ABSTRACT

KRISHNA PRASAD, RAHUL. Learning Actionable Analytics in Software Engineering. (Under the direction of Tim Menzies.)

Software analytics is routinely used by researchers and industrial practitioners for many diverse tasks. Large organizations such as Microsoft routinely make practice data-driven policy development where organizational policies are learned from an extensive analysis of large datasets. However, despite these successes, there exist some limitations to modern software analytic tools —

1. Lack of relevant data to perform the analytics, and
2. Lack of insightful analytics.

This thesis attempts to highlight and offer potential solutions to these pressing problems.

A premise with most of the prior work on software data analytics is that there exists data from which we can learn models. But this premise is not always satisfied. When local data is scarce, sometimes it is possible to use data collected from other projects. Researchers achieve this using transfer learning which seeks to transfer knowledge from some source project and apply it to a target project. Much of the transfer learning methodologies achieve this by using complex dimensionality transformations. However, these methodologies were seldom generalizable and needlessly complex. To address this, this thesis offers a very simple "bellwether" transfer learner. Given N data sets, we find one dataset that which produces the best predictions on all the other projects. We call this the "bellwether" data set. We show that these can then be used for all subsequent analytics. We explore the existence of Bellwethers in a number of domains within software analytics: (a) Code smells detection; (b) Effort estimation; (c) Issue lifetime estimation; (d) Defect Prediction; and (e) Optimization of highly configurable systems. In doing so, we found that bellwethers are

ubiquitous and in general outperform other more complex transfer learners.

The second key challenge in data mining in software engineering is the lack of actionable analytics. Specifically, while there are tools that support discovering likely issues in software engineering, there is a general lack tools that generate "plans"– specific suggestions on what to change in order to improve the quality of software. Additionally, textbooks, tools, subject matter experts, and researchers often disagree on what is the best way to improve software quality. To address this, this thesis introduces XTREE, a framework that analyzes historical logs of defects seen previously in the code and generates a set of plans, presented as useful code changes, to prevent these issues from reoccurring. We discovered that code modules that are changed in response to XTREE's recommendations contain significantly fewer defects than recommendations from previous studies. Further, it was discovered that the code changes endorsed by XTREE significantly overlaps with the changes actually undertaken by developers.

The work on the above above issues has served to highlight several areas that warrant further exploration. First, we believe that we can use "bellwethers" in conjunction with XTREE to generate actionable plans across projects. This will attempt to generate a unified software analytics framework that generates actionable analytics by looking within and across various software projects. Another area that represents a computational bottleneck is the process of discovering the so-called "Bellwethers". The current method undertakes a brute-force approach with an $O(N^2)$ complexity. While this serves as a baseline, we believe we can achieve faster discovery by seeking better ways to do this.

Learning Actionable Analytics in Software Engineering

by
Rahul Krishna Prasad

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

Computer Science

Raleigh, North Carolina

2019

APPROVED BY:

Laurie Williams	Jamie Jennings
Raju Vatsavai	Tim Menzies Chair of Advisory Committee

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER

1

INTRODUCTION

Software analytics is a part of data mining that attempts to link data mined from many different software artifacts to obtain valuable insights so as to inform the decision-making process throughout a software project's life-cycle. Software analytics is currently a hot and promising topic in software engineering research. Over the past decade, a large number of metrics and models have been proposed for measuring various aspects of software projects such as complexity, maintainability, readability, and other important aspects of software quality (e.g., [Hal77; McC76; CK94; BW08]). Researchers and industrial practitioners routinely make extensive use of software analytics. It has be applied in a myriad different ways. For example, to estimate how long it would take to integrate the new code [Cze11], where

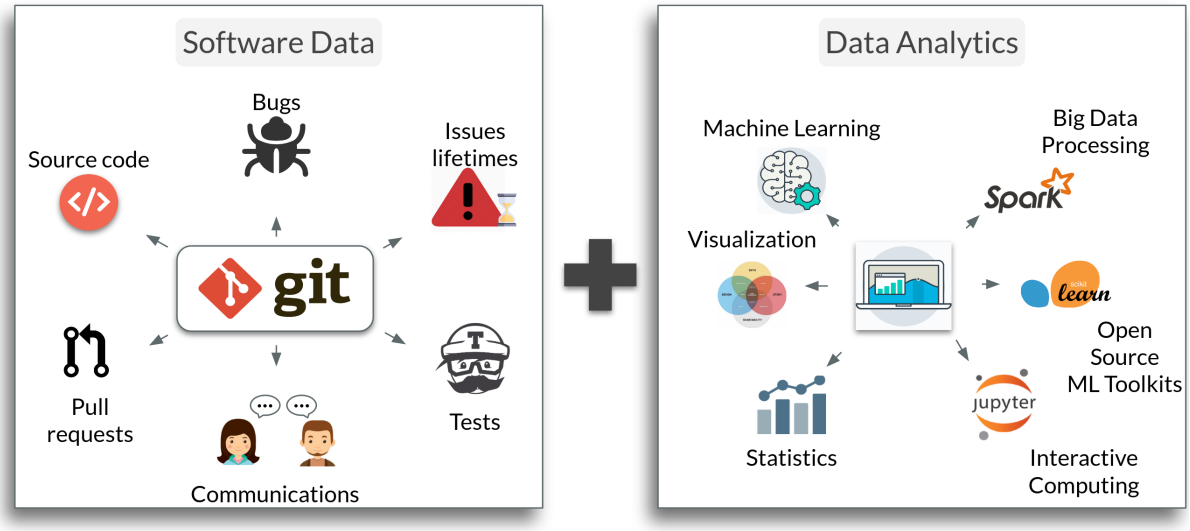


Figure 1.1 Software Analytics

bugs are most likely [Ost04; Men07c], or how long it will take to develop this code [Tur11; Koc12]. In fact, it is now routine for practitioners to treat every planned feature as an "experiment" [Sav16]. In this approach, key performance metrics are carefully monitored and analyzed to judge each proposed feature. Even simple design decisions such as the color of a link are chosen by experiments [LV14]. Large organizations like Microsoft routinely practice data driven policy development where organizational policies are learned from an extensive analysis of large datasets collected from developers [BZ14; The15].

Although software analytics has yielded some very promising results, the current generation of software analytics still has some unique challenges that need to be solved:

1. **What is to be done when there is not enough data?** When a specific project doesn't have historical data, how does one go about learning?
2. **How can one learn lessons from other similar projects?** One way to address the previous challenge is to use *transfer learning*. However, how does one choose an appropriate transfer learner?

3. **Where should one transfer lessons from?** Given the exponential growth in the number of software projects, which project should one transfer lessons from to achieve the best results?
4. **How does one obtain actionable insights?** Most modern software analytics tools are mostly prediction algorithms (e.g. support vector machines, naive bayes, logistic regression, etc). How does one go from prediction to decision making?

While the first two challenges have been addressed with the use of sophisticated machine learning techniques such as *transfer learning*. The last two challenges have yet to be solved. This thesis attempts to offer tools and techniques to help address these challenges.

While there is a lack of historical data for a project, we can use one of many transfer learning techniques. However, given the exponential growth in the number of software projects, there is a major issue of *conclusion instability* with transfer learning methods when applied to software engineering. Conclusion instability may be defined as follows:

"The more data we inspect from more projects, the more our conclusions change."

The problem with conclusion instability is that the lessons transfer from a certain project may no longer hold when new data arrives. Conclusion instability is very unsettling for software project managers struggling to find general policies. In order to support managers, who seek stability in their conclusions, while also allowing new projects to take full benefit of the data from recent projects, in this research we recommend that we must forgo attempts to *generalize* from all data. We believe that a more achievable goal would be to *slow* the pace of conclusion change. The approach proposed in this thesis is to declare one project as the *"bellwether"*[1] which should be used to make conclusions about all other projects. Note that conclusions are stable for as long as this bellwether continues to be the best

[1]According to the Oxford English Dictionary, the "bellwether" is the leading sheep of a flock, with a bell on its neck.

oracle for that community. In the rest of this thesis, we will demonstrate that regardless of the sub-domain of software engineering (code smells detection, effort estimation, defect prediction, or performance optimization), there always exists a bellwether dataset that can be used to train relatively accurate quality prediction models.

A critical component of software analytics that has become drawn a lot of attention of late is the need for actionable advice. Such advice would empower software developers and teams to gain and share insight from their data to make better decisions. Due to the volume of data, finding these insights typically requires some degree of automation. Lack of such insights is a common critique against current generation of software analytics. For example, at the workshop on "Actionable Analytics" in 2015 IEEE conference on Automated Software Engineering, business users were vocal in their complaints about analytics [8]. "Those tools tell us what is, " said one business user, "But they don't tell us what to do". Accordingly, in this thesis we seek new tools that offer guidance on "what to do" within a specific project. The tool assessed in this thesis is the XTREE planning tool. XTREE employs a *cluster + contrast* approach to planning where it (a) Clusters different parts of the software project based on a quality measure (e.g. the number of defects); and (b) Reports the contrast sets between neighboring clusters. Each of these contrast sets represent the difference between these clusters and they can be interpreted as plans. XTREE uses data from within a software project to generate plans. But, in several cases local data may not readily available. To overcome this limitation, we may choose to incorporate our findings from bellwethers to extend XTREE to use the bellwether projects.

1.1 Thesis Statement

> *"Software quality can be improved by generating effective actionable analytics that are stable despite constant change."*

1.2 Thesis Contributions

In summary, the key contributions of this work are:

1. **Addressing conclusion instability with bellwethers:** We show that depending on the source dataset, there can be large variances in the performance of transfer learner. Further, we show that different source datasets can lead to different (and often contradicting) conclusions. This thesis demonstrates that these issues can be potentially addressed using the bellwether dataset.
2. **Empirical evidence for generality of bellwethers:** The more the bellwether effect is explored, the more we learn about its broad applicability. Originally, we explored this just in the context of defect prediction, it has now been shown to work also in:
   - Code smells detection (specifically God Class and Feature Envy);
   - Effort estimation;
   - Issue lifetime estimation;
   - Defect Prediction; and
   - Configuration optimization.
3. **Bellwethers as a baseline transfer learner:** Prior to bellwethers, the research on transfer learning was very domain specific,i.e., evidence of it's effectiveness was only demonstrate in single domain (e.g., defect prediction). In attempts to generalize transfer learners, we

discovered that the transfer learning literature lacks a simple baseline to compare and contrast the various transfer learners.

To address this, we developed a baseline transfer learner called BEETLE (Bellwether Transfer Learner). To the best of our knowledge, our work was the first to offer a baseline for transfer learning and to undertake a case study of all the state-of-the-art transfer learners and validate their usability in domains other than defect prediction with respect to this baseline. In this thesis, we demonstrate that BEETLE is just as effective as other state-of-the-art algorithms such as:

- Transfer Component Analysis (referred to henceforth as TCA+);
- Transfer Naive Bayes (hereafter referred to as TNB) [21];
- Value Cognitive Boosting Learner [22];
- Gaussian Processes Modeling; and
- Linear Regression.

4. **New kinds of software analytics techniques:** Another major component of this research was generating actionable insights in software analytics. In order to do this, we developed a novel algorithm call XTREE. Our results have established that planning is quite successful in producing actions that can reduce the number of defects. We show that actions recommended with XTREE has a significant overlap with the actions actually taken by developers in order to reduce defects.
5. **Rich replication packages:** For all the experiments reported in this thesis, we have made available several replication packages [2] [3].

[2] https://git.io/fNcYY
[3] https://git.io/fh149

1.3 Publications

1.3.1 Published

[1] Krishna, R. & Menzies, T.. *"Bellwethers: A Baseline Method For Transfer Learning"*. In IEEE Transactions on Software Engineering, 2018.

[2] Krishna, R., Menzies, T., & Layman, L. *"Less is more: Minimizing code reorganization using XTREE"*. In Information and Software Technology, 2017.

[3] Krishna, R., Menzies, T., & Fu, W. *"Too much automation? The Bellwether Effect and its Implications for Transfer Learning."* In Intl. Conference on Automated Software Engineering, 2016.

[4] Krishna, R. & Menzies, T. *"Actionable=Cluster+Contrast?"*. In Intl. Conference on Automated Software Engineering Workshop, 2015.

1.3.2 Under Review

[1] Krishna, R. & Menzies, T.. *"Learning Actionable Analytics from Multiple Software Projects"*. Under review in IEEE Transactions on Software Engineering, 2019.

[2] Nair, V., Krishna, R., Menzies, T., Jamshidi, P., *"BEETLE: Finding Better Software Configurations by Borrowing from Old Ones"*. Under review in IEEE TSE, 2019.

1.3.3 Other Collaborative Publications

[1] Krishna, R., Yu, Z., Agrawal, A., Dominguez, M., Wolf, D. *"The 'BigSE' Project: Lessons Learned from Validating Industrial Text Mining. "*. In Intl. Workshop on Big Data Software Engineering (BIGDSE), 2016.

[2] Krishna, R., Agrawal, A., Rahman, A., Sobran, A., & Menzies, T. *"What is the Connection Between Issues, Bugs, and Enhancements? (Lessons Learned from 800+ Software Projects)"*. In ICSE 2018, SEIP.

[3] Wang, J., Yang, Y., Krishna, R., Menzies, T. & Wang, Q., *"Effective Automated Decision Support for Managing Crowdtesting"*. In Intl. Conference on Foundations of Software Engineering, 2019 (accepted);

[4] Chen, J., Nair, V., Krishna, R., & Menzies, T. *"Sampling as a Baseline Optimizer for Search-based Software Engineering"*. In IEEE Transactions on Software Engineering, 2018.

[5] Chen, D., Fu, W., Krishna, R., & Menzies, T. *"Applications of psychological science for actionable analytics"*. In Intl. Conference on Foundations of Software Engineering, 2018.

[6] Agrawal, A., Rahman, A., Krishna, R., Sobran, A. & Menzies, T. *"We Don't Need Another Hero? The Impact of 'Heroes' on Software Development"*. In Intl. Conf. Software Engineering, 2018 SEIP.

[7] Rahman, A., Agrawal, A., Krishna, R., Sobran, A. & Menzies, T. *"Characterizing The Influence of Continuous Integration. Empirical Results from 250+ Open Source and Proprietary Project"*. In Intl. Conference on Foundations of Software Engineering, SWAN 2018.

1.3.4 External Validations

[1] Mensah, S., Keung, J., MacDonell, S. G., Bosu, M. F., & Bennin, K. E. (2018). Investigating the significance of the Bellwether effect to improve software effort prediction: further empirical study. IEEE Transactions on Reliability, 67(3), 1176-1198.

[2] Mensah, S., Keung, J., MacDonell, S. G., Bosu, M. F., & Bennin, K. E. (2017, July). Investigating the significance of bellwether effect to improve software effort estimation. In

Software Quality, Reliability and Security (QRS), 2017 IEEE International Conference on (pp. 340-351). IEEE.

[3] Mensah, S., Keung, J., Bosu, M. F., Bennin, K. E., & Kudjo, P. K. (2017). A Stratification and Sampling Model for Bellwether Moving Window. In SEKE (pp. 481-486).

CHAPTER

2

BACKGROUND

2.1 Transfer Learning

When there is insufficient data to apply data miners to learn defect predictors, *transfer learning* can be used to transfer lessons learned from other *source* projects S to the *target* project T.

Initial experiments with transfer learning offered very pessimistic results. Zimmermann et al. [Zim09b] tried to port models between two web browsers (Internet Explorer and Firefox) and found that cross-project prediction was still not consistent: a model built on Firefox was useful for Explorer, but not vice versa, even though both of them are similar

applications. Turhan's initial experimental results were also very negative: given data from 10 projects, training on $S = 9$ source projects and testing on $T = 1$ target projects resulted in alarmingly high false positive rates (60% or more).

Subsequent research realized that data had to be carefully sub-sampled and possibly transformed before quality predictors from one source are applied to a target project. That work can be divided two ways:

- *Homogeneous* vs *heterogeneous*;
- *Similarity* vs *dimensionality transform*.

Homogeneous, heterogeneous transfer learning operates on source and target data that contain the *same, different* attribute names (respectively). This thesis focuses on homogeneous transfer learning, for the following reason. As discussed in the introduction, we are concerned with an IT manager trying to propose general policies across their IT organization. Organizations are defined by what they do—which is to say that within one organization there is some overlap in task, tools, personnel, and development platforms. This overlap justifies the use of lessons derived from transfer learning.

Hence, all our dataset contain overlapping attributes. In our case these attributes are the metrics gathered for each of the projects. As evidence for this, the datasets explored in this work fall into 4 domains; each domain contains so called "communities" of data sets. Each dataset within a community share the same attributes (see Figure 2.4).

As to other kinds of transfer learning, *similarity* approaches transfer some subset of the rows or columns of data from source to target. For example, the Burak filter [Tur09] builds its training sets by finding the $k = 10$ nearest code modules in S for every $t \in T$. However, the Burak filter suffered from the all too common instability problem (here, whenever the source or target is updated, data miners will learn a new model since different code modules will satisfy the $k = 10$ nearest neighbor criteria). Other researchers [KM11;

Koc15] doubted that a fixed value of k was appropriate for all data. That work recursively bi-clustered the source data, then pruned the cluster sub-trees with greatest "variance" (where the "variance" of a sub-tree is the variance of the conclusions in its leaves). This method combined row selection with row pruning (of nearby rows with large variance). Other similarity methods [Zha15] combine domain knowledge with automatic processing: e.g. data is partitioned using engineering judgment before automatic tools cluster the data. To address variations of software metrics between different projects, the original metric values were discretized by rank transformation according to similar degree of context factors.

Similarity approaches uses data in its raw form and as highlighted above, it suffers from instability issues. This prompted research on *Dimensionality transform* methods. These methods manipulate the raw source data until it matches the target. In the case of defect prediction, a "dimension" was one of the static code attributes of Figure 2.5.

An initial attempt on performing transfer learning with *Dimensionality transform* was undertaken by Ma et al. [Ma12] with an algorithm called transfer naive Bayes (TNB). This algorithm used information from all of the suitable attributes in the training data. Based on the estimated distribution of the target data, this method transferred the source information to weight instances the training data. The defect prediction model was constructed using these weighted training data.

Nam et al. [Nam13] originally proposed a transform-based method that used TCA based dimensionality rotation, expansion, and contraction to align the source dimensions to the target. They also proposed a new approach called TCA+, which selected suitable normalization options for TCA.

The above researchers failed to address the imbalance of classes in datasets they studied. In SE, when a dataset is gathered the samples in them tend to be skewed toward one of the classes. A systematic literature review on software defect prediction carried out by Hall et

al. [Hal12] indicated that data imbalance may be connected to poor performance. They also suggested more studies should be aware of the need to deal with data imbalance. More importantly, they assert that the performance measures chosen can *hide* the impact of imbalanced data on the real performance of classifiers.

An approach proposed by Ryu et al. [Ryu16] showed that using Boosting-SVM combined with class imbalance learner can be used to address skewed datasets. They showed improved performance compared to TNB. More recently, in our previous work [Kri16b], we showed that a very simplistic transfer learner can be developed using the “bellwether" dataset with Random Forest. We reported highly competitive performance scores.

When there are no overlapping attributes (in heterogeneous transfer learning) Nam et al. [NK15] found they could dispense with the optimizer in TCA+ by combining feature selection on the source/target following by a Kolmogorov-Smirnov test to find associated subsets of columns. Other researchers take a similar approach, they prefer instead a canonical-correlation analysis (CCA) to find the relationships between variables in the source and target data [Jin15].

Considering all the attempts at transfer learning sampled above, our reading of these literature suggests a surprising lack of consistency in the choice of datasets, learning methods, and statistical measures while reporting results of transfer learning. Further, there was no baseline approach to compare the algorithms against.

2.1.1 Conclusion Instability

As and when new data arrives, there is a sudden and an unpredictable change in conclusions that are derived from that data source. This uncertainty accompanying a change in data is termed as *conclusion instability*. It manifests itself as large variances in conclusions and

[Fow99] and [Ker05]	[LM06]	[Cam15]	[YM13a]	Dev. Surv
Alt. Classes with Diff. Interfaces				
Combinatorial Explosion [Ker05]				
Comments			11	VL
Conditional Complexity [Ker05]			14	?
Data Class	✓			
Data Clumps				
Divergent Change				
Duplicated Code	✓	✓	1	VH
Feature Envy	✓		8	
Inappropriate Intimacy		✓		L
Indecent Exposure [Ker05]				?
Incomplete Library Class				
Large Class	✓	✓	4	VH
Lazy Class/Freeloader		✓	7	
Long Method	✓	✓	2	VH
Long Parameter List		✓	9	L
Message Chains				H
Middle Man				
Oddball Solution [Ker05]				
Parallel Inheritance Hierarchies				
Primitive Obsession				
Refused Bequest	✓	✓		
Shotgun Surgery	✓			
Solution Sprawl [Ker05]				
Speculative Generality				L
Switch Statements				L
Temporary Field		✓		?

Figure 2.1 Bad smells from different sources. Check marks (✓) denote a bad smell was mentioned. Numbers or symbolic labels (e.g. "VH") denote a priorization comment (and "?" indicates lack of consensus). Empty cells denote some bad smell listed in column one that was not found relevant in other studies. Note: there are many blank cells.

these instabilities usually challenges the validity of the policy decisions made prior to arrival of new data. In addition to making generating general policies very difficult, it also causes practitioners to distrust decisions made from software analytics tools [Has17]. In this work, we define and categorize conclusion instability into two forms: (a) performance instability,

ref	cbo	rfc	lcom	dit	noc	wmc	#projects	size
[Ola07]	+	+	+	-	-	+	6	95-201
[Agg09]	+	+	+	-	-	+	12	86 classess (3-12kloc)
[AB06]	+	+	-				1	1700 (110kloc)
[Bas96a]	+	+	-	+	+	+	8	113
[Bri00]	+	+	-	+	+	+	8	114
[Bri01]	+	+	+	+	-		1	83
[CS00]				+	+		1	32
[Ema01b]				+	-		1	42-69
[Ema99]	+	-	-	-	-	-	1	85
[Tan99]	-	+		-	-	+	3	92
[Yu02]	+	+	+	-	+	+	1	123 (34kloc)
[SK03]	+			+		+	1	706
[ZL06]	+	+	+	-	+	+	1	145
[Gyi05]	+	+	+	+	-	+	1	3677
[Hol09]	+	+	+			+	1	?
[SL08]	+	+	+	+	+	+	3	?
[FN01]	-	+	+	-	-	+	8	113
[TM03]		+	+	+	+		2	64
[Den03]		-		-	-	-	1	3344 modules (2mloc)
[Jan06]	+	+	+	-	-	+	5	395
[Eng09]	+	+		-	-	+	1	1412
[Sha10a]	+	+		-	-	+	2	9713
[Sin10]	+	+	-	-	-	+	1	145
[Gla00]				+	-		1	145
[Ema01a]	-	-	-	-	-	-	1	174
[TV10]	-					-	0	50
[Xu08]	+	+	-	-	-	+	1	145
[Suc03]		+		+	+		2	294
total +	18	20	11	11	8	17		
total -	4	3	7	14	16	4		

KEY: Strong consensus (over 2/3rds); Some consensus (less than 2/3rds); Weak consensus (about half); No consensus

Total percents: "*" denotes majority conclusion in each column

+	* 64%	* 71%	* 39%	39%	29%	* 61%
-	14%	11%	25%	* 50%	* 57%	14%

Figure 2.2 Contradictory conclusions from OO-metrics studies for defect prediction. Studies report significant ("+") or irrelevant ("-") metrics verified by univariate prediction models. Blank entries indicate that the corresponding metric is not evaluated in that particular study. Colors comment on the most frequent conclusion of each column. CBO= coupling between objects; RFC= response for class (#methods executed by arriving messages); LCOM= lack of cohesion (pairs of methods referencing one instance variable, different definitions of LCOM are aggregated); NOC= number of children (immediate subclasses); WMC= #methods per class.

and (b) source instability.

(a) Performance Instability: This can be noticed during ranking studies undertaken to pick a reliable data miner. For instance, many researchers run ranking studies where performance

scores are collected from many classifiers which are ranked for tasks such as defect prediction [Les08; Hal12; EE08; Men10; Gon08; Rad13; Jia08b; WY13; MK09; Li12; Kho10; Jia09; Gho15; Jia08a; Tan16; Fu16a]. These rankings are then used to identify the "best" defect predictor. However, these prediction tasks assume that future events to be predicted will be near identical to past events. Therefore, given data in the from $\{x_{train}, y_{train}\}$, prediction algorithms use this for *training* in order to form a joint distribution $P(X, Y) = P(Y|X)P(Y)$ and estimate the conditional $\hat{P}(Y|X_{test})$. These predictions will be good as long as the data is a close approximation of the underlying distribution. As the source of the data changes, the joint distribution $P(X, Y)$ changes to reflect this new data. This gradual change in the underlying distribution of training data with the arrival of new data is called *data drift*. It is widely accepted that this *drift* is the leading cause of instability of prediction models [QC09; Han06; Sto08]. Performance instability can result in large variances in the quality of predictions. Numerous researchers [Fu16a; AM17] have shown that changing only the data and retaining the same defect predictor can result in statistically significant differences.

(b) *Source Instability:* This arises due to the constant influx of potential new data sources. In methods such as transfer learning, where we translate quality predictors learned in one data set to another, arrival of new data would require changing models all the time as the transfer learners continually exchange new models to the already existing ones. However, as demonstrated in subsequent parts of this section, each new data source can produce completely different and often contradicting conclusions. Identifying a reliable source of data from all the available options is a pressing issue; more so for methods such as transfer learning since they place an inherent faith in quality the data source. If a change in data source can also change the conclusions, then not being able to identify a reliable data source would limit one from leveraging the full benefits of transfer learning.

Impact of these instabilities can be observed in several domains within software engi-

neering. The studies explored in the rest this section sample some instances of instability and its prevalence in the domains of software engineering studied here[1]. Note the vast contradictions in conclusions in each of these domains.

2.1.1.1 Code Smells

Research on software refactoring endorses the use of code-smells as a guide for improving the quality of code as a preventative maintenance. However, as discussed below, a lot of the research on bad-smells suffers from conclusion instability.

There is much contradictory evidence on whether programmers should take heed of these guidelines or ignore them. For instance, a systematic literature review conducted by Tufano et al. [Tuf15] lists dozens of papers that recommend tools for repair and detection of code smells. On the other hand, several other researchers cast doubt on the value of code smells and their use as triggers for change [Man04; YM13a; Sjo13].

Further, this contradiction is also frequently seen among domain experts. Researchers caution that developers' cognitive biases can lead to misleading assertions that some things are important when they are not. According to Passos et al. [Pas11], developers often assume that the lessons they learn from a few past projects are general to all their future projects. They comment, "past experiences were taken into account without much consideration for their context" [Pas11]. This warning is echoed by Jørgensen & Gruschke [JG09]. They report that the supposed software engineering experts seldom use lessons from past projects to improve their future reasoning and that such poor past advice can be detrimental to new projects. [JG09].

Other studies have shown some widely-held views are now questionable given new

[1]Note: Due to relatively recency of the research on estimating lifetime of open issues and comparatively fewer papers, we omit it from this survey of conclusion instability.

evidence. Devanbu et al. examined responses from 564 Microsoft software developers from around the world. They comment programmer beliefs can vary with each project, but do not necessarily correspond with actual evidence in that project [Dev16].

The above remarks seem to hold true for bad smells. As shown in Figure 2.1, there is a significant disagreement on which bad smells are important and relevant to a particular project. In that figure, the first column lists commonly mentioned bad smells and comes from Fowler's 1999 text [Fow99]. The other columns show conclusions from other studies about which bad smells matter most[2]. From this figure, it is easy to note the lack of consensus among developers, text books, and tools. They all disagree on which bad smells are important; just because one developer strongly believes in the importance of a bad smell, it does not mean that the same belief transfers to other developers.

In summary, we seek methods like bellwethers in order to draw stable conclusions. A particular challenge in each of the study in Figure 2.1 is the lack of consistent data source over the period of time these studies were undertaken. In such cases, bellwether datasets can be particularly useful.

2.1.1.2 Defect Prediction

In the area of defect prediction too there are several examples of conclusion instability. As motivating examples, consider the following two findings: (a) Zimmermann et al. [Zim09a] showed that when they learned defect predictors from 622 pairs of projects, in only 4% of pairs, the defect predictors learned from one project pair worked in another. These contradictory conclusions extend to OO metrics as well; and (b) In our previous work, we

[2]The *developer survey* column shows the results of an hour-long whiteboard session with a group of 12 developers from a Washington D.C. web tools development company. Participants worked in a round robin manner to rank the bad smells they thought were important (and any disagreements were discussed with the whole group)

conducted a large scale systematic literature review [Men11a]. We distilled our findings into a list of 28 studies. We noted that they offered contradictory conclusions regarding the effectiveness of OO metrics. These findings are tabulated in Figure 2.2. The figure offers a troubling prospect for managers of a software project. The only concrete finding they can derive from this figure is that response for class is often a useful indicator of defects. Each study makes a clear, but usually different, conclusion regarding the usefulness of other metrics.

In a study of conclusion instability, Turhan [Tur12] showed that the reason for this inconsistency is due to dataset drift. That work reported different kinds of data drift within software engineering data, such as: (1) Source component shift; (2) Domain Shift; (3) Imbalanced Data, etc. Further, he noted that all contribute significantly to the issue of conclusion instability. In our previous work, we offered further evidence to such a drift by demonstrating that different clusters within the data provided completely different models [Men11a]. Further, the models built from specialized regions within a specific data set sometimes perform better than those learned across all data. However, new data is constantly arriving, and finding these specialized regions with new data turns into an arduous task. In such cases, tools like bellwethers offer a way to draw conclusions from a stable project. As long as the bellwether project remains unchanged so does the conclusions we derive from that project.

2.1.1.3 Effort Estimation

As with code smell detection and defect prediction, conclusion instability seems to be an inherent property of the datasets commonly explored in this area [Men05]. For example, consider stability tests conducted on Boehm's COCOMO software effort estimation model by Menzies et al. [Men05]. There, it was found that only the coefficient on lines of code (loc)

was stable while the variance in dozens of other coefficients were extremely large. In fact, in the case of five coefficients, the values even changed from positive to negative across different samples in a cross-validation study.

Other studies on effort estimation also report very similar findings. Jørgensen [Jor04] compared model-based to expert-based methods in 15 different studies. That study reported that: five studies favored expert-based methods, five found no difference, and five favored model-based methods. Similarly, Kitchenham et al. [Kit07a] reviewed seven studies to check the effect of data imported from other organizations as compared with local data for building effort models. Of these seven studies, three found that models from other organizations were not significantly worse than those based on local data, while four found that they were significantly worse. MacDonell and Shepperd [MS07] also performed a review on effort estimation models by replicating Kitchenham et al. [Kit07a]. From a total of 10 studies, two were found to be inconclusive, three supported global models, and five supported local models. Similarly, Mair and Shepperd [MS05] compared regression to analogy methods for effort estimation and found conflicting evidence. From a total of 20 empirical studies, (a) seven recommended regression for building effort estimators; (b) four were indifferent; and (c) nine favored analogy.

2.2 Planning

1. What exactly do you mean by "planning"?

We distinguish planning from prediction for software quality as follows: Quality prediction points to the likelihood of defects. Predictors take the form:

$$out = f(in)$$

where *in* contains many independent features (such as OO metrics) and *out* contains some measure of how many defects are present. For software analytics, the function f is learned via data mining (for static code attributes for instance).

On the other hand, quality planning generates a concrete set of actions that can be taken (as precautionary measures) to significantly reduce the likelihood of defects occurring in the future.

For a formal definition of plans, consider a defective test example Z, planners proposes a plan D to adjust attribute Z_j as follows:

$$\forall \delta_j \in \Delta : Z_j = \begin{cases} Z_j + \delta_j & \text{if } Z_j \text{ is numeric} \\ \delta_j & \text{otherwise} \end{cases}$$

The above plans are described in terms of a range of numeric values. In this case, they represent an increase (or decrease) in some of the static code metrics of Figure 2.5. However, these numeric ranges in and of themselves may not very informative. It would be beneficial to offer a more detailed report on how to go about implementing these plans. For example, to (say) simplify a large bug-prone method, it may be useful to suggest to a developer to reduce its size (e.g., by splitting it across two simpler functions).

2. How to operationalize plans?

In order to implement such plans, developers need some guidance on what to change in order to achieve the desired effect. There are two ways to generate that guidance. One way is to is to use a technique proposed by Nayrolles et al. [NHL18] at MSR 2018. In that approach, we look through the developer's own history to find old examples where they have made the kinds of changes recommended by the plan.

However, this data is not accessible. Therefore, we must seek alternative ways to implement plans. One way is to is to use a reverse engineering technique popularized in recent literature. In these techniques, plans, which are in the form of a range numeric range, are mapped into specific actions that will improve the quality of their designs. There has been a number of recent research by the SE community [SS07; DB06; Kat02; BA09], including one of our previous work [Kri17c], that attempt to tackle this very issue. Stroggylos & Spinellis [SS07] studied the impact of CK-metrics to assert if performing reorganization was useful in reducing bugs and improving software quality, they reported a strong correlation between these metrics and software quality. Du Bois [DB06] conducted a study that explored coupling and cohesion metrics and reported similar findings. Elish & Alshayeb [EA11; EA12] conducted a systematic study to categorize code reorganization procedures in terms of their measurable effect on software quality attributes. Their studies showed how each reorganization action would impact several of the CK-metrics.

For example, consider Figure 2.3. Let's say a planner has recommended the changes shown in Figure 2.3(a). Then, we use 2.3(b) to look-up possible actions developers may take, there we see that performing an "extract method" operation may help alleviate certain defects (this is highlighted in blue). In 2.3(c) we show a simple example of a class where the above operation may be performed. In 2.3(d), we demonstrate how a developer may perform the "extract method".

