Bellwethers: A Baseline Method For Transfer Learning

Rahul Krishna, *Member, IEEE,* Tim Menzies, *Member, IEEE*

Abstract—Software analytics builds quality prediction models for software projects. Experience shows that (a) the more projects studied, the more varied are the conclusions; and (b) project managers lose faith in the results of software analytics if those results keep changing. To reduce this conclusion instability, we propose the use of "bellwethers": given N projects from a community the bellwether is the project whose data yields the best predictions on all others. The bellwethers offer a way to mitigate conclusion instability because conclusions about a community are stable as long as this bellwether continues as the best oracle. Bellwethers are also simple to discover (just wrap a for-loop around standard data miners). When compared to other transfer learning methods (TCA+, transfer Naive Bayes, value cognitive boosting), using just the bellwether data to construct a simple transfer learner yields comparable predictions. Further, bellwethers appear in many SE tasks such as defect prediction, effort estimation, and bad smell detection. We hence recommend using bellwethers as a *baseline method* for transfer learning against which future work should be compared.

Index Terms—Transfer learning, Defect Prediction, Bad smells, Issue Close Time, Effort Estimation, Prediction.

1 Introduction

Researchers and industrial practitioners routinely make extensive use of software analytics for many diverse tasks such as

- Estimating how long it takes to integrate new code [1];
- Predicting where bugs are most likely [2], [3];
- Determining how long it takes to build new code [4], [5].

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 [6], [7]. For more examples of software analytics, see [8], [9].

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. [10], [11], [12] and (b) the similarity based approaches of Kocaguneli, Peters and Turhan et al. [13], [14], [15], [16]. 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. [17] learned defect predictors from 622 pairs of projects *project1, project2*. In only 4% of pairs, predictors from *project1* worked on *project2*. Also, Turhan [18] 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 [19] 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 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 paper, 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 *bellwether method* to benchmark other more complex transfer learners. In other words,

- In the *bellwether method*, we search for the exemplar bellwether project and construct a transfer learner with it. This transfer learner is then used to predict for effects in future data for that community.

This work presents a significant extension to our initial findings on bellwethers [20]:

1) **Generalizing Transfer Learners:** Much of the prior work on transfer learning, including our initial work [20] only

The authors are with the Department of Computer Science, North Carolina State University, USA. E-mail: i.m.ralk@gmail.com, tim@menzies.us
Manuscript received XXXX XX, 20XX; revised XXXX XX, 20XX.

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

explored one domain (defect prediction). Here, we study the effectiveness of transfer learning for
- Code smells detection (specifically God Class and Feature Envy);
- Effort estimation;
- Issue lifetime estimation; and
- Defect Prediction.

2) **Bellwethers as a baseline transfer learner:** Our initial work [20] compared bellwethers against two other transfer learners invented by ourselves. In this paper, we explore other state-of-the-art algorithms such as:
   - Transfer Component Analysis (referred to henceforth as TCA+) [10];
   - Transfer Naive Bayes (hereafter referred to as TNB) [21]; and
   - Value Cognitive Boosting Learner [22].

   In doing the above, we identified that the transfer learning literature lacks a simple baseline to compare and contrast the various transfer learners. To address this, we use the bellwether effect to construct a baseline transfer learner (using the bellwether method). To the best of our knowledge, this is the first report 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.
3) **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. We show that these issues can be potentially addressed using the bellwether dataset.
4) **Richer Replication Package:** We have made available a updated and a much richer replication package at https://goo.gl/jCQ1Le. The newer replication package consists of all the datasets used in this paper, in addition to mechanisms for computation of other statistical measures.

Additionally, this paper makes the following empirical, methodological, and pragmatic contributions. *Empirically*, the key contribution of this paper 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. [23] discuss the 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 all.

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 [15] and then the dimensionality transform methods of Nam, Jing et al were developed. In this paper, 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 [10], [11], [12], [13], [14], [15], [16]. While we cannot claim that such simple baselines are always better (they fail in the case of issue lifetime estimation), the experiments of this paper demonstrate that in some cases other cases (code smell detection and effort estimation) bellwethers can perform better than more complex algorithms.

The rest of this paper is structured as follows. In §2, we discuss the need for baselines and how the bellwether effect can be used for this. In §3, we present the research questions this paper attempts to answer. Following this, in §4, we present an overview of conclusion instability in software engineering. This is followed by §5, there we discuss some work on transfer learning and our proposed approach (the bellwether method) and their implications to software engineering. §6 provides an overview of all the target domains that are studied in this paper. Additionally, for each sub-domain, we discuss our choice of datasets. §7 discusses the research methodology. In §8, we answer each of the research questions that we introduced in section 3. In §9, we discuss the implications of our findings and attempt to answer some other frequently asked questions. In §10, some of the threats to validity of our findings are discussed. Finally, in §11, we conclude this paper with the following statement: bellwethers may not always be the best choice for transfer learning. That said, since bellwethers are so simple to discover and use, it is a reasonable first choice for benchmarking other approaches. To aid in that benchmarking process, all our scripts and sample problems are available on-line in Github[2]. Also, to simplify all future references to this material, the same content has been assigned a digital object identifier in a public-domain repository[3].

2 Baselining with Bellwethers

Different domains can require different approaches. According to Wolpert & Macready [24], 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

2. https://goo.gl/jCQ1Le
3. http://doi.org/10.5281/zenodo.891082

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 [25] strongly recommends comparing supposedly sophisticated systems against simpler alternatives. In the machine learning community, Holte [26] 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. [27] recently proposed baseline methods for effort estimation (for other baseline methods in effort estimation, see Mittas et al. [28]). Shepperd and Macdonnel [29] 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) [30].

In their recent article on baselines in software engineering, Whigham et al. [27] 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 paper 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 Research Questions

RQ1: 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.

RQ2: How does the bellwether dataset fare against within-project dataset?

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, RQ2 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.

RQ3: 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 [20] against 3 other standard transfer learners: (1) TCA+ [10]; (2) Transfer Naive Bayes [21]; and (3) Value Cognitive Boosting [22]. 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.

RQ4: 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 the introduction and further explored in §4) is a major issue in SE and the primary cause of such conclusion instability is the constant influx of new data [31]. 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 RQ3. 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.

RQ5: How effectively do bellwethers mitigate for conclusion instability?

Motivation: In the previous research questions, we established the prevalence of bellwethers (RQ1), we showed its efficacy in constructing a baseline transfer learner (RQ3), and we also showed empirically that we can discover bellwethers early in the project's life-cycle (RQ4). 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 §4.1: (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.

