Learning GENERAL Principles from Hundreds of Software Projects

Suvodeep Majumder, Rahul Krishna and Tim Menzies, *IEEE Fellow*

Abstract—When one exemplar project, which we call the "bellwether", offers the best advice then it can be used to offer advice for many other projects. Such bellwethers can be used to make quality predictions about new projects, even before there is much experience with those new projects.
But existing methods for bellwether transfer are very slow. When applied to the 697 projects studied here, they took 60 days of CPU to find and certify the bellwethers Hence, we propose GENERAL: a novel bellwether detection algorithm based on hierarchical clustering. At each level within a tree of clusters, one bellwether is computed from sibling projects, then promoted up the tree. This hierarchical method is a scalable approach to learning effective models from very large data sets. For example, for nearly 700 projects, the defect prediction models generated from GENERAL's bellwether were just as good as those found via standard methods.

Index Terms—Transfer Learning, Bellwether, Defect Prediction, Software Analytics.

1 Introduction

How should we reason about software quality? Should we use general models that hold over many projects? Or must we use an ever-changing set of ideas that are continually adapted to the task at hand? Or does the truth lie somewhere in-between? To say that another way:

- Are there general principles we can use to guide project management, software standards, education, tool development, and legislation about software?
- Or is software engineering some "patchwork quilt" of ideas and methods where it only makes sense to reason about specific, specialized, and small sets of projects?

If the latter were true then then there would be no stable conclusions about what is best practice for SE (since those best practices would keep changing as we move from project to project). As discussed in section 2.1, such conclusion instability has detrimental implications for *generality*, *trust*, *insight*, *training*, and *tool development*.

Finding general lessons across multiple projects is a complex task. A new approach, that shows much promise, is the "bellwether" method for transferring conclusions between projects [1]–[5] (the bellwether is the leading sheep of a flock, with a bell on its neck). That method:

- Finds a "bellwether" project that is exemplar for the rest;
- Draw conclusions from that project.

This approach has been successfully applied to defect prediction, software effort estimation, bad smell detection, issue lifetime estimation, and configuration optimization. As a method of transferring lessons learned from one project to another, bellwethers have worked better than the Burak filter [6], Ma et al. [7]'s transfer naive Bayes (TNB); and Nam et al. TCA and TCA+, algorithms [8], [9].

In terms of *transfer* and *lessons learned*, such bellwethers have tremendous practical significance. When new projects arrive, then even before there is much experience with those new projects, lessons learned from other projects can be applied to the new project (just by studying the bellwether), Further, since the bellwether is equivalent to the other models, then when new projects appear, their quality can be evaluated even before there is an extensive experience base within that particular project (again, just by studying the bellwether).

But existing methods for bellwether transfer are very slow. When applied to the 697 projects studied here, they took 60 days of CPU to find and certify the bellwethers. There are many reasons for that including how the models were certified (20 repeats with different train/test sets) and the complexity of the analysis procedure (which includes fixing class imbalance and feature selection). But the major cause of this slow down was that those methods required an $O(N^2)$ comparison between $N = 697$ projects.

This paper reports a novel approach that dramatically improves on existing bellwether methods. Our GENERAL method uses hierarchical clustering model to recursively divide a large number of projects into smaller clusters. Starting at the leaves of that tree of clusters, GENERAL finds the bellwethers within sibling projects. That bellwether is then promoted up the tree. The output of GENERAL is the project promoted to the top of the tree of clusters.

This paper evaluates the model built from the project found by GENERAL. We will find that the predictions from this model are as good, or better, than those found via "within" learning (where models are trained and tested on local data) or "all-pairs" learning (where models are found after building models from all pairs of projects). That is

> *Learning from many other projects can be better than learning just from your own local project data.*

GENERAL's clustering methods divide N projects into m groups. In that space, GENERAL only needs to compare $O(m*(N/m)^2)$ projects to find the bellwether. Theoretically and empirically, this means GENERAL will scale up much better than traditional methods. For example:

- This paper applies GENERAL and traditional $O(N^2)$ bellwether to 697 projects. GENERAL and the traditional approach terminated in 1.5 and 72 hours (respectively).