3. Why use automatic methods to find quality plans? Why not just use domain knowledge; e.g. human expert intuition?

Much recent work has documented the wide variety of conflicting opinions among software developers, even those working within the same project. According to Passos et al. [Pas11], developers often assume that the lessons they learn from a few past projects are general to

DIT	NOC	CBO	RFC	FOUT	WMC	NOM	LOC	LCOM
·	·	·	△	·	△	△	△	△

(a) Recommendations from some planner. Here a '△' represents an *increase*, a '▽' and a '·' represents *no-change*.

Action	DIT	NOC	CBO	RFC	FOUT	WMC	NOM	LOC	LCOM
Extract Class			△	▽	△	▽	▽	▽	▽
Extract Method				△		△	△	△	△
Hide Method									
Inline Method				▽		▽	▽	▽	▽
Inline Temp								▽	
Remove Setting Method				▽		▽	▽	▽	▽
Replace Assignment								▽	
Replace Magic Number								△	
Consolidate Conditional				△		△	△	▽	△
Reverse Conditional									
Encapsulate Field						△	△	△	△
Inline Class			▽	△	▽	△	△	△	△

(b) A sample of possible actions developers can take. Here a '△' represents an *increase*, a '▽' represents a *decrease*, and an empty cell represents *no-change*. Taken from [SS07; DB06; Kat02; BA09; EA11; EA12]. The action highlighted in blue shows an action matching XTREE's recommendation.

```
class StoreManagement{
  // ... Some Opertaions
  private static void showMenu() { ... }
  public static void showActoions() {
    int choice;
    do {
      showMenu();
      switch choice {
      case 1: displayAllInventoryItems();
      case 2: findCompanyName();
      case 3: modifyPrice();
      case 4: saveToDisk();
              update();
              System.out.println();
              System.exit(0);
      }
    } while(choice != 4)
  }
  // ... Other Methods
}
```

(c) Before 'extract method'

```
class StoreManagement{
  // ... Some Opertaions
  private static void showMenu() { ... }
  public static void showActoions() {
    int choice;
    do {
      showMenu();
      switch choice {
      case 1: displayAllInventoryItems();
      case 2: findCompanyName();
      case 3: modifyPrice();
      case 4: exit();
      }
    } while(choice <= 4)
  }
  public static void exit() {
          saveToDisk();
          update();
          System.out.println();
          System.exit(0);
  }
  // ... Other Methods
}
```

(d) After 'extract method'

Figure 2.3 An example of how developers might use planning to reduce software defects.

all their future projects. They comment, “past experiences were taken into account without much consideration for their context” [34]. Jorgensen and Gruschke [JG09] offer a similar warning. They report that the supposed software engineering “gurus” rarely use lessons from past projects to improve their future reasoning and that such poor past advice can be detrimental to new projects [JG09]. Other studies have shown some widely-held views are now questionable given new evidence. Devanbu et al. examined responses from 564 Microsoft software developers from around the world. They comment programmer beliefs can vary with each project, but do not necessarily correspond with actual evidence in that project [Dev16].

Given the diversity of opinions seen among humans, it seems wise to explore automatic oracles for planning quality improvement change.

2.2.1 Planning in Classical AI

In classical AI, planning usually refers to generating a sequence of actions that enables an *agent* to achieve a specific *goal* [RN95]. In an idealized situation, it is assumed that the possible initial states of the world is known and so are the description of the desired goal in addition to the set of possible and feasible actions. In such a hypothetical situation, planning results in generating a set of actions that is guaranteed to enable one to reach a desired goal. This can be achieved by classical search-based problem solving approaches or logical planning agents. Such planning tasks now play a significant role in a variety of demanding applications, ranging from controlling space vehicles and robots to playing the game of bridge [Gha04].

As discussed in the rest this section, there are many types of planning. We introduce the premise of those different planning paradigms with respect to classical artificial intelligence.

2.2.1.1 Classical Planning

A simple abstraction of the planning problem is known as classical planning. Classical planning assumes that the initial state is fully-observable and any action is deterministic. Such an assumption enables us to predict the outcome of an action accurately every single time. Further, plans can be defined as sequences of actions, because it is always known in advance which actions will be needed [FN71].

Classical Planning is most useful when the problem is not very complex and the simplifying assumptions mentioned above lead to a development of a well-founded model [WJ95]. In the case of complex domains like software engineering, performing a search of plans is highly inefficient. It becomes very difficult to pick the correct search space, algorithm, and heuristics for finding these plans [Gha04].

2.2.1.2 Probabilistic Planning

Probabilistic planning tackles a slightly different problem in planning — one where the outcomes may be random and are partly controlled by the decision maker [Bel57; Alt99; GHL09]. These kinds of planning problems are usually solved using dynamic programming and reinforcement learning. The key idea here is to represent the planning problem as an optimization problem [Gha04]. Planning for such problems are best achieved when the state-space is small [Gha04]. Additionally, such a planning approach works even with partial observability, i.e., it works even when the planning agent cannot fully observe the underlying problem space. Planning over partially observable states is achieved by maintaining a distribution of probabilities over the possible states and planning based on these distributions [Kae98].

2.2.1.3 Preference-based Planning

The preference based planning is an extension of the above planning schemes with a focus on producing plans that satisfy as many user-defined constraints (preferences) as possible [SP06]. These preferences as defined by the user are generally not hard constrains, rather they define the quality of the plan, which increases as more of the preferences are satisfied. Algorithms that can solve constraint satisfaction problems are well suited to solve these forms of planning problems. Other popular algorithms include PPLAN [SP06] and HTN [BM09]. However, existence of a model precludes the use of this planning approach. This is a major impediment, since for reasons discussed in the following section, not every domain has a reliable model.

2.3 Target Domains

The rest of this work attempts to discover bellwethers and assesses the performance of bellwethers as baseline transfer learning method. For this, we explore 4 domains in SE: code smells, issue lifetime estimation, effort estimation, and defect prediction.

2.3.1 Code Smells

According to Fowler [Fow99], bad smells (a.k.a. code smells) are “a surface indication that usually corresponds to a deeper problem”. Studies suggest a relationship between code smells and poor maintainability or defect proneness [YC13; YM13b; Zaz11] and therefore, smell detection has become an established method to discover source code (or design) problems to be removed through refactoring steps, with the aim to improve software quality and maintenance. Consequently, code smells are captured by popular static analysis tools,

Defect

Community	Dataset	# of instances		# metrics	Nature
		Total	Bugs (%)		
AEEEM	EQ	325	129 (39.81)	61	Class
	JDT	997	206 (20.66)		
	LC	399	64 (9.26)		
	ML	1826	245 (13.16)		
	PDE	1492	209 (13.96)		
Relink	Apache	194	98 (50.52)	26	File
	Safe	56	22 (39.29)		
	ZXing	399	118 (29.57)		
Apache	Ant	1692	350 (20.69)	20	Class
	Ivy	704	119 (16.90)		
	Camel	2784	562 (20.19)		
	Poi	1378	707 (51.31)		
	Jedit	1749	303 (17.32)		
	Log4j	449	260 (57.91)		
	Lucene	782	438 (56.01)		
	Velocity	639	367 (57.43)		
	Xalan	3320	1806 (54.40)		
	Xerces	1643	654 (39.81)		

Code Smells

Community	Dataset	# of instances		# metrics	Nature
		Samples	Smelly (%)		
Feature Envy	wct	25	18 (72.0)	83	Method
	itext	15	7 (47.0)		
	hsqldb	12	8 (67.0)		
	nekohtml	10	3 (30.0)		
	galleon	10	3 (30.0)		
	sunflow	9	1 (11.0)		
	emma	9	3 (33.0)		
	mvnforum	9	6 (67.0)		
	jasml	8	4 (50.0)		
	xmojo	8	2 (25.0)		
	jhotdraw	8	2 (25.0)		
God Class	fitjava	27	2 (7.0)	62	Class
	wct	24	15 (63.0)		
	xerces	17	11 (65.0)		
	hsqldb	15	13 (87.0)		
	galleon	14	6 (43.0)		
	xalan	12	6 (50.0)		
	itext	12	6 (50.0)		
	drjava	9	4 (44.0)		
	mvnforum	9	2 (22.0)		
	jpf	8	2 (25.0)		
	freecol	8	7 (88.0)		

Effort Estimation

Community	Dataset	Samples	Range (min-max)	# metrics
Effort	coc10	95	3.5 - 2673	24
	nasa93	93	8.4 - 8211	
	coc81	63	5.9 - 11400	
	nasa10	17	320 - 3291.8	
	cocomo	12	1 - 22	

Issue Lifetime

Community	Dataset	# of instances		# metrics
		Total	Closed (%)	
camel	1 day	5056	698 (14.0)	18
	7 days		437 (9.0)	
	14 days		148 (3.0)	
	30 days		167 (3.0)	
cloudstack	1 day	1551	658 (42.0)	18
	7 days		457 (29.0)	
	14 days		101 (7.0)	
	30 days		107 (7.0)	
cocoon	1 day	2045	125 (6.0)	18
	7 days		92 (4.0)	
	14 days		32 (2.0)	
	30 days		45 (2.0)	
node	1 day	2045	125 (6.0)	18
	7 days		92 (4.0)	
	14 days		32 (2.0)	
	30 days		45 (2.0)	
deeplearning	1 day	1434	931 (65.0)	18
	7 days		214 (15.0)	
	14 days		76 (5.0)	
	30 days		72 (5.0)	
hadoop	1 day	12191	40 (0.0)	18
	7 days		65 (1.0)	
	14 days		107 (1.0)	
	30 days		396 (3.0)	
hive	1 day	5648	18 (0.0)	18
	7 days		22 (0.0)	
	14 days		58 (1.0)	
	30 days		178 (3.0)	
ofbiz	1 day	6177	1515 (25.0)	18
	7 days		1169 (19.0)	
	14 days		467 (8.0)	
	30 days		477 (8.0)	
qpid	1 day	5475	203 (4.0)	18
	7 days		188 (3.0)	
	14 days		84 (2.0)	
	30 days		178 (3.0)	

Figure 2.4 Datasets from 4 chosen domains.

like PMD[3], CheckStyle[4], FindBugs[5], and SonarQube[6]. Until recently, most detection tools for code smells make use of detection rules based on the computation of a set of metrics, e.g.,

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

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

Figure 2.5 Static code metrics used in defects and code smells data sets.

well-known object-oriented metrics. These metrics are then used to set some thresholds for the detection of a code smell. But these rules lead to far too many false positives making it difficult for practitioners to refactor code [Kri16a].

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

Figure 2.6 Metrics used in issue lifetimes data.

Personnel		Product		System		Other	
Label	Description	Label	Description	Label	Description	Label	Description
ACAP	Analyst Capability	CPLX	Prod. Complexity	DATA	Database size	DOCU	Documentation
APEX	Applications Exp.	SCED	Dedicated Schedule	PVOL	Platform volatility	TOOL	Use of software tools
LEXP	Language Exp.	SITE	Multi-side dev.	RELY	Required Reliability		
MODP	Modern Prog. Practices	TURN	turnaround time	RUSE	Required Reuse		
PCAP	Programmer Capability			STOR	% RAM		
PLEX	Platform Exp.			TIME	% CPU time		
VEXP	Virtual Machine Exp.			VIRT	Machie volatility		
PCON	Personnel Continuity						

Figure 2.7 Metrics used in effort estimation dataset.

Recently, the research community is changing rapidly in terms of defining novel methodologies that incorporate additional information to detect code-smells. Much progress has been made in towards adopting machine learning tools to classify code smells from examples, easing the build of automatic code smell detectors, thereby providing a better-targeted detection. Kreimer [Kre05] proposes an adaptive detection to combine known methods for finding design flaws Large Class and Long Method on the basis of metrics. Khomh et al. [Kho09] proposed a Bayesian approach to detect occurrences of the Blob antipattern on open-source programs. Khomh et al. [Kho11] also presented BDTEX, a GQM approach to build Bayesian Belief Networks from the definitions of antipatterns. Yang et al. [Yan12] study the judgment of individual users by applying machine learning algorithms on code clones. These studies were not included in our comparison as the data was not readily available for us to reuse.

More recently, Fontana et al. [Arc16] in their study of several code smells, considered 74 systems for their analysis and validation. They experimented with 16 different machine learning algorithms. They made available their dataset, which we have adapted for our applications in this study. These datasets were generated using the Qualitas Corpus (QC) of systems [Tem10]. The Qualitas corpus is composed of 111 systems written in Java, characterized by different sizes and belonging to different application domains. Fontana et al. [Arc16] selected a subset of 74 systems for their analysis. The authors computed a large set of object-oriented metrics belonging to class, method, package, and project level. A detailed list of metrics and their definitions are available in appendices of [Arc16]. The code smells repository we use comprises of 22 datasets for two different code smells: Feature envy and God Class. The God Class code smell class level code smell that refers to classes that tend to centralize the intelligence of the system. Feature Envy is a method level smell that tends to use many attributes of other classes (considering also attributes accessed through accessor methods).

The number of samples in these datasets are particularly small. For our analysis, we retained only datasets with at least 8 samples so that the transfer learners used here function reliably. This lead us to a total of 22 datasets shown in Figure 2.4.

2.3.2 Issue Lifetime Estimation

Open source projects use issue tracking systems to enable effective development and maintenance of their software systems. Typically, issue tracking systems collect information about system failures, feature requests, and system improvements. Based on this information and actual project planing, developers select the issues to be fixed. Predicting the time it may take to close an issue has multiple benefits for the developers, managers, and

stakeholders involved in a software project. Predicting issue lifetime helps software developers better prioritize work; helps managers effectively allocate resources and improve consistency of release cycles; and helps project stakeholders understand changes in project timelines and budgets. It is also useful to be able to predict issue lifetime specifically when the issue is created. An immediate prediction can be used, for example, to auto-categorize the issue or send a notification if it is predicted to be an easy fix.

As an initial attempt, Panjer [Pan07] used logistic regression models to classify bugs as closing in 1.4, 3.4, 7.5, 19.5, 52.5, and 156 days, and greater than 156 days. He was able to achieve an accuracy of 34.9%. Giger et al. [Gig10] used models constructed with decision trees to predict for issue lifetimes in Eclipse, Gnome, and Mozilla. They were able obtain a peak precision of 65% by dividing time into 1, 3, 7, 14, 30 days. Zhang et al. [Zha13] developed a comprehensive system to predict lifetime of issues. They used a Markov model with a kNN-based classifier to perform their prediction. More recently, Rees-Jones et al [Rj17] showed that using Hall's CFS feature selector and C4.5 decision tree learner a very reliable prediction of issue lifetime could be made.

Figure 2.4 shows a list of 8 projects used to study issue lifetimes. These projects were selected by our industrial partners since they use, or extend, software from these projects. It forms a part of an ongoing study on prediction of issue lifetime by Rees-Jones et al. [Rj17]. The authors note that one issue in preparing their data was a small number of *sticky* issues. They define sticky issues as one which was not yet closed at the time of data collection. As recommended by Rees-Jones et al. [Rj17], we removed these sticky issues from our datasets.

In raw form, the data consisted of sets of JSON files for each repository, each file contained one type of data regarding the software repository (issues, commits, code contributors, changes to specific files). In order to extract data specific to issue lifetime, we did similar preprocessing and feature extraction on the raw datasets as suggested by [Rj17].

2.3.3 Effort Estimation

The nature of effort estimation and the corresponding data is unlike that of other domains. Firstly, while domains like defect prediction datasets often store several thousand samples of defective and non-defective samples, effort data is usually smaller with only a few dozen samples at most. Secondly, unlike defect dataset or code smells, effort is measured using, say *man-hours*, which is a continuous variable. These differences requires us to significantly modify existing transfer learning techniques to accommodate this kind of data.

Transfer learning attempts have been made in defect prediction before albeit with limited success. Kitchenham et al. [Kit07b] reviewed 7 published transfer studies in effort estimation. They found that in most cases, transferred data generated worse predictors than using within-project information. Similarly, Ye et al. [Yan11] report that the tunings to Boehm's COCOMO model have changed radically for new data collected in the period 2000 to 2009. Kocaguneli et al. [Koc15] used analogy-based effort estimation with relevancy filtering using a method called TEAK for studying transfer learning in effort estimation. He found that it outperforms other approaches such as linear regression, neural networks, and traditional analogy-based reasoners. Since then, however, newer more sophisticated transfer learners have been introduced. Krishna et al. [Kri16b] suggest that relevancy filtering (for defect prediction tasks) would never have been necessary in the first place if researchers had instead hunted for bellwethers. Therefore, in this work, we revisit transfer learning in effort estimation keeping in mind these changing trends.

For our experiments, we consider effort estimation data expressed in terms of the COCOMO ontology: 23 attributes describing a software project, as well as aspects of its personnel, platform, and system features (see Figure 2.7 for details). The data is gathered using Boehm's 2000 COCOMO model. The data was made available by Menzies et al. [Men16] who

show that this model works better than (or just as well as) other models they've previously studied. We use 5 datasets shown in Figure 2.4. Here, COC81 is the original data from 1981 COCOMO book [Boe81b]. This comes from projects dated from 1970 to 1980. NASA93 is NASA data collected in the early 1990s about software that supported the planning activities for the International Space Station. The other datasets are NASA10 and COC05 (the latter is proprietary and cannot be released to the research community). The non-proprietary data (COC81 and NASA93 and NASA10) are available at http://tiny.cc/07wvjy.

2.3.4 Defect Prediction

Human programmers are clever, but flawed. Coding adds functionality, but also defects, so software will crash (perhaps at the most awkward or dangerous time) or deliver wrong functionality. Since programming introduces defects into programs, it is important to test them before they are used. Testing is expensive. According to Lowry et al. software assessment budgets are finite while assessment effectiveness increases exponentially with assessment effort [Low98]. Exponential costs quickly exhaust finite resources, so standard practice is to apply the best available methods only on code sections that seem most critical. One such approach is to use defect predictors learned from static code attributes. Given software described in the attributes of Figures 2.5, 2.6, and 2.7, data miners can learn where the probability of software defects is highest. These static code attributes can be automatically collected, even for very large systems [NB05]. Although other methods like manual code reviews are much more accurate in identifying defects, they take much higher effort to find a defect and also are relatively slower. For example, depending on the review methods, 8 to 20 LOC/minute can be inspected and this effort repeats for all members of the review team, which can be as large as four or six people [Men02]. This is complementary

to defect prediction techniques. These techniques enable developers to target defect-prone areas faster, but do not guide developers toward a particular fix. The defect prediction models are easier to use in that sense that they prioritize *both* code review and testing resources (these areas complement each other).

Moreover, defect predictors often find the location of 70% (or more) of the defects in code [Men07a]. Defect predictors have some level of generality: predictors learned at NASA [Men07a] have also been found useful elsewhere (e.g. in Turkey [Tos10; Tos09]). The success of this method in predictors in finding bugs is markedly higher than other currently-used industrial methods such as manual code reviews. For example, a panel at *IEEE Metrics 2002* [Shu02] concluded that manual software reviews can find ≈60% of defects. In another work, Raffo documents the typical defect detection capability of industrial review methods: around 50% for full Fagan inspections [Fag76] to 21% for less-structured inspections.

Not only do static code defect predictors perform well compared to manual methods, they also are competitive with certain automatic methods. A recent study at ICSE'14, Rahman et al. [Rah14] compared (a) static code analysis tools FindBugs, Jlint, and Pmd and (b) static code defect predictors (which they called "statistical defect prediction") built using logistic regression. They found no significant differences in the cost-effectiveness of these approaches. Given this equivalence, it is significant to note that static code defect prediction can be quickly adapted to new languages by building lightweight parsers that find information like Figure 2.5. The same is not true for static code analyzers– these need extensive modification before they can be used on new languages.

For the above reasons, researchers and industrial practitioners use static attributes to guide software quality predictions. Defect prediction has been favored by most transfer learning researchers. Further, defect prediction models have been reported at Google [Lew13]. Verification and validation (V&V) textbooks [Rak01] advise using static code complexity

attributes to decide which modules are worth manual inspections.

The defect dataset we have used come from 18 projects grouped into 3 communities taken from previous transfer learning studies. The projects measure defects at various levels of granularity ranging from function-level to file-level. Figure 2.4 summarizes all the communities of datasets used in our experiments.

For the reasons discussed in §2.1, we explore homogeneous transfer learning using the attributes shared by a community. That is, this study explores intra-community transfer learning and not cross-community heterogeneous transfer learning.

The first dataset, AEEEM, was used by [NK15]. This dataset was gathered by D'Amborse et al. [D'A12], it contains 61 metrics: 17 object-oriented metrics, 5 previous-defect metrics, 5 entropy metrics measuring code change, and 17 churn-of-source code metrics.

The RELINK community data was obtained from work by Wu et al. [Wu11] who used the Understand tool [7], to measure 26 metrics that calculate code complexity in order to improve the quality of defect prediction. This data is particularly interesting because the defect information in it has been manually verified and corrected. It has been widely used in defect prediction [NK15][Wu11][Bas96b][OA96][Kim11].

In addition to this, we explored two other communities of datasets from the SEACRAFT repository[8]. The group of data contains defect measures from several Apache projects. It was gathered by Jureczko et al. [JM10]. This dataset contains records the number of known defects for each class using a post-release bug tracking system. The classes are described in terms of 20 OO metrics, including CK metrics and McCabes complexity metrics. Each dataset in the Apache community has several versions. There are a total of 38 different datasets. For more information on this dataset see [Kri16a].

[7] http://www.scitools.com/products/
[8] https://zenodo.org/communities/seacraft/

2.4 Evaluation

2.4.1 Evaluating Transfer Learners

2.4.1.1 Evaluation for Continuous Classes

For the effort estimation data in Figure 2.4, the dependent attribute is development effort, measured in terms of calendar hours (at 152 hours per month, including development and management hours). For this, we use general machine learning algorithms as a regressor instead of a classifier.

To evaluate the quality of the learners used for regression, we make use of Standardized Accuracy (SA). The use of SA has been endorsed by several researchers in SE [SM12a; Lan16] Standard Accuracy is computed as below:

$$SA = 1 - \frac{MAR}{\frac{2}{n^2}\sum_{i=1}^{n}\sum_{j=1}^{j<i}|y_i - y_j|} \times 100 \tag{2.1}$$

Where, MAR is the mean of the absolute error for the predictor of interest. E.g. for software project estimation, the average of the absolute difference between the effort predicted and the actual effort the project took.

Higher values of SA are considered to be *better*. Note: Some researchers have endorsed the use other metrics such as MMRE to measure the quality of regressor in effort estimation. We have made available a replication package[9] with this and other metrics. Interested readers are encouraged to use these.

[9] https://goo.gl/jCQ1Le

2.4.1.2 Evaluation for Discrete Classes

In the context of discrete classes, we define positive and negative classes. With defects, instances with one or more defects are considered to belong to the "positive class" and non-defective instances are considered to belong to the "negative class". Similarly in code smell detection (smelly samples belong to "positive class") and in issue lifetime estimation (closed issues belong to "positive class"). Prediction models are not ideal, they therefore need to be evaluated in terms of statistical performance measures.

For classification problems we construct a confusion matrix, with this we can obtain several performance measures such as: (1) *Accuracy*: Percentage of correctly classified classes (both positive and negative); (2) *Recall or pd*: percentage of the target classes (defective instances) predicted. The higher the pd the better ; (3) *False alarm or pf*: percentage of non-defective instances wrongly identified as defective. Unlike pd, lower the pf implies better quality; (4) *Precision*: probability of predicted defects being actually defective. Either a smaller number of correctly predicted faulty modules or a larger number of erroneously predicted defect-free modules would result in a low precision.

There are several trade-offs between the metrics described above. There is a trade-off between recall rate and false alarm rate where attempts to increase recall leads to larger false alarm, which is undesirable. There is also a trade-off between precision and recall where increasing precision lowers recall and vice-versa. These measures alone do not paint a complete picture of the quality of the predictor. Therefore, it is very common to apply performance metrics that incorporate a combination of these metrics. As a result, some authors generally resort to using metrics such as F1 score to assess learners [Fu16b; Kim08; Men07a; Wan16b]. However, there exists a peculiar challenge with using F-measure that is specific to some software engineering problem – the large imbalance between class

variables in the datasets commonly studied here. For instance, consider the datasets studied in this work shown in Figure 2.4. There, a number of datasets have highly skewed samples. In these cases, several researchers caution against use of common performance metrics such as precision or F-measure. Menzies et al. [Men07c] in their 2007 paper showed the negative impact of using these metrics. They caution researchers against the use precision when assessing their detectors. They recommend other more stable measures especially for highly skewed data sets. This concern is echoed by several other researchers in SE [Cha03; KM97; Sha10c]. Kubat & Matwin found that the effect of the negative classes (in our context this refers to bug-free/smell-free/closed issues) has a profound impact on the outcome of these metrics. As a remedy, these authors recommend a new evaluation scheme that combines reliable metrics such as recall (pd) and false-alarm (pf).

One such approach that can combine these metrics is to build a *Receiver Operating Characteristic (ROC)* curve. ROC curve is a plot of Recall versus False Alarm pairing for various predictor cut-off values ranging from 0 to 1. The best possible predictor is the one with an ROC curve that rises as steeply as possible and plateaus at pd=1. Ideally, for each curve, we can measure the *Area Under Curve (AUC)*, to identify the best training dataset. Unfortunately, building an ROC is not straight forward in our case. We have used Random Forest for predicting defects owing to it's superior performance over several other predictors [Les08]. Note that Random Forest lacks a threshold parameter, since this threshold parameter is required in order to generate a set of points to plot the ROC curve, Random Forest is not capable of producing an ROC curve, instead we produce just one point on the ROC curve. It is therefore not possible to compute AUC.

In a previous work, Ma and Cukic [MC07] have shown that other metrics that measure the distance from perfect classification can be substituted for AUC in cases where a ROC curve cannot be generated. Accordingly, we use the the "G-Score" for combining Pd and Pf.

Several authors [Men07c; Sha10c] have previously shown that such a measure is justifiably better than other measures when the test samples have imbalanced distribution in terms of classes. G-Score can by computed by measuring the mean (geometric/harmonic) between the Probability of True Positives (Pd) and Probability of true negatives (1-Pf). The choice of using geometric mean or harmonic mean depends on the variance in Pd/Pf values. Mathematically, it is known that in cases where samples tend to take extreme values (such as Pd=0 or Pf=1) harmonic mean provides estimates that are much more stable and also more conservative in it's estimate compared to geometric mean [Xia99]. Therefore, we propose the use of G-Score, measured as follows:

$$G = \frac{2 \times Pd \times (1 - Pf)}{1 + Pd - Pf} \tag{2.2}$$

In this work, for the sake of consistency with other SE literature, we report the measures of Pd and Pf reported in terms of the G-Score. Also, note that with the formulation in Equation 2.2, *larger* G-scores are better.

2.4.2 Evaluating Planners

It can be somewhat difficult to judge the effects of applying plans to software projects. These plans cannot be assessed just by a rerun of the test suite for three reasons: (1) The defects were recorded by a post release bug tracking system. It is entirely possible it escaped detection by the existing test suite; (2) Rewriting test cases to enable coverage of all possible scenarios presents a significant challenge; and (3) It may take a significant amount of effort to write new test cases that identify these changes as they are made.

To resolve this problem, SE researchers such as Cheng et al. [CJ10], O'Keefe et al. [OC08; OC07], Moghadam [Mog11] and Mkaouer et al. [Mka14] use a *verification oracle* learned

separately from the primary oracle. This oracles assesses how defective the code is before and after some code changes. For their oracle, Cheng, O'Keefe, Moghadam and Mkaouer et al. use the QMOOD quality model [BD02].

A shortcoming of QMOOD is that quality models learned from other projects may perform poorly when applied to new projects [Men13]. Hence, we eschew older quality models like QMOOD and propose a verification oracle based on the *overlap*. between two sets: (1) The changes that developers made, perhaps in response to the issues raised in a post-release issue tracking system; and (2) Plans recommended by an automated planning tool such as XTREE/BELLTREE. Using these two sources of changes, it is possible to compute the extent to which a developer's action matches that of the actions recommended by planners. This is measured using *overlap*:

$$Overlap = \frac{|\mathbb{D} \cap \mathbb{P}|}{|\mathbb{D} \cup \mathbb{P}|} \times 100 \tag{2.3}$$

That is, we measure overlap using the size of the intersections divided by the size of the union of the *changes*. Here $\mathbb{D}$ represents the changes made by the *developers* and $\mathbb{P}$ represents the changes recommended by the *planner*. Accordingly, the larger the intersection between the changes made by the developers to the changes recommended by the planner, the greater the overlap.

As an example, consider Figure 5.5; there we have 2 sets of changes: (1) Changes made by developers ($\mathbb{D}$), and (2) Changes recommended by the planner ($\mathbb{P}$). In each case we have 3 possible actions for every metric: (1) Make no change ('·'), (2) Increase ('+'), and (3) Decrease ('—'). The intersection of the changes represents the number of times the actions taken by the developers is the same as the actions recommended by the planner. This the above example, the intersection, $\mathbb{D} \cap \mathbb{P} = 7$, out of a total of $\mathbb{D} \cup \mathbb{P} = 9$ possible actions. This

	DIT	NOC	CBO	RFC	FOUT	WMC	NOM	LOC	LCOM
Planner ($\mathbb{P}$)	·	·	·	+	·	+	+	+	+
Developer ($\mathbb{D}$)	·	·	−	+	−	+	+	+	+

$$Overlap = \frac{|\mathbb{D} \cap \mathbb{P}|}{|\mathbb{D} \cup \mathbb{P}|} \times 100 = \frac{7}{9} \times 100 = 77.77\%$$

Figure 2.8 A simple example of computing overlap. Here a '+' represents an *increase*, a '−' represents a *decrease*, and a '·' represents *no-change*. Columns shaded in blue indicate a match between developer's change and the recommendation made by a planner.

leads to $Overlap = \frac{7}{9} \times 100 = 77.77\%$.

2.4.3 Statistics

To overcome the inherent randomness introduced by the experiments, we use 30 repeated runs, each time with a different random number seed (we use 30 since that is the minimum needed samples to satisfy the central limit theorem). Researchers have endorsed the use of repeated runs to gather reliable evidence [Vau12]. Thus, we repeat the whole experiment independently several times to provide evidence that the results are reproducible. The repeated runs provide us with a sufficiently large sample size (of size 30) to statistically compare all the datasets.

Each repeated run collects the values of Pd and Pf which are then used to estimate the G-Score using Equation 2.2. (Note: We refrain from performing a cross validation because the process tends to mix the samples from training data (the source) and the test data (other target projects), which defeats the purpose of this study.)

To rank these 30 numbers collected as above, we use the Scott-Knott test recommended by Mittas and Angelis [MA13]. Scott-Knott is a top-down clustering approach used to rank different treatments. If that clustering finds an statistically significant splits in data, then

some statistical test is applied to the two divisions to check if they are statistically significant different. If so, Scott-Knott recurses into both halves.

To apply Scott-Knott, we sorted a list of $l = 40$ values of Equation 2.2 values found in $ls = 4$ different methods. Then, we split l into sub-lists m, n in order to maximize the expected value of differences in the observed performances before and after divisions. E.g. for lists l, m, n of size ls, ms, ns where $l = m \cup n$:

$$E(\Delta) = \frac{ms}{ls} abs(m.\mu - l.\mu)^2 + \frac{ns}{ls} abs(n.\mu - l.\mu)^2$$

We then apply a statistical hypothesis test H to check if m, n are significantly different. In our case, the conjunction of bootstrapping and A12 test. Both the techniques are non-parametric in nature, i.e., they do not make gaussian assumption about the data. As for hypothesis test, we use a non-parametric bootstrapping test as endorsed by Efron & Tibshirani [ET93]. Even with statistical significance, it is possible that the difference can be so small as to be of no practical value. This is known as a "small effect". To ensure that the statistical significance is not due to "small effect" we use effect-size tests in conjunction with hypothesis tests. A popular effect size test used in SE literature is the A12 test. It has been endorsed by several SE researchers [LO02; PC10; AB11; SM12b; Kam07; Koc13]. It was first proposed by Vargha and Delany [VD00b]. In our context, given the performance measure G, the A12 statistics measures the probability that one treatment yields higher G values than another. If the two algorithms are equivalent, then A12 = 0.5. Likewise if $A12 \geq 0.6$, then 60% of the times, values of one treatment are significantly greater that the other. In such a case, it can be claimed that there is *significant effect* to justify the hypothesis test H. In our case, we divide the data if *both* bootstrap sampling and effect size test agree that a division is statistically significant (with a confidence of 99%) and not a small effect

($A12 \geq 0.6$). Then, we recurse on each of these splits to rank G-scores from best to worst.

CHAPTER

3

BELLWETHERS

3.1 Why use Bellwethers?

A premise of software data analytics is that there exists data from which we can learn models. When local data is scarce, sometimes it is possible to use data collected from other projects either at the local site, or other sites. That is, when building software quality predictors, it might be best to look at more than just the local data. To do this, recent research has been exploring the problem of *transferring* data from one project to another for the purposes of data analytics. These research have focused on two methodological variants of transfer learning: (a) dimensionality transform based techniques by Nam, Jing et al. [Nam13; NK15;

Jin15] and (b) the similarity based approaches of Kocaguneli, Peters and Turhan et al. [KM11; Koc15; Tur09; Pet15]. One problem with transfer learning is *conclusion instability* which may be defined as follows:

The more data we inspect from more projects, the more our conclusions change.

The problem with conclusion instability is that the assumptions used to make prior policy decisions may no longer hold. Conclusion instability in software engineering, specifically in transfer learning, is well documented. For example, Zimmermann et al. [Zim09b] learned defect predictors from 622 pairs of projects *project1, project2*. In only 4% of pairs, predictors from *project1* worked on *project2*. Also, Turhan [Men11b] studied defect prediction results from 28 recent studies, most of which offered widely differing conclusions about what most influences software defects.