4 Conclusion Instability in SE

4.1 What is 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 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 [19]. In this paper, we define and categorize conclusion instability into two forms: (a) performance instability, 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 [65], [66], [67], [68], [69], [70], [71], [72], [73], [74], [75], [76], [77], [78], [79], [80]. 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

	[32] and [33]	[34]	[35]	[36]	Dev. Surv
Alt. Classes with Diff. Interfaces					
Combinatorial Explosion					
Comments				11	VL
Conditional Complexity [33]				14	?
Data Class		✓			
Data Clumps					
Divergent Change					
Duplicated Code		✓	✓	1	VH
Feature Envy		✓		8	
Inappropriate Intimacy			✓		L
Indecent Exposure [33]					?
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 [33]					
Parallel Inheritance Hierarchies					
Primitive Obsession					
Refused Bequest		✓	✓		
Shotgun Surgery		✓			
Solution Sprawl [33]					
Speculative Generality					L
Switch Statements					L
Temporary Field			✓		?

Fig. 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.

ref	cbo	rfc	lcom	dit	noc	wmc	#projects	size
[37]	+	+	+	-	-	+	6	95-201
[38]	+	+	+	-	-	+	12	86 classess (3-12kloc)
[39]	+	+	-				1	1700 (110kloc)
[40]	+	+	-	+	+	+	8	113
[41]	+	+	-	+	+	+	8	114
[42]	+	+	+	+	-		1	83
[43]				+	+		1	32
[44]				+	-		1	42-69
[45]	+	-	-	-	-	-	1	85
[46]	-	+		-	-	+	3	92
[47]	+	+	+	-	+	+	1	123 (34kloc)
[48]	+			+		+	1	706
[49]	+	+	+	-	+	+	1	145
[50]	+	+	+	+	-	+	1	3677
[51]	+	+	+			+	1	?
[52]	+	+	+	+	+	+	3	?
[53]	-	+	+	-	-	+	8	113
[54]		+	+	+	+		2	64
[55]		-		-	-	-	1	3344 modules (2mloc)
[56]	+	+	+	-	-	+	5	395
[57]	+	+		-	-	+	1	1412
[58]	+	+		-	-	+	2	9713
[59]	+	+	-	-	-	+	1	145
[60]				+	-		1	145
[61]	-	-	-	-	-	-	1	174
[62]	-					-	0	50
[63]	+	+	-	-	-	+	1	145
[64]		+		+	+		2	294
total +	18	20	11	11	8	17		
total -	4	3	7	14	16	4		
Total percents:		"*" denotes majority conclusion in each column						
+	* 64%	* 71%	* 39%	39%	29%	* 61%		
-	14%	11%	25%	* 50%	* 57%	14%		

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

Fig. 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.

data is called *data drift*. It is widely accepted that this *drift* is the leading cause of instability of prediction models [81], [82], [83]. Performance instability can result in large variances in the quality of predictions. Numerous researchers [80], [84] 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 engineering. The studies explored in the rest this section sample some instances of instability and its prevalence in the domains of software engineering studied here[4]. Note the vast contradictions in conclusions in each of these domains.

4.2 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. [85] 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 [86], [36], [87].

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. [88], 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" [88]. This warning is echoed by

4. 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.

Jørgensen & Gruschke [89]. 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. [89].

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 [90].

The above remarks seem to hold true for bad smells. As shown in Figure 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 [32]. The other columns show conclusions from other studies about which bad smells matter most[5]. 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 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.

4.3 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. [91] 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 conducted a large scale systematic literature review [92]. 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. 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 [93] 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 [92]. 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.

5. 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)

4.4 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 [94]. For example, consider stability tests conducted on Boehm's COCOMO software effort estimation model by Menzies et al. [94]. 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 [95] 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. [96] 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 [97] also performed a review on effort estimation models by replicating Kitchenham et al. [96]. 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 [98] 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.

5 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 summarize the standard approaches to transfer learning, then discusses how we may simplify transfer learning by using bellwethers as a baseline transfer learner.

5.1 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. [17] 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 paper 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 paper fall into 4 domains; each domain contains so called "communities" of data sets. Each dataset within a community share the same attributes (see Figure 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 [15] 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 [13], [14] 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 [99] 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 5.

An initial attempt on performing transfer learning with *Dimensionality transform* was undertaken by Ma et al. [21] 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. [10] 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.

