Too Much Automation? The Bellwether Effect and Its Implications for Transfer Learning

Rahul Krishna, Tim Menzies, and Wei Fu

Computer Science, North Carolina State University, USA
{i.m.ralk, tim.menzies}@gmail.com, wfu@ncsu.edu

ABSTRACT

"Transfer learning": is the process of translating quality predictors learned in one data set to another. Transfer learning has been the subject of much recent research. In practice, that research means changing models all the time as transfer learners continually exchange new models to the current project.

This paper offers a very simple "bellwether" transfer learner. Given N data sets, we find which one produces the best predictions on all the others. This "bellwether" data set is then used for all subsequent predictions (or, until such time as its predictions start failing–at which point it is wise to seek another bellwether).

Bellwethers are interesting since they are very simple to find (just wrap a for-loop around standard data miners). Also, they simplify the task of making general policies in SE since as long as one bellwether remains useful, stable conclusions for N data sets can be achieved just by reasoning over that bellwether.

From this, we conclude (1) this bellwether method is a useful (and very simple) transfer learning method; (2) "bellwethers" are a *baseline method* against which future transfer learners should be compared; (3) sometimes, when building increasingly complex automatic methods, researchers should pause and compare their supposedly more sophisticated method against simpler alternatives.

CCS Concepts

•**Software and its engineering** → Software creation and management;

Keywords

Defect Prediction; Data Mining; Transfer learning

1. INTRODUCTION

When building software quality predictors, it might be best to look are more than just the local data. Researchers in *transfer learning* report that data from other projects can yield better predictors than just using local data [1]. This is especially true when the local data is very scarce. For example, consider a new project based on a technology that, previously, has not been used at this site. If that technology that has been extensively explored elsewhere, then it makes good sense to "borrow" other people's data in order to import other people's quality predictors to the new project.

There are many transfer learning methods such as the the *dimensionality transform* approaches of Nam, Jing et al. [2–4] and the *similarity-based* approaches of Kocaguneli, Peters and Turhan et al. [1, 5–7]. In both approaches, when new code modules are created, these approaches comment on code quality using examples taken from similar projects.

Rahman et al. [8] warn that if quality predictors are always being updated based on the specifics of new data, then those new predictors may suffer from over-fitting. Such over-fitted models are "brittle" in the sense that they can undergo constant changes when new data arrives. That is:

> *When learning from all available data , then what we learn may be always changing whenever the available data is changed.*

Such updates are very common and occur when when considering newly constructed code modules or when we are learning using data from other, newly available, projects (for details on this, see §2.2 and the discussion on the *Burak filter*).

Conclusion instability is unsettling for software project managers struggling to find general policies. Such instability prevents project managers offering clear guidelines on many issues including (a) when some module be inspected; (b) when modules should be refactored; (c) where to focus expensive testing procedures; (d) what return-on-investment might we expect due to decreased defects after purchasing some expensive tool; etc.

How to support those managers, who seek stability in their conclusions, while also allowing new projects to take full benefit from data arriving from all the other projects constantly being completed by other programmers? Perhaps if we cannot *generalize* from all data, a more achievable goal is to *stabilize* the pace of conclusion change. While it may be a fool's errand and wait for eternal and global SE conclusions, one possible approach is for organizations to declare some prior project as the *"bellwether"* [1] that offers predictions that generalize across N projects.

This paper defines and distinguish the *bellwether effect* from the *bellwether method*:

- *The bellwether effect* states that when a community of programmers work on a set of projects, then within that community there exists one exemplary project, called the bellwether, which can define quality predictors for the other projects.
- *The bellwether method* searches for that exemplar project and applies it to all future data generated by that community.

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

ASE'16, September 3–7, 2016, Singapore, Singapore
© 2016 ACM. 978-1-4503-3845-5/16/09...$15.00
http://dx.doi.org/10.1145/2970276.2970339

The rest of this paper explores bellwethers. After some background notes, as well as an explanation of the bellwether method, we ask and answer five research questions:

- **RQ1: How rare are "Bellwethers"?** We explore four "communities" of data containing 3, 5, 5 and 10 projects each. In a result consistent with bellwethers *not* being rare, we find that all these communities have a bellwether; i.e. a single data set from which a superior quality predictor can be generated from the rest of that community.
- **RQ2: How does the bellwether data set fare against local models?** The alternate to transfer learning is to just use the local data to build a quality predictor. To answer this research question, we compare the predictions from the bellwether to predictions from just the local data. In our experiments, the bellwether predictions proved to be *better* than those generated from the local data.
- **RQ3: Is bellwether better than other transfer learning methods?** To answer this question, we compare the data predictor generated from the bellwether to the predictions generated from other transfer learning methods. Our bellwether's predictions were observed to be superior to those other transfer learners.
- **RQ4: Can we predict which data set will be bellwether?** To answer this question, we tried reasoning about the data in candidate bellwethers to see if they shared some property that indicates they will be a useful bellwether. The results of this investigation were not positive since we could find no such property. Hence, our recommended method for finding the bellwether is to try it out against other data sets.
- **RQ5: How much data is required to find the bellwether?** Since RQ4 failed to find a statistical property that selects for bellwethers, then the only way we have to find bellwethers is to compare the performance of pairs of data sets from different projects. A natural question that arises from this experimental approach is RQ5. Our experiments show that program managers need not wait very long to find their bellwethers – a few dozen examples of defective code modules can suffice for creating and testing candidate bellwethers.