From the perspective of transfer learning, this instability means that learners that rely on the source of data would also become unreliable. Conclusion instability is very unsettling for software project managers struggling to find general policies. Hassan [Has17] cautions that managers lose faith in the results of software analytics if those 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; (c) where to focus expensive testing procedures; (d) what return-on-investment might be expected after purchasing an expensive tool; etc.

How to support those managers, who seek stability in their conclusions, while also allowing new projects to take full benefit of the data from recent projects? Perhaps if we cannot *generalize* from all data, a more achievable goal is to *slow* the pace of conclusion change. While it may be a fool's errand to wait for globally stable SE conclusions, one approach is to declare one project as the *"bellwether"*[1] which should be used to make

[1]According to the Oxford English Dictionary, the "bellwether" is the leading sheep of a flock, with a bell on

conclusions about all other projects. Note that conclusions are stable for as long as this bellwether continues to be the best oracle for that community. This "bellwether" project would also act as an excellent source to perform transfer learning.

In this chapter, we first identify a *bellwether effect* and show that it may be used to generate stable conclusions. We offer the following definition:

- *The bellwether effect* states that when a community works on software, then there exists one exemplary project, called the bellwether, which can define predictors for the others.

From this bellwether effect we show that we may construct a *baseline transfer learner* called the *BEETLE* to benchmark other more complex transfer learners. In other words,

- BEETLE searches for the exemplar bellwether project and constructs a transfer learner with it. This transfer learner is then used to predict for effects in future data for that community.

This chapter attempts to make the following empirical, methodological, and pragmatic contributions. *Empirically*, the key contribution of this chapter is the discovery that simple methods can find general conclusions across multiple SE projects. While we *cannot* show that this holds for *all* SE domains, we can report that it has offer satisfactory results *on three out of the four domains that we have studied so far*; i.e. code smell detection, effort estimation, and defect prediction. In one of our domains, issue lifetime estimation, the evidence supporting the usefulness bellwethers was unsatisfactory. But our results show that seeking bellwethers may be a simple starting point to begin to reason about software projects.

Pragmatically, we assert that simple methods should always be preferred to more complex ones– particularly if we hope for those methods to be used widely in the industry. Other researchers agree with our assertion. In a recent paper, Xu et al. [Xu15a] discuss the

its neck.

cost of increasing software complexity: as complexity increases users use fewer and fewer of the available configuration options; i.e. they tend to utilize less and less of the power of that software. This is relevant to transfer learning since standard methods, other than bellwethers, come with so many configuration options that even skilled users have trouble exploiting them.

Methodologically, the simplicity associated with the discovery and the use of bellwethers is encouraging for further research in software engineering. Initial experiments with transfer learning in SE built quality predictors from the *union* of data taken from multiple projects. That approach lead to poor results, so researchers turned to *relevancy filters* to find what small subset of the data was relevant to the current problem [Tur09] and then the dimensionality transform methods of Nam, Jing et al were developed. In this chapter, we demonstrate the use “bellwethers" as a baseline transfer learning method for software analytics. As described in the next section, bellwethers have all the properties desirable for a baseline method such as simplicity of implementation and broad applicability. In the case of transfer learning, such a baseline would have greatly assisted in justifying the need for increasingly complex methods [Nam13; NK15; Jin15; KM11; Koc15; Tur09; Pet15].

While we cannot claim that such simple baselines are always better (they fail in the case of issue lifetime estimation), the experiments of this chapter demonstrate that in some cases other cases (code smell detection and effort estimation) bellwethers can perform better than more complex algorithms.

3.2 Baselining with Bellwethers

Different domains can require different approaches. According to Wolpert & Macready [WM97], no single algorithm can ever be best for all problems. They caution that for every class of

problem where algorithm *A* performs best, there is some other class of problems where *A* will perform poorly. Hence, when commissioning a transfer learner for a new domain, there is always the need for some experimentation to match the particulars of the domain to particular transfer learning algorithms.

When conducting such commissioning experiments, it is methodologically useful to have a *baseline* method; i.e. an algorithm which can generate *floor performance* values. Such baselines let a developer quickly rule out any method that falls "below the floor". With this, researchers and industrial practitioners can achieve fast early results, while also gaining some guidance in all their subsequent experimentation (specifically: "try to beat the baseline").

Using baselines for analyzing algorithms has been endorsed by several experienced researchers. For example, in his textbook on empirical methods for artificial intelligence, Cohen [Coh95] strongly recommends comparing supposedly sophisticated systems against simpler alternatives. In the machine learning community, Holte [Hol93] uses the OneR baseline algorithm as a *scout* that runs ahead of a more complicated learners as a way to judge the complexity of up-coming tasks. In the software engineering community, Whigham et al. [Whi15] recently proposed baseline methods for effort estimation (for other baseline methods in effort estimation, see Mittas et al. [MA13]). Shepperd and Macdonnel [SM12c] argue convincingly that measurements are best viewed as ratios compared to measurements taken from some minimal baseline system. Work on cross-versus within-company cost estimation has also recommended the use of some very simple baseline (they recommend regression as their default model) [Kit07b].

In their recent article on baselines in software engineering, Whigham et al. [Whi15] propose guidelines for designing a baseline implementation that include:

1. Be *simple* to describe and implement;

2. Be *applicable to a range of models*;
3. Be *publicly available* via a reference implementation and associated environment for execution;

In addition to this, we suggest that baselines should also:

4. Offer *comparable performance* to standard methods. While we do not expect a baseline method to out-perform all state-of-the-art methods, for a baseline to be insightful, it needs to offer a level of performance that often approaches the state-of-the-art.

We note that the use of *bellwether method* for transfer learning satisfies all the above criteria. The *bellwether method* is very simple in that it just uses the bellwether dataset to construct a prediction model (without any further complex data manipulation).

As to being *applicable to a wide range of domains*, in this work we apply the bellwether method to several sub-domains in SE, i.e., code-smell detection, effort estimation, issue lifetime estimation, and defect prediction.

As to *public availability*, a full implementation of bellwethers including all the case studies presented here (including working implementations of other transfer learning algorithms and our evaluation methods) are available on-line.

In terms of *comparative performance*, for each model, we compared the bellwether method's performance against the the established state-of-the-art transfer learners reported in the literature. In those comparative results, bellwethers were usually as good, and sometimes even a little better, than the state-of-the art.

The use of bellwethers benefits practitioners and researchers attempting transfer learning in several ways:

1. Researchers can use results of bellwethers as the "sanity checker". Experiments shows that the use of bellwethers for transfer learning is comparable to, and in some cases better than, other complex transfer learners. Consequently, when designing new transfer

learners, researchers can compare their results to bellwether's as a baseline.

2. Practitioners can also use bellwethers as an "off-the-shelf" transfer learner. For example, in three out of the four domains studied here (code-smells, issue lifetimes, effort estimation), there are no established transfer learners. In such cases, we show that practitioners can simply use bellwethers as transfer learners instead of having to develop new transfer learner (or adapt existing ones from other domains).

3.3 Research Questions

This section lists the questions that will guide our investigation.

RQ-3.1: How prevalent are "Bellwethers"?

Motivation: If bellwethers occur infrequently, we cannot rely on them. Hence, this question explores how common bellwethers are.

Approach: To answer this question, we explore four SE domains: defect prediction, effort estimation, issue lifetime estimation, and detection of code smells. Each domain contains multiple "communities" of datasets. For each domain, we ensured that the datasets were as diverse as possible. To this end, data was gathered according to the following rules:

- The data has been used in a prior paper. Each of our datasets for defects, code smells, effort estimation, and issue lifetime estimation has been used previously;
- The communities are quite diverse; e.g. the NASA projects from the effort estimation datasets are proprietary while the others are open source projects. Similarly, the God Class is a class level smell and Feature Envy is a method level design smell.
- In addition, where relevant, the projects also vary in their granularity of data description (in case of defect prediction, we have defects at file, class, or at a function level granularity).

Results: In a result consistent with bellwethers being prevalent, we find that three out of these four domains have a bellwether dataset; i.e. a single dataset from which a superior quality predictor can be generated for the rest of that community.

RQ-3.2: How does the bellwether dataset fare against within-project data?

Motivation: One premise of transfer learning is that using data from other projects are as useful, or better, than using data from within the same project. This research questions tests that this premise holds for bellwethers.

Approach: To answer this question, we reflect on datasets with temporal within-project data. One of our communities in defect prediction (APACHE) comes in multiple versions. Here, each version is a historical release where version i was written before version j where $j > i$. For this community, RQ-3.2 was explored as follows:

- The last version (version N) of each project was set aside as a hold-out.
- Using an older version ($N-1$) we find the bellwether dataset.
- A defect predictor was then constructed on the bellwether dataset.
- The predictor was applied to the latest version (version N).

We compare the above to using the *within-project* data; i.e. for each project:

- The last version (N) of that project was set aside as a hold-out;
- The older version ($N-1$) of that project was then used to train a defect predictor.
- The predictor was then applied to the latest data (N).

Results: In our experiments, the bellwether predictions proved to be as good or better than, those generated from the local data. Note that, as of now, this has been verified only in defect prediction.

RQ-3.3: How well do transfer learners perform across different domains?

Motivation: Our reading of the literature is that for homogeneous transfer learning, the current state of the art is to use TCA+. However, note that this result has only been reported for defect prediction and only for a limited number of datasets. In our previous work we reported that Bellwether was better than relevancy based filtering methods. Here we ask if this is true given newer transfer learning methods and different datasets.

Approach: To answer this question, we compare the "bellwether" method [Kri16b] against 3 other standard transfer learners: (1) TCA+ [Nam13]; (2) Transfer Naive Bayes [Ma12]; and (3) Value Cognitive Boosting [Ryu16]. In addition we modify these learners appropriately for different sub-domains under study.

Results: Our simple *bellwether method's* predictions were observed to be superior than those of other transfer learners in two domains: effort estimation and code smell detection. *Bellwether method's* predictions were a close second in defect prediction.

RQ-3.4: How much data is required to find the bellwether dataset?

Motivation: Our proposal to find bellwethers is to compare the performance of pairs of datasets from different projects in a round robin fashion. However, conclusion instability (as presented in §2.1.1) is a major issue in SE and the primary cause of such conclusion instability is the constant influx of new data [Eka09]. Given this, a natural question that arises from our experimental approach is the amount of data that is required to find the bellwether dataset given the influx of new data.

Approach: To answer this research question, we again consider datasets with historical versions of data similar to RQ-3.3. To discover how much bellwether data is required, we incrementally increase the size of the bellwether dataset. We stop increments when (a) we

notice no statistical improvement in using additional version data, or (b) we notice that there is a deterioration of performance scores using additional version data. Specifically, assuming that the bellwether project contains versions $1, ..., N$, we construct a prediction model with version N and measure the performance scores, then we repeat this by including versions N, $N-1$ and so on. With this, we hope to offer some empirical evidence as to how much data is required to discover the bellwether.

Results: Our experiments show that program managers need not wait very long to find their bellwethers – when there are multiple versions of the bellwether project, project managers need to only use the latest version of that project to perform analytics. Another interesting finding is that in cases with no historical logs, only a few hundred samples usually are sufficient for creating and testing candidate bellwethers.

RQ-3.5: How effectively do bellwethers mitigate for conclusion instability?

Motivation: In the previous research questions, we established the prevalence of bellwethers (RQ-3.1), we showed its efficacy in constructing a baseline transfer learner (RQ-3.3), and we also showed empirically that we can discover bellwethers early in the project's life-cycle (RQ-3.4). Since one of the primary motivation for seeking bellwethers is due to existence of conclusion instability, in this final research question, we ask how one might use the bellwether effect to mitigate the two sources of instability we identify in §2.1.1: (a) performance instability, and (b) source instability.

Approach: To answer this question, we take two steps:

- To verify if the bellwether effect can be used to mitigate performance variations, we reflect on the results of the comparison of various transfer learners (note that, these

also includes bellwethers as a baseline approach). First, we try to determine if different sources of data to construct the transfer learners produces variances in the performance. Then, we determine if the use of bellwethers can address these variances.

- Next, to verify if bellwethers can be used to derive stable lessons in the presence of a variety of data sources. We determine if using different sources of data can lead to different conclusions. Then, we determine how the use of bellwethers can offer stable conclusion.

Results: Our experiments show that all the datasets we have explored in the four domains studied here exhibit both performance instability and source instability. Performance instability causes large variances in performance scores of transfer learners depending on source of the data used. By using the bellwether effect, we may identify the bellwether data set which can then be used as a stable source to construct transfer learners. Further, we show that transfer learners constructed using the bellwether dataset offer statistically and significantly greater performance scores compared to other data sources. The existence of source instability causes different lessons to be derived from different data sources. Bellwether effect can be used to tackle this by identifying a bellwether dataset from the available data sources. The bellwether dataset can then be used to learn lessons. As long as the bellwether dataset remains unchanged, we will (a) obtain the same performance scores for a transfer learner, and (b) the same conclusions from the bellwether dataset.

3.4 Bellwethers in Software Engineering

Bellwethers offer a simple solution to mitigating conclusion instability. Rather than exploring all available data for some eternal conclusions in SE, we seek bellwether datasets that can offer stable solutions over longer stretches of time. When we notice the dataset failing, we may seek different bellwethers. In addition to this, the ability of bellwethers to

offer stable conclusions over long periods of time also simplifies another widely explored problem in SE; i.e., the problem of transfer learning. In this section, we discuss how we may simplify transfer learning by using bellwethers as a baseline transfer learner.

3.4.1 Bellwether Method

In the above section, we sampled some of the work on transfer learning in software engineering. This rest of this work asks the question "is the complexity of §2.1 really necessary?" We believe the answer is *no*. To assert this, we propose a framework that assumes some software manager has a watching brief over N projects (which we will call the *community* "C"). As part of those duties, they can access issue reports and static code attributes of the community. Using that data, this manager will apply the a framework described in Figure 3.1 which comprises of three operators– DISCOVER, TRANSFER, MONITOR.

1. DISCOVER: *Using cross-project data within a community, check if that community has a bellwether dataset.*
   - For all pairs of data from projects in a community $P_i, P_j \in C$;
   - Predict for defects/smells/issue-lifetime/effort in P_j using prediction model from data taken from P_i;
   - A bellwether exists if one P_i generates the most accurate predictions in a majority of $P_j \in C$.
2. TRANSFER: *Using the bellwether, generate prediction models on new project data.* That is, having learned the bellwether on past data, we now apply it to future projects.
3. MONITOR: *Go back to step 1 if the performance statistics seen for new projects during TRANSFER start decreasing.* Specifically,
   - As new data arrives to the projects in a community ...

Figure 3.1.A: Discover
Discover the bellwether dataset for a given community. In a community C, for all pairs of data from projects $P_i, P_j \in C$, do the following: Construct a prediction model with data from project P_i and predict for the target variable in P_j using this model. Note: The term target variable refers to defects, code-smells, issue lifetime, or effort, depending on the community under consideration. Report a bellwether if one P_i generates the best predictions in a majority of $P_j \in C$. Note: The quality of prediction is measured using G-Score for defect-prediction, code smell estimation, and issue-lifetime estimation and by SA for effort estimation.

```
def discover(datasets):
"Identify Bellwether Datasets"
  for data_1, data_2 in datasets:
    def train(data_1):
      "Construct quality predictor"
     return predictor
    def predict(data_1):
      "Predict for quality"
      return predictions
    def score(data_1, data_2):
      "Return accuracy of Prediction"
      return accuracy(train(data_1),\
      test(data_2))

  "Return data with best prediction score"
```

Figure 3.1.B: Transfer *Using the bellwether, construct a transfer learner.*
Construct a transfer learner on the bellwether data. The choice of transfer learners may include any transfer learner used in the literature. For more details on this, see §2.1. Now, apply it to future projects.

```
def transfer(datasets):
  "Transfer Learning with Bellwether Dataset"
  bellwether = discover(datasets)
  def learner(data):
   """
    Construct Transfer Learner, using:
    1. TCA+; 2. TNB; 3. VCB; 4. Bellwether method
   """
  def apply_learner(datasets, learner):
   "Apply transfer learner"
   model = learner(bellwether)
   for data in datasets:
        if data != bellwether:
         train(model)
         test(data)
         yield score(model, data)
```

Figure 3.1.C: Monitor *Keep track of the performance of Bellwethers for transfer learning.*
If the transfer learner constructed in TRANSFER starts to fail, go back to DISCOVER and update the bellwether.

```
def transfer(datasets):
  "Transfer Learning with Bellwether Dataset"
  def fails(data):
   "Return True if predictions deteriorate"
```

Figure 3.1 The Bellwether Framework

- When we note that the prediction performance of bellwether is *statistically poorer* than it was before ...
- Then we can declare that the bellwether has failed[2], that is when we would ideally eschew that bellwether and look for a newer bellwether using the DISCOVER step.

On line 3 in Figure 3.1.A, we just wrap a for-loop around some all pairs of datasets in a community, i.e, data we try every dataset in a round-robin fashion and report the best performing dataset as the bellwether. It is important to note that this will not necessarily lead to a bellwether. Consider a case where all the datasets have very similar performance scores – in such a case it would not be possible to report any dataset as being the bellwether. To identify such similarities in performance, we may use statistical methods such as Scott-Knott tests. If, according to Scott-Knott tests, all the datasets in a community as ranked the same, then we cannot claim that there is a bellwether dataset in that community. However, as discussed later on in this work, we note that this was not the case in any of the four sub-domains we study here. In all cases there is a clear distinction between the best dataset and the worst dataset.

In addition to this simplicity of Figure 3.1. An additional benefit of this DISCOVER-TRANSFER-MONITOR methodology is the ability to optionally replace the Bellwether Method in the TRANSFER stage with any other transfer learner (like TCA+, VCB, TNB, etc.).

[2] we refrained from proposing a numerical threshold because this is a subjective measure. Even with a fixed dataset, it is still subject to vary with several other factors such as the prediction algorithm, the transfer learner, hyper-parameters of several algorithms used here, etc. We therefore recommend a more conservative approach to declaring that the bellwether has failed.

3.5 Experimental Setup

3.5.1 Discovering the bellwether

1. For each community in every sub-domain, we pick a project P_i. We use this as the training set to construct a quality prediction model.
2. Next, we pick another project $P_j \notin P_i$ and retain this as a holdout dataset.
3. Then, for every other project P_k where $k \in 1, \ldots, n;\ k \notin \{i, j\}$, that belong to the same community as $\{P_i, P_j\}$, we evaluate the performance of P_i for P_k according to the evaluation strategy discussed in §4.2.7.
4. We repeat steps 1,2, and 3 for all pairs of projects in a community.

This whole process is repeated 30 times, with different random number seeds. Then, we use the statistical test described in §4.2.8 to rank each project P_i. For every holdout dataset in step 2 above, if there exists one project that returns consistently high performance scores, we label that as the bellwether.

3.5.2 Discovering the best transfer learner

1. For each community in every sub-domain, we pick a project P_i as in §3.5.A. We then use this as the training data to construct the transfer learners (TCA+, TNB, VCB, and Bellwether Method).
2. For every other project P_j where $j \in 1, \ldots, n;\ j \notin i$, that belong to the same community as P_i, we evaluate the performance of each of the transfer learners and use the evaluation strategy discussed in §4.2.7 to evaluate their performance.

Similar to above, the above steps are repeated 30 times, with different random number seeds. Then, we use the statistical test from §4.2.8 to rank each transfer learner.

Defect

Community	Holdout	Test	Bellwether(s)	G-Score(s)	
				med	iqr
AEEEM	EQ	$\forall p \neq EQ$	LC	74	4
	JDT	$\forall p \neq JDT$	LC	75	3
	ML	$\forall p \neq ML$	LC	75	3
	PDE	$\forall p \neq PDE$	LC	75	4
Relink	Apache	$\forall p \neq Apache$	Zxing	67	5
	Safe	$\forall p \neq Safe$	Zxing	66	5
Apache	Ant	$\forall p \neq Ant$	Lucene	66	5
	Ivy	$\forall p \neq Ivy$	Lucene, Poi	64	5
	Camel	$\forall p \neq Camel$	Lucene, Poi	69	7
	Poi	$\forall p \neq Poi$	Lucene, Poi	59	6
	Jedit	$\forall p \neq Jedit$	Lucene	66	4
	Log4j	$\forall p \neq Log4j$	Lucene, Poi	65	5
	Velocity	$\forall p \neq Velocity$	Lucene	67	7
	Xalan	$\forall p \neq Xalan$	Lucene, Poi	68	8
	Xerces	$\forall p \notin Xerces$	Lucene	68	5

Code Smells

Community	Holdout	Test	Bellwether(s)	G-Score(s)	
				med	iqr
Feature Envy	wct	$\forall p \neq wct$	mvnforum	92	3
	itext	$\forall p \neq itext$	mvnforum	92	2
	hsqldb	$\forall p \neq hsqldb$	mvnforum	91	4
	nekohtml	$\forall p \neq nekohtml$	mvnforum	89	4
	galleon	$\forall p \neq galleon$	mvnforum	90	2
	sunflow	$\forall p \neq sunflow$	mvnforum	90	3
	emma	$\forall p \neq emma$	mvnforum	92	1
	jasml	$\forall p \neq jasml$	mvnforum	92	2
	xmojo	$\forall p \neq xmojo$	mvnforum	92	1
	jhotdraw	$\forall p \neq jhotdraw$	mvnforum	92	1
God Class	fitjava	$\forall p \neq fitjava$	xerces, xalan	88	3
	wct	$\forall p \neq wct$	xerces, xalan	88	3
	hsqldb	$\forall p \neq hsqldb$	xerces	87	2
	galleon	$\forall p \neq galleon$	xerces, xalan	90	2
	xalan	$\forall p \neq xalan$	xerces	91	2
	itext	$\forall p \neq itext$	xerces	90	3
	drjava	$\forall p \neq drjava$	xerces, xalan	88	2
	mvnforum	$\forall p \neq mvnforum$	xerces, xalan	90	3
	jpf	$\forall p \neq jpf$	xerces, xalan	90	3
	freecol	$\forall p \neq freecol$	xerces	90	4

Effort Estimation

Community	Holdout	Test	Bellwether(s)	G-Score(s)	
				med	iqr
Effort	coc10	$\forall p \neq coc10$	cocomo	98	2
	nasa93	$\forall p \neq nasa93$	cocomo	99	1
	coc81	$\forall p \neq coc81$	cocomo	98	2
	nasa10	$\forall p \neq nasa10$	cocomo	98	3

Issue Lifetime

Community	Holdout	Test	Bellwether(s)	G-Score(s)	
				med	iqr
1 Day	cloudstack	$\forall p \neq cloudstack$	camel	55	6
	cocoon	$\forall p \neq cocoon$	camel	54	8
	node	$\forall p \neq node$	camel	49	11
	dl4j	$\forall p \neq dl4j$	camel, qpid	55	5
	hadoop	$\forall p \neq hadoop$	camel	57	5
	hive	$\forall p \neq hive$	camel	55	7
	ofbiz	$\forall p \neq ofbiz$	camel	54	4
	qpid	$\forall p \neq qpid$	camel, node	55	7
7 Days	camel	$\forall p \neq camel$	ofbiz	47	7
	cloudstack	$\forall p \neq cloudstack$	ofbiz	47	8
	cocoon	$\forall p \neq cocoon$	ofbiz	48	7
	node	$\forall p \neq node$	ofbiz	48	8
	dl4j	$\forall p \neq dl4j$	ofbiz	47	8
	hadoop	$\forall p \neq hadoop$	ofbiz	46	9
	hive	$\forall p \neq hive$	ofbiz	46	9
	qpid	$\forall p \neq qpid$	ofbiz	47	8
14 Days	camel	$\forall p \neq camel$	qpid	38	5
	cloudstk	$\forall p \neq cloudstk$	qpid	38	5
	cocoon	$\forall p \neq cocoon$	qpid	39	6
	node	$\forall p \neq node$	qpid	37	4
	dl4j	$\forall p \neq dl4j$	qpid	37	4
	hadoop	$\forall p \neq hadoop$	qpid	36	6
	hive	$\forall p \neq hive$	qpid	38	6
	ofbiz	$\forall p \neq ofbiz$	qpid	38	4
	qpid	$\forall p \neq qpid$	qpid	39	5
30 Days	camel	$\forall p \neq camel$	qpid	46	6
	cloudstk	$\forall p \neq cloudstk$	qpid	48	5
	cocoon	$\forall p \neq cocoon$	qpid	47	5
	node	$\forall p \neq node$	qpid	46	6
	dl4j	$\forall p \neq dl4j$	qpid	46	7
	hadoop	$\forall p \neq hadoop$	qpid	47	4
	hive	$\forall p \neq hive$	qpid	48	4
	ofbiz	$\forall p \neq ofbiz$	qpid	47	5
	qpid	$\forall p \neq qpid$	qpid	46	6

Figure 3.2 Discovering Bellwether datasets with a holdout data. We use the experimental setup mentioned in §3.5 to discover these bellwethers.

3.5.3 Understanding The Results

In presenting our results for experiments in §3.5, we adopted a convention that includes tabulated results. The following remarks need to be made regarding our tables:

1. In Figure 3.2, we list the results of performing the experiment in §3.5. The column labeled "Holdout" represents the holdout dataset. The column labeled "Test" represents the test data, i.e., all the remaining data in the community except the holdout. The column

	Bellwether	Local		
	(Lucene) (G-score)	Train	Test	G-Score
Xalan	82	2.6	2.7	56
Ant	68	1.6	1.7	54
Ivy	67	1.4	2	63
Camel	62	1.4	1.6	51
Velocity	57	1.5	1.6	32
Jedit	61	4.2	4.3	77
Log4j	56	1.1	1.2	75
Xerces	58	1.3	1.4	66

Figure 3.3 Bellwether dataset (Lucene) vs. Local Data. Performance scores are G-scores so *higher* values are *better*. Cells highlighted in blue indicate datasets with superior prediction capability. Out of the eight datasets studied here, we note that in five cases the prediction performance of bellwether dataset was superior to within-project dataset.

"Bellwether(s)" shows the dataset that was ranked the best from among the test data (and therefore it is the bellwether dataset). Finally, the column "G-score(s)" is the G-score of training on the bellwether and testing on the holdout dataset.

2. In Figures 3.4, 3.5, 3.6, and 3.7, we list the results of performing the experiment in §3.5 where we compare the bellwether method with other transfer learners. In these figures, the column labeled "source" (the second column) indicates the source from which a transfer learner is built. The remaining datasets within the community are then used as target datasets. The numeric values indicate the *median* performance scores (Standardized Accuracy in case of effort estimation, G-score in the rest), when model is constructed with a "target" dataset and tested against all the "source" datasets, and this processes repeated 30 times for reasons discussed in §5.5.

	Source	**Bellwether**	TCA	TNB
God Class	**xerces**	**90**	75	48
	xalan	89	73	39
	hsqldb	88	0	0
	galleon	87	61	55
	wct	81	58	67
	drjava	80	58	56
	jpf	79	59	65
	mvnforum	74	43	57
	freecol	69	0	0
	fitjava	68	40	0
	itext	62	72	30
	W/T/L	**10/0/1**	1/0/10	0/0/11

	Source	**Bellwether**	TCA	TNB
Feature Envy	**mvnforum**	**92**	57	61
	galleon	84	59	0
	hsqldb	81	57	0
	jhotdraw	81	35	64
	nekohtml	81	52	57
	wct	81	47	0
	itext	74	66	0
	xmojo	74	0	0
	emma	70	74	37
	jasml	66	79	0
	sunflow	47	0	0
	W/T/L	**10/0/1**	1/0/10	0/0/11

Figure 3.4 Code Smells: This figure compares the prediction performance of the bellwether dataset (xalan,mvnforum) against other datasets (other rows). *Bellwether Method* against Transfer Learners (columns) for detecting code smells. The numerical value seen here are the median G-scores from Equation 2 over 30 repeats where one dataset is used as a source and others are used as targets in a round-robin fashion. Higher values are better and cells highlighted in gray produce the best Scott-Knott ranks. The last row in each community indicate Win/Tie/Loss(W/T/L). The *bellwether Method* is the overall best.

3.6 Experimental Results

RQ-3.1: How prevalent is the "Bellwether Effect"?

The bellwether effect points to an exemplar dataset to construct quality predictors from. Ideally, given an adequate transfer learner, such a dataset should produce reasonably high performance scores. Figure 3.2 documents our findings. We use the setup described in §3.5 to discover bellwethers. It is immediately noticeable that for each community there is at least one dataset that provides consistently better predictions when compared to other datasets. For example:

1. *Code Smells datasets:* Here we have two datasets which are frequently ranked high: *Xerces* and *Xalan*. But note that *Xerces* is ranked the best in all the cases. Thus, this would be a

	Source	Bellwether	TCA+	**TNB**	VCB
1 Day	**camel**	17	2	55	10
	node	44	24	55	17
	ofbiz	29	14	53	8
	qpid	44	34	49	19
	deeplearning	51	42	42	15
	cocoon	7	6	34	13
	cloudstack	55	32	32	11
	hive	11	1	22	23
	hadoop	17	0	10	19
	W/T/L	2/0/7	0/0/9	**7/0/2**	2/0/7
7 Days	**ofbiz**	17	3	49	11
	camel	34	6	47	20
	cloudstack	8	27	38	7
	qpid	7	16	38	20
	node	15	33	36	13
	deeplearning	15	20	29	10
	cocoon	0	3	22	16
	hadoop	23	0	18	19
	hive	3	0	7	14
	W/T/L	2/0/7	0/0/9	**7/0/2**	2/0/7

	Source	Bellwether	TCA+	**TNB**	VCB
14 Days	**qpid**	0	0	39	6
	cloudstack	8	8	36	8
	hadoop	0	0	31	22
	deeplearning	4	6	30	17
	camel	1	6	29	18
	cocoon	0	0	19	8
	node	5	4	16	4
	ofbiz	2	2	7	12
	hive	0	0	0	14
	W/T/L	0/0/9	0/0/9	**7/0/2**	2/0/7
30 Days	**qpid**	1	5	47	17
	cloudstack	1	13	38	19
	node	2	10	32	16
	camel	1	1	30	17
	deeplearning	1	2	29	14
	cocoon	2	1	24	12
	ofbiz	4	5	13	5
	hadoop	0	0	7	2
	hive	16	2	6	9
	W/T/L	1/0/8	0/0/9	**8/0/1**	0/0/9

Figure 3.5 Issue Lifetime: This figure compares the prediction performance of the bellwether dataset (qpid) against other datasets (rows) and various transfer learners (columns) for estimating issue lifetime. The numerical value seen here are the median G-scores from Equation 2 over 30 repeats where one dataset is used as a source and others are used as targets in a round-robin fashion. Higher values are better and cells highlighted in gray produce the best Scott-Knott ranks. The last row in each community indicate Win/Tie/Loss(W/T/L). TNB has the overall best Win/Tie/Loss ratio.

bellwether dataset for predicting for the existence of God Classes; this was followed by *hsqldb* with a G-score of 88%. Additionally, when *Xalan* or *Xerces* were absent in Feature Envy, *mvnforum* was a bellwether with a G-score of 92%.

2. *Effort Estimation*: When performing effort estimation, we found that *cocomo* was the bellwether with remarkably high Standardized Accuracy scores of 98%.
3. *Defect datasets:* In the case of defect prediction, Jureczko's bellwether is *Lucene* (with a G-Score of 69%); AEEEM's bellwether is *LC* (with a G-Score of 75%); and Relink's bellwether is *ZXing* (with a G-Score of 68%).
4. *Issue Lifetime:* Finally when predicting for lifetime of issues, we discovered the following

	Source	Bellwether	**TCA+**	TNB	VCB
Apache	**Lucene**	63	**69**	57	64
	Xalan	57	64	59	62
	Camel	60	63	59	44
	Velocity	58	63	51	63
	Ivy	60	62	61	48
	Log4j	60	62	58	62
	Xerces	57	54	58	65
	Ant	61	52	45	55
	Jedit	58	43	57	49
	W/T/L	2/0/7	**6/0/3**	0/0/9	1/2/06
ReLink	**Zxing**	**68**	**67**	53	64
	Safe	38	34	36	31
	Apache	31	31	32	31
	W/T/L	**0/1/1**	0/1/1	1/0/2	0/1/2
AEEEM	**LC**	**75**	**75**	73	61
	ML	73	73	67	51
	PDE	70	71	60	57
	JDT	63	64	68	53
	EQ	59	61	59	57
	W/T/L	0/2/3	**2/2/1**	1/0/4	0/0/5

Figure 3.6 Defect Datasets: This figure compares the prediction performance of the bellwether dataset (Lucene,Zxing,LC) against other datasets (other rows). *Bellwether Method* against Transfer Learners (columns) for detecting defects. The numerical value seen here are the median G-scores from Equation 2 over 30 repeats where one dataset is used as a source and others are used as targets in a round-robin fashion. Higher values are better and cells highlighted in gray produce the best Scott-Knott ranks. The last row in each community indicate Win/Tie/Loss(W/T/L). *TCA+* is the overall best transfer learner.

bellwethers: *camel* for close time of 1 day with G-Scores of around 55%, *ofbiz* for close time of 7 days with a G-score of around 47%, *qpid* for 14 days and 30 days with G-score of around 38%, and 47% s respectively.

Note that in the case of issue lifetime estimation, the G-Scores are particularly low. Here recommend that practitioners monitor the performance of bellwethers and eschew current ones in favor of other better bellwether datasets.