- *S, Majumder, R. Krishna and T. Menzies are with the Department of Computer Science, North Carolina State University, Raleigh, USA. E-mail:[smajumd3,rkrish11]@ncsu.edu, tim@ieee.org*

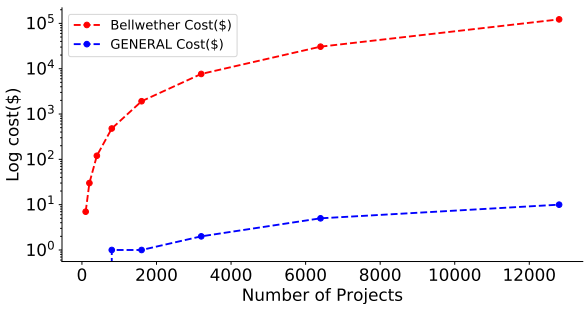

Fig. 1: Hypothetical cost comparison between GENERAL and default Bellwether.

- Figure 1 shows a hypothetical cost comparison in AWS between standard bellwethers and GENERAL when running for 100 to 1,000,000 projects. Note that GENERAL is inherently more scalable.

Using GENERAL, we can explore these research questions:

RQ1: Can hierarchical clustering tame the complexity of bellwether-based reasoning?

> **Theoretically and empirically, the hierarchical reasoning on GENERAL performs much faster than standard bellwether methods.**

RQ2: Is this faster bellwether effective?

> **Measured in terms of predictive performance, the effectiveness of hierarchical bellwethers is very similar to local learning (and these two methods are more effective than the other options explored here).**

RQ3: Does learning from too many projects have detrimental effect?

> **Assessed in terms of *recall*, it is better to learn bellwethers from more data rather than less. But assessed in terms of *false alarms*, while learning bellwethers from many projects is useful, it is possible to learn from too much data.**

RQ4: What exactly did we learn from all those projects?

> **The importance of many key features is not apparent at the local level. To fully appreciate the critical impact on defects of class interface design, it is necessary to conduct a global analysis across hundreds of projects.**

Overall, the contributions of this paper are

- **Hierarchical bellwethers for transfer learner:** We offer a novel hierarchical clustering bellwether algorithm called GENERAL (described in section 3) that finds bellwether in hierarchical clusters, then promotes those bellwether to upper levels The final project that is promoted to the root of the hierarchy is returned as "the" bellwether.
- **Showing inherent generality in SE:** In this study we discover a source data set for transfer learner from a large number of projects, hence proving generality in the SE datasets (where some datasets can act as exemplars for the rest of them for defect prediction).
- **Lessons about software quality that are general to hundreds of software projects:** As said above, in this sample of 697 projects, we find that code interface issues are the dominant factor on software defects.
- **Replication Package:** We have made available a replication package[1] . The replication package consists of all the datasets used in this paper, in addition to mechanisms for computation of other statistical measures.

The rest of this paper is structured as follows. Some background and related work are discussed in section 2. Our algorithm GENERAL is described in section 3. Data collection and experimental setup are in section 4. Followed by evaluation criteria in section 4.4 and performance measures in section 4.5. The results and answers to the research questions are presented in section 5, which is followed by threats to validity in section 6. Finally the conclusion is provided in section 7.

2 Background and Related Work

2.1 Why Seek Generality?

There are many reasons to seek stable general conclusions in software engineering. If our conclusions about best practices for SE projects keep changing, that will be detrimental to generality, trust, insight, training, and tool development.

Generality: Data science for software engineering cannot be called a "science" unless it makes general conclusions that hold across multiple projects. If we cannot offer general rules across a large number of software projects, then it is difficult to demonstrate such generality.

Trust: Hassan [10] cautions that managers lose faith in software analytics if its models keep changing since the assumptions used to make prior policy decisions may no longer hold.

Insight: Kim et al. [11], say that the aim of software analytics is to obtain actionable insights that help practitioners accomplish software development goals. For Tan et al. [12], such insights are a core deliverable. Sawyer et al. agree, saying that insights are the key driver for businesses to invest in data analytics initiatives [13]. Bird, Zimmermann, et al. [14] say that such insights occur when users reflect, and react, to the output of a model generated via software analytics. But if new models keep being generated in new projects, then that exhausts the ability of users to draw insight from new data.

Training: Another concern is what do we train novice software engineers or newcomers to a project? If our models are not stable, then it hard to teach what factors most influence software quality.

Tool development: Further to the last point— if we are unsure what factors most influence quality, it is difficult to design and implement and deploy tools that can successfully improve that quality.