From the above, we conclude that the original motivation for transfer learning in SE might have been misguided. Initial experiments with transfer learning in SE built defect predictors from the *union* of data taken from multiple projects. That approach lead to some very poor results so researchers turned to *relevancy filters* to find what small subset of the data was relevant to the current problem [7]. These relevancy filters generated adequate predictions but introduced the instability problem that motivates this paper. Our bellwether results suggest that relevancy filtering would never have been necessary in the first place if researchers had instead hunted for bellwethers.

2. BACKGROUND

2.1 Defect Prediction

Our example quality predictors are static code attributes defect prediction. Hall et al. offers an extensive review on the defect prediction literature [9]. For an extensive experimental comparison of different learning algorithms for defect prediction, see Lessmann et al. [10]. For brief introduction notes on defect predition, see the rest of this section.

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 [11]. Exponential costs quickly exhaust finite resources so standard practice is to apply the best available methods only on code sections that seem most critical. Any method that focuses on parts of the code can miss defects in other areas so some sampling policy should be used to explore the rest of the system. This sampling policy will always be incomplete, but it is the only option when resources prevent a complete assessment of everything.

One such lightweight sampling policy is defect predictors learned from static code attributes. Given software described in the attributes of Figure 1, data miners can learn where the probability of software defects is highest.

The rest of this section argues that such defect predictors are *easy to use*, *widely-used*, and *useful* to use.

Easy to use: Static code attributes can be automatically collected, even for very large systems [12]. Other methods, like manual code reviews, are far slower and far more labor-intensive. For example, depending on the review methods, 8 to 20 LOC/minute can be inspected and this effort repeats for all members of the review team, which can be as large as four or six people [13].
Widely used: Researchers and industrial practitioners use static attributes to guide software quality predictions. Defect prediction models have been reported at Google [14]. Verification and validation (V&V) textbooks [15] advise using static code complexity attributes to decide which modules are worth manual inspections.

Useful: Defect predictors often find the location of 70% (or more) of the defects in code [16]. Defect predictors have some level of generality: predictors learned at NASA [16] have also been found useful elsewhere (e.g. in Turkey [17, 18]). 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* [19] concluded that manual software reviews can find $\approx$60% of defects. In another work, Raffo documents the typical defect detection capability of industrial review methods: around 50% for full Fagan inspections [20] 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. [21] 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 1. The same is not true for static code analyzers– these need extensive modification before they can be used on new languages.

2.2 Defect Prediction and 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* S projects to the *target* project T.

Initial experiments with transfer learning offered very pessimistic results. Zimmermann et al. [22] 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

wmc	weighted methods per class	
dit	depth of inheritance tree	
noc	number of children	
cbo	coupling between objects	increased when the methods of one class access services of another.
rfc	response for a class	number of methods invoked in response to a message to the object.
lcom	lack of cohesion in methods	number of pairs of methods that do not share a reference to an instance variable.
ca	afferent couplings	how many other classes use the specific class.
ce	efferent couplings	how many other classes is used by the specific class.
npm	number of public methods	
locm3	another lack of cohesion measure	if m, a are the number of $methods, attributes$ in a class number and $\mu(a)$ is the number of methods accessing an attribute, then $lcom3 = ((\frac{1}{a}\sum_j^a \mu(a_j)) - m)/(1-m)$.
loc	lines of code	
dam	data access	ratio of private (protected) attributes to total attributes
moa	aggregation	count of the number of data declarations (class fields) whose types are user defined classes
mfa	functional abstraction	number of methods inherited by a class plus number of methods accessible by member methods of the class
cam	cohesion amongst classes	summation of number of different types of method parameters in every method divided by a multiplication of number of different method parameter types in whole class and number of methods.
ic	inheritance coupling	number of parent classes to which a given class is coupled (includes counts of methods and variables inherited)
cbm	coupling between methods	total number of new/redefined methods to which all the inherited methods are coupled
amc	average method complexity	e.g. number of JAVA byte codes
max_cc	maximum McCabe	maximum McCabe's cyclomatic complexity seen in class
avg_cc	average McCabe	average McCabe's cyclomatic complexity seen in class
defect	defect	Boolean: where defects found in post-release bug-tracking systems.

Figure 1: Sample static code attributes.

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 alarming high false positive rates (60% or more).

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

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

Homogenous, heterogenous transfer learning operates on source and target data that contain the *same, different* attribute names (respectively). This paper focuses on homogenous 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 at some overlap in task, tools, personnel, and development platforms. Hence, data can contain overlapping attributes. As evidence for this, the data sets explored in this paper fall into 4 communities and each community has many overlapping attributes (specifically, our four communities have 20, 23, 26,61 overlapping attributes, see Figure 2).

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 [7] builds its training sets by finding the $k = 10$ nearest code modules in S for every $t \in T$.

(Aside: Note that the Burak filter suffers from the instability problems described in the introduction: 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 [5, 6] 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" are pruned (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 [23] 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. *Dimensionality transform* methods manipulate the raw source data until it matches the target. In the case of defect prediction, a "dimension" might be one of the static code attributes of Figure 1. For example, Nam et al. [2] originally proposed an optimization-based method that used dimensionality rotational and expansion/contraction to align the source dimensions to the target [2]. Subsequently, that team found they could dispense with the optimizer [3] 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, prefer instead a canonical-correlation analysis (CCA) to find the relationships between variables in the source and target data [24].

Our reading of the literature is that dimensionality transform is used mostly in *heterogeneous*, and not *homogeneous*, transfer learning. Hence, our experiments use similarity-based methods.