In summary, in three out of the four domains studied here, there was a clear bellwether

	Source	Bellwether	TCA	TNB
FPA	cocomo	98	90	90
	nasa93	93	85	35
	nasa10	90	53	65
	coc81	83	85	60
	coc10	55	75	73
	W/T/L	3/0/2	2/0/3	0/0/5

Figure 3.7 Effort Estimation: This figure compares the performance of the bellwether dataset (cocomo) against other datasets (rows) and Transfer Learners (columns) for estimating effort. The numerical value seen are the median Standardized Accuracy scores from Equation 3 over 40 repeats. *Bellwether Method* has the best Win/Tie/Loss ratio.

	Lucene 2.4		Lucene 2.4, 2.2		Lucene 2.4, 2.2, 2.0	
	G (mean)	G (iqr)	G (mean)	G (iqr)	G (mean)	G (iqr)
Xalan	83	3	82	3	84	3
Poi	73	5	71	4	72	3
Ivy	69	3	66	2	69	2
Ant	67	2	68	1	70	1
Jedit	62	4	63	3	62	3
Xerces	56	9	52	5	58	5
Velocity	55	4	52	4	55	3
Camel	52	2	54	2	53	2
Log4j	52	6	48	6	50	8

	Lucene 2.4	Lucene 2.4, 2.2	Lucene 2.4, 2.2, 2.0
Samples	341	587	782
Defect %	59	59	55

Figure 3.8 Experiments with incremental discovery of bellwethers. Note that the latest version of lucene (lucene-2.4) has statistically similar performance to using the other older versions of lucene.

dataset for every community. In the case of issue lifetimes, although there was a bellwether, the performances were particular low. Note that this may/may not hold true for other sub-domains of SE. The study on these other domains are beyond the scope of this work but what we can say now is:

	Project	Feature Ranks				
		1st	2nd	3rd	4th	5th
Apache	ant	rfc	loc	cam	ce	cbo
	lucene	loc	cbo	amc	ce	rfc
	jedit	loc	rfc	amc	lcom	avg_cc
	xerces	cbo	loc	cam	rfc	ca
	xalan	loc	amc	cbo	lcom3	rfc
	camel	ca	mfa	cbo	loc	amc
	velocity	mfa	cbo	cam	loc	rfc
	poi	loc	ce	lcom	cbm	rfc
	log4j	wmc	cbo	rfc	amc	loc
	ivy	loc	rfc	cam	ce	amc
AEEEM	JDT	ce	wmc	nbugs	lwmc	cle
	PDE	ntb	cwe	lloc	ce	cle
	EQ	cee	loc	cle	ce	cbo
	LC	cwe	nbugs	ce	cle	lloc
	ML	fanOut	CvsLinEntropy	loc	lloc	npm
Relink	Apache	#LineCodeExe	#Line	#LineCode	RatioCommentToCode	AvgEssential
	Safe	#Stmt	SumCyclomaticStrict	#LineCode	#StmtDecl	#LineCodeExe
	Zxing	#LineCodeDecl	#LineCode	AvgLine	#Line	#StmtDecl

Figure 3.9 An example of source instability in defect datasets studied here. The rows highlighted in gray indicate the bellwether dataset. Note: Space limitations prohibit showing these for the other communities. Interested readers are encouraged to use our replication package to see more examples of source instability in other communities.

Result: Bellwethers are common in several domains of software engineering studied here. ie., in defect prediction, effort estimation, and code-smell detection.

RQ-3.2: How does the bellwether dataset fare against within-project data?

Having established in RQ-3.1 that bellwethers are prevalent in the sub-domains studied here. In Figure 3.3, we compare the predictors built on within-project data against those built with a bellwether. For this question, we only used data from the Apache community since it has releases ordered historically (which is required to test older data against newer

data). Since other sub-domains did not have historically data similar to Apache, we were unable to use them for this research question. For the Apache community, the bellwether dataset was *Lucene*.

As seen in Figure 3.3, the prediction scores with the bellwether is very encouraging in case of the Apache datasets. In 5 out of 8 cases (Ant, $Camel$, Ivy, *Xalan*, and $Velocity$), defect prediction models constructed with *Lucene* as the bellwether performed better than within-project data. In 3 out of 8 cases ($Jedit$, *Xerces*, and $Log4j$), the performance scores of bellwether data were statistically worse than within-project data. Note again that this is true in only one out of our the four domains studied, i.e., defect prediction. Therefore, the following answer to the this research question is limited this domain.

Result: For projects in the Apache Community that were evaluated with the same quality metrics, training a quality prediction model with the Bellwether is better than using within-project data in majority of the cases.

RQ-3.3: How well do transfer learners perform across different domains?

Figures 3.4, 3.5, 3.6, 3.7 show the results of transferring data between different projects in a community for code smell detection, issue lifetime estimation, defect prediction, and effort estimation.

Note that of the three transfer learners studied here, value cognitive boosting (VCB) has some methodological constrains that prevents us from translating it to all the domains. VCB was initially designed for defect prediction. To enable it to work efficiently, the authors propose the use of under-sampling techniques to complement transfer learning. This under-sampling required that the datasets have discrete class variables ($\#defects$) and that the datasets are sufficiently large. Two of the domains considered in this work do

not satisfy these constraints. We could not use VCB in code smell detection because our datasets had small sample size (see Figure 2.4) and therefore under-sampling could not be performed. We could not use VCB in effort estimation either because the class variable was a continuous in nature. Other transfer learners did not have these constraints, therefore we were able to translate them to all the domains relatively easily.

These results are expressed in terms of win/tie/loss (W/T/L) ratios:

1. Code Smells dataset: From Figure 3.4 we note that the baseline transfer learner constructed using the bellwether dataset outperforms the other two approaches with a W/T/L of 10/0/1 in both cases.
2. Issue lifetime dataset: From Figure 3.5, we see that, in this case, TNB outperforms the other three methods. We note a W/T/L ratio for TNB at 7/0/2. The baseline approach has W/T/L of 2/0/7 (for 1 and 7 days), 1/0/8 (for 14 days), and 0/0/9 (30 days).
3. Defects dataset: In the case of Figure 3.6, we note that TCA+ was generally better than the other three methods with an overall W/T/L ratio of 8/3/5. The was followed by the baseline transfer learner with a W/T/L ratio of 2/3/11. Note that this behavior of TCA+ corroborates with previous findings by other researchers [Nam13].
4. Effort datasets: In the case of effort estimation, our results are tabulated in Figure 3.7. In this case, the baseline transfer learner once again outperforms the other two methods with a W/T/L ratio of 3/0/2.

The key point from the above is that no transfer learning method is best in all domains (though we would boast that our bellwether method works best more often than the other transfer learners). Hence, when faced with a new community, software analysts will have to explore multiple transfer learning methods. In that context, it is very useful to have an ordering of methods such that simpler baseline methods are run first before more complex approaches. Note that:

- When such an ordering of methods is available then if the simpler methods achieve acceptable levels of performance, an analyst might decide to stop explore more complex methods.
- We would argue that bellwethers fall very early in that ordering; i.e. bellwethers should be the first simplest transfer learning method tried before other approaches.

That is, although we can't endorse a transfer learner in general, we can offer the bellwether method as a baseline transfer learner which can be used to benchmark other complex transfer learners and seek newer transfer learners that can outperform this baseline. Hence, our answer to this question is:

Result: There is no universal best transfer learner that works across multiple domains. Simpler baseline methods like bellwethers show comparable performances in several domains.

RQ-3.4: How much data is required to find the bellwether dataset?

One of our defect dataset allows for a special kind of analysis – the the Apache community (see Figure 2.4) in the defect datasets has data available as historical versions. Using this dataset, we performed an empirical study to establish the required amount of bellwether data to make reliable predictions. We conducted experiments by incrementally updating the versions of the bellwether dataset until we find no significant increase in performance, i.e., starting from version N (the latest version) we construct a prediction model and measure the performance using G-Score. Next, we include an older version $N-1$ to and construct a prediction model to measure the performance. This process is repeated by incrementally growing the size of the bellwether data by including older versions of the bellwether project. With this, the following empirical observations can be made:

- Figure 3.8 documents the results of this experiment. As previously mentioned, we used the defect datasets from the Apache community in Figure 2.4. In RQ-3.1, it was found the *Lucene* was the bellwether dataset for that community. In experimenting with different versions of *Lucene*, we found that using only the latest version of *Lucene* produced statistically similar results to including the older versions of the data. Also, note that we required only 341 samples to achieve good G-scores.
- In cases where datasets were not available in the form of past versions, we observed that the size of the bellwether dataset is very small. For instance, consider the code-smells dataset, the bellwether datasets had no more than 12 samples. Similarly, in the case of effort estimation, the bellwether dataset had only 12 samples.

Result: Not much data is required to find bellwether dataset. In the case of defect prediction, bellwethers can be found by analyzing only the latest version of the project. Even in domains which lack data in the form if historical versions, we were able to discover bellwethers with as few as 25 samples.

RQ-3.5: How effectively do bellwethers mitigate for conclusion instability?

In §2.1.1, we discussed two sources of conclusion instability, namely performance instability and source instability. We can use the bellwether effect to mitigate these two instabilities as follows:

1. Performance instability causes data mining tools such as prediction algorithms to offer unreliable results (their performance depends on the data source). To address this issue, in this paper, we propose the use of the bellwether effect. This effect can be used to discover the bellwether data and we can use this data set as a reliable source to construct

prediction models. Figures 3.4, 3.5, 3.6, and 3.7 reveal that the bellwether data set can be discovered in all the four domains we have studied here. Additionally, the performance of an appropriate transfer learner (as identified in RQ-3.3) with the bellwether dataset is statistically and significantly better than using other datasets. As long as the bellwether dataset remains unchanged, so will the performance of data mining tools such as transfer learners.

2. Source instability causes vastly different and often contradicting conclusions to be derived from a data source. This sort of instability is very prevalent in several domains of software engineering. An example of source instability in the case of defect prediction[3] is shown in Figure 3.9. This figure shows the rankings of top 5 features that contributed to the construction of the transfer learner (TCA+) for defect prediction tasks. It can be noted that, with every data source, the feature rankings a very different. For instance, if *ant* was used to construct TCA+, one may conclude that rfc (response for class) is the most important feature, but if TCA+ was constructed using *lucene*, then we would find that loc is the most important feature (rfc is only the 5^{th} most important feature). This sort of instability can be addressed by identifying a reliable data source to construct a transfer learner. The bellwether dataset is one such example of a stable data source. As long as the bellwether data is reliable (which can be established using the MONITOR step of Figure 3.1) and the bellwether data remains unchanged, so will the conclusions derived from it.

In summary, we may answer this research question as follows:

[3]Space limitations do not permit us to show these for the other three domains. As a result, we have made available a replication package with instructions to replicate these for all the other domains.

Result: The Bellwether Effect can be used to mitigate conclusion instability because as long as the bellwether dataset remains unchanged, we can (a) obtain consistent performance for a transfer learner, and (b) consistent conclusions from the bellwether dataset.

3.7 Discussion

When reflecting on the findings of this work, there may be four additional questions that arise. These are discussed below:

1. *Can bellwethers mitigate conclusion instability permanently?* No- and we should not expect them to. The aim of bellwethers is to *slow*, but do not necessarily *stop*, the pace of new ideas in software engineering (e.g. as in the paper, new quality prediction models). Sometimes, new ideas are essential. Software engineering is a very dynamic field with a high churn in techniques, platforms, developers and tasks. In such a dynamic environment it is important to change with the times. That said, changing *more* than what is necessary is not desirable– hence this work.
2. *How to detect when bellwether datasets need updating?* The conclusion stability offered by bellwether datasets only lasts as long as the bellwether dataset remains useful. Hence, the bellwether dataset's performance must always be monitored and, if that performance starts to dip, then seek a new bellwether dataset.
3. *What happens if a set of data has no useful bellwether dataset?* In that case, there are numerous standard transfer learning methods that could be used to import lessons learned from other data [KM11; Koc15; He13; Tur09; Pet15; Nam13; NK15; Jin15]. That said, the result here is that all the communities of data explored by this paper had useful bellwether datasets. Hence, we would recommend trying the bellwether method before moving on to more complex methods.

3.8 Summary

In this chapter, we have undertaken a detailed study of transfer learners. Our results show that regardless of the sub-domain of software engineering (code smells, effort, defects or issue lifetimes) or granularity of data (file, class, or method), there exists a bellwether dataset that can be used to train relatively accurate quality prediction models and these bellwethers do not require elaborate data mining methods to discover (just a for-loop around the data sets) and can be found very early in a project's life cycle.

We show that bellwether method is a simple baseline for transfer learning. We note that this baseline approach is of much significance to the transfer learning community since previous results on transfer learning do not translate well when applied to domains other than the ones they were designed for. In such cases, bellwethers can be a useful tool to perform accurate and more importantly, stable transfer. The baseline performance offered by the bellwether method would be especially useful for researchers attempting to develop better transfer learners for different domains in software engineering. Further, bellwethers satisfy all the criteria of a baseline method, introduced in §3.2; i.e., they are simple to code and are applicable to a wide range of domains.

Hence, from a pragmatic engineering perspective there are two main reasons to use bellwethers: (a) they slow down the pace of conclusion change; and (b) they can be use to construct a simple baseline transfer learner with comparable performance to the state-of-the-art.

Finally, we remark that much of the prior work on homogeneous transfer learning, including some of the authors own papers, may have needless complicated the homogeneous transfer learning process. We strongly recommend that when building increasingly complex automatic methods, researchers should pause and compare their supposedly

more sophisticated method against simpler alternatives. Going forward from this paper, we would recommend that the transfer learning community uses bellwethers as a baseline method against which they can test more complex methods.

CHAPTER

4

DISCOVERING BELLWETHERS FASTER

In the preceding chapter, we demonstrated the prevalence of bellwethers. Specifically, we showed that bellwethers appear in domains such as defect prediction, effort estimation, detection of code smells, and in estimation of issue lifetimes. However, these datasets we only experimental in that they exemplified real world data only in the nature of the data and *not* in the scale. For instance, the largest community of defect datasets (Apache) had only 10 projects. Further, the largest project in the Apache community was *Xalan* with 3000 rows of data over 3 releases. These sizes do not represent real world situations, where it is commonplace to analyze several hundered projects [Kri18; Agr18; Rah18]. Additionally, in software engineering domains such as configuration optimization each dataset can poten-

tially be combinatorially large and gathering additional data points can be prohibitively expensive.

Therefore, if one wishes to leverage the bellwether effect, an important problem to address then, to make the usage of bellwethers practicable, is to first ensure that it can be discovered fast. In this chapter, we will highlight and discuss the scalability challenges in greater details and discuss better (faster) discovery methods for bellwethers. The rest of this chapter is structured as follows: in §4.1 we discuss the reason why current methods are slow (§4.1.1) and detail the types of scalability problems that need to be addressed. In §4.2, we propose a novel scaling algorithm that can used to accelerate the discovery of bellwethers and enabling them to be used in real-world scenarios. Note, in §4.2 we explore a real-world case study of performance optimization to evaluate our algorithm. In §4.3, we present our findings as answers to research questions. And finally, in §4.4, we offer a summary of the findings from this chapter.

4.1 On scaling the discovery of bellwethers

4.1.1 What causes the slow down?

To understand why the discovery of bellwethers is slow, we reflect on the *DISCOVERY* phase of the framework presented in §3.4.1 and Figure 3.1.A. Briefly, it works as follows:

1. From N available projects, pick a project P_i.
2. Train a defect prediction model on P_i. Lets call this model $M(P_i)$.
3. Then, for every other project P_k where $k \in 1,\ldots,n;\ k \notin \{i,j\}$, that belong to the same community as $\{P_i, P_j\}$, we evaluate the performance of P_i for P_k according to the evaluation strategy discussed in §4.2.7.

4. Steps 1—3 are repeated for all pairs of projects in N.
5. Finally, the projects are ranked from the best to worst. The best ranked projects are selected to be bellwethers.

It may be noted that the above methodology is a form of exhaustive search. While it worked for the relatively small datasets in defect prediction, effort estimation, and code smell detection (see Figure 2.4), the amount of data in real world situations is sufficiently large that scoring all candidates using every sample is too costly.

To understand the theoretical complexity of the framework, let us say that we have (a) M candidate projects, (b)With at most n measurements (rows), and (c) k features (columns) in each. The classical approach, described above, will construct M models. If we assume that the model construction time is a function of number of samples $f(N)$, then for one round in the round-robin, the computation time will $O(M \cdot f(N))$. Since this is repeated M times for each environment, the total computational complexity is $O(M^2 \cdot f(n))$.

If we take decision trees such as CART/C4.5/ID3 as our chosen model, then the model construction time will be $O(kn \cdot log(n))$. There for the naive approach will have a theoretical worst case complexity of $O(M^2 \cdot kn \cdot log(n))$. When M and/or n is/are extremely large, it becomes necessary to seek alternative methods. Depending on which of M and n is large, the scalability may be categorized into two distinct problems:

1. **Horizontal scaling problem.** This problem occurs when the M, i.e., the volume of datasets is extremely large. Being that our complexity is $O(M^2 \cdot kn \cdot log(n))$, large M leads to a *polynomial* growth in discovery times.
2. **Vertical scaling problem.** This problem occurs when the n, i.e., the size of each dataset is extremely large. Our complexity is $O(M^2 \cdot kn \cdot log(n))$, large n leads to a *quasi-linear* growth in discovery times.

4.2 Scale-up using BEETLE

4.2.1 Case Study: Configuration optimization

With the appearance of continuous software engineering and devops, *configurability* has become a primary concern of software engineers. System administrators today develop and use different versions software programs under running several different workloads and in numerous environments. In doing so, they try to apply software engineering methods to best configure these software systems. Despite their best efforts, the available evidence is that they need to be better assited in making all the configuration decisions. Xu et al. [Xu15b] reports that, when left to their own judgementments, developers ignore up to 80% of configuration options, which exposes them to many potential problems. For this reason, the research community is devoting a lot of effort to configuration studies, as witnessed by many recent software engineering research publications [Sie12; Val15a; Sie15; Sar15; Oh17; Nai17b; Tan18; Nai17a; Nai18].

Without automatic support, humans find it difficult to settle on their initial choice for software configurations. The available evidence [VA17; JC16; Her11] shows that system administrators frequently make poor configuration choices. Typically, off-the-shelf defaults are used, which often behave poorly. There are various examples presented in the literature which have established that choosing default configuration can lead to sub-optimal performance. For instance, Van Aken et al. report that the default MySQL configurations in 2016 assume that it will be installed on a machine that has 160MB of RAM (which, at that time, was incorrect by, at least, an order of magnitude) [VA17]. Also, Herodotou et al. [Her11] report that default settings for Hadoop results in the *worst possible* performance.

Traditional approaches to finding good configuration are very resource intensive. A

typical approach uses sensitivity analysis [Sal00], where performance models are learned by measuring the performance of the system under a limited number of sampled configurations. While this approach is cheaper and more effective than manual exploration, it still incurs the expense of extensive data collection about the software [Guo13; Sar15; Sie12; Nai17a; Nai17b; Nai18; Oh17; Guo17; JC16]. This is undesirable since this data collection has to be repeated if ever the software is updated or the environment of the system changes abruptly. While we cannot tame the pace of change in modern software systems, we can reduce the data collection effort required to react to that change.

As a more concerete example, consider an organization that runs, say, N heavy Apache Spark workloads on the cloud. To optimize the performance of Apache Spark on the given workloads, the DevOps Team need to find the optimal solutions for each of these workloads, i.e., conduct performance optimization N times. This setup has two major shortcomings–
Hardware Change: Even though the DevOps engineer of a software system performs a performance optimization for a specific workload in its staging environment, as soon as the software is moved to the production environment the optimal configuration found previously may be inaccurate. This problem is further accentuated if the production environment changes due to the ever-expanding cloud portfolios. It has been reported that cloud providers expand their cloud portfolio more than 20 times in a year [Ec2].
Workload Change: The developers of a database system can optimize the system for a read-heavy workload, however, the optimal configuration may change once the workload changes to, say, a write-heavy counterpart. The reason is that if the workload changes, different functionalities of the software might get activated more often and so the nonfunctional behavior changes too. This means that as soon as a new workload is introduced (new feature in the organization's product) or if the workload changes, the process of performance optimization needs to be repeated.

Given the fragility of traditional performance optimization, it is imperative that we develop a method to learn from our *previous experiences* and hence reduce the burden of having to find optimum configurations ad nauseam. Formally, this is called "transfer learning"; i.e., the transfer of information from selected "*source*" software configurations running on one environment to learn a model for predicting the performance of some "*target*" configurations in a different environment. We discuss transfer learning in much detail in §2.1. Transfer learning can only be useful in cases where the source environment is similar to the target environment. If the source and the target are not similar, knowledge should not be transferred. In such situations, transfer learning can be unsuccessful and can lead to a *negative transfer*. Prior work on transfer learning focused on "*What to transfer*" and "*How to transfer*", by implicitly assuming that the source and target are related to each other. However, those work failed to address "*From where (whence) to transfer*" [PY10]. Jamshidi et al. [Jam17b] alluded to this and explained when transfer learning works but, did not provide a method which can help in selecting a suitable source.

Fortunately, the issue of identifying a suitable source is not an uncommon problem in software engineering. In fact, we have addressed this very issue in the previous chapter using the bellwether effect.

There exists a computational bottleneck with using the bellewether effect as highlighted previously. The challenge posed by this domain falls under the vertical scaling problem. In the case of performance optimization, it is imperative that the number of measurements are minimized. This is because of two reasons:

1. *Combinatorial Explosion.* Configurable software systems frequently have tens to hundereds of possible configuration. It is therefore not practical to expect to train a Machine Learner on such vast spaces.
2. *Cost of data collection.* Measuring the performance of each configuration is rather ex-

pensive. Therefore, we are also limited by how data we can gather before the allocated budget is exceeded.

To address these two problems, we must be able to discover the bellwether project as fast as possible in order not to lose too much time or money. In this chapter, we present a transfer learning framework called Bellwether Transfer Learner (henceforth referred to as BEETLE) that scales to arbitrarily large configuration problems, performing much better than the prior state of the art. Further, it does so using fewer measurements than existing state-of-the-art methods. We say this since, comparitively, BEETLE has the lowest CPU cost (and we conjecture that this is so since BEETLE makes the best use of old configurations).

4.2.2 Problem Formalization

Configuration: A software system, $\mathbb{S}$, may offer a number of configuration options that can be changed. We denote the total number of configuration options of a software system $\mathbb{S}$ as N. A configuration option of the software system can either be a (1) continuous numeric value or a (2) categorical value. This distinction is very important since it impacts the choice of machine learning algorithms. The configuration options in all software systems studied in this thesis are a combination of both *categorical* and *continuous* in nature. The learning algorithm used here namely, *Regression Trees*, are particularly well suited to handle such a combination of continuous and categorical data.

A configuration is represented by c_i, where i represents the i^{th} configuration of a system. A set of all configurations is called the *configuration space*, denoted as $\mathbb{C}$. Formally, $\mathbb{C}$ is a Cartesian product of all possible options $\mathbb{C} = \text{Dom}(c_1) \times \text{Dom}(c_2) \times ... \times \text{Dom}(c_N)$, where $\text{Dom}(c_i)$ is either $\mathbb{R}$ (Real Numbers) or $\mathbb{B}$ (Catergorical/Boolean value) and N is the number of configuration options.

	ATOMIC	USE_LFS	SECURE	LATENCY (μs)
c_1	0	0	0	100
c_2	0	0	1	150
⋮	⋮	⋮	⋮	⋮
c_N	1	1	1	400

Figure 4.1 Some configuration options for SQLite.

As a simple example, consider a subset of configuration options from SQLite, i.e., $\mathbb{S} \equiv$ SQLite. This is shown in Figure 4.1. The subset of SQLite offers three configuration options namely, ATOMIC (atomic delete), USE_LFS (use large file storage), and SECURE (secure delete), ie., $N = 3$. The last column contains the *latency* in μs when various combinations of these options are chosen.

Environment: As defined by Jamshidi et al. [Jam17b], the different ways a software system is deployed and used is called its *environment* (e). The environment is usually defined in terms of: (1) ***workload*** (w): the input which the system operates upon; (2) ***hardware*** (h): the hardware on which the system is running; and (3) ***version*** (v): the state of the software.

Note that, other environmental changes might be possible (e.g., JVM version used, etc.). For example, consider software system Apache Storm, here we must ensure that an appropriate JVM is installed in an environment before it can be deployed in that environment. Indeed, the selection of one version of a JVM over another can have a profound performance impact. However, the perceived improvement in the performance is due to the optimizations in JVM, not the original software system being studied. Therefore, in this work, we do not alter these other factors which do not have a direct impact on the performance of the software system.

The following criteria is used to define an environment:

1. Environmental factors of the software systems that we can vary in the deployment stack of the system. This prevents us from varying factors such as the JVM version, CPU

frequency, system software, etc., which define the deployment stack and not the software system.

2. Common changes developers choose to alter in the software system. In practice, it is these factors that affect the performance of systems the most [Jam17b; Jam17a; Val17; Val15b].
3. Factors that are most amenable for transfer learning. Preliminary studies have shown that factors such as workload, hardware, and software version lend themselves very well to transfer learning [Jam17b; Jam17a].

Formally, we say an environment is $\mathbf{e} = \{w, h, v\}$ where $w \subseteq W$, $h \subseteq H$, and $v \subseteq V$. Here, W, H, V are the space of all possible hardware changes H; all possible software versions V, and all possible workload changes W. With this, the environment space is defined as $\mathbb{E} \subset \{W \times H \times V\}$, i.e., a subset of environmental conditions **e** for various workloads, hardware, and environments.

Performance: For each enviroment **e**, the instances in our data are of the form $\{(c_1, y_1), ..., (c_N, y_N)\}$, where c_i is a vector of configurations of the i-th example and it has a corresponding performance measure $y_i \in Y_{S,c,e}$ associated with it. We denote the performance measure associated with a given configuration (c_i) by $y = f(c^i)$. We consider the problem of finding the near-optimal configurations (c^*) such that $f(c^*)$ is better than other configurations in $C_{A,e}$, i.e.,

$$f(c^*) \leq f(c) \; \forall c \in C_{A,h,w,v} \setminus c^* \quad \text{for min objective}$$
$$f(c^*) \geq f(c) \; \forall c \in C_{A,h,w,v} \setminus c^* \quad \text{for max objective}$$

Bellwethers: In the context of performance optimization, the bellwether effect states that: *For a configurable system, when performance measurements are made under different environments, then among those environments there exists one exemplary environment, called the bellwether, which can be used determine near optimum configuration for other environ-*

ments for that system. We show that, when performing transfer learning, there are exemplar source environments called the bellwether environment(s) ($\mathbb{B} = e_{s1}, e_{s2}, ..., e_{sn} \subset E$), which are the best source environment(s) to find near-optimal configuration for the rest of the environments ($\forall e \in E \setminus \mathbb{B}$).

Problem Statement: The problem statement of this work:

> Within a pre-defined budget constraint, find a near-optimal configuration for a target environment (S_{e_t}), by learning from the measurements ($\langle c, y \rangle$) for the same system operating in different source environments (S_{e_s}).

In other words, we aim to reuse the measurements from a system operating in an environment to optimize the same system operating in the different environment thereby reducing the number of measurements required to find the near-optimal configuration.

4.2.3 BEETLE: Bellwether Transfer Learner

This section describes BEETLE, a bellwether based approach that finds the near-optimal configuration using the knowledge in the "bellwether" environment. BEETLE can be separated into two main steps: (i) *Discovery:* finding the bellwether environment, and (ii) *Transfer:* using the bellwether environment to find the near-optimal configuration for target environments. These steps will are explained in greater detail in §4.2.3.1 and §4.2.3.2. We outline it below,

1. *Discovery:* Leverages the existence of the bellwether effect to *discover* which of the available environments are best suited to be a *source enviroment* (known as the *bellwether environment*). To do this, BEETLE uses a **racing algorithm** to sequentially evaluate candidate environments [Bir02]. In short,
   (a) A fraction (about 10%) of all available data is sampled. A prediction model is built

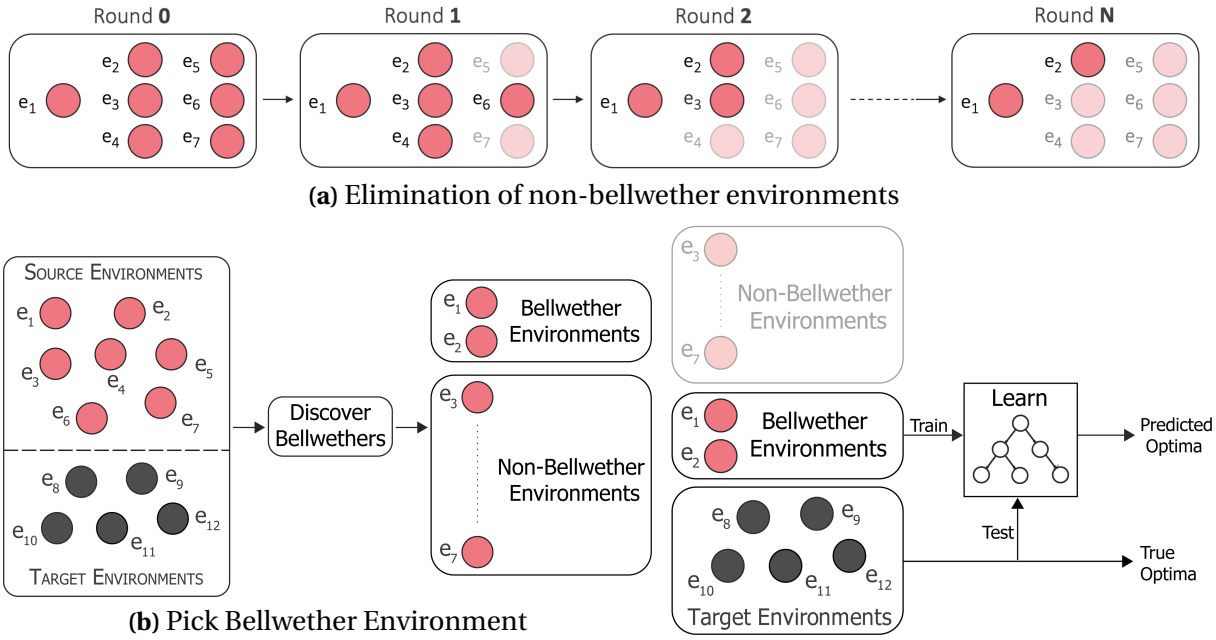


Figure 4.2 BEETLE framework

with these sampled datasets.

(b) Each enviroment is used as a *source* to build a prediction model and all the others are used as *targets* in a round-robin fashion.

(c) Performance of all the enviroments are measured and are statistically ranked from the best source environemnt to the worst. Environments with a *poor* performance (i.e., those ranked last) are eliminated.

(d) For the remaining enviroments, another 10% of the samples are added and the steps (a)–(c) are repeated.

(e) When the ranking order doesn't change for a fixed number of repeats, we terminate the process and nominate the best ranked enviroment(s) as the bellwether.

2. *Transfer:* Next, to perform transfer learning, we just use these bellwether environments to train a performance prediction model with **Regression Trees**.

```
def FindBellwether(srcs, frac, budget, ↩
    thres, lives):
 while lives or cost > budget:
  "Sample configurations"
  for i, src in enumerate(srcs):
   samp_src[i].add_sample(src, step_size)
  "Get cost"
  cost += get_cost(samp_src)
  "Evaluate pair-wise performances"
  cur_ranks = get_perf_rankings(sampled)
  "Loose life if no sources are removed"
  if prev_ranks == cur_ranks:
   lives -= 1
   prev_ranks = cur_ranks
   continue
  else:
   "Remove non-bellwether environments"
   samp_sources = remove_non_bellws(↩
       cur_ranks)
 return sample_sources[cur_rank[0]]
```

Figure 4.3 Pseudocode for *Discovery*

```
def BEETLE(sources, target, budget):
 "Find the bellwether environments"
 bellw = FindBellwether(sources, frac, ↩
     budget, thres, lives)
 "If more than one bellwethers, pick one at random"
 if len(bellw) > 1:
  bellw = random.choice(bellw)
 "Train a prediction model with the bellwether"
 prediction_model = regTree.train(bellw)
 "Predict the performance various configurations in target"
 best_config = prediction_model.↩
     predict_best(target)
 "Return the configuration with the best performance"
 return best_config
```

Figure 4.4 Pseudocode for *Transfer*

We conjecture that once a *bellwether source environment* is identified, it is possible to build a simple transfer model without any complex methods and still be able to discover near-optimal configurations in a target environment.

4.2.3.1 *Discovery*: Finding Bellwether Environments

In the previous work on bellwethers [KM18], the discovery process involved a round-robin experimentation comprised of the following steps:

1. Pick an enviroment $\mathbf{e}_i$ from the space of all available enviroments, i.e., $\mathbf{e}_i \in \mathbb{E}$.
2. Use $\mathbf{e}_j$ as a *source* to build a prediction model.
3. Using all the *other* enviroments $\mathbf{e}_j \in \mathbb{E}$ and $\mathbf{e}_j \neq \mathbf{e}_i$ as the *target*, determine the prediction performance of $\mathbf{e}_i$.
4. Next, repeat the steps by choosing a different $\mathbf{e}_i \in \mathbb{E}$
5. Finally, rank the performances of all the enviroments and pick the best ranked enviroment(s) as bellwether(s).

The above methodology is a form of an exhaustive search. While it worked for the relatively small datasets in [Kri16c; KM18], the amount of data in this work is sufficiently large that scoring all candidates using every sample is too costly. More formally, let us say that we have M candidate enviroments with N measurements each. The classical approach, described above, will construct M models. If we assume that the model construction time is a function of number of samples $f(N)$, then for one round in the round-robin, the computation time will $O(M \cdot f(N))$. Since this is repeated M times for each enviroment, the total computational complexity is $O(M^2 \cdot f(n))$. When M and/or N is/are extremely large, it becomes necessary to seek alternative methods. Therefore, in this work, we use a racing algorithm to achieve computational speedups.