1. http://tiny.cc/bellwether

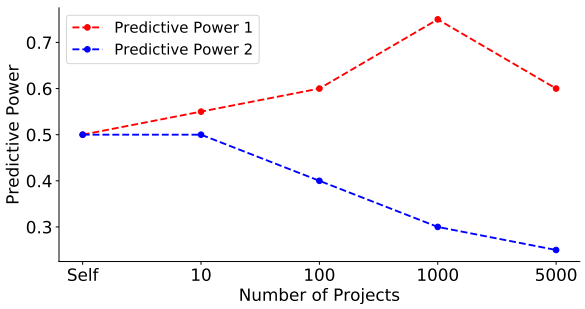

Fig. 2: Two hypothetical results about how training set size might effect the efficacy of quality prediction for software projects.

2.2 Why Shun Generality?

Just to balance the above argument, we add that sometimes it is possible to learn from *too much data*. Petersen and Wohlin [15] argue that for empirical SE, context matters. That is, they would predict that one model will not cover all projects and that tools that report generality over many software projects need to also know the *communities* within which those conclusions apply. Hence, this work divides into (a) automated methods for finding sets of projects in the same *community*; and (b) within each *community*, find the model that works best.

The *size* of the communities found in this way would have a profound impact on how we should reason about software engineering. Consider the hypothetical results of Figure 2. The **BLUE** curve shows some quality predictor that (hypothetically) gets better, the more projects it learns from (i.e. higher levels in the hierarchical cluster). After about learning from 1000 projects, the **BLUE** curve's growth stops and we would say that community size here was around cluster size in level 1. In this case, while we could not offer a single model that covers *all* of SE, we could offer a handful of models, each of which would be relevant to project clusters at that level.

Now consider the hypothetical **RED** curve of Figure 2. Here, we see that (hypothetically) learning from more projects makes quality predictions worse which means the our 10,000 projects break up into "communities" of size one. In this case, (a) principles about what is "best practice" for different software projects would be constantly changing (whenever we jump from small community to small community); and (b) the generality issues would be becoming open and urgent concerns for the SE analytics community.

In summary, the above two sections lead to our research question **RQ4**: does learning from too many projects have detrimental effects. Later in this paper, we will return to this issue.

2.3 Why Transfer Knowledge?

In this section, we ask "Why even bother to transfer lessons learned between projects?". In several recent studies [16]–[18] with readily-available data from SE repositories, numerous authors report the locality effect in SE; i.e. general models outperformed by specialized models localized to particular parts of the data. For example. Menzies et al. explored local vs global learning in defect prediction and effort estimation [17] and found that learning rules from specific local data was more effective than learning rules from the global space.

On the other hand, Herbold et al. [19] offered an opposite conclusion. In their study regarding global vs local model for cross-project defect prediction, they saw that local models offered little to no improvement over models learned from all the global data. One explanation for this discrepancy is the size of number of projects that they explored. Menzies, Herbold et al. explored less than two dozen projects which raises issues of external validity in their conclusions. Accordingly, here, we explore nearly 700 projects. As shown below, the results of this paper agree more with Herbold et al. than Menzies et al. since we show that one global model (learned from a single bellwether projects) does just as well as anything else.

Apart from the above discrepancy in research results, there are many other reasons to explore learning from many projects. Those reasons falls into four groups:

(a) The lesson on big data is that that the *more* training data, the *better* the learned model. Vapnik [20] discusses examples where models accuracy improves to nearly 100%, just by training on 10^2 times as much data. This effect has yet to be seen in SE data [21] but that might just mean we have yet to use enough training data (hence, this study).

(b) We need to learn from more data since there is very little agreement on what has been learned to far: Another reason to try generalizing across more SE data is that, among developers, there is little agreement on what many issues relating to software:

- According to Passos et al. [22], 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" [22].
- Jørgensen & Gruschke [23] offer a similar warning. They report that the suppose software engineering "gurus" rarely use lessons from past projects to improve their future reasoning and that such poor past advice can be detrimental to new projects. [23].
- 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 [24].

The good news is that using software analytics, we can correct the above misconceptions. If data mining shows that doing XYZ is bug prone, then we could guide developers to avoid XYZ. But will developers listen to us? If they ask "are we sure XYZ causes problems?", can we say that we have mined enough projects to ensure that XYZ is problematic?