3. BELLWETHERS: A NEW APPROACH

The previous section sampled some of the work on transfer learning in software engineering. This rest of this paper asks the question "is the complexity of §2.2 really necessary?"

To answer this question, we propose a process 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 three operators- GENERATE, APPLY, MONITOR:

1. GENERATE: *Using historical data, check if the community has bellwether.* See if data miners can predict for the number of issues, given the static code attributes.
   - For all pairs of data from projects $P_i, P_j \in C$;
   - Predict for issues in P_j using a quality predictor learned from data taken from P_i;
   - Report a bellwether if one P_i generates the most accurate predictions in a majority of $P_j \in C$.
2. APPLY: *Using the bellwether, generate quality predictors 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 during APPLY start decreasing*.

Note the simplicity of this approach– just wrap a for-loop around some data miners. Note also that these steps use *none* of the machine described in §2.2.

4. RESEARCH QUESTIONS

RQ1: How rare are "Bellwethers"?

If bellwethers occur infrequently, we cannot rely on them. Hence, this question explores how common are bellwethers. To this end, we applied the GENERATE method described above to the four communities shown in Figure 2. This data was selected according to the following rules:

- The data has been used in prior transfer learning paper; e.g. [3];
- The communities are quite diverse; e.g. the NASA projects are proprietary while the others are open source projects.
- In addition, the projects also vary in their granularity of data description (file, class, or function level).

RQ2: How does the bellwether fare against local models?

One premise of transfer learning is that using data from other projects is as useful, or better, then using data from the local project. This research questions tests that this premise holds for bellwethers.

To answer this question, we implemented APPLY as follows. One of our communities (APACHE) comes in multiple versions; e.g. in APACHE, the XALAN system has versions 2.4, 2.5, 2.6, 2.7. Each versions are historical releases where version i was written before version j where $j > i$. RQ2 was explored in this community as follows:

- The last version of each project was set aside as a hold-out.
- GENERATE was then applied across the older versions within the community to find the bellwether.
- A defect predictor was then learned from the older data seen in the bellwether.
- The predictor was then applied to the latest data.

We compare the above to *local learning*; i.e. for each project:

- The last version of that project was set aside as a hold-out;
- The older versions of that project were then used to train a defect predictor.
- The predictor was then applied to the latest data.

Note that:

- The local learner only ever uses data from earlier in the *same project*;
- While the bellwether uses data from *any member* of the community.

RQ3: Is bellwether better than other transfer learning methods?

Our reading of the literature is that dimensionality-reduction transfer learning is the preferred choice for heterogeneous transfer learning while, for the homogeneous transfer learning studied here, similarity based approaches are the norm. Hence, we compare bellwether against two similarity based transfer learners: the first classic Burak filter from 2009 [7] as well as a more recent *mixed* approach that uses a small sample of the target along with the available source data [25].

RQ4: Can we predict which data set will be bellwether?

This question tries to reason about bellwether dataset to identify characteristics, if any, that make it unique.

To do this, we compare the distributions of the code quality metrics that make up the bellwether data with that of the other datasets. We employ a multiple comparison test [26] [27].

RQ5: How much data is required to find the bellwether?

A core process in all the above is the GENERATE step. If this requires too much data to find bellwethers, then that would indicate developers should eschew bellwethers in favor of standard transfer learning. Hence, it is important to ask how much data is required before a community can find adequate bellwethers.

5. METHODOLOGY

5.1 Benchmark Datasets

This study uses 120 data sets grouped into 4 communities taken from previous transfer learning studies.. The projects measure defects at defect levels of granularity ranging from function-level to file-level Figure 2 summarizes all the communities of datasets used in our experiments.

For the reasons discussed in §2.2, 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 [3]. This dataset was gathered by D'Amborse et al. [28], 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. [29] who used the Understand tool [2], 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

[2]http://www.scitools.com/products/

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)		
NASA	cm	998	126 (12.63)	23	Function
	jm	25157	5103 (20.28)		
	kc	588	108 (18.37)		
	mc	13630	292 (2.14)		
	mw	770	81 (10.52)		

Figure 2: 120 Defect Datasets from 4 communities. The # metrics columns shows the number of metrics that are shared by all members of that community.

information in it has been manually verified and corrected. It has been widely used in defect prediction [3] [29] [30] [31] [32].

In addition to this, we explored two other communities of datasets from the PROMISE repository[3]. The first set of group contains defect measures from several Apache projects. It was gathered by Jureczko et al. [33]. 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 [34].

Further, we used 5 proprietary datasets from NASA containing similar metrics [35]. For the sake of consistency, we cleaned up the dataset so that they all share the same metrics. These datasets measure McCabe and Halstead's cyclomatic complexity metrics in addition to other complexity metrics such as parameter count and percentage comments.

5.2 Learning Methods

There are many ways to predict defects. A comprehensive study on the same was conducted by Lessmann et al. [10]. They endorsed the use of Random Forests [36] for defect prediction over several other methods. 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.

[3]http://openscience.us/repo/

It is known that the fraction of defects in the training samples affects the performance of defect predictors. Figure 2 shows that in most datasets, the percentage of defective samples varies between 10% to 20% (except in a few, projects like log4j where it is 58%). Handling this class imbalance has been shown to improve the quality of defect prediction. Pelayo and Dick [37] report that the defect prediction is improved by SMOTE [38]. 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 *non-defective, defective* examples until the training data had only *100,50* non-defective, defective examples (respectively).

Important methodological note: sub-sampling was only applied to *training* data (so the test data remains unchanged).

