp. 46–57.
- [D’A10] D’Ambros, M. et al. “An extensive comparison of bug prediction approaches”. *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*. IEEE. 2010, pp. 31–41.
- [Dev96] Devanbu, P. et al. “Analytical and empirical evaluation of software reuse metrics”. *Proceedings of IEEE 18th International Conference on Software Engineering*. IEEE. 1996, pp. 189–199.
- [Dev16] Devanbu, P. et al. “Belief & evidence in empirical software engineering”. *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE. 2016, pp. 108–119.
- [Die17] Dieste, O. et al. “Empirical evaluation of the effects of experience on code quality and programmer productivity: an exploratory study”. *Empirical Software Engineering* **22.5** (2017), pp. 2457–2542.
- [Dua17] Dua, D. & Graff, C. *UCI Machine Learning Repository*. 2017.
- [D2] D’Ambros, M. et al. “Evaluating defect prediction approaches: a benchmark and an extensive comparison”. *Empirical Software Engineering* **17.4-5** (2012), pp. 531–577.
- [End03] Endres, A. & Rombach, H. D. *A handbook of software and systems engineering: Empirical observations, laws, and theories*. Pearson Education, 2003.
- [Eri04] Ericsson, K. A. “Deliberate practice and the acquisition and maintenance of expert performance in medicine and related domains”. *Academic medicine* **79.10** (2004), S70–S81.
- [Eri93] Ericsson, K. A. et al. “The role of deliberate practice in the acquisition of expert performance.” *Psychological review* **100.3** (1993), p. 363.
- [Fan19] Fan, Y. et al. “The Impact of Changes Mislabeled by SZZ on Just-in-Time Defect Prediction”. *IEEE Transactions on Software Engineering* (2019).

- [Fen08] Fenton, N. et al. “On the effectiveness of early life cycle defect prediction with Bayesian Nets”. *Empirical Software Engineering* **13.5** (2008), p. 499.
- [Fra03] Fraser, S. et al. “Discipline and Practices of TDD: (Test Driven Development)”. *Companion of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA '03. Anaheim, CA, USA: Association for Computing Machinery, 2003, 268–270.
- [Fu16a] Fu, W. et al. “Tuning for software analytics: Is it really necessary?” *Information and Software Technology* **76** (2016), pp. 135–146.
- [Fu16b] Fu, W. et al. “Why is differential evolution better than grid search for tuning defect predictors?” *arXiv preprint arXiv:1609.02613* (2016).
- [Fuc17] Fucci, D. et al. “A Dissection of the Test-Driven Development Process: Does It Really Matter to Test-First or to Test-Last?” *IEEE Transactions on Software Engineering* **43.7** (2017), pp. 597–614.
- [Fuk14] Fukushima, T. et al. “An empirical study of just-in-time defect prediction using cross-project models”. *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM. 2014, pp. 172–181.
- [Gho15] Ghotra, B. et al. “Revisiting the impact of classification techniques on the performance of defect prediction models”. *37th ICSE-Volume 1*. IEEE Press. 2015, pp. 789–800.
- [Cit] *GitHub Inc - provides hosting for software development version control using Git*. <https://github.com/>. Accessed: 2019-03-18.
- [Gol83] Goldberg, A. & Robson, D. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., 1983.
- [Gor15] Gormley, C. & Tong, Z. *Elasticsearch: the definitive guide: a distributed real-time search and analytics engine*. " O'Reilly Media, Inc.", 2015.
- [Gra00] Graves, T. L. et al. “Predicting fault incidence using software change history”. *IEEE Transactions on software engineering* **26.7** (2000), pp. 653–661.
- [Gra13] Grazioli, F. “An Analysis of Student Performance During the Introdution of the PSP: An Empirical Cross Course Comparrison”. PhD thesis. Universidad de la Republica, 2013.
- [Hal03] Hall, M. A. & Holmes, G. “Benchmarking attribute selection techniques for discrete class data mining”. *IEEE Transactions on Knowledge and Data engineering* **15.6** (2003), pp. 1437–1447.
- [Has17] Hassan, A. *Remarks made during a presentation to the UCL Crest Open Workshop*. 2017.
- [Has05] Hassan, A. E. & Holt, R. C. “The top ten list: dynamic fault prediction”. *21st IEEE International Conference on Software Maintenance (ICSM'05)*. 2005, pp. 263–272.