Figure 3.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.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 §5.1. 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.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"
```

Fig. 3: The Bellwether Framework

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. [66] 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. [22] 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 [20], 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. [11] 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 [12].

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. This partly motivated our study.

5.2 Bellwether Method

In the above section, we sampled some of the work on transfer learning in software engineering. This rest of this paper asks the question "is the complexity of §5.1 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 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 ...
   - 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[6], 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.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 paper, 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. 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.).

6 Target Domains

The rest of this paper 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.

6.1 Code Smells

According to Fowler [32], 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 [100], [101], [102] 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, like PMD[7], CheckStyle[8], FindBugs[9], and SonarQube[10]. Until recently, most detection tools for code smells make use of detection rules based on the computation of a set of metrics, e.g., well-known object-oriented metrics. 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 [103].

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 [104] 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. [105] proposed a Bayesian approach to detect occurrences of the Blob antipattern on open-source programs. Khomh et al. [106] also presented BDTEX, a GQM approach to build Bayesian Belief Networks from the definitions of antipatterns. Yang et al. [107] 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.

6. 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.

7. https://github.com/pmd/pmd
8. http://checkstyle.sourceforge.net/
9. http://findbugs.sourceforge.net/
10. http://www.sonarqube.org/

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)	

Fig. 4: Datasets from 4 chosen domains.

More recently, Fontana et al. [108] 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 [109]. The Qualitas corpus is composed of 111 systems written in Java, characterized by different sizes and belonging to different application domains. Fontana et al. [108] 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 [108]. 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 4.

6.2 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

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								

Fig. 5: Static code metrics used in defects and code smells data sets.

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

Fig. 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						

Fig. 7: Metrics used in effort estimation dataset.

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 [110] 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. [111] 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. [112] 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 [113] 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 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. [113]. 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. [113], 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 [113].

6.3 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. [30] 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. [114] report that the tunings to Boehms COCOMO model have changed radically for new data collected in the period 2000 to 2009. Kocaguneli et al. [14] 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. [20] 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 paper, 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 7 for details). The data is gathered using Boehm's 2000 COCOMO model. The data was made available by Menzies et al. [115] 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 4. Here, COC81 is the original data from 1981 COCOMO book [116]. 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.

6.4 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 [117]. 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 5, 6, and 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 [118]. 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 [119]. 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 [120]. Defect predictors have some level of generality: predictors learned at NASA [120] have also been found useful elsewhere (e.g. in Turkey [121], [122]). 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* [123] 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 [124] 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. [125] 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 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 [126]. Verification and validation (V&V) textbooks [127] 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 4 summarizes all the communities of datasets used in our experiments.

For the reasons discussed in §5.1, 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 [11]. This dataset was gathered by D'Amborse et al. [128], 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. [129] who used the Understand tool [11], 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 [11][129][130][131][132].

In addition to this, we explored two other communities of datasets from the SEACRAFT repository[12]. The group of data contains defect measures from several Apache projects. It was gathered by Jureczko et al. [133]. 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 [103].

7 Methodology

7.1 Learning Methods

In our datasets, the class variable (defects, code-smells, closed issues, and effort) belong to two categories:

1) *Discrete classes:* The classes in the case of defect prediction, detection of code-smells, and close time of issues have *two* discrete classes. We therefore use learners as *binary classifiers*.
2) *Continuous classes:* The class variable in the case of effort estimation takes on continuous values. Here we use learners as *regression* algorithms.

There are many binary classifiers to predict defects (smells or issue lifetime). A comprehensive study on defect prediction was conducted by Lessmann et al. [65]. They endorsed the use of Random Forests [134] for defect prediction over several other methods. This was also true in detecting code smells [108]. When a specific transfer learner did not endorse the use of any classification/regression scheme, we used Random Forests (Note: If an explicit reference was made regarding using a specific prediction algorithm by the authors of other transfer learners used in this paper, we use those predictors instead of random forest. e.g., VCB endorses the use of SVMs).

Random Forests is an ensemble learning method that builds several decision trees on randomly chosen subsets of data. The final reported prediction is the mode of predictions by the trees.

When the class variable if discrete (as in binary classification), it is known that the fraction of "positive" class samples in the training data affects the performance of predictors. Figure 4 shows that in most datasets, the percentage of "positive samples" (i.e., samples that are defective, smelly, or closed) vary between 10% to 40% (except in a few, projects like log4j for instance where it is 58%). Handling this class imbalance has been shown to improve the quality of prediction.

Pelayo and Dick [135] report that the defect prediction is improved by SMOTE [136]. SMOTE works by under-sampling majority-class examples and over-sampling minority class examples to balance the training data prior to applying prediction models.

After an extensive experimentation, in this study, we randomly sub-sampled examples until the training data had only positive and negative classes in a ratio of 1:2.

Important methodological notes:

1) sub-sampling was only applied to *training* data (so the test data remains unchanged).