It turns out that developers are not the only one's confused about how various factors influence software projects. Much recent research calls into question the "established wisdoms" of SE field. For example, here is a list of recent conclusions that contradict prior conclusions:

- In stark contrast to much prior research, pre- and post-release failures are not connected [25];

- Static code analyzers perform no better than simple statistical predictors [26];
- The language construct GOTO, as used in contemporary practice, is rarely considered harmful [27];
- Strongly typed languages are not associated with successful projects [28];
- Test-driven development is not any better than "test last" [29];
- Delayed issues are not exponentially more expensive to fix [30];

Note that if the reader disputes any of the above, then we ask how would you challenge the items on this list? Where would you get the data, from enough projects, to successfully refute the above? And where would you get that data? And how would you draw conclusions from that large set? Note that the answers to these questions requires learning from multiple projects. Hence, this paper.

(c) Imported data can be more useful than local data: Another benefit of importing data from other projects is that, sometimes, that imported data can be better than the local information. For example, Rees-Jones reports in one study that while building predictors for Github close time for open source projects [31] using data from other projects performs much better then building models using *local learning* (because there is better information *there* than *here*).

(d) When there is insufficient local data, learning from other projects is very useful: When developing new software in novel areas, it is useful to draw on the relevant experience from related areas with a larger experience base.This is particularly true when developers are doing something that is novel to them, but has been widely applied elsewhere For example, Clark and Madachy [32] discuss 65 types of software they see under-development by the US Defense Department in 2015. Some of these types are very common (e.g. 22 ground-based communication systems) but other types are very rare (e.g. only one avionics communication system). (e.g. workers on flight avionics might check for lessons learned from ground-based communications). Developers working in an uncommon area (e.g. avionics communications) might want to transfer in lessons from more common areas (e.g. ground-based communication).

2.4 How to Transfer Knowledge

This art of moving data and/or lessons learned from one project or another is *Transfer Learning*. When there is insufficient current 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. [33] 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. Successful transfer learning can have two variants -

- **Heterogeneous Transfer Learning:** This type of transfer learning operates on source and target data that contain the different attributes.
- **Homogeneous Transfer Learning:** This kind of transfer learning operates on source and target data that contain the same attributes. This paper explores scalable methods for homogeneous transfer.

Another way to divide transfer learning is the approach that is followed. There are 2 approaches that are frequently used in many research: **similarity-based approaches** and **dimensional transforms**.

Similarity-Based Approaches: In this approach we can transfer some/all subset of the rows or columns of data from source to target. For example, the Burak filter [6] 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 [34], [35] 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 [36] 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.

Dimensional Transformation: In this approach we manipulate the raw source data until it matches the target. An initial attempt on performing transfer learning with Dimensionality transform was undertaken by Ma et al. [7] 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. [8] 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, When there are no overlapping attributes (in heterogeneous transfer learning) Nam et al. [9] 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 [37].

Considering all the attempts at transfer learning sampled above, suggested a surprising lack of consistency in the choice of datasets, learning methods, and statistical mea-

sures while reporting results of transfer learning. This issue was addressed by "Bellwether" suggested by Krishna et al. [38], [39]. which is a simple transfer learning technique is defined in 2- folds namely the Bellwether effect and the Bellwether method:

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

In their paper Krishna et al. performed experiment with communities of 3, 5 and 10 projects in each, and showed that (a) bellwethers are not rare, (b) their prediction performance is better than local learning, and (c) they do fairly well when compared with the state-of-the-art transfer learning methods discussed above. This motivated us to use bellwethers as our choice of method for transfer learning to search for generality in SE datasets.

That said, Krishna et al. warn that in order to find bellwether we need to do a $N * (N-1)$ comparison; i.e. standard bellwethers have complexity $O(N^2)$ (N being the number of projects in community).

$$Bellwether\ complexity = O(N^2) \quad (1)$$

The goal of this paper is to find ways to reduce the Equation 1 complexity.

3 About GENERAL

Our proposed improvement to bellwethers is called GENERAL. The core intuition of this new approach is that if many projects are similar, then we do not need to run comparisons between all pairs of projects. When such similar projects exist, if may suffice to just compare a small number of representative examples.

Accordingly, the rest of this paper performs the following experiment:

1) Summarize the projects via **feature extraction** (see §3.1).
2) Using some clustering algorithm, group all our data into sets of similar projects.
3) The groups are themselves grouped into super-groups, then super-super-groups, etc to form a tree. This step requires a **hierarchical clustering** algorithm (see §3.2).
4) Starting at the leaves of that tree, the best project is discovered by training on each project, and test its model on all others in that leaf cluster. This step needs a **data mining** algorithm to generate models (see §3.3) and a comparison method to **select the best model** (see §3.4).
5) Once bellwether from each group is pushed up the tree, then steps 4,5 are repeated, recursively.
6) The project pushed to the root of the tree is then used as the bellwether for all the projects.