- [Has09] Hassan, A. E. “Predicting faults using the complexity of code changes”. *Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society. 2009, pp. 78–88.
- [Hat08] Hattori, L. P. & Lanza, M. “On the nature of commits”. *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press. 2008, pp. III–63.
- [He09] He, H. & Garcia, E. A. “Learning from imbalanced data”. *IEEE Transactions on knowledge and data engineering* **21.9** (2009), pp. 1263–1284.
- [He15] He, P. et al. “An empirical study on software defect prediction with a simplified metric set”. *Information and Software Technology* **59** (2015), pp. 170–190.
- [He21] He, S. et al. “A survey on automated log analysis for reliability engineering”. *ACM Computing Surveys (CSUR)* **54.6** (2021), pp. 1–37.
- [He12] He, Z. et al. “An investigation on the feasibility of cross-project defect prediction”. *Automated Software Engineering* **19.2** (2012), pp. 167–199.
- [He13] He, Z. et al. “Learning from open-source projects: An empirical study on defect prediction”. *2013 ACM/IEEE international symposium on empirical software engineering and measurement*. IEEE. 2013, pp. 45–54.
- [Hej06] Hejlsberg, A. et al. *The C# programming language*. Adobe Press, 2006.
- [Her13] Herbold, S. “Training data selection for cross-project defect prediction”. *Proceedings of the 9th international conference on predictive models in software engineering*. 2013, pp. 1–10.
- [Hin08] Hindle, A. et al. “What do large commits tell us?: a taxonomical study of large commits”. *Proceedings of the 2008 international working conference on Mining software repositories*. ACM. 2008, pp. 99–108.
- [Hin12] Hindle, A. et al. “On the naturalness of software”. *2012 34th International Conference on Software Engineering (ICSE)*. IEEE. 2012, pp. 837–847.
- [Hoa19] Hoang, T. et al. “DeepJIT: An End-to-End Deep Learning Framework for Just-in-Time Defect Prediction”. *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. 2019, pp. 34–45.
- [Hos18] Hosseini, S. et al. “A benchmark study on the effectiveness of search-based data selection and feature selection for cross project defect prediction”. *Information and Software Technology* **95** (2018), pp. 296–312.
- [Hua17] Huang, Q. et al. “Supervised vs unsupervised models: A holistic look at effort-aware just-in-time defect prediction”. *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE. 2017, pp. 159–170.
- [Hua19] Huang, Q. et al. “Revisiting supervised and unsupervised models for effort-aware just-in-time defect prediction”. *Empirical Software Engineering* **24.5** (2019), pp. 2823–2862.

- [Hud18] Huda, S. et al. “An ensemble oversampling model for class imbalance problem in software defect prediction”. *IEEE access* **6** (2018), pp. 24184–24195.
- [Hum95] Humphrey, W. S. *A Discipline for Software Engineering*. Vol. 640. Reading, MA: Addison-Wesley Longman Publishing Co., Inc., 1995, p. 789.
- [Jia13] Jiang, T. et al. “Personalized defect prediction”. *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Ieee. 2013, pp. 279–289.
- [Jin16] Jing, X.-Y. et al. “An improved SDA based defect prediction framework for both within-project and cross-project class-imbalance problems”. *IEEE Transactions on Software Engineering* **43.4** (2016), pp. 321–339.
- [Jin15] Jing, X. et al. “Heterogeneous cross-company defect prediction by unified metric representation and CCA-based transfer learning”. *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM. 2015, pp. 496–507.
- [Joh99] Johnson, P. M. & Disney, A. M. “A Critical Analysis of PSP Data Quality: Results from aCase Study”. *Empirical Softw. Engg.* **4.4** (1999), pp. 317–349.
- [Jør09] Jørgensen, M. & Gruschke, T. M. “The Impact of Lessons-Learned Sessions on Effort Estimation and Uncertainty Assessments”. *Software Engineering, IEEE Transactions on* **35.3** (2009), pp. 368–383.
- [Kam12] Kamei, Y. et al. “A large-scale empirical study of just-in-time quality assurance”. *IEEE Transactions on Software Engineering* **39.6** (2012), pp. 757–773.
- [Kam16] Kamei, Y. et al. “Studying just-in-time defect prediction using cross-project models”. *Empirical Software Engineering* **21.5** (2016), pp. 2072–2106.