2) Authors of several transfer learners studied here recommend using different predictors. When replicating their studies, we adhere to their recommendations.
3) SMOTE is only applicable for classification problems (defect prediction, code smell detection, and issue lifetime prediction). When performing regression for estimation of effort, we don't apply SMOTE.

7.2 Evaluation Strategy

7.2.1 Evaluation for Continuous Classes

For the effort estimation data in Figure 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 the same learning methods as in §7.1 used 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 [137], [138] Standard Accuracy is computed as below:

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

Where, MAR is the mean of the absolute error for the predictor of interest. E.g. for software project estimation, the average of the

11. http://www.scitools.com/products/
12. https://zenodo.org/communities/seacraft/

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[13] with this and other metrics. Interested readers are encouraged to use these.

7.2.2 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 [139], [140], [120], [141]. 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 paper shown in Figure 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. [3] 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 [142], [143], [144]. 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 [65]. 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 [145] 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 [3], [144] 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 [146]. Therefore, we propose the use of G-Score, measured as follows:

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

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, *larger* G-scores are better.

7.3 Statistics

To overcome the inherent randomness introduced by Random Forests and SMOTE, 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 [147]. 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. (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 [28]. 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 values found in $ls = 4$ different methods. Then, we

13. https://goo.gl/jCQ1Le

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 [148, p220-223]. 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 [149], [150], [151], [152], [153], [154]. It was first proposed by Vargha and Delany [155]. 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.

7.4 Experimental Setup

- **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 according to the learning method described in §7.1.
  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 §7.2.
  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 §7.3 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.
- **Discovering the best transfer learner:**
  1) For each community in every sub-domain, we pick a project P_i as in §7.4.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 §7.2 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 §7.3 to rank each transfer learner.

7.5 Understanding These Results

In presenting our results for experiments in §7.4, we adopted a convention that includes tabulated results. The following remarks need to be made regarding our tables:

- In Figure 8, we list the results of performing the experiment in §7.4. 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(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.
- In Figures 10, 11, 12, and 13, we list the results of performing the experiment in §7.4 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.

8 Results

RQ1: 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 8 documents our findings. We use the setup described in §7.4 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 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 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

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

Fig. 8: Discovering Bellwether datasets with a holdout data. We use the experimental setup mentioned in §7.4 to discover these bellwethers.

	Bellwether	Local		
	(Lucene)	Train	Test	
	(G-score)			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

Fig. 9: Bellwether dataset (Lucene) vs. Local Data. Performance scores are G-scores so *higher* values are *better*. Cells highlighted in gray 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.

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 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:

Result 1
Bellwethers are common in several domains of software engineering studied here. ie., in defect prediction, effort estimation, and code-smell detection.

	Source	**Baseline**	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	**Baseline**	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

Fig. 10: 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.

	Source	Baseline	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	Baseline	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

Fig. 11: 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.

RQ2: How does the bellwether dataset fare against within-project dataset?

Having established in RQ1 that bellwethers are prevalent in the sub-domains studied here. In Figure 9, 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 9, 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 *Log*4*j*), 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 2
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.

RQ3: How well do transfer learners perform across different domains?

Figures 10, 11, 12, 13 show the results of transferring data between different projects in a community for code smell detection, issue lifetime estimation, defect prediction, and effort estimation.

	Source	Baseline	**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

Fig. 12: 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.

	Source	Baseline	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

Fig. 13: 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.

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 paper do not satisfy these constraints. We could not use VCB in code smell detection because our datasets had small sample size (see Figure 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 10 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 11, 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 12, 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 [10].
4) Effort datasets: In the case of effort estimation, our results are tabulated in Figure 13. 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:

	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

Fig. 14: 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.

Result 3
There is no universal best transfer learner that works across multiple domains. Simpler baseline methods like bellwethers show comparable performances in several domains.

RQ4: 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 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 14 documents the results of this experiment. As previously mentioned, we used the defect datasets from the Apache community in Figure 4. In RQ1, 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 4
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.

RQ5: How effectively do bellwethers mitigate for conclusion instability?

In §4.1, 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 10, 11, 12, and 13 reveal that the bellwether data set can be discovered in three out of the four domains we have studied here. Additionally, the performance of an appropriate transfer learner (as identified in RQ3) 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[14] is shown in Figure 15. 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 are 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) and the bellwether data remains unchanged, so will the conclusions derived from it.

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

Result 5
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.

9 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 paper.
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

14. 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.

	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	CountLineCodeExe	CountLine	CountLineCode	RatioCommentToCode	AvgEssential
	Safe	CountStmt	SumCyclomaticStrict	CountLineCode	CountStmtDecl	CountLineCodeExe