Instead of evaluating every available instance to determine the best source environment, *Racing algorithms* take the following steps:

- Sample a small fraction of instances from the original environments to minimize computational costs.
- Evaluate the performance of environments statistically.
- Discarded the environments with the poorest performance.
- Repeated the process with the remaining datasets with slightly larger sample size.

Figure 4.2a shows how BEETLE eliminates inferior environments at every iteration (thus reducing the overall number of environments evaluated). Since each iteration only uses a small sample of the available data, the model building time also reduces significantly. It has been shown that racing algorithms are extremely effective in model selection when the size of the data is arbitrarily large [Bir02; LN13].

In Figure 4.2c, we illustrate the discovery of the bellwether environments with an example. Here, there are two groups of environments:

(i) Group 1: Environments $e_1, e_2, ..., e_7$, for which performance measurements have been

gathered. One or more these environment(s) are potentially bellwether(s).

(ii) Group 2: Environments $e_8, e_9, ..., e_{12}$, these represent the target environments, for which need to determine an optimal configuration.

In the discovery process, BEETLE's objective is to *find bellwethers* from among the environments in Group 1. And, later in the *Transfer* phase, we use the bellwether environments to find the near-optimal configuration for the target environments from Group 2.

Fig. 4.3 outlines a pseudocode for the algorithm used to find bellwethers. The key steps are listed below:

- *Lines 3–5*: Randomly sample a small subset of configurations from the source environments. The size of the subset (of configurations) is controlled by a predefined parameter *frac*, which defines the percent of configurations to be sampled in each iteration.
- *Line 6–7*: Calculate sampling cost for the configurations.
- *Line 8–9*: Use the sampled configurations from each environment as a *source* build a prediction model with regression trees. For all the remaining environments, this regression tree model is used to predict for optimum configuration. After using every enviroment as a *source*, the environments are ranked from best to worst using the evaluation criteria discussed in §4.2.7.
- *Line 10–14*: We check to see if the rankings of the enviroments have changed since the last iteration. If not, then a "life" is lost. We go back to *Line 3* and repeat the process. When all lives are expired, or we run out of the budget, the search process terminates. This acts as an early stopping criteria, we need not sample more data if those samples do not help in improving the outcome.
- *Line 15–17*: If there is some change in the rankings, then new configuration samples are informative and the environments that are *ranked last* are eliminated. These environments are not able to find near-optimal configurations for the other environments and

therefore cannot be bellwethers.

- *Line 18*: Once we have exhausted all the lives or the sampling budget, we simply return the source project with the best rank. These would be the bellwether enviroments.

On *line 6–7* we measure the sampling cost. In our case, we use the number of samples as a proxy for cost. This is because each measurement consumes computational resources which in turn has a monetary cost. Therefore, it is commonplace to set a budget and sample such that the budget is honored. Our choice of using the number of measurements as a cost measure was an engineering judgment, this can be replaced by any user-defined cost function such as (1) the actual cost, or (2) the wall-clock time. The accuracy of either of the above are depenedent on the business context. If one is either constrained by runtime or there is large variance in the measurements of time per configuration, then the wallclock time might be a more reasonable measure. On the other hand, if the cost of measurements is the limiting factor, it makes sense to use the actual measurement cost. Using number of samples encompasses these two factors since it both costs more money and takes time to obtain more samples. In the ideal case, we would like to have performance measurements for all possible configurations of a software system. But this is not practical because certain systems have over 2^{50} unique configurations.

It is entirely possible for the *FindBellwether* method to identify multiple bellwethers (e.g., in the case of 4.2c the bellwethers were e_1 and e_2). When mutliple bellwethers are found, we may use (a) any one of the bellwether enviroments at random, (b) use all the enviroments, or (c) use heuristics based on human intuition. In this work, we pick one enviroment from among the bellwethers at random. As long as the chosen project is among the bellwether enviroments, the results remain unchanged.

The BEETLE approach assumes that a *fixed* set of enviroments exist from which we pick one or more bellwethers. But, approach would work just as well where new measurements

from new enviroments are added. Specifically, when more environments are added into a project, it is possible that the newly added environment could be the bellwether. Therefore, we recommend repeating *FindBellwether* method prior to using the new enviroment. Note that, repeating *FindBellwether* for new enviroments would add minimal computational overhead since the measurements have already been made for the new enviroments. Also note that, this approach of revisiting *FindBellwether* on availability of new data, has been previously been proposed in other domains in software engineering [Kri16c; KM18].

4.2.3.2 *Transfer*: Using the Bellwether Environments

Once the bellwether environment is identified, it can be used to find the near-optimal configurations of target environments. As shown in Fig. 4.4, *FindBellwether* eliminates enviroments that are not potentially bellwethers and returns only the candidate bellwether enviroments. For the remaining target enviroments, we use the model built with the bellwether enviroments to identify the near optimal configurations.

Figure 4.4 outlines the pseudocode used to perform the transfer. The key steps are listed below:

- *Line 2-3*: We use the *FindBellwether* from 4.3 to identify the bellwether enviroments.
- *Line 4-6*: If there exists more than one bellwether, we randomly chose one among them be used as the bellwether enviroment.
- *Line 7-8*: The configurations from the bellwether and their corresponding performance measures are used to build a prediction model using regression trees.
- *Line 9-10*: Predict the performances of various configurations from the target enviroment.
- *Line 11-12*: Return the best configuration for the target.

Note that, on *Line 10*, we use *regression trees* to make predictions. It has been the most preferred prediction algorithm in this domain [Guo13; Sar15; Nai17a]. This is primarily

because much of the data used in this domain are a mixture numerical and categorical attributes. Given configuration measurement in the form $\{(c_i, y_i)\}$, c_i is a vector of categorical/numeric values and y_i is a continuous numeric value. For such data, regression trees are the best suited prediction algorithms [Nai17b; Nai18; Guo17; Val17]. This is because the regression trees are built by recursively splitting the configuration vectors into a root node configuration and subsets of children node configurations. This splitting is based on a set of splitting rules that work equally well for both categorical and numeric data.

4.2.4 Other Transfer Learners

This section describes the methods we use to compare BEETLE against. These alternatives are (a) two state-of-the-art transfer learners for performance optimization: Valov et al. [Val17] and Jamshidi et al. [Jam17a]; and (b) a non-transfer learner: Nair et al. [Nai17b].

4.2.4.1 Transfer Learning with Linear Regression

Valov et al. [Val17] proposed an approach for transferring performance models of software systems across platforms with *different hardware settings*. The method consists of the following two components:

1. *Performance prediction model:* The configurations on a source hardware are sampled using *Sobol* sampling. The number of configurations is given by $T \times N_f$, where $T = 3, 4, 5$ is the *training coefficient* and N_f is the number of configuration options. These configurations are used to construct a *Regression Tree* model.
2. *Transfer Model:* To transfer the predictions from the source to the target, a linear regression model is used since it was found to provide good approximations of the transfer function. To construct this model, a small number of random configurations are ob-

tained from *both the source and the target*. Note that this is a shortcoming since, without making some preliminary measurements on the target, one cannot begin to perform transfer learning.

4.2.4.2 Transfer Learning with Gaussian Process

Jamshidi et al. [Jam17a] took a slightly different approach to transfer learning. They used Multi-Task Gaussian Processes (GP) to find the relatedness between the performance measures in source and the target. The relationships between input configurations were captured in the GP model using a covariance matrix that defined the kernel function to construct the Gaussian processes model. To encode the relationships between the measured performance of the source and the target, a scaling factor is used with the above kernel.

The new kernel function is defined as follows:

$$k(s,t,f(s),f(t)) = k_t(s,t) \times k_{xx}(f(s),f(t)), \tag{4.1}$$

where $k_t(s,t)$ represents the multiplicative scaling factor. $k_t(s,t)$ is given by the correlation between source f(s) and target f(t) function, while k_{xx} is the covariance function for input environments (s & t). The essence of this method is that the kernel captures the interdependence between the source and target environments.

4.2.4.3 Non-Transfer Learning Performance Optimization

A performance optimization model with no transfer was proposed by Nair et al. [Nai17b] in FSE '17. It works as follows:

1. Sample a small set of measurements of configurations from the target environment.
2. Construct performance model with regression trees.

Table 4.1 Overview of the real-world subject systems. $|C|$:Number of Configurations sampled per environment, N=Number of configuration options, $|E|$: Number of Environments, $|H|$: Hardware, $|W|$: Workloads, and $|V|$: Versions.

System	Language	$\{\|C\|, N, \|E\|\}$	H	W	V	Unchanged
x264	C, Assembly	4000, 16, 21	NUC/4/1.30/15/SSD NUC/2/2.13/7/SSD Station/2/2.8/3/SCSI, AWS/1/2.4/1.0/SSD, AWS/1/2.4/0.5/SSD, Azure/1/2.4/3/SCSI	8/2, 32/11, 128/44	r2389, r2744	Memory, CPU, background services
SPEAR	C, Assembly	16384, 14, 10	NUC/4/1.30/15/SSD, NUC/2/2.13/7/SSD, Station/2/2.8/3/SCSI, AWS/1/2.4/1.0/SSD, AWS/1/2.4/0.5/SSD, Azure/1/2.4/3/SCSI	(in #variables/#clauses), 774/5934, 1008/7728, 1554/11914, 978/7498	1.2, 2.7	Memory, CPU, background services
SQLite	C	1000, 14, 15	NUC/4/1.30/15/SSD, NUC/2/2.13/7/SSD, Station/2/2.8/3/SCSI, AWS/1/2.4/1.0/SSD, AWS/1/2.4/0.5/SSD, Azure/1/2.4/3/SCSI	write–seq, read–batc, read–rand, read–seq	3.7. 6.3, 3.19.0.0	Memory, CPU, background services
SaC	C	846, 50, 5	NUC/4/1.30/15/SSD, NUC/2/2.13/7/SSD, Station/2/2.8/3/SCSI, AWS/1/2.4/1.0/SSD, AWS/1/2.4/0.5/SSD, Azure/1/2.4/3/SCSI	random matrix generator, particle filtering, differential, equation solver, k-means, optimal matching, nbody, simulation, conjugate, gradient, garbage collector.	1.0.0	Memory, CPU, background services
Storm	Clojure	2048, 12, 4	NUC/4/1.30/15/SSD, NUC/2/2.13/7/SSD, Station/2/2.8/3/SCSI, AWS/1/2.4/1.0/SSD, AWS/1/2.4/0.5/SSD, Azure/1/2.4/3/SCSI	WordCount, RollingCount, RollingSort, SOL	Storm 0.9.5 + Zookeeper 3.4.11	JVM machine, Zookeeper Options, Memory, CPU, background services

3. Predict for near-optimal configurations.

The key distinction here is that unlike transfer learners, that use a *different source environment* to build to predict for near-optimal configurations in a target environment, a non-transfer method such as this uses configurations *from within the target* environment to predict for near-optimal configurations.

4.2.5 Evaluation

4.2.6 Subject Systems

In this study, we selected five configurable software systems from different domains, with different functionalities, and written in different programming languages. We selected these real-world software systems since their characteristics cover a broad spectrum of scenarios. Briefly,

1. SPEAR is an industrial strength bit-vector arithmetic decision procedure and Boolean satisfiability (SAT) solver. It is designed for proving software verification conditions, and it is used for bug hunting. It consists of a binary configuration space with 14 options with 2^{14} or 16384 configurations. We measured how long it takes to solve an SAT problem in all 2^{14} configurations in 10 environments.
2. X264 is a video encoder that compresses video files and has 16 configurations options to adjust output quality, encoder types, and encoding heuristics. Due to the cost of sampling the entire configuration space, we randomly sample 4000 configurations in 21 environments.
3. SQLITE is a lightweight relational database management system, which has 14 configuration options to change indexing and features for size compression. Due to the cost of sampling and a limited budget, we use 1000 randomly selected configurations in 15 different environments.
4. SAC is a compiler for high-performance computing. The SaC compiler implements a large number of high-level and low-level optimizations to tune programs for efficient parallel executions. It has 50 configuration options to control optimization options. We measure the execution time of the program for 846 configurations in 5 enviroments.

5. STORM is a distributed stream processing framework which is used for data analytics. We measure the latency of the benchmark in 2,048 randomly selected configurations in 4 environments.

Table 4.1 lists the details of the software systems used in this work. Here, $|N|$ is the number of configuration options available in the software system. If the options for each configuration is *binary*, then there can be as much as $2^{|N|}$ possible configurations for a given system[1], since it is not possible for us measure the performance of all possible configurations, we measure the performance of a subset of the $2^{|N|}$ samples, this subset is denoted by $|C|$. The performance of each of the $|C|$ configurations are measured under different hardware (H), workloads (W), and software versions (V). A unique combination of H, W, V constitutes an enviroment which is denoted by E. Note that, measuring the performance of $|C|$ configurations in each of the $|E|$ enviroments can be very costly and time consuming. Therefore, instead of all combinations of $H \times W \times V$, we measure the performance in only a subset of the enviroments (the total number is denoted by $|E|$).

4.2.7 Evaluation Criterion

4.2.7.1 Inadequacies of Conventional Evaluation Criteria

Typically, performance models are evaluated based on accuracy or error using measures such as *Mean Magnitude of Relative error* (abbreviated as $MMRE$). MMRE is calculated as follows:

$$MMRE = \frac{|predicted - actual|}{actual} \cdot 100$$

While seemingly intutive, it has recently been shown that exact measures like MMRE can be somewhat misleading to assess configurations. There has been a lot of criticism leveled

[1]On the other hand, if there are $|o|$ possible options, then there may be $|o|^N$ possible configurations.

against MMRE. MMRE along with other accuracy statistics such as MBRE (which stands for Mean Balanced Relative Error) have been shown to cause conclusion instability [MS12; Myr05; Fos03].

In order to overcome this, recently we have argued against the use of MMRE values, in favor of using *rank-based metrics* that compute the difference between the *relative rankings* of the performance scores [Nai17b; Nai18]. The key intuition behind *relative rankings* is that the raw accuracy (as measured by MMRE) is less important than the rank-ordering of configuraions from best to worst. As long as a model can preserve the order of the rankings of the configurations (from best to worst), we can still determine which configuration is the most optimum. We can quantify this by measuring the differences in ranking between the actual rank and the predicted rank. More formally, rank-difference R^{δ} is measured as:

$$R^{\delta} = |Rank_{Predicted} - Rank_{Actual}|$$

We note that rank difference, although slightly less susceptible to instability compared to

	Value	Rank
Actual	0.09	100
Predicted	0.11	10
Difference	0.02	90

$$MMRE = \frac{0.11 - 0.09}{0.09} \times 100 = 22\%$$

$$R^{\delta} = |10 - 100| = 90$$

Now, let's say the $min = 0.09$ and $max = 0.11$. Then,

$$NAR = \frac{0.11 - 0.09}{0.11 - 0.09} \times 100 = 100\%$$

Figure 4.5 A contrived example to illustrate the challenges with MMRE and rank based measures

MMRE, is still not particularly informative. This is because it ignores the distribution of performance scores and *a small difference in performance measure can lead to a large rank difference and vice-versa* [Tru18].

To illustrate the challenges with R^{δ} and $MMRE$, consider the example in Figure 4.5 where we are trying to find a configuration with the *minimum* value. Here, although the difference between the predicted value and the actual value is only 0.02, the rank difference R^{δ} is 90. But this does not tell us if $R^{\delta} = 90$ is good or bad. While, in the same Figure 4.5, when we calculate MMRE we get an error of only 22%, this may convey a false sense of accuracy. In the same example, let us say that the maximum value permissible is 0.11, then according to Figure 4.5, our predicted value for the best performance (which recall is supposed to the lowest) is the *highest permissible value* of 0.11.

Therefore, to obtain a realistic estimate of optimality of a configuration, we must use a measure that does not *under-estimate* the difference between the actual best configuration and the predicted best configuration (as with $MMRE$), netiher should we *over-estimate* the difference (as with rank-difference). In other words, we seek a robust measure of optimality.

4.2.7.2 NAR: A more robust metric

To overcome the challenges above, in this work, we propose a measure called *Normalized Absolute Residual* (NAR). It represents the ratio of (a) difference between the actual performance value of the optimal configuration and the predicted performace value of the optimal configuration, and (b) The absolute difference between the *maximum* and *minimum* possible performace values. Formally, it can be defined as:

$$NAR = \frac{|min(f(c)) - f(c^*)|}{max(f(c)) - min(f(c))} \cdot 100 \tag{4.2}$$

Where $min(f(c))$ is the value of the true minima of configuration c, $f(c^*)$ is the predicted value of the minima, and $max(f(c))$ is the largest performance value of a configuration. This measure is equivalent to Absolute Residual between predicted and actual, normalized to lie between 0% to 100% (hence the name *Normalized Absolute Residual* or NAR). According to this formulation, the *lower the* NAR, the better. Reflecting back on Figure 4.5, we see that the NAR is 100% which is exact what is expected when a predicted "minima" (0.11) is equal to the actual "maxima" (also 0.11).

Further, it is worth noting that, NAR is actually a variant of the *Generational Distance* or *Inverted Generation Distance* used very commonly in search-based software engineering literature [Wan16a; Che18; Deb02].

4.2.8 Statistical Validation

Our experiments are all subjected to inherent randomness introduced by sampling configurations or by a different source and target environments. To overcome this, we use 30 repeated runs, each time with a different random number seed. The repeated runs provide us with sufficiently large sample size for statistical comparisons. Each repeated run collects the values of NAR.

To rank these 30 numbers collected as above, we use the Scott-Knott test recommended by Mittas and Angelis [MA13]:

- A list of treatments, sorted by their mean value, are split at the point that maximizes the expected value of the difference in their mean before after the split.
- That split is accepted if, between the two splits, (a) there is a statistically significant difference using a hypothesis test $\mathbb{H}$, and (b) the difference between the two splits is *not* due to a small effect.

- If the split is acceptable, the algorithm then recurses on both splits.
- Once no more splits are found, they are "ranked" smallest to largest (based on their median value).

In our work, in order to judge the statistical significance we use a non-parametric bootstrap test with 95% confidence [ET93]. Also, to make sure that the statistical significance is not due to the presence of small effects, we use an A12 test [VD00a]. Briefly, the A12 test measures the probability that one split has a lower NAR values than another. If the two splits are equivalent, then $A12 = 0.5$. Likewise if $A12 \geq 0.6$, then 60% of the times, values of one split are significantly smaller that the other. In such a case, it can be claimed that there is a *significant effect* to justify the hypothesis test. We use these two tests (bootstrap and A12) since these are non-parametric and have been previously demonstrated to be informative [LO02; PC10; AB11; SM12b; Kam07; Koc13].

4.3 Experimental Results

RQ-4.1: Does there exist a Bellwether Environment?

Purpose: The first research question seeks to establish the presence of bellwether environments within different environments of a software system. If there exists a bellwether environment, then identifying that environment can greatly reduce the cost of finding a near-optimal configuration for different environments.

Approach: For each subject software system, we use the environments to perform a pairwise comparison as follows:

1. We pick one environment as a source and evaluate all configurations to construct a regression tree model.

X264

Rank	Dataset	Median	IQR
1	x264_18	0.35	1.82
1	x264_9	0.35	1.62
2	x264_10	0.94	8.25
2	x264_7	0.94	8.25
2	x264_11	1.62	7.46
3	x264_16	2.33	12.18
3	x264_2	2.33	12.18
3	x264_6	2.82	5.35
3	x264_20	3.65	13.74
4	x264_19	6.95	41.97
4	x264_3	8.68	49.78
4	x264_17	13.61	32.32
4	x264_13	16.42	51.65
4	x264_15	20.14	50.68
5	x264_14	27.24	42.74
5	x264_0	28.63	49.77

SAC

Rank	Dataset	Median	IQR
1	sac_6	0.27	0.14
2	sac_4	0.96	4.26
2	sac_8	1.04	3.67
2	sac_9	2.29	4.98
3	sac_5	10.8	89.65

STORM

Rank	Dataset	Median	IQR
1	storm_feature9	0.0	0.0
1	storm_feature8	0.0	0.0
1	storm_feature6	0.0	0.01
1	storm_feature7	0.01	0.04

SPEAR

Rank	Dataset	Median	IQR
1	spear_7	0.1	0.1
1	spear_6	0.1	0.2
1	spear_1	0.1	0.1
1	spear_9	0.1	0.5
1	spear_8	0.1	0.2
1	spear_0	0.1	0.91
2	spear_5	0.28	0.3
3	spear_4	0.6	1.17
4	spear_2	1.09	5.31
5	spear_3	1.89	4.48

SQLITE

Rank	Dataset	Median	IQR
1	sqlite_17	0.8	1.13
1	sqlite_59	2.0	3.44
1	sqlite_19	2.0	4.88
2	sqlite_44	1.96	6.91
2	sqlite_16	2.52	7.41
2	sqlite_73	2.82	7.24
2	sqlite_45	3.47	11.86
2	sqlite_10	3.88	6.92
2	sqlite_96	4.94	6.04
2	sqlite_79	5.64	5.24
2	sqlite_11	6.64	5.75
2	sqlite_52	6.84	7.95
2	sqlite_97	7.68	13.71
3	sqlite_18	13.17	54.68
3	sqlite_94	27.43	47.66

Figure 4.6 Median NAR of 30 repeats. Median NAR is the normalized absolute residual values as described in Equation 4.2, and IQR the difference between 75th percentile and 25th percentile found during multiple repeats. Lines with a dot in the middle (—●—), show the median as a round dot within the IQR. All the results are sorted by the median NAR: a lower median value is better. The left-hand column (*Rank*) ranks the various techniques where lower ranks are better. Overall, we find that there is always at least one environment, denoted in blue, that is much superior (lower NAR) to others.

2. The remaining environments are used as targets. For every target environment, we use the regression tree model constructed above to predict for the best configuration.
3. Then, we measure the NAR of the predictions (see §4.2.7.2).
4. Afterwards, we repeat steps 1, 2, and 3 for all the other source environments and gather the outcomes.

We repeat the whole process above 30 times and use the Scott-Knott test to rank each environment best to worst.

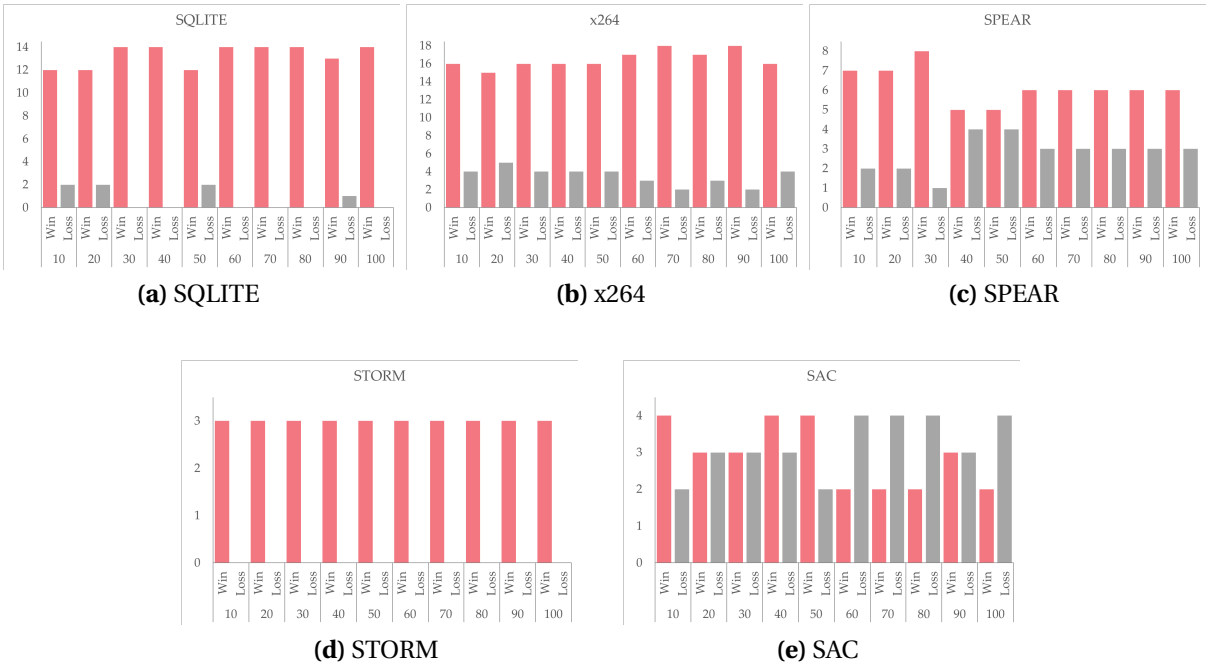
Summary: Our results are shown in Fig. 4.6. Overall, we find that there is always at least one

environment (the bellwether environment) in all the subject systems, that is much superior to others. Note that, STORM is an interesting case, here all the environments are ranked 1, which means that all the environments are equally useful as a bellwether environment —in such cases, any randomly selected environment could serve as a bellwether. Further, we note that the variance in the bellwether environments are much lower compared to other environments. Low variance indicates the low median NAR is not an effect of randomness in our experiments and hence increases our confidence in the existence of bellwethers.

Please note, in this specific experiment, we use *all* measured configurations (i.e., 100% of $|C|$ in Table 4.1) to determine if bellwethers exist. This ensures that the existance of bellwethers is not biased by how we sampled the configuration space. Later, in RQ-4.2, we will restrict our study to determine what fraction of the samples would be adequate to find the bellwethers.

One may be tempted to argue that the answer to this question trivially could be answered as "yes" since it is unlikely that all environments exhibit identical performance and there will always be some environment that can make better predictions. However, observe that the environments ranked first performs much better than the rest (with certain exceptions), and hence, the difference between the bellwether environment and others is not coincidental.

Result: There exists environments in each subject system, which act as the bellwether environment and hence can be used to find the near-optimal configuration for the rest of the environments.



(a) SQLITE **(b)** x264 **(c)** SPEAR

(d) STORM **(e)** SAC

Figure 4.7 Win/Loss analysis of learning from the bellwether environment and target environment using Scott Knott. The x-axis represents the percentage of data used to build a model and y-axis is the count. BEETLE wins in all software systems (since BEETLE wins more times than losses) except for SAC– and only when we have measured more that 50% of the data.

RQ-4.2: How many measurements are required to discover bellwether environments?

Purpose: The bellwether environments found in RQ-4.1 required us to use 100% of the measured performance values from all the environments[2]. Sampling all configurations may not be practical, since that may take an extremely long time [Jam17b]. Thus, in this research question, we ask if we can find the bellwether environments sooner using fewer samples.

[2]Note, except for SPEAR, we only have measured a subset of all possible configuration space since we were limited by the time and the cost required to make exhaustive measurements

Table 4.2 Effectiveness of source selection method.

Subject System	Bellwether Environment		Predicted Bellwether Environment	
	Median	**IQR**	**Median**	**IQR**
SQLite	0.8	1.13	1.8	2.48
Spear	0.1	0.1	0.1	0
x264	0.35	1.62	0.9	1.06
Storm	0.0	0.0	0.0	0.0
SaC	0.27	0.14	0.63	7.4

Approach: We used the racing algorithm discussed in Section §4.2.3.1. To see if our proposed method is effective, we compare the performance of bellwether environment with the predicted bellwether environment. It works as follows:

1. We start from 10% of configurations from each environment and assume that every environment is a potential bellwether environment.
2. Then, we increment the number of configurations in steps of 10% and measure the NAR values.
3. We rank the environments and eliminate those that do not show much promise. A detailed description of how this is accomplished can be found in §4.2.3.
4. We repeat the above steps until we cannot eliminate any more environments.

Result: Table 4.2 summarizes our findings. We find that,

- In all 5 cases, using **at most** 10% of the configurations we find one of the bellwether environments that are found using 100% of the measured configurations. In Table 4.2, the second and the third column represent the median and IQR of the NAR values found using the Bellwether environment, which is found using 100% of the configurations (ground truth). The fourth and the fifth column (under Predicted Bellwether Environment)

represent the median, and the IQR values found using the racing algorithm.

- The NAR of the predicted bellwether environments with 10% of the configurations is different by < 1% from the bellwether found at 100%.

These results are most encouraging in that we need only about 10% of the samples to determine the bellwether:

> ***Result:*** The bellwether environment can be recognized using only a fraction of the measurements (under 10%). Encouragingly, the identified bellwether environments have similar NAR values to the bellwether environment with 100% of samples. More running fewer configuration takes less time and is cheaper.

RQ-4.3: How does BEETLE compare to state-of-the-art methods?

Purpose: The main motivation of this work is to show that the source environment can have a significant impact on transfer learning. In this research question, we seek to compare BEETLE with other state-of-the-art transfer learners by Jamshidi et al. [Jam17a] and Valov et al. [Val17].

Approach: We perform transfer learning the methods proposed by Valov et al. [Val17] and Jamshidi et al. [Jam17a]. Then we measure the NAR values and compare them statistically using Skott-Knott tests. Finally, we rank the methods from best to worst based on their Skott-Knott ranks.

Result: Our results are shown in Fig. 4.8. In this figure, the best transfer learner is ranked 1. We note that in 4 out of 5 cases, BEETLE performs just as well as (or better than) the state-of-the-art. This result is encouraging in that it points to a significant impact on choosing a good source environment can have on the performance of transfer learners. Further, in Figure 4.9 we compare the number of performance measurements required to construct

SAC

Rank	Learner	Median	IQR
1	Jamshidi et al. [Jam17a]	1.58	5.39
2	BEETLE	6.89	99.1
2	Valov et al. [Val17]	6.99	99.24

SPEAR

Rank	Learner	Median	IQR
1	Jamshidi et al. [Jam17a]	0.70	1.29
1	BEETLE	0.79	1.40
1	Valov et al. [Val17]	1.11	1.98

SQLITE

Rank	Learner	Median	IQR
1	BEETLE	5.41	9.28
2	Valov et al. [Val17]	6.96	12.91
3	Jamshidi et al. [Jam17a]	18.51	50.85

STORM

Rank	Learner	Median	IQR
1	BEETLE	0.04	0.06
1	Jamshidi et al. [Jam17a]	0.86	20.69
2	Valov et al. [Val17]	2.47	53.98

X264

Rank	Learner	Median	IQR
1	BEETLE	8.67	27.01
2	Valov et al. [Val17]	16.99	41.24
3	Jamshidi et al. [Jam17a]	43.58	28.39

Figure 4.8 Comparison between state-of-the-art transfer learners and BEETLE. The best transfer learner is shaded blue. The "ranks" shown in the left-hand-side column come from the statistical analysis described in §4.2.8.

the transfer learners (note the logarithmic scale on the vertical axis). The total number of available samples for each software system is shown in the second column of Table 4.1 (see values corresponding to $|C|$). Of these we used only:

1. *x264*: 10.21% of 4000 samples
2. *SQLite*: 11.42% of 1000 samples
3. *Spear*: 13.79% of 16384 samples
4. *SaC*: 15.4% of 846 samples
5. *Storm*: 17.40% of 2048 samples

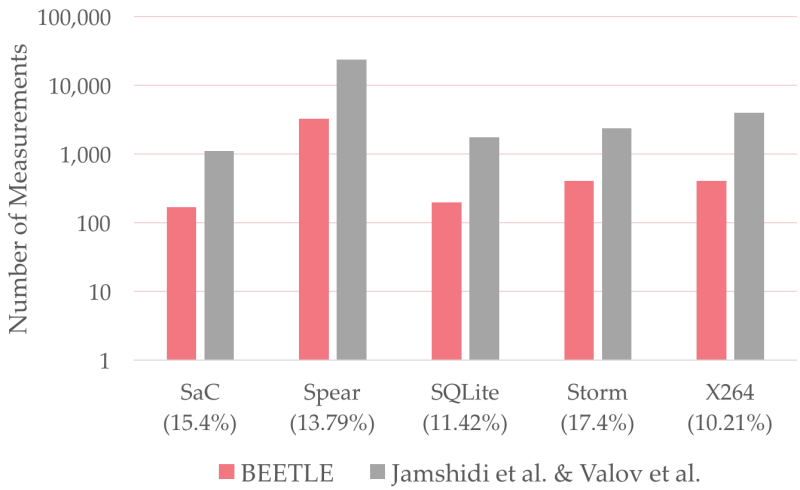


Figure 4.9 BEETLE v/s state-of-the-art transfer learners. The numbers in parenthesis represent the numbers of measurements BEETLE uses in comparison to the state-of-the-art learners.

Based on these results, we note that BEETLE requires far fewer measurements compared to the other transfer-learning methods. That is,

Result: In most software systems, BEETLE performs just as well as (or better than) other state-of-the-art transfer learners for performance optimization using far fewer measurements.

4.4 Summary

Our approach, BEETLE, exploits the bellwether effect—there are one or more bellwether environments which can be used to find good configurations for the rest of the environments. We also propose a new transfer learning method, called BEETLE, which exploits this phenomenon. As shown in this chapter, BEETLE can quickly identify the bellwether environments with only a few measurements ($\approx 10\%$) and use it to find the near-optimal solutions in the target environments. Further, after extensive experiments with five highly-configurable systems demonstrating that BEETLE:

- Identifies suitable sources to construct transfer learners;
- Finds near-optimal configurations with only a small number of measurements (an average of $\leq 13.5\% \approx \frac{1}{7}^{th}$ of the available number of samples);
- Performs as well as non-transfer learning approaches; and
- Performs as well as state-of-the-art transfer learners.

Based on our experiments, we demonstrate our initial problem–"whence to learn?" is an important question, and,

> A *good source* with a simple transfer learner is *better than source agnostic* complex transfer learners.

We show BEETLE can help answer this question effectively.