5.3 Evaluation Strategy

In our context, we consider modules with defects as positive instances and those without as negative instances. Prediction models are not ideal, they therefore need to be evaluated in terms of statistical performance measures. On classification 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 fewer the false negative results. ; (3) *False alarm or pf*: percentage of non-defective instances wrongly identified as defective. Unlike pf, lower the pd better the 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. There is also a trade-off between precision and recall. 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. One such approach 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 [10]. Random Forest lacks a threshold parameter, it is capable of producing just one point on the ROC curve. It is therefore not possible to compute AUC. In a previous work, Ma and Cukic [39] have shown that distance from perfect classification (ED) can be substituted for AUC in cases where a ROC curve cannot be generated. ED measures the distance between obtained (Pd, Pf) pair and the ideal point on the ROC space (1, 0), weighted by cost function θ. It is given by:

$$ED = \sqrt{\theta.(1 - Pf)^2 + (1 - \theta).Pd^2} \quad (1)$$

Note that for ED, the *smaller* the distance, the *better* the predictor.

Setting θ to (e.g.) 0.5 places equal weights on Pd and Pf. From a software engineering perspective, it is more important to reduce

misclassification of defective module that to reduce false classification of fault-free modules. With this in mind, we have set θ as 0.6, thereby placing more weight on Pd.

In this work, we report only on ED-measures. However, for the reader of this work who wishes to use different performance measures, we have made available a replication package[4] in order to facilitate the computation of other statistical measures.

5.4 Statistics

To overcome the inherent randomness introduced by Random Forests and SMOTE, we use 40 repeated runs, each time with a different random number seed (we use 40 since that is more than the 30 samples needed to satisfy the central limit theorem). The repeated runs provide us with a sufficiently large sample size to statistically compare all the datasets. Each run collects the values of ED (Equation 1). (Note: We refrain from performing a cross validation because the process tends to mix the samples from training data (the bellwether) and the test data (the other projects), which defeats the purpose of this study.)

To rank these 40 numbers collected as above, we use the Scott-Knott test recommended by Mittas and Angelis [40]. Scott-Knott is a top-down clustering approach used to rank different treatments. If that clustering finds an interesting division of the 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 1 values found in $ls = 4$ different methods. Then, we split l into sub-lists m, n in order to maximize the expected value of differences in the observed performances before and after divisions. E.g. for lists l, m, n of size ls, ms, ns where $l = m \cup n$:

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

We then apply a statistical hypothesis test H to check if m, n are significantly different (in our case, the conjunction of A12 and bootstrapping). If so, Scott-Knott recurses on the splits. In other words, 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 ($A12 \geq 0.6$).

For a justification of the use of non-parametric bootstrapping, see Efron & Tibshirani [41, p220-223]. For a justification of the use of effect size tests see Shepperd and MacDonell [42]; Kampenes [43]; and Kocaguenli et al. [44]. These researchers warn that even if a hypothesis test declares two populations to be "significantly" different, then that result is misleading if the "effect size" is very small. Hence, to assess the performance differences we first must rule out small effects using A12, a test recently endorsed by Arcuri and Briand [45].

6. RESULTS

6.1 RQ1: How rare are "Bellwethers"?

Figures 3 show the results of GENERATE within our four communities. It is immediately noticeable that for each community there is one data set that provides consistently better predictions when compared to other datasets. For example: Apache's bellwether is Lucene; NASA's bellwether is MC; AEEEM's bellwether is LC; and Relink's bellwether is Safe. Thhat is, all the communities studied here have a bellwether. Hence:

Research answer 1
Our results suggest bellwethers are not rare.

6.2 RQ2: How does the bellwether fare against local models?

Figure 4 compares ED scores of defect predictors built on local models against those built with a bellwether. For this question, we used data from the Apache community since it has the versions required to test older data against newer data.

As see in the figure, the prediction scores with the bellwether is very encouraging in case of the Apache datasets. In all cases, except for Jedit, defect prediction models constructed with the Lucene bellwether performs as well as local data. In some cases (Xerces), Lucene performs much better than local data. Therefore, the answer to the second research question is:

Research answer 2
For projects evaluated with the same quality metrics, training a defect prediction model with the Bellwether is just as good as doing so with local data.

RQ3: Is bellwether better than other transfer learning methods?

Figure 5 shows results comparing three homogenous transfer learning methods: bellwether, the classic Burak filter (denoted "Turhan09") and Turhan's subsequent update to that method (denoted "Turhan11"). We note that in usual case, bellwethers perfrom much better than those other methods. Hence:

Research answer 3
The bellwether method out-performs standard homogenous transfer learning methods.

6.3 RQ4: Can we predict which data set will be bellwether?

To study existing trends in the bellwether data, we tried to identify the existence of statistical similarities between the distributions of the bellwether samples and the samples from our test cases. To do this, we performed a multiple comparison test using Kruskal-Wallis H-Test. The Kruskal-Wallis H-test tests the null hypothesis that the population median of two or more groups are equal. It is a non-parametric version of the ANOVA test.

In each group, we compared the distribution of the metric values of bellwether dataset and all other datasets. If the null hypothesis, as formulated above, is rejected, that means there doesn't exist a statistically significant similarity between medians of the bellwether and the test data for that metric.