	Zxing	CountLineCodeDecl	CountLineCode	AvgLine	CountLine	CountStmtDecl

Fig. 15: 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.

from other data [13], [14], [156], [15], [16], [10], [11], [12]. 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.

10 Threats to Validity

10.1 Sampling Bias

Sampling bias threatens any classification experiment; what matters in one case may or may not hold in another case. For example, even though we use 100+ open-source datasets in this study which come from several sources, they were all supplied by individuals.

That said, this paper shares this sampling bias problem with every other data mining paper. As researchers, all we can do is document our selection procedure for data (as done in §3) and suggest that other researchers try a broader range of data in future work.

10.2 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 [65]. We note that recent studies showed that different classifiers are highly complementary, despite obtaining similar performances. Thus, the usage of Random Forests is not bulletproof; but it can certainly act as a baseline for other algorithms. Exploration of these learners is part of our future work. 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.

10.3 Evaluation Bias

This paper uses one measure of prediction quality, G (see Equation 2). Other quality measures often used in software engineering to quantify the effectiveness of prediction [145] [3] [139] (discussed in §7.2). A comprehensive analysis using these measures may be performed with our replication package.

10.4 Random 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 parameter bias is theoretically a threat, as researchers we have used the default parameters in all situations. As researchers, all we can do is document our selection procedure for data (as done in §3) and suggest that other researchers try a broader range of data in future work.

10.5 Parameter Bias

With all the transfer learners and predictors discussed here, there are a number of internal parameters that have been set by default. The result of changing these parameters may (or may not) have a significant impact on the outcomes of this study. However, it must be noted that the possible number of combinations of these parameters is combinatorial in nature. There do however exist a growing number of literature on parameter optimization in SE. However, a encompassing this beyond the scope of this current paper.

Hence, we conclude that although parameter bias is a possible threat, as researchers we have used the default parameters in all situations sake of consistency. We recommend that other researchers attempt to toggle these parameters with the use of tuning algorithms in to validate (or possibly refute) our findings.

11 Conclusion

In this paper, 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. 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 §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.

Acknowledgements

The work is partially funded by NSF awards #1506586 and #1302169.

References

[1] J. Czerwonka, R. Das, N. Nagappan, A. Tarvo, and A. Teterev, "Crane: Failure prediction, change analysis and test prioritization in practice – experiences from windows," in *Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on*, march 2011, pp. 357 –366.

[2] T. J. Ostrand, E. J. Weyuker, and R. M. Bell, "Where the bugs are," in *ISSTA '04: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*. New York, NY, USA: ACM, 2004, pp. 86–96.

[3] T. Menzies, A. Dekhtyar, J. Distefano, and J. Greenwald, "Problems with Precision: A Response to "Comments on 'Data Mining Static Code Attributes to Learn Defect Predictors'"," *IEEE Transactions on Software Engineering*, vol. 33, no. 9, pp. 637–640, sep 2007. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4288197

[4] B. Turhan, A. Tosun, and A. Bener, "Empirical evaluation of mixed-project defect prediction models," in *Software Engineering and Advanced Applications (SEAA), 2011 37th EUROMICRO Conference on*. IEEE, 2011, pp. 396–403.

[5] E. Kocaguneli, T. Menzies, A. Bener, and J. Keung, "Exploiting the essential assumptions of analogy-based effort estimation," *IEEE Transactions on Software Engineering*, vol. 28, pp. 425–438, 2012, available from http://menzies.us/pdf/11teak.pdf.

[6] A. Begel and T. Zimmermann, "Analyze this! 145 questions for data scientists in software engineering," in *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. ACM, June 2014. [Online]. Available: http://research.microsoft.com/apps/pubs/default.aspx?id=208800

[7] C. Theisen, K. Herzig, P. Morrison, B. Murphy, and L. Williams, "Approximating attack surfaces with stack traces," in *ICSE'15*, 2015.

[8] T. Zimmermann and T. Menzies, "Software analytics: So what?" *IEEE Software*, vol. 30, no. 4, pp. 0031–37, 2013.

[9] C. Bird, T. Menzies, and T. Zimmermann, *The Art and Science of Analyzing Software Data*. Elsevier, 2015.

[10] J. Nam, S. J. Pan, and S. Kim, "Transfer defect learning," in *Proceedings - International Conference on Software Engineering*, 2013, pp. 382–391.

[11] J. Nam and S. Kim, "Heterogeneous defect prediction," in *Proc. 2015 10th Jt. Meet. Found. Softw. Eng. - ESEC/FSE 2015*. New York, New York, USA: ACM Press, 2015, pp. 508–519. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2786805.2786814

[12] X. Jing, F. Wu, X. Dong, F. Qi, and B. Xu, "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)*, pp. 496–507, 2015.

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

[14] E. Kocaguneli, T. Menzies, and E. Mendes, "Transfer learning in effort estimation," *Empirical Software Engineering*, vol. 20, no. 3, pp. 813–843, jun 2015. [Online]. Available: http://link.springer.com/10.1007/s10664-014-9300-5

[15] B. Turhan, T. Menzies, A. B. Bener, and J. Di Stefano, "On the relative value of cross-company and within-company data for defect prediction," *Empirical Software Engineering*, vol. 14, no. 5, pp. 540–578, 2009.

[16] F. Peters, T. Menzies, and L. Layman, "LACE2: Better privacy-preserving data sharing for cross project defect prediction," in *Proceedings - International Conference on Software Engineering*, vol. 1, 2015, pp. 801–811.

[17] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. Murphy, "Cross-project defect prediction: a large scale experiment on data vs. domain vs. process," in *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.

[18] T. Menzies, A. Butcher, A. Marcus, T. Zimmermann, and D. Cok, "Local vs. global models for effort estimation and defect prediction," in *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. IEEE, nov 2011, pp. 343–351. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6100072

[19] A. Hassan, "Remarks made during a presentation to the ucl crest open workshop," March 2017.

[20] R. Krishna, T. Menzies, and W. Fu, "Too Much Automation? The Bellwether Effect and Its Implications for Transfer Learning," in *ASE'16*, 2016.

[21] Y. Ma, G. Luo, X. Zeng, and A. Chen, "Transfer learning for cross-company software defect prediction," *Information and Software Technology*, vol. 54, no. 3, pp. 248–256, 2012.

[22] D. Ryu, O. Choi, and J. Baik, "Value-cognitive boosting with a support vector machine for cross-project defect prediction," *Empir. Softw. Eng.*, vol. 21, no. 1, pp. 43–71, feb 2016. [Online]. Available: http://link.springer.com/10.1007/s10664-014-9346-4

[23] T. Xu, L. Jin, X. Fan, Y. Zhou, S. Pasupathy, and R. Talwadker, "Hey, you have given me too many knobs!: Understanding and dealing with over-designed configuration in system software," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: ACM, 2015, pp. 307–319. [Online]. Available: http://doi.acm.org/10.1145/2786805.2786852

[24] D. H. Wolpert and W. G. Macready, "No free lunch theorems for optimization," *Trans. Evol. Comp*, vol. 1, no. 1, pp. 67–82, Apr. 1997. [Online]. Available: http://dx.doi.org/10.1109/4235.585893

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

[26] R. C. Holte, "Very Simple Classification Rules Perform Well on Most Commonly Used Datasets," *Machine Learning*, vol. 11, p. 63, 1993.

[27] P. A. Whigham, C. A. Owen, and S. G. Macdonell, "A baseline model for software effort estimation," *ACM Trans. Softw. Eng. Methodol.*, vol. 24, no. 3, pp. 20:1–20:11, May 2015. [Online]. Available: http://doi.acm.org/10.1145/2738037