CHAPTER

5

FROM PREDICTION TO PLANNING

5.1 Why do we need Planning?

Despite the success of software analytics, there exists some drawbacks with current software analytic tools. At a recent workshop on "Actionable Analytics" at the 2015 IEEE conference on Automated Software Engineering, business users were very vocal in their complaints about analytics [HM15]. "Those tools tell us *what is*," said one business user, "But they don't tell us *what to do*".

Accordingly, in this research, we seek new tools that support actionable analytics that offer clear guidance on "what to do" about a specific software project. We seek new tools

since the current generation of software analytics tools are mostly *prediction* algorithms such as support vector machines [CV95], naive Bayes classifiers [Les08], logistic regression [Les08], etc. For example, defect prediction tools report what combinations of software project features predict for some dependent variable (such as the number of defects). Note that this is a different task to *planning*, which answers the question: what to *change* in order to *improve* quality.

More specifically, we seek plans that offer *least* changes in software but most *improve* the *quality* where:

- *Quality* = defects reported by the development team;
- *Improvement* = lowered likelihood of future defects.

		Data source		
		Within	Cross	
Task	Prediction	TSE '07 [Men07b]	EMSE '09 [Tur09] ASE '16 [Kri16c] TSE '18 [KM18]	TSE '17 [Nam17]
	Planning	IST '17 [Kri17b]	This work	Future work
		Homogeneous		Heterogeneous

Figure 5.1 Relationship of this work to our prior research. Within project trained and tested data miners using data from the same project. Cross projects train on one project, then test on another. Homogeneous learning requires the attribute names to be identical in the training and test set. Heterogeneous learning relaxes that requirement; i.e. the attribute names might change from the training to the test set.

5.1.1 Relationship to Previous Work

As for the connections to previous research, this chapter is an extension of those results. As shown in Figure 5.1, originally in 2007 we explored software quality prediction in the

context of training and testing within the same software project [Men07b]. After that we found ways in 2009 to train these predictors on some projects, then test them on others [Tur09]. Subsequent work in 2016 found that bellwethers were a simpler and effective way to implement transfer learning [Kri16c], which worked well for a wide range of software analytics tasks [KM18]. Meanwhile, in the area of planning, we conducted some limited within-project planning in 2017 on recommending what to change in software [Kri17b]. This current article now addresses a much harder question: can plans be generated from one project and applied to the another? While answering this question, we have endeavored to avoid our mistakes from the past, e.g., the use of overly complex methodologies to achieve a relatively simpler goal. Accordingly, this work experiments with bellwethers to see if this simple method works for planning as with prediction.

One assumption across much of our work is the *homogeneity* of the learning, i.e., although the training and testing data may belong to different projects, they share the same attributes [Kri16c; Kri17b; KM18; Men07b; Tur09]. Since that is not always the case, we have recently been exploring heterogeneous learning where attribute names may change between the training and test sets [Nam17]. Heterogeneous planning is primary focus of our future work.

this work extends a short abstract presented at the IEEE ASE'17 Doctoral Symposium [Kri17a]. Most of this work, including all experiments, did not appear in that abstract.

5.2 Research Questions

RQ-5.1: How effective is within-project planning with XTREE?

In this research question, we assess the effectiveness of XTREE when it is trained on past data from *within* a project. This research question follows RQ-5.1, where we showed that changes recommended by XTREE has a significant overlap with changes actually undertaken by developers. XTREE recommends changes so that the project might have have fewer defects in the next version, however correlation does not necessarily imply causation, it is theoretically possible that the recommendations generated by XTREE could be misleading.

The results of this work are in accordance with our previous findings [Kri17b] and show that *XTREE* is an effective planner for automatically recommending useful plans.

Result: For within-project planning, we see that greater the overlap with XTREE's recommendations the larger the number of defects reduced in a subsequent release of a project. This was true in 8 out of 9 projects studied here.

RQ-5.2: How does within-project XTREE compare with other threshold based methods?

Alternative methods for planning make use of statistically determined thresholds over source code metrics for reducing defects. Recent work by Shatnawi [Sha10b], Alves et al. [Alv10], and Oliveira et al. [Oli14] assume that unusually large measurements in source code metrics point to larger likelihood of defects and these should be avoided since, if left unchanged, they would lead to defect-prone code.

Our results show that their assumption is not effective to generate actionable plans and

that XTREE is a better way to plan quality improvement.

Result: For within-project planning, plans generated by XTREE is significantly superior to plans generated by other threshold based methods.

RQ-5.3: How many changes do the planners propose?

In the final research question, we ask how many attributes are recommended to be changed by different planners.

This result has a lot of practical significance since developers have a hard time following those plans that recommend too many changes.

Result: XTREE recommends the least number of changes compared to other planners, while also producing the best overall performance (measured in terms of defect reduction).

5.3 Planning in Software Analytics

Planning is extensively explored in artificial intelligence research. There, it usually refers to generating a sequence of actions that enables an *agent* to achieve a specific *goal* [RN95]. This can be achieved by classical search-based problem solving approaches or logical planning agents. Such planning tasks now play a significant role in a variety of demanding applications, ranging from controlling space vehicles and robots to playing the game of bridge [Gha04]. Some of the most common planning paradigms include: (a) classical planning [WJ95]; (b) probabilistic planning [Bel57; Alt99; GHL09]; and (c) preference-based planning [SP06; BM09]. Existence of a model precludes the use of each of these planning approaches. This is a limitation of all these planning approaches since not every domain has a reliable model.

A survey of literature reveals at least two other kinds of planning research in SE. Each kind is distinguishable by *what* is being changed.

- In *test-based planning*, some optimization is applied to reduce the number of tests required to achieve to a certain goal or the time taken before tests yield interesting results [TG06; YH12; Blu13].
- In *process-based planning* some search-based optimizer is applied to a software process model to infer high-level business plans about software projects. Examples of that kind of work include our own prior studies combining simulated annealing with the COCOMO models or Ruhe et al.'s work on next release planning in requirements engineering [RG03; Ruh10].

In software engineering, the planning problem translates to proposing changes to software artifacts. These are usually a hybrid task combining probabilistic planning and preference-based planning using search-based software engineering techniques [Har09; Har11]. These search-based techniques are evolutionary algorithms that propose actions guided by a fitness function derived from a well established domain model. Examples of algorithms used here include GALE, NSGA-II, NSGA-III, SPEA2, IBEA, MOEA/D, etc. [**deb00a**; Kra15; Zit02; ZK04; DJ14; Cui05; ZL07]. As with traditional planning, these planning tools all require access to some trustworthy models that can be used to explore some highly novel examples. In some software engineering domains there is ready access to such models which can offer assessment of newly generated plans. Examples of such domains within software engineering include automated program repair [Wei09; Le 12; Le 15], software product line management [Say13; MP14; Hen15], automated test generation [And07; And10], etc.

However, not all domains come with ready-to-use models. For example, consider all the intricate issues that may lead to defects in a product. A model that includes *all* those potential issues would be very large and complex. Further, the empirical data required

to validate any/all parts of that model can be hard to find. Worse yet, our experience has been that accessing and/or commissioning a model can be a labor-intensive process. For example, in previous work [Men07d] we used models developed by Boehm's group at the University of Southern California. Those models took as inputs project descriptors to output predictions of development effort, project risk, and defects. Some of those models took decades to develop and mature (from 1981 [Boe81a] to 2000 [Boe00]). Lastly, even when there is an existing model, they can require constant maintenance lest they become out-dated. Elsewhere, we have described our extensions to the USC models to enable reasoning about agile software developments. It took many months to implement and certify those extensions [Ii09; Lem09]. The problem of model maintenance is another motivation to look for alternate methods that can be quickly and automatically updated whenever new data becomes available.

In summary, for domains with readily accessible models, we recommend the kinds of tools that are widely used in the search-based software engineering community such as GALE, NSGA-II, NSGA-III, SPEA2, IBEA, particle swarm optimization, MOEA/D, etc. In other cases where this is not an option, we propose the use of data mining approaches to create a quasi-model of the domain and make of use observable states from this data to generate an estimation of the model. Examples of such data mining approaches are described below. These include three methods described in the rest of this section: Alves et al. [Alv10], Shatnawi [Sha10b], and Oliveira et al. [Oli14]

5.3.1 Alves

Alves et al. [Alv10] proposed an unsupervised approach that uses the underlying statistical distribution and scale of the OO metrics. It works by first weighting each metric value

according to the source lines of code (SLOC) of the class it belongs to. All the weighted metrics are then normalized by the sum of all weights for the system. The normalized metric values are ordered in an ascending fashion (this is equivalent to computing a density function, in which the x-axis represents the weight ratio (0-100%), and the y-axis the metric scale).

Alves et al. then select a percentage value (they suggest 70%) which represents the "normal" values for metrics. The metric threshold, then, is the metric value for which 70% of the classes fall below. The intuition is that the worst code has outliers beyond 70% of the normal code measurements i.e., they state that the risk of there existing a defect is moderate to high when the threshold value of 70% is exceeded.

Here, we explore the correlation between the code metrics and the defect counts with a univariate logistic regression and reject code metrics that are poor predictors of defects (i.e. those with $p > 0.05$). For the remaining metrics, we obtain the threshold ranges which are denoted by $[0, 70\%)$ ranges for each metric. The plans would then involve reducing these metric range to lie within the thresholds discovered above.

5.3.2 Shatnavi

Shatnawi [Sha10b] offers a different alternative Alves et al by using VARL (Value of Acceptable Risk Level). This method was initially proposed by Bender [Ben99] for his epidemiology studies. This approach uses two constants (p_0 and p_1) to compute the thresholds which Shatnawi recommends to be set to $p_0 = p_1 = 0.05$. Then using a univariate binary logistic regression three coefficients are learned: α the intercept constant; β the coefficient for maximizing log-likelihood; and p_0 to measure how well this model predicts for defects. (Note: the univariate logistic regression was conducted comparing metrics to defect counts.

Any code metric with $p > 0.05$ is ignored as being a poor defect predictor.)

Thresholds are learned from the surviving metrics using the risk equation proposed by Bender:

$$Defective\ if\ Metric > VARL$$

Where,

$$VARL = p^{-1}(p_0) = \frac{1}{\beta}\left(\log\left(\frac{p_1}{1-p_1}\right) - \alpha\right)$$

In a similar fashion to Alves et al., we deduce the threshold ranges as $[0, VARL)$ for each selected metric. The plans would again involve reducing these metric range to lie within the thresholds discovered above.

5.3.3 Oliveira

Oliveira et al. in their 2014 paper offer yet another alternative to absolute threshold methods discussed above [Oli14]. Their method is still unsupervised, but they propose complementing the threshold by a second piece of information called the *relative threshold*. This measure denotes the percentage of entities the upper limit should be applied to. These have the following format:

$$p\%\ of\ the\ entities\ must\ have\ M \leq k$$

Here, M is an OO metric, k is the upper limit of the metric value, and p (expressed as %) is the minimum percentage of entities are required to follow this upper limit. As an example Oliveira et al. state, " ... 85% of the methods should have $CC \leq 14$. Essentially, this threshold expresses that high-risk methods may impact the quality of a system when they represent more than 15% of the whole population of methods ..."

The procedure attempts derive these values of (p, k) for each metric M. They define

a function `Compliance(p,k)` that returns the percentage of system that follows the rule defined by the relative threshold pair (p,k). They then define two penalty functions: (1) `penalty1(p,k)` that penalizes if the compliance rate is less than a constant $Min\%$, and (2) `penalty2(k)` to define the distance between k and the median of preset $Tail$-th percentile. (Note: according to Oliveira et al., median of the tail is an idealized upper value for the metric, i.e., a value representing classes that, although present in most systems, have very high values of M). They then compute the total penalty as `penalty` = `penalty1(p,k)` + `penalty2(k)`. Finally, the relative threshold is identified as the pair of values (p,k) that has the lowest total `penalty`. After obtaining the (p,k) for each OO metric. As in the above two methods, the plan would involve ensuring the for every metric M $p\%$ of the entities have a value that lies between $(0,k]$[1].

5.4 XTREE

5.4.1 Construction

XTREE is a cluster delta algorithm that avoids the problem of verbose Δs. XTREE is our *preferred planning algorithm* that makes recommendations of what changes should be made to code modules.

Planning with XTREE is comprised of three steps namely, (a) Frequent pattern mining; (b) Decision tree construction; and (c) Planning with random walk traversal.

Step-1: Frequent pattern mining. The first step in XTREE is to determine which metrics are most often changed together. The OO metrics are not independent of each other. In other words, changing one metric (say *LOC*) would lead to a corresponding change in other

[1]If certain metrics already satisfy this criterion, they remain unchanged

	rfc	loc	dit	cbo	Bugs
1.java	0.6	100	1	4	0
2.java	0.9	223	4	5	1
3.java	1.1	290	5	7	1
4.java	2.1	700	10	12	3
5.java	2.3	800	11	15	3

(a)

	rfc	loc	dit	cbo
1.java	A	A	A	A
2.java	A	A	B	A
3.java	A	A	B	A
4.java	B	B	C	B
5.java	B	B	C	B

(b)

	Items
1.java	rfc_A, loc_A, dit_A, cbo_A
2.java	rfc_A, loc_A, dit_B, cbo_A
3.java	rfc_A, loc_A, dit_B, cbo_A
4.java	rfc_B, loc_B, dit_C, cbo_B
5.java	rfc_B, loc_B, dit_C, cbo_B

(c)

Items (`min_sup=60`)	Support
rfc_A	60
loc_A	60
~~dit_A~~	~~40~~
$\{rfc_A, loc_A\}, \{loc_A, cbo_A\}, \ldots$	60
$\{rfc_A, loc_A, cbo_A\}$	60
~~$\{rfc_A, loc_A, cbo_A, dit_{B,C}\}$~~	~~40~~

(d)

Figure 5.2 To determine which of metrics are usually changed together, we use frequent itemset mining. Our dataset is continuous in nature (see (a)) so we first discretize using Fayyad-Irani [FI93]; this gives us a representation shown in (b). Next, we convert these into "transactions" where each file contains a list of discretized OO-metrics (see (c)). Then we use the *FP-growth* algorithm to mine frequent itemsets. We return the *maximal frequent itemset* (as in (d)). Note: in (d) the row in green is the maximal frequent itemset.

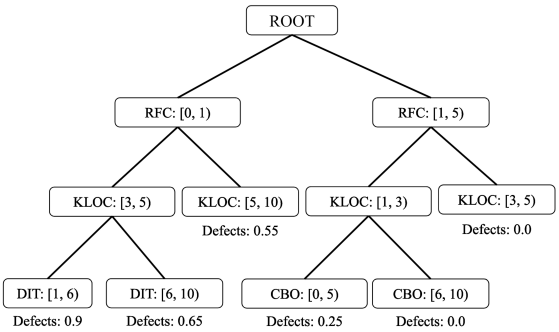
metrics (such as *CBO*). We refrain from using correlation to determine which metrics change together because correlation measures the existence of a monotonic relationships between two metrics. We cannot assume that the metrics are monotonically related; moreover, it is possible that more than two metrics are related to each other. Therefore, we use frequent pattern mining [Han07], which represents a more generalized relationship between metrics, to detect which of the metrics change together.

Our instrumentation is shown in Figure 5.2. We use the FP-Growth algorithm [Han07] to identify the *maximal frequent itemset* (highlighted in green in Figure 5.2 (d)). This represents the longest set of metrics that change together atleast *support*% (in our case 60%)

Algorithm 1 N-ary Decision Tree

```
procedure NARY_DTREE(train)
    features = train[train.columns[:-1]]
    for f ∈ features do
        Find splits using Fayyad-Irani method
        Compute expected entropy of splits
    end for
    f_best ← Feature with lowest mean entropy
    Tree ← Tree.add_node(f_best)
    D_v ← Induced sub-datasets from train based on
f_best
    for d ∈ D_v do
        Tree_v ← NARY_DTREE(d)
        Tree ← Tree_v
    end for
return Tree
end procedure
```

(a) Decision Tree Algorithm



(b) Example decision tree

Figure 5.3 To build the decision tree, we find the most informative feature,i.e., the feature which has the lowest mean entropy of splits and construct a decision tree recursively in a top-down fashion as show above.

of the time. The following steps use the *maximal frequent itemset* to guide the generation of plans.

Step-2: Decision tree construction. Having discovered which metrics change together, we next establish what range of values for each metrics point to a high likelihood of defects. For this we use a decision tree algorithm (see Figure 5.3). For this, XTREE uses a multi-interval discretizer based on an iterative dichotomization scheme, first proposed by Fayyad and Irani [FI93]. This method converts the values for each code metric into a small number of nominal ranges. It works as follows:

- A code metric is split into r ($r = 2$) ranges, each range is of size n_r and is associated with a set of defect counts x_r with standard deviation σ_r.
- The best split for that range is the one that minimizes the expected value of the defect

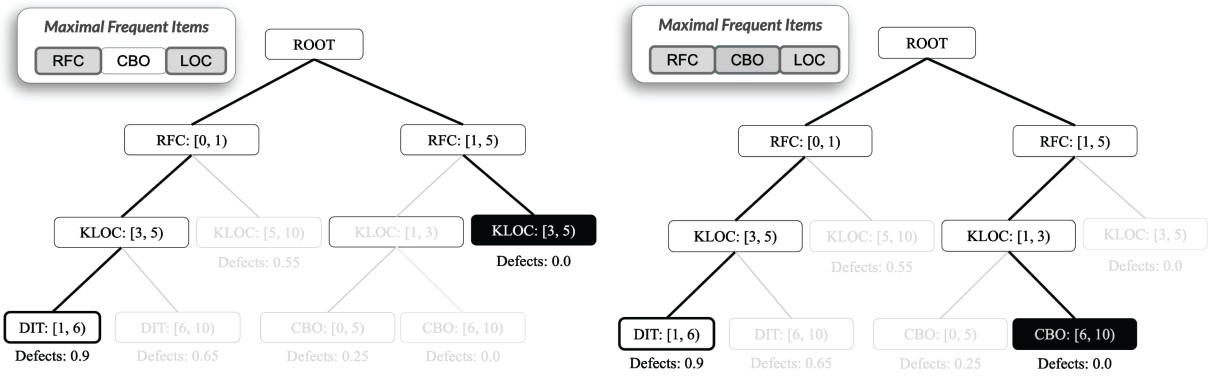


Figure 5.4 For ever test instance, we pass it down the decision tree constructed in Figure 5.3. The node it lands is called the "start". Next we find all the "end" nodes in the tree, i.e., those which have the lowest likelihood of defects (labeled in black below). Finally, perform a random-walk to get from "start" to "end". We use the mined itemsets from Figure 5.2 to guide the walk. When presented with multiple paths, we pick the one which has the largest overlap with the frequent items. e.g., in the below example, we would pick path (b) over path (a).

variance, after the split; i.e. $\sum_r \frac{n_r}{n}\sigma_x$ (where $n = \sum_r n_r$).

- This discretizer then recurses on split to find other splits in a recursive fashion. As suggested by Fayyad and Irani, minimum description length (MDL) is used as a termination criterion for the recursive partitioning.

When discretization finishes, each code metric M has a final expected value M_v for the defect standard deviation across all the discretized ranges of that metric. Iterative dichomization sorts the metrics by M_v to find the code metric that best splits the data i.e., the code metric with smallest M_v.

A decision tree is then constructed on the discretized metrics. The metric that generated the best split forms the root of the tree with its discrete ranges acting as the nodes.

When all the metrics are arranged this way, the process is very similar to a hierarchical clustering algorithm that groups together code modules with similar defect counts and

some shared ranges of code metrics. For our purposes, we score each cluster found in this way according to the probability of classes with known defects. For example, in the case of Figure 5.3(b), if $rfc = [0,1)$, $KLOC = [3,5)$, and $DIT = [1,6)$ then the probability of defect is 0.9.

Step-3: Random Walk Traversal. With the last two steps, we now know (1) which metrics change together and (2) what ranges of metrics indicate a high likelihood of defects. with this information, XTREE builds plans from the branches of the tree as follows. Given a "defective" test instance, we ask:

1. Which *current* node does the test instance fall into?
2. What are all the *desired* nodes the test case would want to emulate? These would be nodes with the *lowest* defect probabilities.

Finally, we implement a random-walk [Yin18; Sha16] model to find paths that lead from the *current* node the *desired* node. Of all the paths that lead from the *current* node to the *desired* node, we select the path that has the highest overlap with the *maximal frequent itemset*. As an example, consider Figure 5.4. Here, of the two possible paths Figure 5.4(a) and Figure 5.4(b), we choose that latter because it traverses through all the metrics in the maximal frequent itemset.

5.4.2 How are plans generated?

The path taken by the random-walk is used to generate a plan. For example, in the case of Figure 5.4, it works as follows:

1. The test case finds itself on the far left, i.e., the "current node" has: $RFC : [0,1)$, $KLOC : [3,5)$ and $DIT : [1,6)$
2. After implementing the random walk, we find that "desired" node is on the far right

(highlighed in black)

3. The path taken to get from the "current node" to the "desired node" would require that the following changes be made.
   - $RFC : [0,1) \longrightarrow [1,5)$;
   - $KLOC : [0,1) \longrightarrow [1,3)$; and
   - $CBO : [6,10)$

   The plan would then be these ranges of values.

5.5 Cross-project Planning with BELLTREE

Many methods have been proposed for transferring data or lessons learned from one project to another. Transfer learning with bellwethers is just a matter of calling existing learners inside a for-loop. For all the training data from different projects, a bellwether learner conducts a round-robin experiment where a model is learned from project, then applied to all others. The *bellwether* is that project which generates the best performing model. The *bellwether effect*, states that models learned from this bellwether performs as well as, or better than, other transfer learning algorithms.

For the purposes of prediction, we have shown previously that bellwethers are remarkably effective for many different kinds of SE tasks such as (i) defect prediction, (ii) performance optimization, (iii) effort estimation, and (iv) detecting code smells [KM18]. Since planning takes from prediction, it is valuable to check the value of bellwethers for the purposes of planning. Note also that use of bellwethers enables us to generate plans from different data sets from across different projects.

BELLTREE extends the three bellwether operators defined in our previous work [KM18] on bellwethers: DISCOVER, PLAN, VALIDATE. That is:

1. DISCOVER: *Check if a community has bellwether.* This step is similar to our previous technique used to discover bellwethers [Kri16c]. We see if standard data miners can predict for the number of defects, given the static code attributes. This is done as follows:
   - For a community C obtain all pairs of data from projects $P, Q, R, S...$ such that $x, y \in C$;
   - Predict for defects in y using a quality predictor learned from data taken from x;
   - Report a bellwether if one x generates consistently high predictions in a majority of $y \in C$.

   Note, since the above steps perform an all-pairs comparison, the theoritical complexity of the DISCOVER phase will be be $O(N^2)$ where N is the number of projects.
2. PLAN: *Using the bellwether, we generate plans that can improve a new project.* That is, having learned the bellwether on past data, we now construct a decision tree similar to within-project XTREE. We then use the same methodology to generate the plans.
3. VALIDATE: *Go back to step 1* if the performance statistics seen during PLAN fail to generate useful actions.

5.6 Experimental Setup

It can be somewhat difficult to judge the effects of applying plans to software projects. These plans cannot be assessed just by a rerun of the test suite for three reasons: (1) The defects were recorded by a post release bug tracking system. It is entirely possible it escaped detection by the existing test suite; (2) Rewriting test cases to enable coverage of all possible scenarios presents a significant challenge; and (3) It may take a significant amount of effort to write new test cases that identify these changes as they are made.

To resolve this problem, SE researchers such as Cheng et al. [CJ10], O'Keefe et al. [OC08; OC07], Moghadam [Mog11] and Mkaouer et al. [Mka14] use a *verification oracle* learned

separately from the primary oracle. This oracles assesses how defective the code is before and after some code changes. For their oracle, Cheng, O'Keefe, Moghadam and Mkaouer et al. use the QMOOD quality model [BD02]. A shortcoming of QMOOD is that quality models learned from other projects may perform poorly when applied to new projects [Men13]. As a results, we eschew using these methods in favor of evaluation strategies discussed in the rest of this section.

5.6.1 The $\mathbb{K}$-test

In order to measure the extent to which the recommendations made by planning tools matches those undertaken by the developers, we assess the impact making those changes would have on an upcoming release of a project. For this purpose, we propose the $\mathbb{K}$-test.

We say that a project $\mathbb{P}$ is released in versions $\mathbb{V} \in \{\mathbb{V}_i, \mathbb{V}_j, \mathbb{V}_k\}$. Here, in terms of release dates, $\mathbb{V}_i$ precedes $\mathbb{V}_j$, which in turn precedes $\mathbb{V}_k$. We will use these three sets for *train*, *test*, and *validation*, respectively. These three sets are used as follows:

1. First, train the planner on version $\mathbb{V}_i$. Note: this could either be data that is either from a previous release, or it could be data from the bellwether project.
2. Next, use the planner to generate plans to reduce defects for files that were reported to be buggy in version $\mathbb{V}_j$.
3. Finally, on version $\mathbb{V}_k$, for *only* the files that were reported to be buggy in the previous release, we measure the OO metrics.

Having obtained the changes at version $\mathbb{V}_k$ we can now (a) measure the *overlap* between plans recommended by the planner and the developer's actions, and (b) count the number of defects reduced (or possibly increased) when compared to the previous release. Using these two measures, we can assess the impact of implementing these plans. Details on

	DIT	NOC	CBO	RFC	FOUT	WMC	NOM	LOC	LCOM
Version $\mathbb{V}_k$	3	4	4	2	5	2.5	3	400	6
$\mathbb{P} \rightarrow \mathbb{V}_{k+1}$	·	·	·	[4,7]	·	[3,6]	[4,7]	[1000,2000]	[1,4]
$\mathbb{D} \rightarrow \mathbb{V}_{k+1}$	3	4	3	5	3	5	4	1500	2

$$Overlap = \frac{|\mathbb{D} \cap \mathbb{P}|}{|\mathbb{D} \cup \mathbb{P}|} \times 100 = \frac{7}{9} \times 100 = 77.77\%$$

Figure 5.5 A simple example of computing overlap. Here a '·' represents *no-change*. Columns shaded in gray indicate a match between developer's changes and planner's recommendations.

measuring each of these are discussed in the subsequent parts of this section.

To compute that overlap, we proceeded as follows. Consider two sets of changes:

1. $\mathbb{D}$: The changes that developers made, perhaps in response to the issues raised in a post-release issue tracking system;
2. $\mathbb{P}$: The plans recommended by an automated planning tool, *overlap* attempts to compute the extent to which a developer's action matches that of the actions recommended by planners.

To measure this *overlap*, we use Jaccard similarity:

$$Overlap = \frac{|\mathbb{D} \cap \mathbb{P}|}{|\mathbb{D} \cup \mathbb{P}|} \times 100 \tag{5.1}$$

In other words, we measure the ratio of the size of the intersection between the developers plans and the size of all possible *changes*. Note that the *larger* the intersection between the changes made by the developers to the changes recommended by the planner, then the *greater* the overlap.

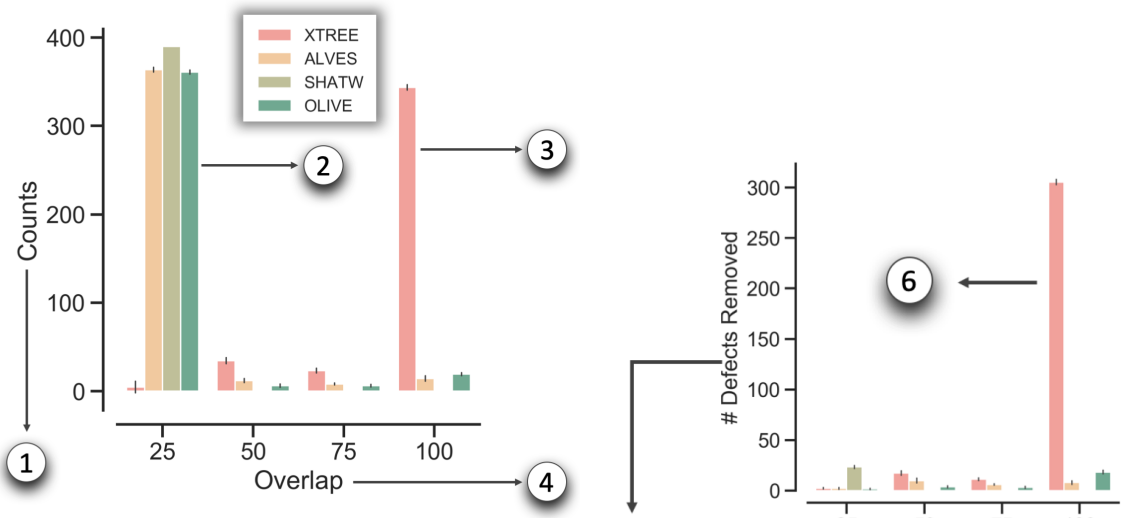
An simple example of how overlap is computed is illustrated in Figure 5.5. Here, we have 9 metrics and let's say a defective file version $\mathbb{V}_k$ has metric values corresponding to row labeled Version $\mathbb{V}_k$. The row labeled $\mathbb{P} \rightarrow \mathbb{V}_{k+1}$ contains set of treatments recommended by a planner $\mathbb{P}$ for version $\mathbb{V}_{k+1}$ (note that the recommendations are ranges of values rather than actual numbers). Finally, the row labeled $\mathbb{D} \rightarrow \mathbb{V}_{k+1}$ are the result of a developer taking

certain steps to possibly reduce the defects in the file for version $\mathbb{V}_{k+1}$. We see that in two cases (CBO and FOUT) the developers actions led to changes in metrics that were not prescribed by the planner. But in 7 cases, the developers actions matched the changes prescribed by the planner. Computing overlap as per Equation 5.1, produces an overlap value of 77%.

5.6.2 Presentation of Results

Using the $\mathbb{K}$-test and overlap counts defined above, we can measure the overlap between the planners' recommendations and developers actions. With this, plot three kinds of charts to discuss our results:

1. *Overlap vs. Counts*: A plot of overlap ranges (x-axis) versus the count of files that have that specific overlap range (on the y-axis). This is illustrated in Figure 5.6. Here the overlap counts (x-axis) have 4 ticks: 0 (labeled 100). We see that, in the case of XTREE, the number of files that have between 76%−100% overlap is significantly larger than any other overlap range. This implies that most of the changes recommended by $XTREE$ are exactly what the developers would have actually done. On the other hand, for the other three planners (Alves, Shatnawi, and Oliveira) the number of files that have between 0%−25% overlap is significantly larger than any other overlap range. This means that those planners' recommendation are seldom what developers actually do.
2. *Overlap vs. Defects reduced*: Just because there is an overlap, it does not necessarily mean that the defects were actually reduced. To measure what impact overlaps between planners' recommendations and developers actions have on reduction of defects, we plot a chart of overlap (x-axis) against the actual number of defects reduced. This is illustrated in Figure 5.6. The key distinction between this chart and the previous chart is



(1) Counts the number of files

(2) Number of files which have an overlap within the range between 0% to 25%

(3) Number of files which have an overlap within the range between 76% to 100%

(4) Overlap ranges from 0% to 100% in steps of 25%

(5) A count of #defects removed or #defects added.

(6) & (7) Number of defects removed (or added) at 76%–100% overlap. Note: the scales are different. Usually, #defects removed ≫ #defects added.

Figure 5.6 Sample charts to illustrate the format used to present the results.

the y-axis, here the y-axis represents the number of defects reduced. Larger y-axis values for larger overlaps are desirable because this means that more the developers follow a planners' actions, higher the number of defects reduced.

3. Overlap vs. Defects increased: It is also possible that defects are increased as a result of overlap. To measure what impact overlaps between planners' recommendations and

developers actions have on *increasing* defectiveness, we plot a chart of overlap (x-axis) against the actual number of defects increased. This is illustrated in Figure 5.6. The key distinction between this chart and the previous two charts is the y-axis, here the y-axis represents the number of defects *increased*. Lower y-axis values for larger overlaps are desirable because this means that more the developers follow a planners' actions, lower the number of defects increased.

5.7 Experimental Results

RQ-5.1: Do planners recommendations match developer actions?

To answer this question, we measure the *overlap* between the planners' recommendations and the developer's actions. To measure this, we split the available data into training, testing, and validation sets. That is, given versions $\mathbb{V}_1, \mathbb{V}_2, \mathbb{V}_3$...., we,

1. *train* the planners on version $\mathbb{V}_1$; then
2. *generate plans* using the planners for version $\mathbb{V}_2$;
3. then *validate* the effectiveness of those plans on $\mathbb{V}_2$ using the $\mathbb{K}$-test.

Then, we repeat the process by training on $\mathbb{V}_2$, testing on $\mathbb{V}_3$, and validating on version $\mathbb{V}_4$, and so on. For each of these {*train*, *test*, *validation*} sets, we measure the *overlap* and categorize them into 4 ranges:

- very little, i.e. $0-25\%$;
- some, i.e. $26\%-50\%$;
- more, i.e. $51\%-75\%$;
- mostly, i.e. $76\%-100\%$.