If the bellwether bore any resemblance to the test data, we would expect to see several metrics with statistically significant similarity. For instance, in the Apache projects in Figure 6, we noticed that there doesn't exist any noticeable similarities. Further, as highlighed in Figure 7, the same effect was noticed in all the other projects as well. A mere reflection on the distribution or feature importance is not sufficient to determine of a specific dataset is a bellwether or not. Therefore, our response to the third research question is as follows:

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

	Ant		Camel		Ivy		Poi		Velocity		Xalan		Xerces		Jedit		Log4j		Lucene	
	Med	Iqr	Med	Iqr	Med	Iqr	Med	Iqr	Med	Iqr	Med	Iqr	Med	Iqr	Med	Iqr	Med	Iqr	Med	Iqr
Ant			0.33	0.01	0.32	0.01	0.28	0.01	0.28	0.02	0.3	0.01	0.3	0.01	0.34	0.01	0.28	0.02	0.28	0
Camel	0.61	0.01			0.62	0.02	0.57	0.01	0.47	0.01	0.53	0.01	0.42	0.01	0.71	0.03	0.42	0.02	0.4	0.01
Ivy	0.41	0.01	0.39	0.02			0.43	0.03	0.33	0.01	0.38	0.02	0.37	0.01	0.51	0.04	0.35	0.01	0.32	0.01
Poi	0.6	0.03	0.54	0.02	0.57	0.03			0.59	0.03	0.42	0.04	0.33	0.01	0.66	0.03	0.36	0.01	0.33	0.01
Velocity	0.49	0.02	0.42	0.01	0.64	0.01	0.67	0.01			0.54	0.01	0.44	0.02	0.73	0.02	0.51	0.01	0.49	0.01
Xalan	0.56	0.01	0.52	0.02	0.62	0.02	0.59	0.01	0.56	0.01			0.46	0.01	0.66	0.01	0.46	0.01	0.45	0
Xerces	0.71	0.01	0.51	0.01	0.67	0.01	0.62	0.02	0.62	0.01	0.55	0.01			0.71	0.01	0.49	0.01	0.48	0.01
Jedit	0.35	0.01	0.38	0.01	0.48	0.01	0.37	0.01	0.47	0.01	0.28	0	0.37	0			0.31	0.01	0.34	0.01
Log4j	0.62	0.01	0.42	0.01	0.59	0.01	0.56	0	0.39	0.01	0.53	0.02	0.39	0	0.68	0.03			0.49	0.03

	CM		JM		KC		MW		MC	
	Med	Iqr	Med	Iqr	Med	Iqr	Med	Iqr	Med	Iqr
CM			0.41	0	0.69	0	0.49	0	0.37	0
JM	0.77	0			0.65	0	0.69	0	0.38	0
KC	0.79	0	0.44	0			0.59	0	0.4	0
MW	0.62	0	0.57	0	0.41	0			0.32	0

	EQ		JDT		ML		PDE		LC	
	Med	Iqr	Med	Iqr	Med	Iqr	Med	Iqr	Med	Iqr
EQ			0.62	0.01	0.35	0.01	0.4	0.01	0.34	0.01
JDT	0.36	0.02			0.28	0.03	0.28	0.01	0.28	0
ML	0.42	0.01	0.51	0.02			0.39	0.02	0.37	0.01
PDE	0.32	0.01	0.54	0.01	0.34	0.01			0.36	0.01

	Apache		ZXing		Safe	
	Med	Iqr	Med	Iqr	Med	Iqr
Apache			0.51	0.09	0.46	0
Zxing	0.79	0			0.51	0

Figure 3: Identifying the "Bellwether", This figure compares the prediction performance of the bellwether (highlighted in bold) against other datasets. "Med" refers to the median value seen in 40 repeats. "IQR" refers to the 75th-25th percentile seen in those 40 repeats (and the IQR is very small). All performance figures here are the ED from Equation 1 so *lower* values are *better*. Cells highlighted in gray produce the best performance with the highest Scott-Knott ranks.

		Bellwether		Local	
		Med	Iqr	Med	Iqr
Apache	Ant	0.3	0.01	0.36	0.01
	Camel	0.37	0.02	0.43	0.01
	Poi	0.23	0.02	0.36	0.01
	Xalan	0.36	0.01	0.45	0
	Xerces	0.44	0.01	0.62	0.01
	Ivy	0.28	0.01	0.28	0.01
	Velocity	0.41	0.01	0.41	0
	Log4j	0.39	0.01	0.39	0.01
	Jedit	0.43	0.01	0.38	0.02

Figure 4: Bellwether vs. Local Data. Performance scores are ED so *lower* values are *better*. We note that in all datasets except for Jedit, the performance of Bellwether dataset is just as well as local data.

Research answer 4
Although there seemingly always exists a Bellwether dataset, identifying this dataset by reflecting on distribution of the data is not trivial.

RQ5: How much data is required to find the bellwether?

As yet, we do not have a theoretical analysis offering a lower bound for the the number of examples required for finding the bellwethers. What we do have is the following empirical observation: all the above results were achieved using the sub-sampling methods of §5.2; i.e. 100 randomly selected non-defective modules and 50 randomly selected defective modules. That is:

Research answer 5
Bellwethers can be found after projects have discovered a few dozen examples of defective modules.