[28] N. Mittas and L. Angelis, "Ranking and clustering software cost estimation models through a multiple comparisons algorithm," *IEEE Trans. Software Eng.*, vol. 39, no. 4, pp. 537–551, 2013.

[29] M. J. Shepperd and S. G. MacDonell, "Evaluating prediction systems in software project estimation," *Information & Software Technology*, vol. 54, no. 8, pp. 820–827, 2012.

[30] B. A. Kitchenham, E. Mendes, and G. H. Travassos, "Cross versus within-company cost estimation studies: A systematic review," *IEEE Trans. Softw. Eng.*, vol. 33, no. 5, pp. 316–329, May 2007. [Online]. Available: http://dx.doi.org/10.1109/TSE.2007.1001

[31] J. Ekanayake, J. Tappolet, H. C. Gall, and A. Bernstein, "Tracking concept drift of software projects using defect prediction quality," in *Mining Software Repositories, 2009. MSR'09. 6th IEEE International Working Conference on*. IEEE, 2009, pp. 51–60.

[32] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley Longman, 1999.

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

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

[35] A. Campbell, "SonarQube: Open source quality management," 2015, website: tiny.cc/2q4z9x.

[36] A. Yamashita and L. Moonen, "Do developers care about code smells? an exploratory survey," in *Reverse Engineering (WCRE), 2013 20th Working Conference on*, Oct 2013, pp. 242–251.

[37] H. Olague, L. Etzkorn, S. Gholston, and S. Quattlebaum, "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*, vol. 33, no. 6, pp. 402–419, 2007.

[38] K. Aggmakarwal, Y. Singh, A. Kaur, and R. Malhotra, "Empirical analysis for investigating the effect of object-oriented metrics on fault proneness: a replicated case study," *Software Process: Improvement and Practice*, vol. 14, no. 1, January 2009.

[39] E. Arisholm and L. Briand, "Predicting fault prone components in a JAVA legacy system," *2006 ACM/IEEE international symposium on Empirical software engineering*, p. 17, 2006.

[40] V. Basili, L. Briand, and W. Melo, "A validation of object-oriented design metrics as quality indicators," *Software Engineering, IEEE Transactions*, vol. 22, no. 10, pp. 751–761, 1996.

[41] L. Briand, J. Wust, J. Daly, and D. V. Porter, "Exploring the relationships between design measures and software quality in object-oriented systems," *Journal of Systems and Software*, vol. 51, no. 3, pp. 245–273, 2000.

[42] L. Briand, J. Wust, and H. Lounis, "Replicated case studies for investigating quality factors in object-oriented designs," *Empirical Software Engineering*, vol. 6, no. 1, pp. 11–58, 2001.

[43] M. Cartwright and M. Shepperd, "An empirical investigation of an object-oriented software system," *Software Engineering, IEEE Transactions*, vol. 26, no. 8, pp. 786–796, 2000.

[44] K. el Emam, W. Melo, and J. Machado, "The prediction of faulty classes using object-oriented design metrics," *Journal of Systems and Software*, vol. 56, no. 1, pp. 63–75, 2001.

[45] K. el Emam, S. Benlarbi, N. Goel, and S. Rai, "A validation of object-oriented metrics," *National Research Council of Canada, NRC/ERB,*, vol. 1063, 1999.

[46] M. Tang, M. Kao, and M. Chen, "An empirical study on object-oriented metrics," *Software Metrics Symposium*, vol. Proceedings. Sixth International, pp. 242–249, 1999.

[47] P. Yu, T. Systa, and H. Muller, "Predicting fault-proneness using oo metrics an industrial case study," *Sixth European Conference on Software Maintenance and Reengineering*, pp. 99–107, 2002.

[48] R. Subramanyam and M. S. Krishnan, "Empirical analysis of ck metrics for object-oriented design complexity: implications for software defects," *IEEE Transactions on Software Engineering*, vol. 29, no. 4, pp. 297–310, April 2003.

[49] Y. Zhou and H. Leung, "Empirical analysis of object-oriented design metrics for predicting high and low severity faults," *Software Engineering, IEEE Transactions*, vol. 32, no. 10, pp. 771–789, 2006.

[50] T. Gyimothy, R. Ferenc, and I. Siket, "Empirical validation of object-oriented metrics on open source software for fault prediction," *Software Engineering, IEEE Transactions*, vol. 31, no. 10, pp. 897–910, 2005.

[51] T. Holschuh, M. Pauser, K. Herzig, T. Zimmermann, R. Premraj, and A. Zeller, "Predicting defects in SAP Java code: An experience report," *Software Engineering - Companion Volume, 2009. ICSE-Companion 2009. 31st International Conference*, pp. 172–181, 2009.

[52] R. Shatnawi and W. Li, "The effectiveness of software metrics in identifying error-prone classes in post-release software evolution process," *Journal of Systems and Software*, vol. 81, no. 11, pp. 1868–1882, 2008.

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

[54] M. Thongmak and P. Muenchaisri, "Predicting faulty classes using design metrics with discriminant analysis," *Software Engineering Research and Practice*, pp. 621–627, 2003.

[55] G. Denaro, L. Lavazza, and M. Pezze, "An empirical evaluation of object oriented metrics in industrial setting," *The 5th CaberNet Plenary Workshop, Porto Santo, Madeira Archipelago, Portugal*, 2003.

[56] A. Janes, M. Scotto, W. Pedrycz, B. Russo, M. Stefanovic, and G. Succi, "Identification of defect-prone classes in telecommunication software systems using design metrics," *Information Sciences*, vol. 176, no. 24, pp. 3711–3734, 2006.

[57] M. English, C. Exton, I. Rigon, and B. Cleary, "Fault detection and prediction in an open-source software project," *Proceedings of the 5th International Conference on Predictor Models in Software Engineering*, no. 1-11, 2009.

[58] R. Shatnawi, "A quantitative investigation of the acceptable risk levels of object-oriented metrics in open-source systems," *IEEE Transactions on Software Engineering*, vol. 36, no. 2, pp. 216–225, 2010.

[59] Y. Singh, A. Kaur, and R. Malhotra, "Empirical validation of object-oriented metrics for predicting fault proneness models," *Software Quality Journal*, vol. 18, no. 1, pp. 3–35, 2010.

[60] D. Glasberg, K. el Emam, W. Memo, and N. Madhavji, "Validating object-oriented design metrics on a commercial JAVA application," *NRC 44146*, 2000.

[61] K. el Emam, S. Benlarbi, N. Goel, and S. Rai, "The confounding effect of class size on the validity of object-oriented metrics," *Software Engineering, IEEE Transactions*, vol. 27, no. 7, pp. 630–650, 2001.

[62] M. Thapaliyal and G. Verma, "Software defects and object oriented metrics-an empirical analysis," *International Journal of Computer Applications*, vol. 9/5, 2010.

[63] J. Xu, D. Ho, and L. Capretz, "An empirical validation of object-oriented design metrics for fault prediction," *Journal of Computer Science*, pp. 571–577, July 2008.

[64] G. Succi, "Practical assessment of the models for identification of defect-prone classes in object-oriented commercial systems using design metrics,," *Journal of Systems and Software*, vol. 65, no. 1, pp. 1–12, January 2003.

[65] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch, "Benchmarking Classification Models for Software Defect Prediction: A Proposed Framework and Novel Findings," *IEEE Trans. Softw. Eng.*, vol. 34, no. 4, pp. 485–496, jul 2008. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4527256

[66] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell, "A systematic literature review on fault prediction performance in software engineering," *IEEE Transactions on Software Engineering*, vol. 38, no. 6, pp. 1276–1304, Nov 2012.

[67] K. O. Elish and M. O. Elish, "Predicting defect-prone software modules using support vector machines," *JSS*, vol. 81, no. 5, pp. 649–660, 2008.

[68] T. Menzies, Z. Milton, B. Turhan, B. Cukic, Y. Jiang, and A. Bener, "Defect prediction from static code features: current results, limitations, new approaches," *Automated Software Engineering*, vol. 17, no. 4, pp. 375–407, 2010.

[69] I. Gondra, "Applying machine learning to software fault-proneness prediction," *Journal of Systems and Software*, vol. 81, no. 2, pp. 186–195, 2008.

[70] D. Radjenović, M. Heričko, R. Torkar, and A. Živkovič, "Software fault prediction metrics: A systematic literature review," *Information and Software Technology*, vol. 55, no. 8, pp. 1397–1418, 2013.

[71] Y. Jiang, B. Cukic, and Y. Ma, "Techniques for evaluating fault prediction models," *Empirical Software Engineering*, vol. 13, no. 5, pp. 561–595, 2008.

[72] S. Wang and X. Yao, "Using class imbalance learning for software defect prediction," *IEEE Transactions on Reliability*, vol. 62, no. 2, pp. 434–443, 2013.