Figure 5.7 shows the results of planning with several planners: XTREE, Alves, Shatnawi, and

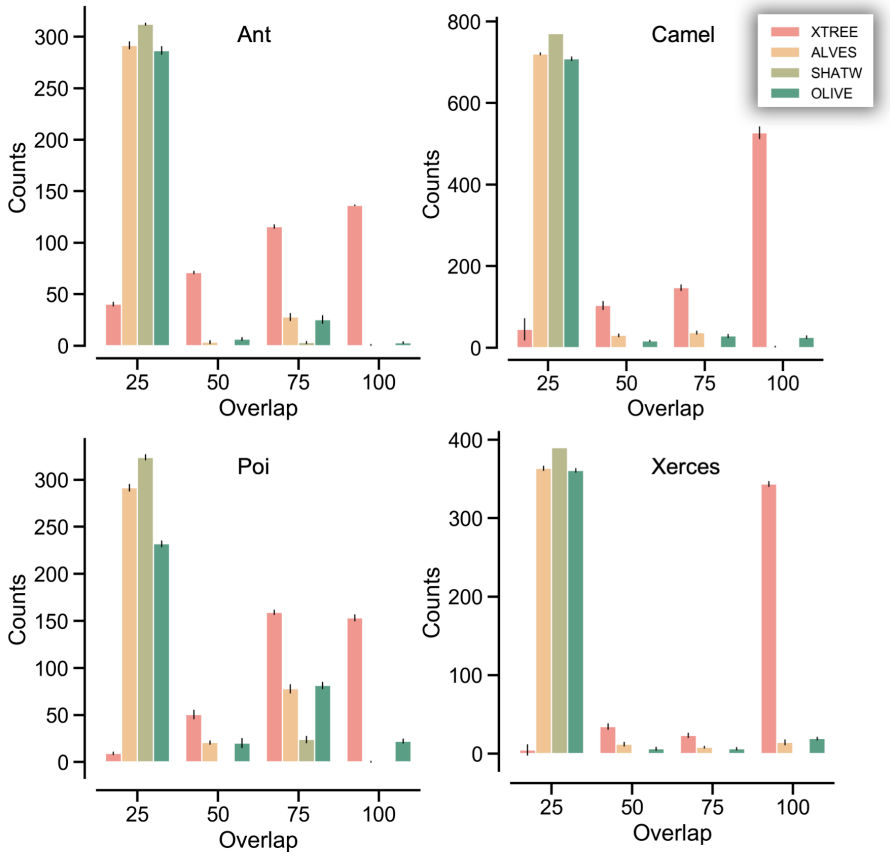


Figure 5.7 A count of number of test instances where the developer changes overlaps a planner recommendation. The overlaps (in the x-axis) are categorized into four ranges for every dataset (these are $0 \leq$ *Overlap* ≤ 25, $26 \leq$ *Overlap* ≤ 50, $51 \leq$ *Overlap* ≤ 75, and $76 \leq$ *Overlap* ≤ 100). For each of the overlap ranges, we count the the number of instances in the validation set where overlap between the planner's recommendation and the developers changes fell in that range. Note: *Higher counts* for larger overlap is *better*, e.g., $Count([75,100]) > Count([0,25))$ is considered better.

Oliveira. Note, for the sake of brevity, we illustrate results for 4 projects– Ant, Camel, Poi, and Xerces. A full set set results for all projects are available at `https://git.io/fjkNM`.

We observe a clear dichotomy in our results.

- All outlier statistics based planners (i.e., those of Alves, Shatnawi, and Oliveira) have overlaps only in the range of 0% to 25%. This means that *most of the developers actions did not match the recommendations proposed by these planners.*

- In the case of XTREE, the largest number of files had an overlap of 76% to 100% and second largest was between 51% to 75%. This means that, in a majority of cases developers actions are 76% to 100% similar to XTREE's recommendations. At the very least, there was an 51% similarity between XTREE's recommendations and developers actions.

We observe this trend in all 18 datasets– XTREE significantly outperformed other threshold based planners in terms of the overlap between the plans and the actual actions undertaken by the developers. Note that reason the results are very negative about the methods of Alves, Shatnawi, Oliveira, et al. is because their recommendations would be very hard to operationalize (since those recommendations were seldom seen in the prior history of a project). Thus, our response to this research question can be summarized as follows:

Result: XTREE significantly outperforms all the other outlier statistics based planners. Further, in all the projects studied here, most of the developer actions to fix defects in a file has as 76%–100% overlap with the recommendations offered by XTREE.

RQ-5.2: Do planners' recommendation lead to reduction in defects?

In the previous research question measured the extent to which a planner's recommendations matched the actions taken by developers to fix defects in their files. But, the existence of a high overlap in most files does not necessarily mean that the defects are actually reduced. Likewise, it is also conceivable that that defects are added due to other actions the developer took during their development. Thus, it is important to ask how many defects are reduced, and how many are added, in response to larger overlap with the planners' recommendations.

Our experimental methodology to answer this research question is as follows:

- Like before, we measure the *overlap* between the planners' recommendations developers'

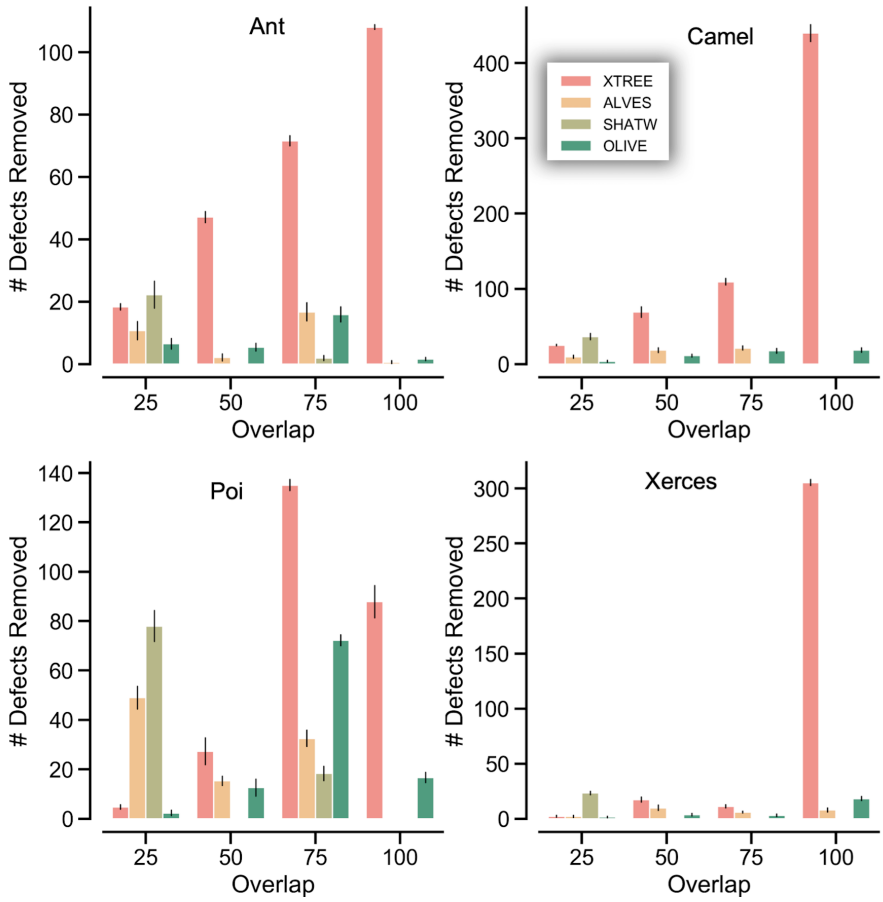


Figure 5.8 A count of total number *defects reduced* as a result each planners' recommendations. The overlaps are again categorized into four ranges for every dataset (denoted by $min \leq Overlap < max$). For each of the overlap ranges, we count the total number of *defects reduced* and in the validation set for the classes that were defective in the test set as a result of overlap between the planner's recommendation and the developers changes that fell in the given range

actions.

- Next, we plot the aggregate number defects reduced and in file with overlap values ranging from 0% to 100% in bins of size 25% (for ranges of 0−25%, 26−50%, 51−75%, and 76−100%).

Similar to RQ-5.1, we compare XTREE with three other outlier statistics based planners of Alves et al., Shatnawi, and Oliveira, for the overall number of defects reduced and number

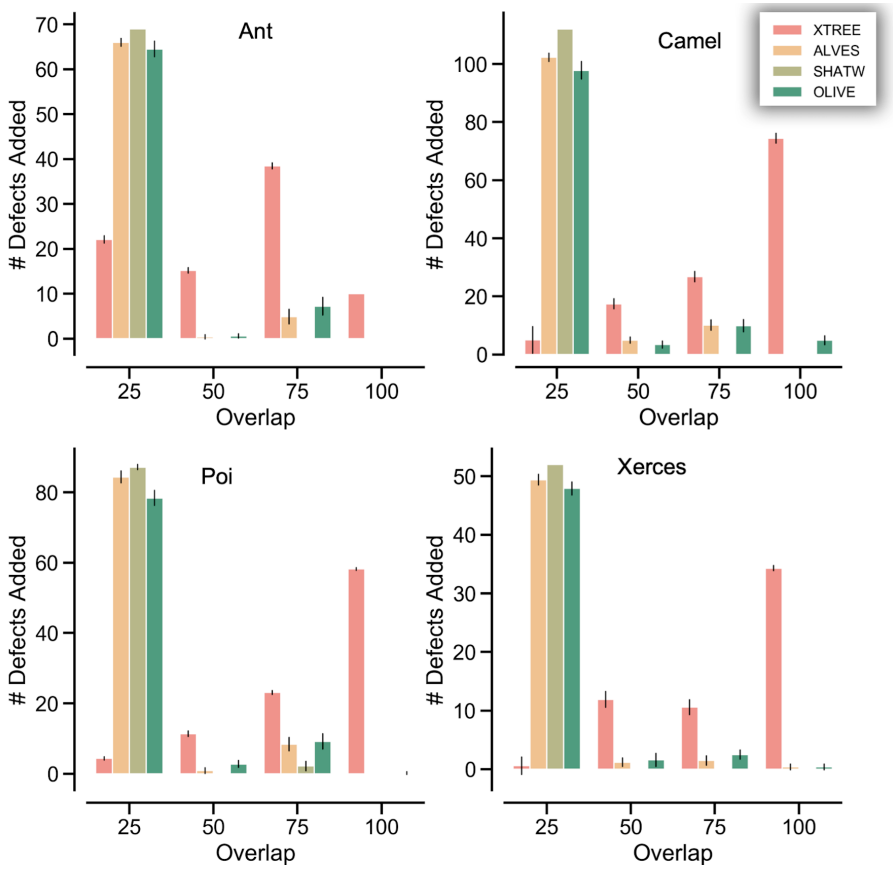


Figure 5.9 A count of total number *defects increased* as a result each planners' recommendations. The overlaps are again categorized into four ranges for every dataset (denoted by $min \leq Overlap < max$). For each of the overlap ranges, we count the total number of *defects increased* in the validation set for the classes that were defective in the test set as a result of overlap between the planner's recommendation and the developers changes that fell in the given range

of defects added. We prefer planners that have a large number defects reduced for higher overlap ranges are considered better.

Figure 5.8 and Figure 5.9 show the results of planning with several planners: XTREE, Alves, Shatnawi, and Oliveira. Note that, similar to the previous research question, we only illustrate results for 4 projects– Ant, Camel, Poi, and Xerces. A full set of results for RQ-5.2 for all projects are available at `https://git.io/fjIvG`.

We make the following observations from in our results:

1. *Defects Decreased*: Figure 5.8 plots the number of defects *removed* in files with various overlap ranges. It is desirable to see larger defects removed with larger overlap. We note that:
   - When compared to other planners, the number of defects removed as a result of recommendations obtained by XTREE is significantly larger. This trend was noted in all the projects we studied here.
   - In the cases of Ant, Camel, and Xerces there are large number of defect reduced when the overlap lies between 76% and 100%. Poi is an exception– here, we note that the largest number of defects are removed when the overlap is between 51% and 75%.
2. *Defects Increased*: Figure 5.9 plots the number of defects *added* in files with various overlap ranges. It is desirable to see lower number of defects added with larger overlap. We note that:
   - When compared to other planners, the number of defects added as a result of recommendations obtained by XTREE is comparatively larger. This trend was noted in all the projects we studied here. This is to be expected since, developers actions seldom match the recommendations of these other planners.
   - In all the cases the number of defects removed was significantly larger than the number of defects added. For example, in the case of Camel, 420+ defects were removed at 76% – 100% overlap and about 70 defects were added (i.e., 6× more defects were removed than added). Likewise, in the case of Xerces, over 300 defects were removed and only about 30 defects were added (i.e., 10× more defects were removed than added).

The ratio of defects removed to the number of defects added is very important to asses. Figure 5.10 plots this ratio at 76% – 100% overlap (it applied equally for the other overlap ranges as they have far fewer defects removed and added). From this chart, we note that out of 18 datasets, in 14 cases XTREE lead to a significant reduction in defects. For example,

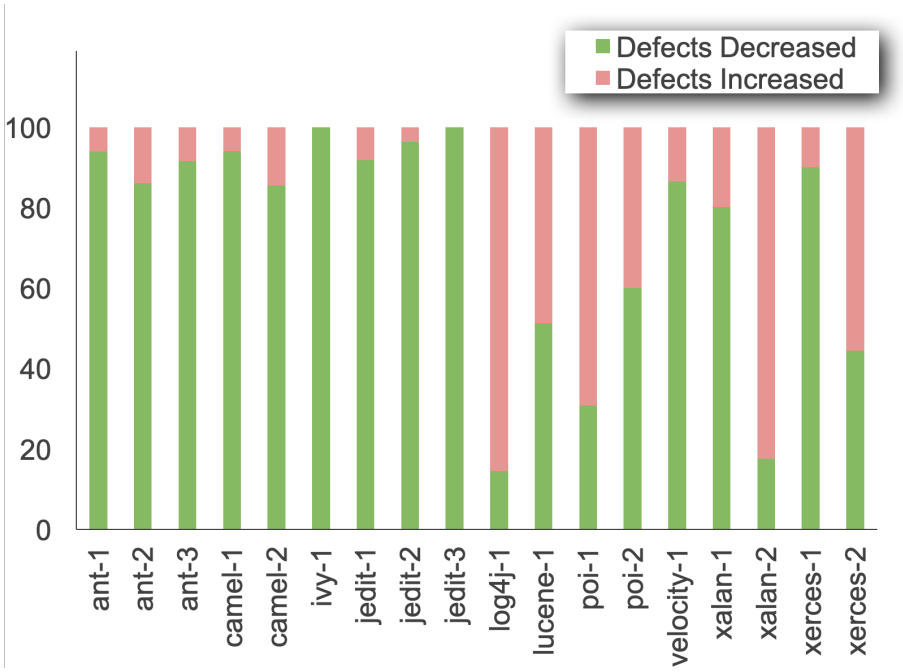


Figure 5.10 A count of total number *defects reduced* and *defects increased* as a result each planners' recommendations. The overlaps are again categorized into four ranges for every dataset (denoted by $min \leq Overlap < max$). For each of the overlap ranges, we count the total number of *defects reduced* and *defects increased* in the validation set for the classes that were defective in the test set as a result of overlap between the planner's recommendation and the developers changes that fell in the given range

in the case of Ivy and Log4j, there were no defects added at all.

However, in 4 cases, there were more defects added than there were removed. Given the idiosyncrasies of real world projects, we do not presume that developers will always take actions as suggested by a planner. This may lead to defects being increased, however, based on our results we notice that this is not a common occurrence. In summary, our response to this research question is as follows:

Result: Plans generated by XTREE are superior to other outlier statistics based planners in all 10 projects. Planning with XTREE leads to the far larger number of defects reduced as opposed to defects added in 9 out of 10 projects studied here.

	[0,25)			[25,50)			[50,75)			[75,100]		
	RAND	XTREE	BELLTREE	RAND	XTREE	BELLTREE	RAND	XTREE	BELLTREE	RAND	XTREE	BELLTREE
ant-1	13	13	12	3	33	19	0	27	10	0	62	54
ant-2	0	6	13	0	42	33	0	27	27	0	114	61
ant-3	22	18	6	1	71	42	0	47	27	0	108	124
camel-1	76	29	10	0	90	30	0	52	20	0	226	98
camel-2	36	25	30	0	109	100	0	69	68	0	439	277
ivy-1	1	4	12	0	10	42	0	5	13	0	12	25
jedit-1	13	9	11	8	35	44	0	39	50	0	136	108
jedit-2	28	24	10	1	77	34	0	36	39	0	107	115
jedit-3	18	30	28	1	67	75	0	28	35	0	70	86
log4j-1	5	1	0	0	7	14	0	3	8	0	8	50
poi-1	1	0	7	5	0	80	0	2	19	0	81	90
poi-2	78	4	0	18	135	0	0	27	2	0	87	83
velocity-1	51	2	6	0	25	15	0	39	32	0	90	48
xalan-1	22	6	2	105	43	51	13	60	66	0	409	230
xalan-2	110	0	6	0	38	49	0	102	54	0	83	298
xerces-1	23	2	11	0	11	13	0	17	24	0	305	49
xerces-2	7	0	2	0	3	11	0	6	18	0	117	165

Figure 5.11 A count of total number *defects reduced* with XTREE and BELLTREE. *Higher* values at *Larger* overlaps are better.

RQ-5.3: Are cross-project plans generated by BELLTREE as effective as within-project plans of XTREE?

In the previous two research questions, we made an assumption that there are past releases that can be used to construct the planners. However, this may not always be the case. For new project, it is quite possible that there are not any historical data to construct the planners. In such cases, SE literature proposes the use of *transfer learning*. In this work, we leverage the so-called *bellwether* effect to identify a bellwether project. Having done so, we construct a planner quite similar to XTREE with the exception that the training data comes

	[0,25)			[25,50)			[50,75)			[75,100]		
	RAND	XTREE	BELLTREE	RAND	XTREE	BELLTREE	RAND	XTREE	BELLTREE	RAND	XTREE	BELLTREE
ant-1	15	1	3	0	10	11	0	2	1	0	4	2
ant-2	63	9	1	0	33	10	0	11	2	0	20	4
ant-3	69	22	9	0	38	33	0	15	11	0	10	20
camel-1	36	10	5	0	11	25	0	6	14	0	14	31
camel-2	112	5	2	0	26	15	0	17	9	0	74	15
ivy-1	6	1	0	0	3	2	0	2	1	0	0	0
jedit-1	37	3	2	2	20	10	0	11	6	0	12	6
jedit-2	15	2	5	0	8	19	0	2	11	0	4	12
jedit-3	3	1	1	0	1	0	0	1	1	0	0	0
log4j-1	73	1	2	1	14	7	0	13	2	0	47	7
poi-1	190	1	1	6	7	1	0	5	0	0	182	6
poi-2	87	4	0	2	23	7	0	11	5	0	58	184
velocity-1	21	1	4	4	3	14	0	3	17	0	14	10
xalan-1	152	2	3	21	46	29	6	33	31	0	101	217
xalan-2	506	27	3	0	25	48	0	87	32	0	388	101
xerces-1	52	0	0	0	10	1	0	11	1	0	34	1
xerces-2	169	4	0	0	14	11	0	9	12	0	146	34

Figure 5.12 A count of total number *defects reduced* with XTREE and BELLTREE. *Lower* values at *Larger* overlaps are better.

from the bellwether project. This variant of our planner that uses the bellwether project is called the BELLTREE (see §5.5 for more details).

To answer this research question, we train XTREE on within-project data and generate plans for reducing the number of defects. We then compare this with plans derived from the bellwether data and BELLTREE. We hypothesized that since bellwethers have been demonstrated to be efficient in prediction tasks, learning from the bellwethers for a specific community of projects would produce performance scores comparable to within-project data. Our experimental methodology to answer this research question is as follows:

1. Like before, we measure the *overlap* between the planners' recommendations developers' actions.
2. Next, we tabulate the aggregate number defects reduced (Figure 5.11) and the number of defects increased (Figure 5.12) in files with overlap values ranging from 0% to 100% in bins of size 25% (for ranges of 0−25%, 26−50%, 51−75%, and 76−100%).

Similar to previous research questions, we compare XTREE with BELLTREE and a random oracle (RAND). We prefer planners that have a large number defects reduced for higher overlap ranges and planner that have lower number of defects added are are considered better.

We make the following observations from in our results:

1. *Defects Decreased*: Figure 5.12 we tabulate the number of defects *removed* in files with various overlap ranges for XTREE and BELLTREE. It is desirable to see larger defects removed with larger overlap. We note that there is no clear winner, i.e., BELLTREE performs just as well as XTREE.
2. *Defects Increased*: Figure 5.11 we tabulate the number of defects *added* in files with various overlap ranges for XTREE and BELLTREE. It is desirable to see lower defects removed with larger overlap. We again note that there is no clear winner, i.e., BELLTREE performs just as well as XTREE.
3. In all the cases the number of defects removed was significantly larger than the number of defects added.

In summary, our response to this research question is as follows:

Result: The effectiveness of BELLTREE and XTREE are similar. If within-project data is available, we recommend using XTREE. If not, BELLTREE is a viable alternative.

5.8 Discussion

When discussing these results with colleagues, we are often asked the following questions.

1. Why use automatic methods to find quality plans? Why not just use domain knowledge; e.g. human expert intuition? Recent research has documented the wide variety of conflicting opinions among software developers, even those working within the same project. According to Passos et al. [Pas11], developers often assume that the lessons they learn from a few past projects are general to all their future projects. They comment, "past experiences were taken into account without much consideration for their context". Jorgensen and Gruschke [JG09] offer a similar warning. They report that the supposed software engineering "gurus" rarely use lessons from past projects to improve their future reasoning and that such poor past advice can be detrimental to new projects [JG09]. Other studies have shown some widely-held views are now questionable given new evidence. Devanbu et al. examined responses from 564 Microsoft software developers from around the world. They comment programmer beliefs can vary with each project, but do not necessarily correspond with actual evidence in that project [Dev16]. Given the diversity of opinions seen among humans, it seems wise to explore automatic oracles for planning.

5.9 Summary

Most software analytic tools that are currently in use today are mostly prediction algorithms. These algorithms are limited to making predictions. We extend this by offering "planning": a novel technology for prescriptive software analytics. Our planner offers users a guidance on what action to take in order to improve the quality of a software project. Our preferred planning tool is BELLTREE, which performs cross-project planning with encouraging results.

With our BELLTREE planner, we show that it is possible to reduce several hundred defects in software projects.

It is also worth noting that BELLTREE is a novel extension of our prior work on (1) the bellwether effect, and (2) within-project planning with XTREE. In this work, we show that it is possible to use bellwether effect and within-project planning (with XTREE) to perform cross-project planning using BELLTREE, without the need for more complex transfer learners. Our results from Figure 5.7 show that BELLTREE is just as good as XTREE, and both XTREE/BELLTREE are much better than other planners.

Further, we can see that both BELLTREE and XTREE recommend changes to very few metric, while other unsupervised planners such as Shatnawi, Alves, and Olivera, recommend changing most of the metrics. This is not practical in many real world scenarios.

Hence our overall conclusion is to endorse the use of planners like XTREE (if local data is available) or BELLTREE (otherwise).

CHAPTER

6

FUTURE WORK

6.1 Future Work: Large Scale Transfer

While our work has generated several specific results about software projects projects, it has failed (thus far) to provide general lessons that are demonstrably useful across a large number of software projects. This, surely, is the utimate goal of software analysis. Indeed, we must attempt to do better, given the ready availability of many software projects and the scalablility of automated analytics.

A resonable explanation for the limited conclusions from automated analytics is the amount of data that is being used in software analytics. A typical research paper in soft-

ware analytics studies less than a few dozen projects. Such small samples can never be representative of something as diverse as software engineering.

Perhaps it is time to stop making limited conclusions from small samples of data set of software projects. To be able to discover general conclusions in software engineering, we propose learning from a much larger corpus than ever used before (10,000+ projects). To that end, the future work of after this thesis will we explore innovative and scalable transfer learning methods based on very fast clustering and transfer learners based on very fast stream mining algorithms that use incremental hyper-parameter optimization.

With these innovations should enable use to explore 1500+ commercial projects maintained by our industrial colleagues at Raleigh's Research Triangle Park; as well as the 10,000+ projects currently available from collaborative coding platforms such as Github. Using this data, the future work will ask the following question:

Can transfer learning between 10,000+ project generate insightful models?

In order to answer the previous question, we propose watching for new releases of the projects that are being studied. When these new releases do occur, we can apply our current models to those new releases. This should demonstrate the value (or lack thereof) of our models on new data.

6.2 Threats to Validity

6.2.1 Sampling Bias

Sampling bias threatens any empirical experiment; what matters in one case may or may not hold in another case. For example, even though we use a number of open-source datasets in this study which come from several sources, they were all supplied by individuals.

That said, this work shares this sampling bias problem with every other data mining paper. As researchers, all we can do is document our selection procedure for data and suggest that other researchers try a broader range of data in future work.

6.2.2 Learner Bias

For building the quality predictors in this study, we elected to use random forests. We chose this learner because past studies shows that, for prediction tasks, the results were superior to other more complicated algorithms [Les08] and can act as a baseline for other algorithms.

Apart from this choice, one limitation to our current study is that we have focused here on homogeneous transfer learning (where the attributes in source and target are the same). The implications for heterogeneous transfer learning (where the attributes in source an target have different names) are not yet clear. We have some initial results suggesting that a bellwether-like effect occurs when learning across the communities but those results are very preliminary. Hence, for the moment, we would conclude:

- For the homogeneous case, we recommend using bellwethers rather than similarity-based transfer learning.
- For the heterogeneous case, we recommend using dimensionality transforms.

6.2.3 Evaluation Bias

This work uses a number of measures of prediction quality, for example, to discover if bellwethers exist, we use G-Score for example (see Equation 2.2). Other quality measures often used in software engineering to quantify the effectiveness of prediction [MC07; Men07c; Fu16b]. A comprehensive analysis using these measures may be performed with our replication package.

6.2.4 Order Bias

With random forest and SMOTE, there is invariably some degree of randomness that is introduced by both the algorithms. Random Forest, as the name suggests, randomly samples the data and constructs trees which it then uses in an ensemble fashion to make predictions.

To mitigate these biases, we run the experiments 30 times (the reruns are equal to 30 in keeping with the central limit theorem). Note that the reported variations over those runs were very small. Hence, we conclude that while order bias is theoretically a problem, it is not a major problem in the particular case of this study.

BIBLIOGRAPHY

[Agg09] Aggmakarwal, K. et al. “Empirical analysis for investigating the effect of object-oriented metrics on fault proneness: a replicated case study”. *Software Process: Improvement and Practice* **14**.1 (2009).

[AM17] Agrawal, A. & Menzies, T. “"Better Data" is Better than "Better Data Miners" (Benefits of Tuning SMOTE for Defect Prediction)”. *CoRR* **abs/1705.03697** (2017). arXiv: 1705.03697.

[Agr18] 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*. ICSE-SEIP ’18. Gothenburg, Sweden: ACM, 2018, pp. 245–253.

[Alt99] Altman, E. *Constrained Markov decision processes*. Vol. 7. CRC Press, 1999.

[Alv10] Alves, T. L. et al. “Deriving metric thresholds from benchmark data”. *2010 IEEE Int. Conf. Softw. Maint.* IEEE, 2010, pp. 1–10.

[And07] Andrews, J. H. et al. “Nighthawk: A Two-Level Genetic-Random Unit Test Data Generator”. *IEEE ASE’07*. 2007.

[And10] Andrews, J. H. et al. “Genetic Algorithms for Randomized Unit Testing”. *IEEE Transactions on Software Engineering* (2010).

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

[AB11] Arcuri, A. & Briand, L. “A practical guide for using statistical tests to assess randomized algorithms in software engineering”. *ICSE’11*. 2011, pp. 1–10.

[AB06] Arisholm, E. & Briand, L. “Predicting fault prone components in a JAVA legacy system”. *2006 ACM/IEEE international symposium on Empirical software engineering* (2006), p. 17.

[Ec2] *AWS EC2 Document History*. `http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/DocumentHistory.html`.

[BM09] Baier, S. S.J. A. & McIlraith, S. A. “HTN planning with preferences”. *21st Int. Joint Conf. on Artificial Intelligence*. 2009, pp. 1790–1797.

[BD02] Bansiya, J. & Davis, C. G. “A Hierarchical Model for Object-Oriented Design Quality Assessment”. *IEEE Trans. Softw. Eng.* **28**.1 (2002), pp. 4–17.

[Bas96a] Basili, V. et al. “A validation of object-oriented design metrics as quality indicators”. *Software Engineering, IEEE Transactions* **22**.10 (1996), pp. 751–761.

[Bas96b] Basili, V. R. et al. “A validation of object-oriented design metrics as quality indicators”. *Software Engineering, IEEE Transactions on* **22**.10 (1996), pp. 751–761.

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

[Bel57] Bellman, R. “A Markovian Decision Process”. *Indiana Univ. Math. J.* **6** (4 1957), pp. 679–684.

[Ben99] Bender, R. “Quantitative Risk Assessment in Epidemiological Studies Investigating Threshold Effects”. *Biometrical Journal* **41**.3 (1999), pp. 305–319.

[Bir02] Birattari, M. et al. “A racing algorithm for configuring metaheuristics”. *Proceedings of the 4th Annual Conference on Genetic and Evolutionary Computation*. Morgan Kaufmann Publishers Inc. 2002, pp. 11–18.

[Blu13] Blue, D. et al. “Interaction-based test-suite minimization”. *Proc. the 2013 Intl. Conf. Software Engineering*. IEEE Press. 2013, pp. 182–191.

[Boe81a] Boehm, B. *Software Engineering Economics*. Prentice Hall, 1981.

[Boe00] Boehm, B. et al. *Software Cost Estimation with Cocomo II*. Prentice Hall, 2000.

[Boe81b] Boehm, B. W. et al. *Software engineering economics*. Vol. 197. Prentice-hall Englewood Cliffs (NJ), 1981.

[Bri00] Briand, L. et al. “Exploring the relationships between design measures and software quality in object-oriented systems”. *Journal of Systems and Software* **51**.3 (2000), pp. 245–273.

[Bri01] Briand, L. et al. “Replicated case studies for investigating quality factors in object-oriented designs”. *Empirical Software Engineering* **6**.1 (2001), pp. 11–58.

[BA09] Bryton, S. & Abreu, F. B. e. “Strengthening refactoring: towards software evolution with quantitative and experimental grounds”. *Software Engineering Advances, 2009. ICSEA’09. Fourth Intl. Conf.* IEEE. 2009, pp. 570–575.

[BW08] Buse, R. P. & Weimer, W. R. “A metric for software readability”. *Proceedings of the 2008 international symposium on Software testing and analysis*. ACM. 2008, pp. 121–130.

[Cam15] Campbell, A. *SonarQube: Open Source Quality Management*. 2015.

[CS00] Cartwright, M. & Shepperd, M. “An empirical investigation of an object-oriented software system”. *Software Engineering, IEEE Transactions* **26**.8 (2000), pp. 786–796.

[Cha03] Chawla, N. V. “C4. 5 and imbalanced data sets: investigating the effect of sampling method, probabilistic estimate, and decision tree structure”. *Proceedings of the ICML*. Vol. 3. 2003.

[Che18] Chen, J. et al. “" Sampling" as a Baseline Optimizer for Search-based Software Engineering”. *Transactions on Software Engineering* (2018).

[CJ10] Cheng, B. & Jensen, A. “On the Use of Genetic Programming for Automated Refactoring and the Introduction of Design Patterns”. *Proc. 12th Annual Conf. Genetic and Evolutionary Computation*. GECCO ’10. Portland, Oregon, USA: ACM, 2010, pp. 1341–1348.

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

[Coh95] Cohen, P. R. *Empirical Methods for Artificial Intelligence*. MIT Press, 1995.

[CV95] Cortes, C. & Vapnik, V. “Support-vector networks”. *Machine learning* **20**.3 (1995), pp. 273–297.

[Cui05] Cui, X et al. “Document clustering using particle swarm optimization”. *Intelligence Symposium, 2005.* (2005).

[Cze11] Czerwonka, J. et al. “CRANE: Failure Prediction, Change Analysis and Test Prioritization in Practice – Experiences from Windows”. *Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth Intl. Conference on*. 2011, pp. 357 –366.

[D’A12] D’Ambros, M. et al. “Evaluating defect prediction approaches: a benchmark and an extensive comparison”. *Empir. Softw. Eng.* **17**.4-5 (2012), pp. 531–577.

[DJ14] Deb, K & Jain, H. “An Evolutionary Many-Objective Optimization Algorithm Using Reference-Point-Based Nondominated Sorting Approach, Part I: Solving

Problems With Box Constraints". *Evolutionary Computation, IEEE Transactions on* **18**.4 (2014), pp. 577–601.

[Deb02] Deb, K. et al. "A fast and elitist multiobjective genetic algorithm: NSGA-II". *IEEE transactions on evolutionary computation* **6**.2 (2002), pp. 182–197.

[Den03] Denaro, G. et al. "An empirical evaluation of object oriented metrics in industrial setting". *The 5th CaberNet Plenary Workshop, Porto Santo, Madeira Archipelago, Portugal* (2003).

[Dev16] Devanbu, P. et al. "Belief & evidence in empirical software engineering". *Proceedings of the 38th International Conference on Software Engineering*. ACM. 2016, pp. 108–119.

[DB06] Du Bois, B. *A study of quality improvements by refactoring*. 2006.

[ET93] Efron, B. & Tibshirani, R. J. *An introduction to the bootstrap*. Mono. Stat. Appl. Probab. London: Chapman and Hall, 1993.

[Eka09] Ekanayake, J. et al. "Tracking concept drift of software projects using defect prediction quality". *Mining Software Repositories, 2009. MSR'09. 6th IEEE International Working Conference on*. IEEE. 2009, pp. 51–60.

[EA11] Elish, K. & Alshayeb, M. "A classification of refactoring methods based on software quality attributes". *Arabian Journal for Science and Engineering* **36**.7 (2011), pp. 1253–1267.

[EA12] Elish, K. & Alshayeb, M. "Using Software Quality Attributes to Classify Refactoring to Patterns." *JSW* **7**.2 (2012), pp. 408–419.

[EE08] Elish, K. O. & Elish, M. O. "Predicting defect-prone software modules using support vector machines". *JSS* **81**.5 (2008), pp. 649–660.

[Ema99] Emam, K. el et al. "A validation of object-oriented metrics". *National Research Council of Canada, NRC/ERB,* **1063** (1999).

[Ema01a] Emam, K. el et al. "The Confounding Effect of Class Size on the Validity of Object-Oriented Metrics". *Software Engineering, IEEE Transactions* **27**.7 (2001), pp. 630–650.