		Bellwether		Turhan09		Turhan11	
		Med	Iqr	Med	Iqr	Med	Iqr
Apache	Ant	0.31	0.01	0.77	0.01	0.77	0.1
	Camel	0.42	0.02	0.42	0.01	0.43	0.01
	Ivy	0.32	0.01	0.4	0.01	0.41	0.01
	Poi	0.3	0	0.42	0.01	0.48	0.04
	Velocity	0.41	0.01	0.77	0	0.77	0.01
	Xerces	0.51	0.01	0.77	0.01	0.77	0
	Jedit	0.3	0.01	0.65	0.02	0.65	0.03
	Log4j	0.48	0.01	0.43	0.01	0.57	0.06
	Xalan	0.45	0.01	0.41	0	0.42	0.01
AEEEM	EQ	0.35	0.01	0.77	0	0.77	0.01
	JDT	0.28	0.01	0.77	0.01	0.77	0.02
	ML	0.35	0.02	0.77	0.01	0.77	0.03
	PDE	0.41	0	0.75	0.01	0.76	0.02
Relink	Apache	0.46	0.02	0.77	0	0.77	0.02
	ZXing	0.51	0.01	0.77	0.01	0.77	0.03
NASA	CM	0.37	0	0.69	0	0.75	0.02
	JM	0.38	0	0.42	0.01	0.69	0
	KC	0.4	0	0.69	0	0.57	0
	MW	0.32	0	0.76	0.02	0.7	0

Figure 5: Bellwether vs. Homogeneous Transfer Learning methods (Turhan09 [7] and Turhan11 [25]). All results are ED so *lower* values are *better*. Cells highlighted in gray indicate datasets with superior prediction capability.

7. THREATS TO VALIDITY

7.1 Sampling Bias

Sampling bias threatens any classification experiment; what matters in one case may or may not hold in another case. For example,

	wmc	dit	noc	cbo	rfc	lcom	ca	ce	npm	lcom3	loc	dam	moa	mfa	cam	ic	cbm	amc	max_cc	avg_cc
ant	✓	·	·	✓	·	·	·	·	·	·	·	·	·	·	·	·	✓	·	·	·
camel	·	✓	·	✓	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·
ivy	✓	·	·	·	✓	·	✓	·	·	·	·	·	·	·	·	·	·	·	·	✓
jedit	✓	·	·	·	·	·	·	·	·	·	·	✓	·	·	✓	✓	✓	·	·	·
log4j	·	·	·	·	✓	·	·	·	·	·	·	·	·	·	✓	·	·	✓	·	·
poi	·	✓	·	·	·	·	✓	✓	·	·	·	✓	·	✓	✓	·	·	·	✓	✓
velocity	·	✓	·	✓	✓	·	✓	·	✓	·	·	·	·	·	·	·	·	·	·	·
xalan	·	·	·	✓	✓	·	·	·	✓	·	·	·	·	·	·	·	·	·	✓	✓
xerces	✓	✓	·	·	·	·	·	·	·	·	·	·	·	✓	·	·	·	·	·	✓

Figure 6: Results of Kruskal-Wallis H-Test comparing the bellwether dataset (Lucene) with the other datasets from the Apache. Each dataset contains 20 Static Code Metrics (for a description of each of these metrics, please refer to [46]). The rows contain the test data, and the columns contain the metrics. A "✓" symbol represents a significant statistical similarity (with a 95% confidence interval) and a "." represents a no similarity.

Group	Dataset	# metrics		Bellwether
		Significant	%	
AEEEM	EQ	26/61	42	LC
	JDT	12/61	19	
	ML	28/61	46	
	PDE	13/61	21	
Relink	Apache	0/26	0	Safe
	ZXing	11/26	42	
Apache	Ant	3/20	15	Lucene
	Ivy	4/20	20	
	Camel	2/20	10	
	Poi	8/20	40	
	Jedit	5/20	25	
	Log4j	3/20	15	
	Velocity	5/20	25	
	Xalan	5/20	25	
	Xerces	4/20	20	
NASA	cm	0/21	0	mc
	jm	0/21	0	
	kc	0/21	0	
	mw	0/21	0	

Figure 7: Results of Kruskal-Wallis H-Test comparing the bellwether datasets with the test datasets.

even though we use 120 open-source data sets in this study (Figure 2) which come from several sources (Apache and NASA were obtained from the PROMISE repository and ReLink and AEEEM were obtained from [24]), 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 §4) and suggest that other researchers try a broader range of data in future work.

7.2 Learner Bias

For building the defect predictors in this study, we elected to use random forests. We chose this learner because past studies shows that, for defect prediction, the results were superior to other more complicated algorithms [10] and can act as a baseline for other algorithms.

Apart from learner choice, one limitation to our current study is that we have focused here on homogenous transfer learning (where the attributes in source and target have the same name). The implications for heterogeneous transfer learning (where the attributes in source an target have different names) are no clear. We have some initial results suggesting that an bellwether-like effect occurs when learning across the communities of Figure 2 but those results are very preliminary. Hence, for the moment, we would conclude:

- For the homogenous case, we recommend using bellwethers rather than similarity-based transfer learning.
- For the heterogenous case, we recommend using dimensionality tranforms.

7.3 Evaluation Bias

This paper uses one measure of prediction quality, ED (see Equation 1). Other quality measures often used in software engineering to quantify the effectiveness of prediction [39] [47] [48] (discussed in §5.3). A comprehensive analysis using these measures is left for future work.

7.4 Order Bias

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

To mitigate these biases, we run the experiments 40 times (the reruns are greater than 30 in keeping with the central limit theorem). Note that the reported variations over those runs were very small (see the low IQR values in Tables 3, 4, and 5). Hence, we conclude that while order bias is theoretically a problem, it is not a major problem in the particular case of this study.