- [Ker06] Kersten, M. & Murphy, G. C. “Using task context to improve programmer productivity”. *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*. 2006, pp. 1–11.
- [Kim11] Kim, M. et al. “An empirical investigation into the role of API-level refactorings during software evolution”. *Proceedings of the 33rd International Conference on Software Engineering*. ACM. 2011, pp. 151–160.
- [Kim16] Kim, M. et al. “The Emerging Role of Data Scientists on Software Development Teams”. *Proceedings of the 38th International Conference on Software Engineering*. ICSE '16. Austin, Texas: ACM, 2016, pp. 96–107.
- [Koc15] Kocaguneli, E. et al. “Transfer learning in effort estimation”. *Empirical Software Engineering* **20.3** (2015), pp. 813–843.
- [Koc16] Kochhar, P. S. et al. “A large scale study of multiple programming languages and code quality”. *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. Vol. 1. IEEE. 2016, pp. 563–573.

- [Kon19] Kondo, M. et al. “The impact of feature reduction techniques on defect prediction models”. *Empirical Software Engineering* **24.4** (2019), pp. 1925–1963.
- [Kon20] Kondo, M. et al. “The impact of context metrics on just-in-time defect prediction”. *Empirical Software Engineering* **25.1** (2020), pp. 890–939.
- [Kor08a] Koru, A. G. et al. “An investigation into the functional form of the size-defect relationship for software modules”. *IEEE Transactions on Software Engineering* **35.2** (2008), pp. 293–304.
- [Kor08b] Koru, A. G. et al. “Theory of relative defect proneness”. *Empirical Software Engineering* **13.5** (2008), pp. 473–498.
- [Kri18] Krishna, R. & Menzies, T. “Bellwethers: A baseline method for transfer learning”. *IEEE Transactions on Software Engineering* (2018).
- [Kri16] Krishna, R. et al. “Too much automation? The bellwether effect and its implications for transfer learning”. *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. 2016, pp. 122–131.
- [Lam17] Lam, A. N. et al. “Bug localization with combination of deep learning and information retrieval”. *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*. IEEE. 2017, pp. 218–229.
- [Lar15] Laradji, I. H. et al. “Software defect prediction using ensemble learning on selected features”. *Information and Software Technology* **58** (2015), pp. 388–402.
- [LaT20] LaToza, T. D. et al. “Explicit programming strategies”. *Empirical Software Engineering* (2020), pp. 1–34.
- [Li17a] Li, J. et al. “Software defect prediction via convolutional neural network”. *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. IEEE. 2017, pp. 318–328.
- [Li12] Li, M. et al. “Sample-based software defect prediction with active and semi-supervised learning”. *Automated Software Engineering* **19.2** (2012), pp. 201–230.
- [Li17b] Li, Y. et al. “An Empirical Study to Revisit Productivity across Different Programming Languages”. *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE. 2017, pp. 526–533.
- [Li18] Li, Z. et al. “Cost-sensitive transfer kernel canonical correlation analysis for heterogeneous defect prediction”. *Automated Software Engineering* **25.2** (2018), pp. 201–245.
- [Liu19] Liu, C. et al. “A two-phase transfer learning model for cross-project defect prediction”. *Information and Software Technology* **107** (2019), pp. 125–136.
- [Lo15] Lo, D. et al. “How practitioners perceive the relevance of software engineering research”. *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM. 2015, pp. 415–425.

- [Lu14] Lu, H. et al. “Defect Prediction between Software Versions with Active Learning and Dimensionality Reduction”. *2014 IEEE 25th International Symposium on Software Reliability Engineering*. 2014, pp. 312–322.
- [Ma12] Ma, Y. et al. “Transfer learning for cross-company software defect prediction”. *Information and Software Technology* **54.3** (2012), pp. 248–256.
- [Män15] Mäntylä, M. V. et al. “On rapid releases and software testing: a case study and a semi-systematic literature review”. *Empirical Software Engineering* **20.5** (2015), pp. 1384–1425.
- [Mat18] Mathew, G. et al. “Finding trends in software research”. *IEEE Transactions on Software Engineering* (2018).
- [Mat10] Matsumoto, S. et al. “An analysis of developer metrics for fault prediction”. *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*. ACM. 2010, p. 18.