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

[74] M. Li, H. Zhang, R. Wu, and Z.-H. Zhou, "Sample-based software defect prediction with active and semi-supervised learning," *Automated Software Engineering*, vol. 19, no. 2, pp. 201–230, 2012.

[75] T. M. Khoshgoftaar, K. Gao, and N. Seliya, "Attribute selection and imbalanced data: Problems in software defect prediction," in *Tools with Artificial Intelligence (ICTAI), 2010 22nd IEEE International Conference on*, vol. 1. IEEE, 2010, pp. 137–144.

[76] Y. Jiang, J. Lin, B. Cukic, and T. Menzies, "Variance analysis in software fault prediction models," in *Software Reliability Engineering, 2009. ISSRE'09. 20th International Symposium on*. IEEE, 2009, pp. 99–108.

[77] B. Ghotra, S. McIntosh, and A. E. Hassan, "Revisiting the impact of classification techniques on the performance of defect prediction models," in *37th ICSE-Volume 1*. IEEE Press, 2015, pp. 789–800.

[78] Y. Jiang, B. Cukic, and T. Menzies, "Can data transformation help in the detection of fault-prone modules?" in *Proceedings of the 2008 workshop on Defects in large software systems*. ACM, 2008, pp. 16–20.

[79] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto, "Automated parameter optimization of classification techniques for defect prediction models," in *ICSE 2016*. ACM, 2016, pp. 321–332.

[80] W. Fu, T. Menzies, and X. Shen, "Tuning for software analytics: Is it really necessary?" *IST*, vol. 76, pp. 135–146, 2016.

[81] J. Quionero-Candela, M. Sugiyama, A. Schwaighofer, and N. D. Lawrence, *Dataset shift in machine learning*. The MIT Press, 2009.

[82] D. J. Hand, "Classifier Technology and the Illusion of Progress," *ArXiv Mathematics e-prints*, Jun. 2006.

[83] A. Storkey, "When training and test sets are different: Characterizing learning transfer," in *Dataset Shift in Machine Learning*. The MIT Press, dec 2008, pp. 2–28. [Online]. Available: https://doi.org/10.7551/mitpress/9780262170055.003.0001

[84] A. Agrawal and T. Menzies, ""better data" is better than "better data miners" (benefits of tuning SMOTE for defect prediction)," *CoRR*, vol. abs/1705.03697, 2017. [Online]. Available: http://arxiv.org/abs/1705.03697

[85] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, and D. Poshyvanyk, "When and Why Your Code Starts to Smell Bad," in *2015 IEEE/ACM 37th IEEE Int. Conf. Softw. Eng.* IEEE, May 2015, pp. 403–414. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7194592

[86] M. Mantyla, J. Vanhanen, and C. Lassenius, "Bad smells - humans as code critics," in *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on*, Sept 2004, pp. 399–408.

[87] D. Sjoberg, A. Yamashita, B. Anda, A. Mockus, and T. Dyba, "Quantifying the effect of code smells on maintenance effort," *Software Engineering, IEEE Transactions on*, vol. 39, no. 8, pp. 1144–1156, Aug 2013.

[88] C. Passos, A. P. Braun, D. S. Cruzes, and M. Mendonca, "Analyzing the impact of beliefs in software project practices," in *ESEM'11*, 2011.

[89] M. Jørgensen and T. M. Gruschke, "The impact of lessons-learned sessions on effort estimation and uncertainty assessments," *Software Engineering, IEEE Transactions on*, vol. 35, no. 3, pp. 368 –383, May-June 2009.

[90] P. Devanbu, T. Zimmermann, and C. Bird, "Belief & evidence in empirical software engineering," in *Proceedings of the 38th International Conference on Software Engineering*. ACM, 2016, pp. 108–119.

[91] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. Murphy, "Cross-project defect prediction," in *ESEC/FSE'09*, August 2009.

[92] T. Menzies, A. Butcher, A. Marcus, T. Zimmermann, and D. Cok, "Local vs. global models for effort estimation and defect prediction," in *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 2011, pp. 343–351.

[93] B. Turhan, "On the dataset shift problem in software engineering prediction models," *Empirical Software Engineering*, vol. 17, pp. 62–74, 2012.

[94] T. Menzies, Z. Chen, D. Port, and J. Hihn, "Simple software cost estimation: Safe or unsafe?" in *Proceedings, PROMISE workshop, ICSE 2005*, 2005, available from http://menzies.us/pdf/05safewhen.pdf.

[95] M. Jorgensen, "Realism in assessment of effort estimation uncertainty: It matters how you ask," *IEEE Trans. Softw. Eng.*, vol. 30, no. 4, pp. 209–217, 2004.

[96] B. Kitchenham, E. Mendes, and G. H. Travassos, "Cross versus within-company cost estimation studies: A systematic review," *IEEE Trans. Softw. Eng.*, vol. 33, no. 5, pp. 316–329, 2007, member-Kitchenham, Barbara A.

[97] S. G. MacDonell and M. J. Shepperd, "Comparing local and global software effort estimation models – reflections on a systematic review," in *Proceedings of the First International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 401–409.

[98] C. Mair and M. Shepperd, "The consistency of empirical comparisons of regression and analogy-based software project cost prediction," in *Empirical Software Engineering, 2005. 2005 International Symposium on*, nov. 2005, p. 10 pp.

[99] F. Zhang, A. Mockus, I. Keivanloo, and Y. Zou, "Towards building a universal defect prediction model with rank transformed predictors," *Empirical Software Engineering*, pp. 1–39, 2015. [Online]. Available: http://dx.doi.org/10.1007/s10664-015-9396-2

[100] A. Yamashita and S. Counsell, "Code smells as system-level indicators of maintainability: An empirical study," *Journal of Systems and Software*, vol. 86, no. 10, pp. 2639–2653, 2013.

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

[102] N. Zazworka, M. A. Shaw, F. Shull, and C. Seaman, "Investigating the impact of design debt on software quality," in *Proceedings of the 2nd Workshop on Managing Technical Debt*. ACM, 2011, pp. 17–23.

[103] R. Krishna, T. Menzies, and L. Layman, "Less is More: Minimizing Code Reorganization using XTREE," *CoRR*, vol. abs/1609.03614, 2016. [Online]. Available: http://arxiv.org/abs/1609.03614

[104] J. Kreimer, "Adaptive Detection of Design Flaws," *Electronic Notes in Theoretical Computer Science*, vol. 141, no. 4, pp. 117–136, 2005. [Online]. Available: //www.sciencedirect.com/science/article/pii/S1571066105051844

[105] F. Khomh, S. Vaucher, Y. G. Guhneuc, and H. Sahraoui, "A bayesian approach for the detection of code and design smells," in *2009 Ninth International Conference on Quality Software*, Aug 2009, pp. 305–314.

[106] F. Khomh, S. Vaucher, Y.-G. Guhneuc, and H. Sahraoui, "Bdtex: A gqm-based bayesian approach for the detection of antipatterns," *Journal of Systems and Software*, vol. 84, no. 4, pp. 559 – 572, 2011, the Ninth International Conference on Quality Software. [Online]. Available: //www.sciencedirect.com/science/article/pii/S0164121210003225

[107] J. Yang, K. Hotta, Y. Higo, H. Igaki, and S. Kusumoto, "Filtering clones for individual user based on machine learning analysis," in *2012 6th International Workshop on Software Clones (IWSC)*, June 2012, pp. 76–77.

[108] F. Arcelli Fontana, M. V. Mäntylä, M. Zanoni, and A. Marino, "Comparing and experimenting machine learning techniques for code smell detection," *Empir. Softw. Eng.*, vol. 21, no. 3, pp. 1143–1191, jun 2016. [Online]. Available: http://dx.doi.org/10.1007/s10664-015-9378-4http://link.springer.com/10.1007/s10664-015-9378-4

[109] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble, "Qualitas corpus: A curated collection of java code for empirical studies," in *2010 Asia Pacific Software Engineering Conference (APSEC2010)*, Dec. 2010, pp. 336–345.

[110] L. D. Panjer, "Predicting eclipse bug lifetimes," in *Proceedings of the Fourth International Workshop on Mining Software Repositories*, ser. MSR '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 29–. [Online]. Available: http://dx.doi.org/10.1109/MSR.2007.25

[111] E. Giger, M. Pinzger, and H. Gall, "Predicting the fix time of bugs," in *Proceedings of the 2Nd International Workshop on Recommendation Systems for Software Engineering*, ser. RSSE '10. New York, NY, USA: ACM, 2010, pp. 52–56. [Online]. Available: http://doi.acm.org/10.1145/1808920.1808933

[112] H. Zhang, L. Gong, and S. Versteeg, "Predicting bug-fixing time: An empirical study of commercial software projects," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 1042–1051. [Online]. Available: http://dl.acm.org/citation.cfm?id=2486788.2486931

[113] M. Rees-jones, M. Martin, C. College, and T. Menzies, "Better Predictors for Issue Lifetime," , pp. 1–8, 2017.