[Ema01b] Emam, K. el et al. "The prediction of faulty classes using object-oriented design metrics". *Journal of Systems and Software* **56**.1 (2001), pp. 63–75.

[Eng09] English, M. et al. “Fault detection and prediction in an open-source software project”. *Proceedings of the 5th International Conference on Predictor Models in Software Engineering* 1-11 (2009).

[Fag76] Fagan, M. “Design and Code Inspections to Reduce Errors in Program Development”. *IBM Systems Journal* **15**.3 (1976).

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

[FN71] Fikes, R. E. & Nilsson, N. J. “STRIPS: A new approach to the application of theorem proving to problem solving”. *Artificial intelligence* **2**.3-4 (1971), pp. 189–208.

[FN01] Fioravanti, F. & Nesi, P. “A study on fault-proneness detection of object-oriented systems”. *Software Maintenance and Reengineering, 2001. Fifth European Conference* (2001), pp. 121–130.

[Fos03] Foss, T. et al. “A Simulation Study of the Model Evaluation Criterion MMRE”. *IEEE Transactions on Software Engineering* (2003).

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

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

[Fu16b] Fu, W. et al. “Tuning for software analytics: Is it really necessary?” *Information and Software Technology* **76** (2016), pp. 135 –146.

[Gha04] Ghallab, M. et al. *Automated Planning: theory and practice*. Elsevier, 2004.

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

[Gig10] Giger, E. et al. “Predicting the Fix Time of Bugs”. *Proceedings of the 2Nd International Workshop on Recommendation Systems for Software Engineering*. RSSE ’10. Cape Town, South Africa: ACM, 2010, pp. 52–56.

[Gla00] Glasberg, D. et al. “Validating object-oriented design metrics on a commercial JAVA application”. *NRC 44146* (2000).

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

[Guo13] Guo, J. et al. “Variability-aware performance prediction: A statistical learning approach”. *International Conference on Automated Software Engineering*. IEEE. 2013.

[Guo17] Guo, J. et al. “Data-efficient performance learning for configurable systems”. *Empirical Software Engineering* (2017).

[GHL09] Guo, X. & Hernández-Lerma, O. “Continuous-time Markov decision processes”. *Continuous-Time Markov Decision Processes* (2009), pp. 9–18.

[Gyi05] Gyimothy, T. et al. “Empirical validation of object-oriented metrics on open source software for fault prediction”. *Software Engineering, IEEE Transactions* **31**.10 (2005), pp. 897–910.

[Hal12] Hall, T. et al. “A Systematic Literature Review on Fault Prediction Performance in Software Engineering”. *IEEE Transactions on Software Engineering* **38**.6 (2012), pp. 1276–1304.

[Hal77] Halstead, M. H. et al. *Elements of software science*. Vol. 7. Elsevier New York, 1977.

[Han07] Han, J. et al. “Frequent pattern mining: current status and future directions”. *Data mining and knowledge discovery* **15**.1 (2007), pp. 55–86.

[Han06] Hand, D. J. “Classifier Technology and the Illusion of Progress”. *ArXiv Mathematics e-prints* (2006). eprint: `math/0606441`.

[Har11] Harman, M et al. “Search based software engineering: Techniques, taxonomy, tutorial”. *Search* **2012** (2011), pp. 1–59.

[Har09] Harman, M. et al. “Search based software engineering: A comprehensive analysis and review of trends techniques and applications”. *Department of Computer Science, King’s College London, Tech. Rep. TR-09-03* (2009).

[Has17] Hassan, A. *Remarks made during a presentation to the UCL Crest Open Workshop*. 2017.

[He13] He, Z. et al. “Learning from open-source projects: An empirical study on defect prediction”. *Empirical Software Engineering and Measurement, 2013 ACM/IEEE International Symposium on*. IEEE. 2013, pp. 45–54.

[Hen15] Henard, C. et al. “Combining Multi-Objective Search and Constraint Solving for Configuring Large Software Product Lines”. *2015 IEEE/ACM 37th IEEE Intl. Conf. Software Engineering*. Vol. 1. 2015, pp. 517–528.

[Her11] Herodotou, H. et al. “Starfish: a self-tuning system for big data analytics.” *Conference on Innovative Data Systems Research*. 2011.

[HM15] Hihn, J. & Menzies, T. “Data Mining Methods and Cost Estimation Models: Why is it So Hard to Infuse New Ideas?” *2015 30th IEEE/ACM Intl. Conf. Automated Software Engineering Workshop (ASEW)*. 2015, pp. 5–9.

[Hol09] Holschuh, T. et al. “Predicting defects in SAP Java code: An experience report”. *Software Engineering - Companion Volume, 2009. ICSE-Companion 2009. 31st International Conference* (2009), pp. 172–181.

[Hol93] Holte, R. C. “Very Simple Classification Rules Perform Well on Most Commonly Used Datasets”. *Machine Learning* **11** (1993), p. 63.

[Ii09] Ii, P. G. et al. “Understanding the value of software engineering technologies”. *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society. 2009, pp. 52–61.

[JC16] Jamshidi, P. & Casale, G. “An Uncertainty-Aware Approach to Optimal Configuration of Stream Processing Systems”. *Symposium on the Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*. IEEE, 2016.

[Jam17a] Jamshidi, P. et al. “Transfer learning for improving model predictions in highly configurable software”. *Symposium on Software Engineering for Adaptive and Self-Managing Systems*. IEEE. 2017.

[Jam17b] Jamshidi, P. et al. “Transfer learning for performance modeling of configurable systems: An exploratory analysis”. *International Conference on Automated Software Engineering*. IEEE Press. 2017.

[Jan06] Janes, A. et al. “Identification of defect-prone classes in telecommunication software systems using design metrics”. *Information Sciences* **176**.24 (2006), pp. 3711–3734.

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

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

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

[Jin15] Jing, X. et al. “Heterogeneous Cross-Company Defect Prediction by Unified Metric Representation and CCA-Based Transfer Learning Categories and Subject Descriptors”. *Proceeding of the 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2015)* (2015), pp. 496–507.

[Jor04] Jorgensen, M. “Realism in Assessment of Effort Estimation Uncertainty: It Matters How You Ask”. *IEEE Trans. Softw. Eng.* **30**.4 (2004), pp. 209–217.

[JG09] 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.

[JM10] Jureczko, M. & Madeyski, L. “Towards identifying software project clusters with regard to defect prediction”. *Proc. 6th Int. Conf. Predict. Model. Softw. Eng. - PROMISE ’10*. New York, New York, USA: ACM Press, 2010, p. 1.

[Kae98] Kaelbling, L. P. et al. “Planning and acting in partially observable stochastic domains”. *Artificial intelligence* **101**.1 (1998), pp. 99–134.

[Kam07] Kampenes, V. B. et al. “A systematic review of effect size in software engineering experiments”. *Information & Software Technology* **49**.11-12 (2007), pp. 1073–1086.

[Kat02] Kataoka, Y. et al. “A quantitative evaluation of maintainability enhancement by refactoring”. *Software Maintenance, 2002. Proceedings. Intl. Conf.* IEEE. 2002, pp. 576–585.

[Ker05] Kerievsky, J. *Refactoring to Patterns*. Addison-Wesly Professional, 2005.

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

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

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

[Kim08] Kim, S. et al. “Classifying software changes: Clean or buggy?” *IEEE Transactions on Software Engineering* **34**.2 (2008), pp. 181–196.

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

[Kit07a] Kitchenham, B. et al. “Cross versus Within-Company Cost Estimation Studies: A Systematic Review”. *IEEE Trans. Softw. Eng.* **33**.5 (2007). Member-Kitchenham, Barbara A., pp. 316–329.

[Kit07b] Kitchenham, B. A. et al. “Cross Versus Within-Company Cost Estimation Studies: A Systematic Review”. *IEEE Trans. Softw. Eng.* **33**.5 (2007), pp. 316–329.

[KM11] Kocaguneli, E. & Menzies, T. “How to find relevant data for effort estimation?” *Empirical Software Engineering and Measurement (ESEM), 2011 International Symposium on*. IEEE. 2011, pp. 255–264.

[Koc12] Kocaguneli, E. et al. “Exploiting the essential assumptions of analogy-based effort estimation”. *IEEE Transactions on Software Engineering* **38**.2 (2012), pp. 425–438.

[Koc13] Kocaguneli, E. et al. “Distributed development considered harmful?” *Proceedings - International Conference on Software Engineering*. 2013, pp. 882–890.

[Koc15] Kocaguneli, E. et al. “Transfer learning in effort estimation”. *Empirical Software Engineering* **20**.3 (2015), pp. 813–843.

[Kra15] Krall, J. et al. “Gale: Geometric active learning for search-based software engineering”. *IEEE Transactions on Software Engineering* **41**.10 (2015), pp. 1001–1018.

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

[Kri17a] Krishna, R. “Learning effective changes for software projects”. *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press. 2017, pp. 1002–1005.

[KM18] Krishna, R. & Menzies, T. “Bellwethers: A Baseline Method For Transfer Learning”. *IEEE Transactions on Software Engineering* (2018), pp. 1–1.

[Kri16a] Krishna, R. et al. “Less is More: Minimizing Code Reorganization using XTREE”. *CoRR* **abs/1609.03614** (2016).

[Kri16b] Krishna, R. et al. “Too Much Automation? The Bellwether Effect and Its Implications for Transfer Learning”. *ASE’16*. 2016.

[Kri16c] Krishna, R. et al. “Too much automation? the bellwether effect and its implications for transfer learning”. *Proc. 31st IEEE/ACM Intl. Conf. Automated Software Engineering - ASE 2016*. New York, New York, USA: ACM Press, 2016, pp. 122–131.

[Kri17b] Krishna, R. et al. “Less is more: Minimizing code reorganization using XTREE”. *Information and Software Technology* (2017). arXiv: 1609.03614.

[Kri17c] Krishna, R. et al. “Less is more: Minimizing code reorganization using XTREE”. *Information and Software Technology* **88** (2017), pp. 53–66.

[Kri18] Krishna, R. et al. “What is the Connection Between Issues, Bugs, and Enhancements?: Lessons Learned from 800+ Software Projects”. *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*. ICSE-SEIP ’18. Gothenburg, Sweden: ACM, 2018, pp. 306–315.

[KM97] Kubat, M., Matwin, S., et al. “Addressing the curse of imbalanced training sets: one-sided selection”. *ICML*. Vol. 97. Nashville, USA. 1997, pp. 179–186.

[Lan16] Langdon, W. B. et al. “Exact Mean Absolute Error of Baseline Predictor, MARP0”. *Information and Software Technology* **73** (2016), pp. 16 –18.

[LM06] Lanza, M. & Marinescu, R. *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*. Springer Verlag, 2006, p. 207.

[Le 12] Le Goues, C. et al. “A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each”. *2012 34th Intl. Conf. Software Engineering (ICSE)*. IEEE, 2012, pp. 3–13.

[Le 15] Le Goues, C. et al. “The ManyBugs and IntroClass Benchmarks for Automated Repair of C Programs”. *IEEE Transactions on Software Engineering* **41**.12 (2015), pp. 1236–1256.

[LO02] Leech, N. L. & Onwuegbuzie, A. J. “A Call for Greater Use of Nonparametric Statistics.” *Annual Meeting of the Mid-South Educational Research Association*. ERIC, 2002.

[Lem09] Lemon, B et al. “Applications of Simulation and AI Search: Assessing the Relative Merits of Agile vs Traditional Software Development”. *IEEE ASE’09*. 2009.

[Les08] Lessmann, S. et al. “Benchmarking Classification Models for Software Defect Prediction: A Proposed Framework and Novel Findings”. *IEEE Trans. Softw. Eng.* **34**.4 (2008), pp. 485–496.

[Lew13] Lewis, C. et al. “Does Bug Prediction Support Human Developers? Findings from a Google Case Study”. *Proceedings of the 2013 International Conference on Software Engineering*. ICSE ’13. San Francisco, CA, USA: IEEE Press, 2013, pp. 372–381.

[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.

[LV14] Linares-Vásquez, M. et al. “Mining energy-greedy API usage patterns in Android apps: an empirical study”. *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM. 2014, pp. 2–11.

[LN13] Loh, P.-L. & Nowozin, S. “Faster hoeffding racing: Bernstein races via jackknife estimates”. *International Conference on Algorithmic Learning Theory*. Springer. 2013, pp. 203–217.

[Low98] Lowry, M. et al. “Towards a theory for integration of mathematical verification and empirical testing”. *Automated Software Engineering, 1998. Proceedings. 13th IEEE International Conference on*. IEEE. 1998, pp. 322–331.

[MC07] Ma, Y. & Cukic, B. “Adequate and Precise Evaluation of Quality Models in Software Engineering Studies”. *Predictor Models in Software Engineering, 2007. PROMISE’07: ICSE Workshops 2007. International Workshop on*. 2007, pp. 1–1.

[Ma12] Ma, Y. et al. “Transfer learning for cross-company software defect prediction”. *Information and Software Technology* **54**.3 (2012), pp. 248–256.

[MS07] MacDonell, S. G. & Shepperd, M. J. “Comparing Local and Global Software Effort Estimation Models – Reflections on a Systematic Review”. *Proceedings of the First International Symposium on Empirical Software Engineering and Measurement*. ESEM ’07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 401–409.

[MS05] Mair, C. & Shepperd, M. “The consistency of empirical comparisons of regression and analogy-based software project cost prediction”. *Empirical Software Engineering, 2005. 2005 International Symposium on*. 2005, 10 pp.

[Man04] Mantyla, M. et al. “Bad smells - humans as code critics”. *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on*. 2004, pp. 399–408.

[McC76] McCabe, T. J. “A complexity measure”. *IEEE Transactions on software Engineering* 4 (1976), pp. 308–320.

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

[Men02] Menzies, T. et al. “Model-based Tests of Truisms”. *Proceedings of IEEE ASE 2002*. Available from `http://menzies.us/pdf/02truisms.pdf`. 2002.

[Men05] Menzies, T. et al. “Simple Software Cost Estimation: Safe or Unsafe?” *Proceedings, PROMISE workshop, ICSE 2005*. Available from `http://menzies.us/pdf/05safewhen.pdf`. 2005.

[Men07a] Menzies, T. et al. “Data Mining Static Code Attributes to Learn Defect Predictors”. *IEEE Transactions on Software Engineering* (2007). Available from `http://menzies.us/pdf/06learnPredict.pdf`.

[Men07b] Menzies, T. et al. “Data mining static code attributes to learn defect predictors”. *IEEE transactions on software engineering* **33**.1 (2007), pp. 2–13.

[Men07c] Menzies, T. et al. “Problems with Precision: A Response to" comments on’data mining static code attributes to learn defect predictors’"”. *IEEE Transactions on Software Engineering* **33**.9 (2007), pp. 637–640.

[Men07d] Menzies, T. et al. “The business case for automated software engineering”. *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. ACM. 2007, pp. 303–312.

[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.

[Men11a] Menzies, T. et al. “Local vs. global models for effort estimation and defect prediction”. *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society. 2011, pp. 343–351.

[Men11b] 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.

[Men13] Menzies, T. et al. “Local versus Global Lessons for Defect Prediction and Effort Estimation”. *IEEE Transactions on Software Engineering* **39**.6 (2013), pp. 822–834.

[Men16] Menzies, T. et al. “Negative results for software effort estimation”. *Empirical Software Engineering* (2016), pp. 1–26.

[MP14] Metzger, A. & Pohl, K. “Software product line engineering and variability management: achievements and challenges”. *Proc. on Future of Software Engineering*. ACM. 2014, pp. 70–84.

[MA13] Mittas, N. & Angelis, L. “Ranking and Clustering Software Cost Estimation Models through a Multiple Comparisons Algorithm”. *IEEE Trans. Software Eng.* **39**.4 (2013), pp. 537–551.

[Mka14] Mkaouer, M. W. et al. “Recommendation System for Software Refactoring Using Innovization and Interactive Dynamic Optimization”. *Proc. 29th ACM/IEEE Intl. Conf. Automated Software Engineering*. ASE ’14. Vasteras, Sweden: ACM, 2014, pp. 331–336.

[Mog11] Moghadam, I. H. “Search Based Software Engineering: Third Intl. Symposium, SSBSE 2011, Szeged, Hungary, September 10-12, 2011. Proceedings”. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011. Chap. Multi-level Automated Refactoring Using Design Exploration, pp. 70–75.

[MS12] Myrtveit, I. & Stensrud, E. “Validity and Reliability of Evaluation Procedures in Comparative Studies of Effort Prediction Models”. *Empirical Software Engineering* (2012).

[Myr05] Myrtveit, I. et al. “Reliability and Validity in Comparative Studies of Software Prediction Models”. *IEEE Transactions on Software Engineering* (2005).

[NB05] Nagappan, N. & Ball, T. “Static Analysis Tools as Early Indicators of Pre-Release Defect Density”. *ICSE 2005, St. Louis*. 2005.

[Nai17a] Nair, V. et al. “Faster discovery of faster system configurations with spectral learning”. *Automated Software Engineering* (2017).

[Nai17b] Nair, V. et al. “Using bad learners to find good configurations”. *Foundations of Software Engineering* (2017).

[Nai18] Nair, V. et al. “Finding Faster Configurations using FLASH”. *Transactions on Software Engineering (Accepted)* (2018).

[Nam17] Nam, J. et al. “Heterogeneous Defect Prediction”. *IEEE Transactions on Software Engineering* **PP**.99 (2017), pp. 1–1.

[NK15] Nam, J. & Kim, S. “Heterogeneous defect prediction”. *Proc. 2015 10th Jt. Meet. Found. Softw. Eng. - ESEC/FSE 2015*. New York, New York, USA: ACM Press, 2015, pp. 508–519.

[Nam13] Nam, J. et al. “Transfer defect learning”. *Proceedings - International Conference on Software Engineering*. 2013, pp. 382–391.

[NHL18] Nayrolles, M. & Hamou-Lhadj, A. “CLEVER: Combining Code Metrics with Clone Detection for Just-In-Time Fault Prevention and Resolution in Large Industrial Projects”. *Mining Software Repositories*. 2018.

[Oh17] Oh, J. et al. “Finding near-optimal configurations in product lines by random sampling”. *Foundations of Software Engineering*. ACM. 2017.

[OA96] Ohlsson, N. & Alberg, H. “Predicting fault-prone software modules in telephone switches”. *Software Engineering, IEEE Transactions on* **22**.12 (1996), pp. 886–894.

[OC08] O’Keeffe, M. & Cinnéide, M. O. “Search-based Refactoring: An Empirical Study”. *J. Softw. Maint. Evol.* **20**.5 (2008), pp. 345–364.

[OC07] O’Keeffe, M. K. & Cinneide, M. O. “Getting the Most from Search-based Refactoring”. *Proc. 9th Annual Conf. Genetic and Evolutionary Computation*. GECCO ’07. London, England: ACM, 2007, pp. 1114–1120.

[Ola07] Olague, H. et al. “Empirical Validation of Three Software Metrics Suites to Predict Fault-Proneness of Object-Oriented Classes Developed Using Highly Iterative or Agile Software Development Processes”. *Software Engineering, IEEE Transactions* **33**.6 (2007), pp. 402–419.

[Oli14] Oliveira, P. et al. “Extracting relative thresholds for source code metrics”. *2014 Software Evolution Week - IEEE Conf. Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*. IEEE, 2014, pp. 254–263.

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

[PY10] Pan, S. J. & Yang, Q. “A survey on transfer learning”. *Transactions on knowledge and data engineering* (2010).

[Pan07] Panjer, L. D. “Predicting Eclipse Bug Lifetimes”. *Proceedings of the Fourth International Workshop on Mining Software Repositories*. MSR ’07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 29–.

[Pas11] Passos, C. et al. “Analyzing the Impact of Beliefs in Software Project Practices”. *ESEM’11*. 2011.

[Pet15] Peters, F. et al. “LACE2: Better privacy-preserving data sharing for cross project defect prediction”. *Proceedings - International Conference on Software Engineering*. Vol. 1. 2015, pp. 801–811.

[PC10] Poulding, S. & Clark, J. A. “Efficient software verification: Statistical testing using automated search”. *IEEE Transactions on Software Engineering* **36**.6 (2010), pp. 763–777.

[QC09] Quionero-Candela, J. et al. *Dataset shift in machine learning*. The MIT Press, 2009.

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

[Rah18] Rahman, A. et al. “Characterizing the Influence of Continuous Integration: Empirical Results from 250+ Open Source and Proprietary Projects”. *Proceedings of the 4th ACM SIGSOFT International Workshop on Software Analytics*. SWAN 2018. Lake Buena Vista, FL, USA: ACM, 2018, pp. 8–14.

[Rah14] Rahman, F. et al. “Comparing static bug finders and statistical prediction”. *Proceedings of the 36th International Conference on Software Engineering*. ACM. 2014, pp. 424–434.

[Rak01] Rakitin, S. *Software Verification and Validation for Practitioners and Managers, Second Edition*. Artech House, 2001.

[Rj17] Rees-jones, M. et al. “Better Predictors for Issue Lifetime”. (2017), pp. 1–8.

[Ruh10] Ruhe, G. *Product release planning: methods, tools and applications*. CRC Press, 2010.

[RG03] Ruhe, G. & Greer, D. “Quantitative studies in software release planning under risk and resource constraints”. *Empirical Software Engineering, 2003. ISESE 2003. Proceedings. 2003 Intl. Symposium on*. IEEE. 2003, pp. 262–270.

[RN95] Russell, S. & Norvig, P. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Egnlewood Cliffs, 1995.

[Ryu16] Ryu, D. et al. “Value-cognitive boosting with a support vector machine for cross-project defect prediction”. *Empir. Softw. Eng.* **21**.1 (2016), pp. 43–71.

[Sal00] Saltelli, A. et al. *Sensitivity analysis*. Vol. 1. Wiley New York, 2000.

[Sar15] Sarkar, A. et al. “Cost-efficient sampling for performance prediction of configurable systems (t)”. *International Conference on Automated Software Engineering*. IEEE. 2015.

[Sav16] Savor, T. et al. “Continuous deployment at Facebook and OANDA”. *Proceedings of the 38th International Conference on Software Engineering Companion*. ACM. 2016, pp. 21–30.

[Say13] Sayyad, A. S. et al. “Scalable product line configuration: A straw to break the camel’s back”. *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th Intl. Conf.* IEEE. 2013, pp. 465–474.

[Sha16] Sharma, A. et al. “GraphJet: real-time content recommendations at twitter”. *Proceedings of the VLDB Endowment* **9**.13 (2016), pp. 1281–1292.

[Sha10a] Shatnawi, R. “A Quantitative Investigation of the Acceptable Risk Levels of Object-Oriented Metrics in Open-Source Systems”. *IEEE Transactions on Software Engineering* **36**.2 (2010), pp. 216–225.

[Sha10b] Shatnawi, R. “A Quantitative Investigation of the Acceptable Risk Levels of Object-Oriented Metrics in Open-Source Systems”. *IEEE Transactions on Software Engineering* **36**.2 (2010), pp. 216–225.

[SL08] Shatnawi, R. & Li, W. “The effectiveness of software metrics in identifying error-prone classes in post-release software evolution process”. *Journal of Systems and Software* **81**.11 (2008), pp. 1868–1882.

[Sha10c] Shatnawi, R. “A quantitative investigation of the acceptable risk levels of object-oriented metrics in open-source systems”. *IEEE Transactions on software engineering* **36**.2 (2010), pp. 216–225.

[SM12a] Shepperd, M. & MacDonell, S. “Evaluating prediction systems in software project estimation”. *Information and Software Technology* **54**.8 (2012), pp. 820–827.

[SM12b] Shepperd, M. J. & MacDonell, S. G. “Evaluating prediction systems in software project estimation”. *Information & Software Technology* **54**.8 (2012), pp. 820–827.

[SM12c] Shepperd, M. J. & MacDonell, S. G. “Evaluating prediction systems in software project estimation”. *Information & Software Technology* **54**.8 (2012), pp. 820–827.

[Shu02] Shull, F. et al. “What We Have Learned About Fighting Defects”. *Proceedings of 8th International Software Metrics Symposium, Ottawa, Canada*. 2002, pp. 249–258.

[Sie12] Siegmund, N. et al. “Predicting performance via automated feature-interaction detection”. *International Conference on Software Engineering*. IEEE. 2012.

[Sie15] Siegmund, N. et al. “Performance-influence Models for Highly Configurable Systems”. *FSE'15*. 2015, pp. 284–294.

[Sin10] Singh, Y. et al. “Empirical validation of object-oriented metrics for predicting fault proneness models”. *Software Quality Journal* **18**.1 (2010), pp. 3–35.

[Sjo13] Sjoberg, D. et al. “Quantifying the Effect of Code Smells on Maintenance Effort”. *Software Engineering, IEEE Transactions on* **39**.8 (2013), pp. 1144–1156.

[SP06] Son, T. C. & Pontelli, E. “Planning with preferences using logic programming”. *Theory and Practice of Logic Programming* **6**.5 (2006), pp. 559–607.

[Sto08] Storkey, A. “When Training and Test Sets Are Different: Characterizing Learning Transfer”. *Dataset Shift in Machine Learning*. The MIT Press, 2008, pp. 2–28.

[SS07] Stroggylos, K. & Spinellis, D. “Refactoring–Does It Improve Software Quality?” *Software Quality, 2007. WoSQ'07: ICSE Workshops 2007. Fifth Intl. Workshop on*. IEEE. 2007, pp. 10–10.

[SK03] Subramanyam, R. & Krishnan, M. S. “Empirical analysis of CK metrics for object-oriented design complexity: implications for software defects”. *IEEE Transactions on Software Engineering* **29**.4 (2003), pp. 297–310.

[Suc03] Succi, G. “Practical assessment of the models for identification of defect-prone classes in object-oriented commercial systems using design metrics,” *Journal of Systems and Software* **65**.1 (2003), pp. 1–12.

[TG06] Tallam, S. & Gupta, N. “A concept analysis inspired greedy algorithm for test suite minimization”. *ACM SIGSOFT Software Engineering Notes* **31**.1 (2006), pp. 35–42.

[Tan18] Tang, C. et al. “Searching for high-performing software configurations with metaheuristic algorithms”. *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. ACM. 2018, pp. 354–355.

[Tan99] Tang, M. et al. “An empirical study on object-oriented metrics”. *Software Metrics Symposium* **Proceedings. Sixth International** (1999), pp. 242–249.

[Tan16] Tantithamthavorn, C. et al. “Automated parameter optimization of classification techniques for defect prediction models”. *ICSE 2016*. ACM. 2016, pp. 321–332.

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

[TV10] Thapaliyal, M. & Verma, G. “Software Defects and Object Oriented Metrics-An Empirical Analysis”. *International Journal of Computer Applications* **9/5** (2010).

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

[TM03] Thongmak, M. & Muenchaisri, P. “Predicting Faulty Classes using Design Metrics with Discriminant Analysis”. *Software Engineering Research and Practice* (2003), pp. 621–627.

[Tos09] Tosun, A. et al. “Practical Considerations of Deploying AI in Defect Prediction: A Case Study within the Turkish Telecommunication Industry”. *PROMISE'09*. 2009.

[Tos10] Tosun, A. et al. “AI-Based Software Defect Predictors: Applications and Benefits in a Case Study”. *Twenty-Second IAAI Conference on Artificial Intelligence*. 2010.

[Tru18] Trubiani, C. et al. “Performance issues? Hey DevOps, mind the uncertainty!” *IEEE Software* (2018).

[Tuf15] Tufano, M. et al. “When and Why Your Code Starts to Smell Bad”. *2015 IEEE/ACM 37th IEEE Int. Conf. Softw. Eng.* IEEE, 2015, pp. 403–414.

[Tur12] Turhan, B. “On the dataset shift problem in software engineering prediction models”. *Empirical Software Engineering* **17** (1 2012), pp. 62–74.

[Tur09] Turhan, B. et al. “On the relative value of cross-company and within-company data for defect prediction”. *Empirical Software Engineering* **14**.5 (2009), pp. 540–578.

[Tur11] Turhan, B. et al. “Empirical evaluation of mixed-project defect prediction models”. *Software Engineering and Advanced Applications (SEAA), 2011 37th EUROMICRO Conference on*. IEEE. 2011, pp. 396–403.

[Val15a] Valov, P. et al. “Empirical comparison of regression methods for variability-aware performance prediction”. *Proceedings of the 19th International Conference on Software Product Line*. ACM. 2015, pp. 186–190.

[Val15b] Valov, P. et al. “Empirical comparison of regression methods for variability-aware performance prediction”. *International Conference on Software Product Line*. ACM. 2015.

[Val17] Valov, P. et al. “Transferring performance prediction models across different hardware platforms”. *International Conference on Performance Engineering*. ACM. 2017.

[VA17] Van Aken, D. et al. “Automatic Database Management System Tuning Through Large-scale Machine Learning”. *International Conference on Management of Data*. ACM. 2017.

[VD00a] 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* (2000).

[VD00b] 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.

[Vau12] Vaux, D. L. et al. “Replicates and repeats—what is the difference and is it significant?” *EMBO reports* **13**.4 (2012), pp. 291–296.

[Wan16a] Wang, S. et al. “A practical guide to select quality indicators for assessing pareto-based search algorithms in search-based software engineering”. *ICSE'16*. IEEE. 2016, pp. 631–642.

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

[Wan16b] 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.

[Wei09] Weimer, W. et al. “Automatically finding patches using genetic programming”. *Proc. Intl. Conf. Software Engineering*. IEEE, 2009, pp. 364–374.

[Whi15] Whigham, P. A. et al. “A Baseline Model for Software Effort Estimation”. *ACM Trans. Softw. Eng. Methodol.* **24**.3 (2015), 20:1–20:11.

[WM97] Wolpert, D. H. & Macready, W. G. “No Free Lunch Theorems for Optimization”. *Trans. Evol. Comp* **1**.1 (1997), pp. 67–82.

[WJ95] Wooldridge, M. & Jennings, N. R. “Intelligent agents: Theory and practice”. *The knowledge engineering review* **10**.2 (1995), pp. 115–152.

[Wu11] Wu, R. et al. “ReLink”. *Proc. 19th ACM SIGSOFT Symp. 13th Eur. Conf. Found. Softw. Eng. - SIGSOFT/FSE '11*. New York, New York, USA: ACM Press, 2011, p. 15.

[Xia99] Xia, D.-F. et al. “A proof of the arithmetic mean-geometric mean-harmonic mean inequalities”. *RGMIA research report collection* **2**.1 (1999).

[Xu08] Xu, J. et al. “An Empirical Validation of Object-Oriented Design Metrics for Fault Prediction”. *Journal of Computer Science* (2008), pp. 571–577.

[Xu15a] Xu, T. et al. “Hey, You Have Given Me Too Many Knobs!: Understanding and Dealing with Over-designed Configuration in System Software”. *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2015. Bergamo, Italy: ACM, 2015, pp. 307–319.

[Xu15b] Xu, T. et al. “Hey, you have given me too many knobs!: understanding and dealing with over-designed configuration in system software”. *Foundations of Software Engineering*. ACM. 2015.

[YM13a] Yamashita, A. & Moonen, L. “Do developers care about code smells? An exploratory survey”. *Reverse Engineering (WCRE), 2013 20th Working Conference on*. 2013, pp. 242–251.

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

[YM13b] Yamashita, A. & Moonen, L. “Exploring the impact of inter-smell relations on software maintainability: An empirical study”. *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press. 2013, pp. 682–691.

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

[Yan11] Yang, Y. et al. “Local bias and its impacts on the performance of parametric estimation models”. *Proceedings of the 7th International Conference on Predictive Models in Software Engineering - Promise ’11*. New York, New York, USA: ACM Press, 2011, pp. 1–10.

[Yin18] Ying, R. et al. “Graph convolutional neural networks for web-scale recommender systems”. *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. ACM. 2018, pp. 974–983.

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

[Yu02] Yu, P. et al. “Predicting Fault-Proneness using OO Metrics An Industrial Case Study”. *Sixth European Conference on Software Maintenance and Reengineering* (2002), pp. 99–107.

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

[Zha15] Zhang, F. et al. “Towards building a universal defect prediction model with rank transformed predictors”. English. *Empirical Software Engineering* (2015), pp. 1–39.

[Zha13] Zhang, H. et al. “Predicting Bug-fixing Time: An Empirical Study of Commercial Software Projects”. *Proceedings of the 2013 International Conference on Software Engineering*. ICSE ’13. San Francisco, CA, USA: IEEE Press, 2013, pp. 1042–1051.

[ZL07] Zhang, Q. & Li, H. “MOEA/D: A Multiobjective Evolutionary Algorithm Based on Decomposition”. *Evolutionary Computation, IEEE Transactions on* **11**.6 (2007), pp. 712–731.

[ZL06] Zhou, Y. & Leung, H. “Empirical analysis of object-oriented design metrics for predicting high and low severity faults”. *Software Engineering, IEEE Transactions* **32**.10 (2006), pp. 771–789.

[Zim09a] Zimmermann, T. et al. “Cross-Project Defect Prediction”. *ESEC/FSE’09*. 2009.

[Zim09b] Zimmermann, T. et al. “Cross-project defect prediction: a large scale experiment on data vs. domain vs. process”. *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. ACM. 2009, pp. 91–100.

[ZK04] Zitzler, E. & Künzli, S. “Indicator-Based Selection in Multiobjective Search”. *Parallel Problem Solving from Nature - PPSN VIII*. Vol. 3242. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2004, pp. 832–842.

[Zit02] Zitzler, E. et al. “SPEA2: Improving the Strength Pareto Evolutionary Algorithm for Multiobjective Optimization”. *Evolutionary Methods for Design, Optimisation, and Control*. CIMNE, Barcelona, Spain, 2002, pp. 95–100.