- [McI17] McIntosh, S. & Kamei, Y. “Are fix-inducing changes a moving target? a longitudinal case study of just-in-time defect prediction”. *IEEE Transactions on Software Engineering* **44.5** (2017), pp. 412–428.
- [Men06] Menzies, T. et al. “Data mining static code attributes to learn defect predictors”. *IEEE transactions on software engineering* **33.1** (2006), pp. 2–13.
- [Men08] Menzies, T. et al. “Implications of ceiling effects in defect predictors”. *Proceedings of the 4th international workshop on Predictor models in software engineering*. 2008, pp. 47–54.
- [Men10] Menzies, T. et al. “Defect prediction from static code features: current results, limitations, new approaches”. *Automated Software Engineering* **17.4** (2010), pp. 375–407.
- [Men11] Menzies, T. et al. “Local vs. global models for effort estimation and defect prediction”. *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. IEEE. 2011, pp. 343–351.
- [Men12] Menzies, T. et al. “Local versus global lessons for defect prediction and effort estimation”. *IEEE Transactions on software engineering* **39.6** (2012), pp. 822–834.
- [Men17] Menzies, T. et al. “Are delayed issues harder to resolve? Revisiting cost-to-fix of defects throughout the lifecycle”. *Empirical Software Engineering* **22.4** (2017), pp. 1903–1935.
- [Mil93] Mills, H. D. “Cleanroom engineering”. *Advances in Computers* **36** (1993), p. 1.
- [Mil83] Mills, H. “Software productivity in the enterprise”. *Software Productivity*. Little, Brown, 1983, pp. 265–270.
- [Mis11] Misirli, A. T. et al. “Ai-based software defect predictors: Applications and benefits in a case study”. *AI Magazine* **32.2** (2011), pp. 57–68.
- [Mit13] Mittas, N. & Angelis, L. “Ranking and Clustering Software Cost Estimation Models through a Multiple Comparisons Algorithm”. *IEEE Trans SE* **39.4** (2013), pp. 537–551.

- [Mon17] Monden, A. et al. “Examining software engineering beliefs about system testing defects”. *It Professional* **19.2** (2017), pp. 58–64.
- [Mun17] Munaiah, N. et al. “Curating GitHub for engineered software projects”. *Empirical Software Engineering* **22.6** (2017), pp. 3219–3253.
- [MH19] Murphy-Hill, E. et al. “What Predicts Software Developers’ Productivity?” *IEEE Transactions on Software Engineering* (2019).
- [Nag15] Nagappan, M. et al. “An empirical study of goto in C code from GitHub repositories”. *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM. 2015, pp. 404–414.
- [Nag05] Nagappan, N. & Ball, T. “Use of relative code churn measures to predict system defect density”. *Proceedings of the 27th international conference on Software engineering*. ACM. 2005, pp. 284–292.
- [Nam13] Nam, J. et al. “Transfer defect learning”. *2013 35th international conference on software engineering (ICSE)*. IEEE. 2013, pp. 382–391.
- [Nam17] Nam, J. et al. “Heterogeneous defect prediction”. *IEEE Transactions on Software Engineering* **44.9** (2017), pp. 874–896.
- [Ngu11] Nguyen, V. et al. “An analysis of trends in productivity and cost drivers over years”. *Proceedings of the 7th International Conference on Predictive Models in Software Engineering*. 2011, pp. 1–10.
- [Ni17] Ni, C. et al. “A cluster based feature selection method for cross-project software defect prediction”. *Journal of Computer Science and Technology* **32.6** (2017), pp. 1090–1107.
- [Nic19] Nichols, W. R. “The End to the Myth of Individual Programmer Productivity”. *IEEE Software* **36.5** (2019), pp. 71–75.
- [Nor93] Norman, D. A. *Things That Make Us Smart: Defending Human Attributes in the Age of the Machine*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1993.
- [Nor11] Norvig, P. *The Unreasonable Effectiveness of Data*. Youtube. 2011. url: <https://www.youtube.com/watch?v=yvDCzhhbjYWs>.
- [Ost05] Ostrand, T. J. et al. “Predicting the location and number of faults in large software systems”. *IEEE Transactions on Software Engineering* **31.4** (2005), pp. 340–355.
- [Özt17] Öztürk, M. M. “Which type of metrics are useful to deal with class imbalance in software defect prediction?” *Information and Software Technology* **92** (2017), pp. 17–29.