[114] Y. Yang, L. Xie, Z. He, Q. Li, V. Nguyen, B. Boehm, and R. Valerdi, "Local bias and its impacts on the performance of parametric estimation models," in *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. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2020390.2020404

[115] T. Menzies, Y. Yang, G. Mathew, B. Boehm, and J. Hihn, "Negative results for software effort estimation," *Empirical Software Engineering*, pp. 1–26, 2016. [Online]. Available: http://dx.doi.org/10.1007/s10664-016-9472-2

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

[117] M. Lowry, M. Boyd, and D. Kulkami, "Towards a theory for integration of mathematical verification and empirical testing," in *Automated Software Engineering, 1998. Proceedings. 13th IEEE International Conference on*. IEEE, 1998, pp. 322–331.

[118] N. Nagappan and T. Ball, "Static analysis tools as early indicators of pre-release defect density," in *ICSE 2005, St. Louis*, 2005.

[119] T. Menzies, D. Raffo, S. on Setamanit, Y. Hu, and S. Tootoonian, "Model-based tests of truisms," in *Proceedings of IEEE ASE 2002*, 2002, available from http://menzies.us/pdf/02truisms.pdf.

[120] T. Menzies, J. Greenwald, and A. Frank, "Data mining static code attributes to learn defect predictors," *IEEE Transactions on Software Engineering*, January 2007, available from http://menzies.us/pdf/06learnPredict.pdf.

[121] A. Tosun, A. Bener, and R. Kale, "AI-based software defect predictors: Applications and benefits in a case study," in *Twenty-Second IAAI Conference on Artificial Intelligence*, 2010.

[122] A. Tosun, A. Bener, and B. Turhan, "Practical considerations of deploying ai in defect prediction: A case study within the Turkish telecommunication industry," in *PROMISE'09*, 2009.

[123] F. Shull, V. B. ad B. Boehm, A. Brown, P. Costa, M. Lindvall, D. Port, I. Rus, R. Tesoriero, and M. Zelkowitz, "What we have learned about fighting defects," in *Proceedings of 8th International Software Metrics Symposium, Ottawa, Canada*, 2002, pp. 249–258.

[124] M. Fagan, "Design and code inspections to reduce errors in program development," *IBM Systems Journal*, vol. 15, no. 3, 1976.

[125] F. Rahman, S. Khatri, E. T. Barr, and P. Devanbu, "Comparing static bug finders and statistical prediction," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 424–434.

[126] C. Lewis, Z. Lin, C. Sadowski, X. Zhu, R. Ou, and E. J. Whitehead Jr., "Does bug prediction support human developers? findings from a google case study," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 372–381. [Online]. Available: http://dl.acm.org/citation.cfm?id=2486788.2486838

[127] S. Rakitin, *Software Verification and Validation for Practitioners and Managers, Second Edition*. Artech House, 2001.

[128] M. D'Ambros, M. Lanza, and R. Robbes, "Evaluating defect prediction approaches: a benchmark and an extensive comparison," *Empir. Softw. Eng.*, vol. 17, no. 4-5, pp. 531–577, aug 2012. [Online]. Available: http://link.springer.com/10.1007/s10664-011-9173-9

[129] R. Wu, H. Zhang, S. Kim, and S.-C. Cheung, "ReLink," in *Proc. 19th ACM SIGSOFT Symp. 13th Eur. Conf. Found. Softw. Eng. - SIGSOFT/FSE '11*. New York, New York, USA: ACM Press, 2011, p. 15.

[130] V. R. Basili, L. C. Briand, and W. L. Melo, "A validation of object-oriented design metrics as quality indicators," *Software Engineering, IEEE Transactions on*, vol. 22, no. 10, pp. 751–761, 1996.

[131] N. Ohlsson and H. Alberg, "Predicting fault-prone software modules in telephone switches," *Software Engineering, IEEE Transactions on*, vol. 22, no. 12, pp. 886–894, 1996.

[132] S. Kim, H. Zhang, R. Wu, and L. Gong, "Dealing with noise in defect prediction," in *Software Engineering (ICSE), 2011 33rd International Conference on*. IEEE, 2011, pp. 481–490.

[133] M. Jureczko and L. Madeyski, "Towards identifying software project clusters with regard to defect prediction," in *Proc. 6th Int. Conf. Predict. Model. Softw. Eng. - PROMISE '10*. New York, New York, USA: ACM Press, 2010, p. 1. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1868328.1868342

[134] L. Breiman, "Random forests," *Machine learning*, pp. 5–32, 2001. [Online]. Available: http://link.springer.com/article/10.1023/A:1010933404324

[135] L. Pelayo and S. Dick, "Applying Novel Resampling Strategies To Software Defect Prediction," in *NAFIPS 2007 - 2007 Annu. Meet. North Am. Fuzzy Inf. Process. Soc.* IEEE, jun 2007, pp. 69–72. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4271036

[136] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "SMOTE: Synthetic minority over-sampling technique," *J. Artif. Intell. Res.*, vol. 16, 2002.

[137] M. Shepperd and S. MacDonell, "Evaluating prediction systems in software project estimation," *Information and Software Technology*, vol. 54, no. 8, pp. 820–827, 2012.

[138] W. B. Langdon, J. Dolado, F. Sarro, and M. Harman, "Exact mean absolute error of baseline predictor, marp0," *Information and Software Technology*, vol. 73, pp. 16 – 18, 2016. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0950584916000057

[139] W. Fu, T. Menzies, and X. Shen, "Tuning for software analytics: Is it really necessary?" *Information and Software Technology*, vol. 76, pp. 135 – 146, 2016. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0950584916300738

[140] S. Kim, E. J. Whitehead Jr, and Y. Zhang, "Classifying software changes: Clean or buggy?" *IEEE Transactions on Software Engineering*, vol. 34, no. 2, pp. 181–196, 2008.

[141] S. Wang, T. Liu, and L. Tan, "Automatically learning semantic features for defect prediction," in *Proceedings of the 38th International Conference on Software Engineering*. ACM, 2016, pp. 297–308.

[142] N. V. Chawla, "C4. 5 and imbalanced data sets: investigating the effect of sampling method, probabilistic estimate, and decision tree structure," in *Proceedings of the ICML*, vol. 3, 2003.

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

[144] R. Shatnawi, "A quantitative investigation of the acceptable risk levels of object-oriented metrics in open-source systems," *IEEE Transactions on software engineering*, vol. 36, no. 2, pp. 216–225, 2010.

[145] Y. Ma and B. Cukic, "Adequate and precise evaluation of quality models in software engineering studies," in *Predictor Models in Software Engineering, 2007. PROMISE'07: ICSE Workshops 2007. International Workshop on*, May 2007, pp. 1–1.

[146] D.-F. Xia, S.-L. Xu, and F. Qi, "A proof of the arithmetic mean-geometric mean-harmonic mean inequalities," *RGMIA research report collection*, vol. 2, no. 1, 1999.

[147] D. L. Vaux, F. Fidler, and G. Cumming, "Replicates and repeatswhat is the difference and is it significant?" *EMBO reports*, vol. 13, no. 4, pp. 291–296, 2012.

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

[149] N. L. Leech and A. J. Onwuegbuzie, "A call for greater use of nonparametric statistics." in *Annual Meeting of the Mid-South Educational Research Association*. ERIC, 2002.

[150] S. Poulding and J. A. Clark, "Efficient software verification: Statistical testing using automated search," *IEEE Transactions on Software Engineering*, vol. 36, no. 6, pp. 763–777, 2010.

[151] A. Arcuri and L. Briand, "A practical guide for using statistical tests to assess randomized algorithms in software engineering," in *ICSE'11*, 2011, pp. 1–10.

[152] M. J. Shepperd and S. G. MacDonell, "Evaluating prediction systems in software project estimation," *Information & Software Technology*, vol. 54, no. 8, pp. 820–827, 2012.

[153] V. B. Kampenes, T. Dybå, J. E. Hannay, and D. I. K. Sjøberg, "A systematic review of effect size in software engineering experiments," *Information & Software Technology*, vol. 49, no. 11-12, pp. 1073–1086, 2007.

[154] E. Kocaguneli, T. Zimmermann, C. Bird, N. Nagappan, and T. Menzies, "Distributed development considered harmful?" in *Proceedings - International Conference on Software Engineering*, 2013, pp. 882–890.

[155] A. Vargha and H. D. Delaney, "A critique and improvement of the cl common language effect size statistics of mcgraw and wong," *Journal of Educational and Behavioral Statistics*, vol. 25, no. 2, pp. 101–132, 2000.

[156] Z. He, F. Peters, T. Menzies, and Y. Yang, "Learning from open-source projects: An empirical study on defect prediction," in *Empirical Software Engineering and Measurement, 2013 ACM/IEEE International Symposium on*. IEEE, 2013, pp. 45–54.