8. CONCLUSIONS AND DISCUSSION

When historical data is limited or not available (e.g. perhaps due the project being in its infancy), developers might seek data from other projects. Our results show that regardless of the granularity of data (see the file, class, file values of Figure 2), there exists a bellwether data set that can be used to train relatively more accurate defect prediction models. This bellwether does 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 (after uncovering a few dozen defective code modules).

As discussed in the introduction, the results of this paper cast some doubts on the results that originally motivated much of the transfer learning results in defect prediction since the original 2009 Turhan paper on the Burak filter [7]. Our bellwether results suggest the relevancy filtering of the Burak filter would never have been necessary in the first place if researchers had instead discovered bellwethers.

Finally, when discussing this work, there are three frequently asked questions:

1. *Do bellwethers guarantee permanent conclusion stability?* 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 necessary is not desirable– hence this paper.

2. *How to detect when bellwethers need updating?* The conclusion stability offered by bellwethers only lasts as long as the bellwether remains useful. Hence, bellwether performance must always be monitoried and, if that performance starts to dip, then seek a new bellwether.

3. *What happens if a set of data has no useful bellwether?* In that case, there are numerous standard transfer learning methods that could be used to import lessons learned from other data [1–7,49]. That said, the result here is that all the communities of data explored by this paper had useful bellwethers. Hence, we would recommend trying bellwethers before moving on to more complex methods.

Further to this last point, in his text on empirical software engineering, Cohen [50] recommends benchmarking supposedly more sophisticated methods 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.

9. ACKNOWLEDGEMENTS

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

10. REFERENCES

[1] Fayola Peters, Tim Menzies, and Lucas Layman. LACE2: Better privacy-preserving data sharing for cross project defect prediction. In *Proceedings - International Conference on Software Engineering*, volume 1, pages 801–811, 2015.

[2] Jaechang Nam, Sinno Jialin Pan, and Sunghun Kim. Transfer defect learning. In *Proceedings - International Conference on Software Engineering*, pages 382–391, 2013.

[3] Jaechang Nam and Sunghun Kim. Heterogeneous defect prediction. In *Proc. 2015 10th Jt. Meet. Found. Softw. Eng. - ESEC/FSE 2015*, pages 508–519, New York, New York, USA, 2015. ACM Press.

[4] X. Jing, G. Wu, X. Dong, F. Qi, and B. Xu. Heterogeneous cross-company defect prediction by unified metric representation and cca-based transfer learning. In *FSE'15*, 2015.

[5] Ekrem Kocaguneli and Tim Menzies. How to find relevant data for effort estimation? In *Empirical Software Engineering and Measurement (ESEM), 2011 International Symposium on*, pages 255–264. IEEE, 2011.

[6] Ekrem Kocaguneli, Tim Menzies, and Emilia Mendes. Transfer learning in effort estimation. *Empirical Software Engineering*, 20(3):813–843, 2014.

[7] Burak Turhan, Tim Menzies, Ayşe B Bener, and Justin Di Stefano. On the relative value of cross-company and within-company data for defect prediction. *Empirical Software Engineering*, 14(5):540–578, 2009.

[8] Foyzur Rahman, Daryl Posnett, and Premkumar Devanbu. Recalling the "imprecision" of cross-project defect prediction. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 61:1–61:11, New York, NY, USA, 2012. ACM.

[9] Tracy Hall, Sarah Beecham, David Bowes, David Gray, and Steve Counsell. A Systematic Review of Fault Prediction Performance in Software Engineering. *IEEE Transactions on Software Engineering*, 38(6):1276–1304, 2011.

[10] Stefan Lessmann, Bart Baesens, C. Mues, and S. Pietsch. Benchmarking Classification Models for Software Defect Prediction: A Proposed Framework and Novel Findings. *IEEE Trans. Softw. Eng.*, 34(4):485–496, jul 2008.

[11] Michael Lowry, Mark Boyd, and Deepak Kulkami. Towards a theory for integration of mathematical verification and empirical testing. In *Automated Software Engineering, 1998. Proceedings. 13th IEEE International Conference on*, pages 322–331. IEEE, 1998.

[12] Nachiappan Nagappan and Thomas Ball. Static analysis tools as early indicators of pre-release defect density. In *ICSE 2005, St. Louis*, 2005.

[13] Tim Menzies, David Raffo, Siri on Setamanit, Ying Hu, and Sina Tootoonian. Model-based tests of truisms. In *Proceedings of IEEE ASE 2002*, 2002. Available from http://menzies.us/pdf/02truisms.pdf.

[14] Chris Lewis, Zhongpeng Lin, Caitlin Sadowski, Xiaoyan Zhu, Rong Ou, and E. James Whitehead Jr. Does bug prediction support human developers? findings from a google case study. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 372–381, Piscataway, NJ, USA, 2013. IEEE Press.

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

[16] T. Menzies, J. Greenwald, and A. Frank. Data mining static code attributes to learn defect predictors. *IEEE Trans. Softw Eng.*, 33(1):2–13, Jan 2007. Available from http://menzies.us/pdf/06learnPredict.pdf.

[17] 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.

[18] 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.

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

[20] M. Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(3), 1976.

[21] Foyzur Rahman, Sameer Khatri, Earl T Barr, and Premkumar Devanbu. Comparing static bug finders and statistical prediction. In *Proc. International Conference on Software Engineering*, pages 424–434. ACM, 2014.

[22] Thomas Zimmermann, Nachiappan Nagappan, Harald Gall, Emanuel Giger, and Brendan 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*, pages 91–100. ACM, 2009.

[23] Feng Zhang, Audris Mockus, Iman Keivanloo, and Ying Zou. Towards building a universal defect prediction model with rank transformed predictors. *Empirical Software Engineering*, pages 1–39, 2015.