- [Pan10] Pan, S. J. et al. “Domain adaptation via transfer component analysis”. *IEEE Transactions on Neural Networks* **22.2** (2010), pp. 199–210.

- [Pan20] Pandey, S. K. et al. “BPDET: An effective software bug prediction model using deep representation and ensemble learning techniques”. *Expert Systems with Applications* **144** (2020), p. 113085.
- [Pan14] Panichella, A. et al. “Cross-project defect prediction models: L’union fait la force”. *2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*. IEEE. 2014, pp. 164–173.
- [Par11] Parnin, C. & Orso, A. “Are automated debugging techniques actually helping programmers?” *Proceedings of the 2011 international symposium on software testing and analysis*. ACM. 2011, pp. 199–209.
- [Pas18] Pascarella, L. et al. “Re-evaluating method-level bug prediction”. *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 2018, pp. 592–601.
- [Pas19] Pascarella, L. et al. “Fine-grained just-in-time defect prediction”. *Journal of Systems and Software* **150** (2019), pp. 22–36.
- [Pas11] Passos, C. et al. “Analyzing the Impact of Beliefs in Software Project Practices”. *ESEM’11*. 2011.
- [Pau06] Paulk, M. C. “Factors affecting personal software quality” (2006).
- [Pau10] Paulk, M. C. “The impact of process discipline on personal software quality and productivity”. *Software Quality Professional* **12.2** (2010), p. 15.
- [Pau05] Paulk, M. C. “An empirical study of process discipline and software quality”. PhD thesis. University of Pittsburgh, 2005.
- [Ped11] Pedregosa, F. et al. “Scikit-learn: Machine Learning in Python”. *Journal of Machine Learning Research* **12** (2011), pp. 2825–2830.
- [Pet13a] Peters, F. et al. “Balancing privacy and utility in cross-company defect prediction”. *IEEE Transactions on Software Engineering* **39.8** (2013), pp. 1054–1068.
- [Pet13b] Peters, F. et al. “Better cross company defect prediction”. *2013 10th Working Conference on Mining Software Repositories (MSR)*. IEEE. 2013, pp. 409–418.
- [Pet15] Peters, F. et al. “LACE2: Better privacy-preserving data sharing for cross project defect prediction”. *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press. 2015, pp. 801–811.
- [Pla02] Planning, S. “The economic impacts of inadequate infrastructure for software testing”. *National Institute of Standards and Technology* (2002).
- [Pop99] Pope, G. U. et al. *Sacred Kural Of Tiruvalluva Nayanar*. Asian Educational Services, 1999.
- [Rah13a] Rahman, F. & Devanbu, P. “How, and why, process metrics are better”. *2013 35th International Conference on Software Engineering (ICSE)*. IEEE. 2013, pp. 432–441.

- [Rah12] Rahman, F. et al. "Recalling the "" imprecision"" of cross-project defect prediction". *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. 2012, pp. 1–11.
- [Rah13b] Rahman, F. et al. "Sample size vs. bias in defect prediction". *Proceedings of the 2013 9th joint meeting on foundations of software engineering*. ACM. 2013, pp. 147–157.
- [Rah14] Rahman, F. et al. "Comparing Static Bug Finders and Statistical Prediction". *Proceedings of the 36th International Conference on Software Engineering*. ICSE 2014. Hyderabad, India: Association for Computing Machinery, 2014, 424–434.
- [Rat17] Rathore, S. S. & Kumar, S. "Towards an ensemble based system for predicting the number of software faults". *Expert Systems with Applications* **82** (2017), pp. 357–382.
- [Ray16] Ray, B. et al. "On the "Naturalness" of Buggy Code". *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. 2016, pp. 428–439.
- [Ray14] Ray, B. et al. "A large scale study of programming languages and code quality in github". *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM. 2014, pp. 155–165.
- [Rhm20] Rhmann, W. et al. "Software fault prediction based on change metrics using hybrid algorithms: An empirical study". *Journal of King Saud University-Computer and Information Sciences* **32.4** (2020), pp. 419–424.
- [Rom11] Romano, D. & Pinzger, M. "Using source code metrics to predict change-prone Java interfaces". *2011 27th IEEE International Conference on Software Maintenance (ICSM)*. 2011, pp. 303–312.
- [Rom08] Rombach, D. et al. "Teaching disciplined software development". *Journal of Systems and Software* **81.5** (2008), pp. 747–763.
- [Ros15] Rosen, C. et al. "Commit guru: analytics and risk prediction of software commits". *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM. 2015, pp. 966–969.
- [Ryu16a] Ryu, D. & Baik, J. "Effective multi-objective naïve Bayes learning for cross-project defect prediction". *Applied Soft Computing* **49** (2016), pp. 1062–1077.
- [Ryu15] Ryu, D. et al. "A hybrid instance selection using nearest-neighbor for cross-project defect prediction". *Journal of Computer Science and Technology* **30.5** (2015), pp. 969–980.
- [Ryu16b] Ryu, D. et al. "Value-cognitive boosting with a support vector machine for cross-project defect prediction". *Empirical Software Engineering* **21.1** (2016), pp. 43–71.
- [Ryu17] Ryu, D. et al. "A transfer cost-sensitive boosting approach for cross-project defect prediction". *Software Quality Journal* **25.1** (2017), pp. 235–272.
- [Sac66] Sackman, H. et al. *Exploratory experimental studies comparing online and offline programing performance*. Tech. rep. SYSTEM DEVELOPMENT CORP SANTA MONICA CA, 1966.

- [Saw13] Sawyer, R. “BI’s Impact on Analyses and Decision Making Depends on the Development of Less Complex Applications”. *Principles and Applications of Business Intelligence Research*. IGI Global, 2013, pp. 83–95.
- [Sca14] Scandariato, R. et al. “Predicting vulnerable software components via text mining”. *IEEE Transactions on Software Engineering* **40.10** (2014), pp. 993–1006.
- [Sel18] Sela, A. & Ben-Gal, H. “Big Data Analysis of Employee Turnover in Global Media Companies, Google, Facebook and Others”. 2018, pp. 1–5.
- [Shi10] Shihab, E. et al. “Understanding the impact of code and process metrics on post-release defects: a case study on the eclipse project”. *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*. 2010, pp. 1–10.
- [Shr20a] Shrikanth, N. & Menzies, T. “Assessing practitioner beliefs about software defect prediction”. *2020 IEEE/ACM 42nd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE. 2020, pp. 182–190.
- [Shr20b] Shrikanth, N. & Menzies, T. “What disconnects practitioner belief and empirical evidence?” *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings*. 2020, pp. 286–287.
- [Shr21a] Shrikanth, N. & Menzies, T. *Simplifying Software Defect Prediction (via the “early bird” Heuristic)*. Under Review. 2021.
- [Shr21b] Shrikanth, N. et al. “Assessing practitioner beliefs about software engineering”. *Empirical Software Engineering* **26.4** (2021), pp. 1–32.
- [Shr21c] Shrikanth, N. et al. “Early Life Cycle Software Defect Prediction. Why? How?” *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE. 2021, pp. 448–459.
- [Sie15] Siers, M. J. & Islam, M. Z. “Software defect prediction using a cost sensitive decision forest and voting, and a potential solution to the class imbalance problem”. *Information Systems* **51** (2015), pp. 62–71.
- [Ś05] Śliwerski, J. et al. “When Do Changes Induce Fixes?” *Proceedings of the 2005 International Workshop on Mining Software Repositories*. MSR ’05. St. Louis, Missouri: ACM, 2005, pp. 1–5.
- [Son18] Song, Q. et al. “A comprehensive investigation of the role of imbalanced learning for software defect prediction”. *IEEE Transactions on Software Engineering* **45.12** (2018), pp. 1253–1269.
- [Sta] *Stack overflow*. 2008 Accessed: 2021-11-24.
- [Sun12] Sun, Z. et al. “Using coding-based ensemble learning to improve software defect prediction”. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* **42.6** (2012), pp. 1806–1817.

- [Tan15] Tan, M. et al. “Online defect prediction for imbalanced data”. *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. Vol. 2. IEEE. 2015, pp. 99–108.
- [Tan16a] Tan, S.-Y. & Chan, T. “Defining and conceptualizing actionable insight: a conceptual framework for decision-centric analytics”. *arXiv preprint arXiv:1606.03510* (2016).
- [Tan18] Tantithamthavorn, C. et al. “The Impact of Class Rebalancing Techniques on the Performance and Interpretation of Defect Prediction Models”. *IEEE Transactions on Software Engineering* (2018), pp. 1–1.