[24] Xiaoyuan Jing, Fei Wu, Xiwei Dong, Fumin Qi, and Baowen 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)*, pages 496–507, 2015.

[25] Burak Turhan, Ayşe Tosun, and Ayşe Bener. Empirical evaluation of mixed-project defect prediction models. In *Software Engineering and Advanced Applications (SEAA), 2011 37th EUROMICRO Conference on*, pages 396–403. IEEE, 2011.

[26] Yoav Benjamini. Simultaneous and selective inference: Current successes and future challenges. *Biometrical Journal*, 52(6):708–721, 2010.

[27] G Rupert Jr et al. *Simultaneous statistical inference*. Springer Science & Business Media, 2012.

[28] Marco D'Ambros, Michele Lanza, and Romain Robbes. Evaluating defect prediction approaches: a benchmark and an extensive comparison. *Empir. Softw. Eng.*, 17(4-5):531–577, aug 2012.

[29] Rongxin Wu, Hongyu Zhang, Sunghun Kim, and Shing-Chi Cheung. ReLink. In *Proc. 19th ACM SIGSOFT Symp. 13th Eur. Conf. Found. Softw. Eng. - SIGSOFT/FSE '11*, page 15, New York, New York, USA, 2011. ACM Press.

[30] Victor R Basili, Lionel C Briand, and Walcélio L Melo. A validation of object-oriented design metrics as quality indicators. *Software Engineering, IEEE Transactions on*, 22(10):751–761, 1996.

[31] Niclas Ohlsson and Hans Alberg. Predicting fault-prone software modules in telephone switches. *Software Engineering, IEEE Transactions on*, 22(12):886–894, 1996.

[32] Sunghun Kim, Hongyu Zhang, Rongxin Wu, and Liang Gong. Dealing with noise in defect prediction. In *Software Engineering (ICSE), 2011 33rd International Conference on*, pages 481–490. IEEE, 2011.

[33] Marian Jureczko and Lech Madeyski. Towards identifying software project clusters with regard to defect prediction. In *Proc. 6th Int. Conf. Predict. Model. Softw. Eng. - PROMISE '10*, page 1, New York, New York, USA, 2010. ACM Press.

[34] Rahul Krishna, Tim Menzies, and Lucas Layman. Which "Bad Smells" Can Be Ignored? In *Submitted to FSE '16*, 2016.

[35] Martin Shepperd, Qinbao Song, Zhongbin Sun, and Carolyn Mair. Data Quality: Some Comments on the NASA Software Defect Datasets. *IEEE Trans. Softw. Eng.*, 39(9):1208–1215, sep 2013.

[36] L Breiman. Random forests. *Machine learning*, pages 5–32, 2001.

[37] Lourdes Pelayo and Scott Dick. Applying Novel Resampling Strategies To Software Defect Prediction. In *NAFIPS 2007 - 2007 Annu. Meet. North Am. Fuzzy Inf. Process. Soc.*, pages 69–72. IEEE, jun 2007.

[38] Nitesh V. Chawla, Kevin W. Bowyer, Lawrence O. Hall, and W. Philip Kegelmeyer. SMOTE: Synthetic minority over-sampling technique. *J. Artif. Intell. Res.*, 16, 2002.

[39] 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*, pages 1–1, May 2007.

[40] Nikolaos Mittas and Lefteris Angelis. Ranking and clustering software cost estimation models through a multiple comparisons algorithm. *IEEE Trans. Software Eng.*, 39(4):537–551, 2013.

[41] Bradley Efron and Robert J Tibshirani. *An introduction to the bootstrap*. Mono. Stat. Appl. Probab. Chapman and Hall, London, 1993.

[42] Martin J. Shepperd and Steven G. MacDonell. Evaluating prediction systems in software project estimation. *Information & Software Technology*, 54(8):820–827, 2012.

[43] Vigdis By Kampenes, Tore Dybå, Jo Erskine Hannay, and Dag I. K. Sjøberg. A systematic review of effect size in software engineering experiments. *Information & Software Technology*, 49(11-12):1073–1086, 2007.

[44] Ekrem Kocaguneli, Thomas Zimmermann, Christian Bird, Nachiappan Nagappan, and Tim Menzies. Distributed development considered harmful? In *Proceedings - International Conference on Software Engineering*, pages 882–890, 2013.

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

[46] Tim Menzies, Andrew Butcher, Andrian Marcus, Thomas Zimmermann, and David Cok. Local vs. global models for effort estimation and defect prediction. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, pages 343–351. IEEE, nov 2011.

[47] Tim Menzies, Alex Dekhtyar, Justin Distefano, and Jeremy Greenwald. Problems with Precision: A Response to "Comments on 'Data Mining Static Code Attributes to Learn Defect Predictors'". *IEEE Transactions on Software Engineering*, 33(9):637–640, sep 2007.

[48] W. Fu, T. Menzies, and X. Shen. Tuning for software analytics: is it really necessary? *Information and Software Technology (submitted)*, 2016. Read on-line at https://goo.gl/Jp5VIm.

[49] Zhimin He, Fayola Peters, Tim Menzies, and Ye Yang. Learning from open-source projects: An empirical study on defect prediction. In *Empirical Software Engineering and Measurement, 2013 ACM/IEEE International Symposium on*, pages 45–54. IEEE, 2013.

[50] Paul R Cohen. *Empirical methods for artificial intelligence*, volume 139. MIT press Cambridge, 1995.