- [Tan16b] Tantithamthavorn, C. et al. “Automated parameter optimization of classification techniques for defect prediction models”. *ICSE 2016*. ACM. 2016, pp. 321–332.
- [Tan18] Tantithamthavorn, C. et al. “The impact of automated parameter optimization on defect prediction models”. *IEEE Transactions on Software Engineering* **45.7** (2018), pp. 683–711.
- [Tho97] Thomas, W. M. et al. “An analysis of errors in a reuse-oriented development environment”. *Journal of Systems and Software* **38.3** (1997), pp. 211–224.
- [Thi] *Tiruvalluvanayan arulicceyta Tirukkural = The 'Sacred' Kural of Tiruvalluva-Nayanar*. <https://archive.org/details/tiruvalluvanayan00tiruuoft/mode/2up>. Accessed: 2020-04-18.
- [Ton18] Tong, H. et al. “Software defect prediction using stacked denoising autoencoders and two-stage ensemble learning”. *Information and Software Technology* **96** (2018), pp. 94–111.
- [Tur13] Turhan, B. et al. “Empirical evaluation of the effects of mixed project data on learning defect predictors”. *Information and Software Technology* **55.6** (2013), pp. 1101–1118.
- [Val12] Vallespir, D. & Nichols, W. “An Analysis of Code Defect Injection and Removal in PSP”. *Proceedings of the TSP Symposium 2012*. Carnegie Mellon University. Pittsburgh, 2012.
- [Val16] Vallespir, D. & Nichols, W. “Quality Is Free, Personal Reviews Improve Software Quality at No Cost.” *Software Quality Professional* **18.2** (2016).
- [Var00] Vargha, A. & Delaney, H. D. “A critique and improvement of the CL common language effect size statistics of McGraw and Wong”. *Journal of Educational and Behavioral Statistics* **25.2** (2000), pp. 101–132.
- [Vas15] Vasilescu, B. et al. “Quality and productivity outcomes relating to continuous integration in GitHub”. *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM. 2015, pp. 805–816.
- [Wal18] Walkinshaw, N. & Minku, L. “Are 20% of Files Responsible for 80% of Defects?” *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. ESEM '18. Oulu, Finland: ACM, 2018, 2:1–2:10.
- [Wan18a] Wan, Z. et al. “Perceptions, expectations, and challenges in defect prediction”. *IEEE Transactions on Software Engineering* (2018).

- [Wan13] Wang, S. & Yao, X. “Using class imbalance learning for software defect prediction”. *IEEE Transactions on Reliability* **62.2** (2013), pp. 434–443.
- [Wan16a] Wang, S. et al. “Automatically learning semantic features for defect prediction”. *Proceedings of the 38th International Conference on Software Engineering*. ACM. 2016, pp. 297–308.
- [Wan18b] Wang, S. et al. “Deep semantic feature learning for software defect prediction”. *IEEE Transactions on Software Engineering* (2018).
- [Wan16b] Wang, T. et al. “Multiple kernel ensemble learning for software defect prediction”. *Automated Software Engineering* **23.4** (2016), pp. 569–590.
- [Wey08] Weyuker, E. J. et al. “Do too many cooks spoil the broth? using the number of developers to enhance defect prediction models”. *Empirical Software Engineering* **13.5** (2008), pp. 539–559.
- [Wil19] Willett, W. N. W. H. J. M. J. M. D. B. A. *PSP Student Assignment Data*. 2019.
- [Woh02] Wohlin, C. “Is prior knowledge of a programming language important for software quality?”. *Proceedings International Symposium on Empirical Software Engineering*. IEEE. 2002, pp. 27–34.
- [Woh12] Wohlin, C. et al. *Experimentation in software engineering*. Springer Science & Business Media, 2012.
- [Wu18] Wu, F. et al. “Cross-project and within-project semisupervised software defect prediction: A unified approach”. *IEEE Transactions on Reliability* **67.2** (2018), pp. 581–597.
- [Xia16a] Xia, X. et al. “Hydra: Massively compositional model for cross-project defect prediction”. *IEEE Transactions on software Engineering* **42.10** (2016), pp. 977–998.
- [Xia16b] Xia, X. et al. “Predicting crashing releases of mobile applications”. *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. 2016, pp. 1–10.
- [Xia17] Xia, X. et al. “What do developers search for on the web?”. *Empirical Software Engineering* **22.6** (2017), pp. 3149–3185.
- [Xia19] Xia, X. et al. “How practitioners perceive coding proficiency”. *Proceedings of the 41st International Conference on Software Engineering*. IEEE Press. 2019, pp. 924–935.
- [Xu19] Xu, Z. et al. “Software defect prediction based on kernel PCA and weighted extreme learning machine”. *Information and Software Technology* **106** (2019), pp. 182–200.
- [Yan19a] Yan, M. et al. “Characterizing and identifying reverted commits”. *Empirical Software Engineering* **24.4** (2019), pp. 2171–2208.
- [Yan20] Yan, M. et al. “Just-In-Time Defect Identification and Localization: A Two-Phase Framework”. *IEEE Transactions on Software Engineering* (2020).

- [Yan19b] Yang, X. et al. “An Empirical Study on Progressive Sampling for Just-in-Time Software Defect Prediction” (2019).
- [Yan17] Yang, X. et al. “TLEL: A two-layer ensemble learning approach for just-in-time defect prediction”. *Information and Software Technology* **87** (2017), pp. 206–220.
- [Yan16] Yang, Y. et al. “Effort-aware just-in-time defect prediction: simple unsupervised models could be better than supervised models”. *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM. 2016, pp. 157–168.
- [Yao20] Yao, J. & Shepperd, M. “Assessing software defection prediction performance: Why using the Matthews correlation coefficient matters”. *Proceedings of the Evaluation and Assessment in Software Engineering*. 2020, pp. 120–129.
- [Yat19] Yatish, S. et al. “Mining software defects: should we consider affected releases?” *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE. 2019, pp. 654–665.
- [Ye14] Ye, X. et al. “Learning to rank relevant files for bug reports using domain knowledge”. *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 2014, pp. 689–699.
- [Zha17] Zhang, F. et al. “The Use of Summation to Aggregate Software Metrics Hinders the Performance of Defect Prediction Models”. *IEEE Transactions on Software Engineering* **43.5** (2017), pp. 476–491.
- [Zha14] Zhang, F. et al. “Towards building a universal defect prediction model”. *Proceedings of the 11th Working Conference on Mining Software Repositories*. 2014, pp. 182–191.
- [Zha16a] Zhang, F. et al. “Cross-project defect prediction using a connectivity-based unsupervised classifier”. *Proceedings of the 38th International Conference on Software Engineering*. ACM. 2016, pp. 309–320.
- [Zha16b] Zhang, F. et al. “The use of summation to aggregate software metrics hinders the performance of defect prediction models”. *IEEE Transactions on Software Engineering* **43.5** (2016), pp. 476–491.
- [Zha16c] Zhang, F. et al. “Towards building a universal defect prediction model with rank transformed predictors”. *Empirical Software Engineering* **21.5** (2016), pp. 2107–2145.
- [Zha10a] Zhang, H. & Wu, R. “Sampling program quality”. *2010 IEEE International Conference on Software Maintenance*. IEEE. 2010, pp. 1–10.
- [Zha10b] Zhang, H. et al. “On the value of learning from defect dense components for software defect prediction”. *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*. 2010, pp. 1–9.

- [Zha15] Zhang, Y. et al. “An empirical study of classifier combination for cross-project defect prediction”. *2015 IEEE 39th Annual Computer Software and Applications Conference*. Vol. 2. IEEE. 2015, pp. 264–269.
- [Zho19] Zhou, T. et al. “Improving defect prediction with deep forest”. *Information and Software Technology* **114** (2019), pp. 204–216.
- [Zho18] Zhou, Y. et al. “How far we have progressed in the journey? an examination of cross-project defect prediction”. *ACM Transactions on Software Engineering and Methodology (TOSEM)* **27.1** (2018), pp. 1–51.
- [Zim07] Zimmermann, T. et al. “Predicting defects for eclipse”. *Third International Workshop on Predictor Models in Software Engineering (PROMISE’07: ICSE Workshops 2007)*. IEEE. 2007, pp. 9–9.
- [Zim09] Zimmermann, T. et al. “Cross-project defect prediction: a large scale experiment on data vs. domain vs. process”. *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. 2009, pp. 91–100.
- [Zou18] Zou, W. et al. “How practitioners perceive automated bug report management techniques”. *IEEE Transactions on Software Engineering* (2018).