Note that when the clustering algorithm divides the data into m clusters, then the complexity of this method (which we call GENERAL) is:

$$GENERAL\ complexity = O(m * (N/m)^2) \quad (2)$$

Figure 1 contrasts the computational cost of Equation 2 with Equation 1 (and that figure assumes $m = 20$, which is the division constant we used in these experiments– see below.). As seen in that figure, the $O(m*(N/m)^2)$ analysis is inherently more scalable than the $O(N^2)$ analysis required by standard bellwether.

To operationalize steps 1,2,3,4,5 listed above, we need to make some lower-level engineering decisions. The rest of this section documents those decisions.

3.1 Feature Extraction

Prior to anything else, we must summarize our projects. Xenos [40] distinguishes between *product metrics* (e.g. counts of lines of code); and *process metrics* about the development process (e.g. number of file revisions). Using the Understand tool [41], we calculated 21 product and 5 process metrics to build defect prediction models (see Table 1). These product metrics are calculated from snapshots from every 6 months of the data. The process metrics are computed using the change history in the six-months period before the split date via manual collection of data using scripts. The data collected for this project is summarized in Figure 3 and Figure 4.

Understand is a widely used tool in software analytics [36], [42]–[46]. The advantage of using this tool is that much of the tooling needed for this kind of large scale analysis is already available. On the other hand, it also means that we can only reason about the features that Understand can report– which could be a threat to the validity of the conclusions reached. As shown below, the Table 1 metrics were shown to be effective for our task. Nevertheless, in future work, this study needs to be repeated whenever new tools allow for the widespread collection of different kinds of features.

3.2 Hierarchical Clustering

After data collection, comes the hierarchical clustering needed for step 2. For this purpose, we followed the advice from the scikit.learn [47] documentation that recommends the Balanced Iterative Reducing and Clustering using Hierarchies (BIRCH) algorithm for hierarchical clustering for large sample datasets that might contain spurious outliers [48]. BIRCH has the ability to incrementally and dynamically cluster incoming, multi-dimensional data in an attempt to maintain best quality clustering. BIRCH also has the ability to identify data points that are not part of the underlying pattern (so it can effectively identifying

Language	Projects
Java	290
CPP	241
C	116
CS	42
Pascal	8

Fig. 3: Distribution of projects depending on languages. Many projects use combinations of languages to achieve their results. Here, we show majority language used in each project.

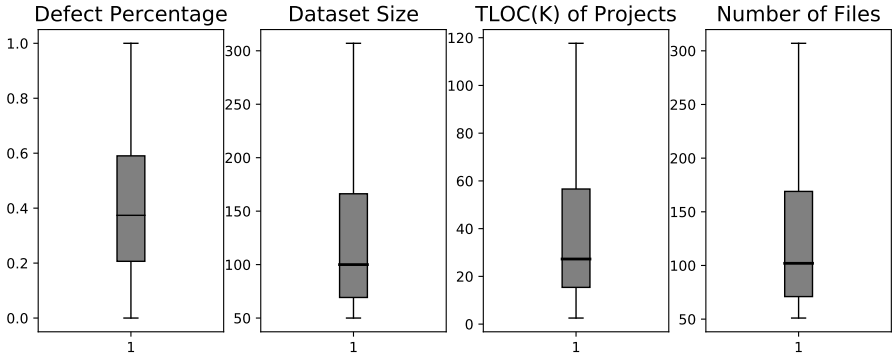

Fig. 4: Distribution of projects depending on Defect Percentage, Data set Size, Lines of Code and Number of Files.

Metric	Metric level	Metric Name	Metric Description
Product	File	LOC	Lines of Code
		CL	Comment Lines
		NSTMT	Number of Statements
		NFUNC	Number of Functions
		RCC	Ratio Comments to Code
		MNL	Max Nesting Level
	Class	WMC	Weighted Methods per Class
		DIT	Depth of Inheritance Tree
		RFC	Response For a Class
		NOC	Number of Immediate Sub-classes
		CBO	Coupling Between Objects
		LCOM	Lack of Cohesion in Methods
		NIV	Number of instance variables
		NIM	Number of instance methods
		NOM	Number of Methods
		NPBM	Number of Public Methods
		NPM	Number of Protected Methods
		NPRM	Number of Private Methods
	Methods	CC	McCabe Cyclomatic Complexity
		FANIN	Number of Input Data
		FANOUT	Number of Output Data
Process	File	NREV	Number of revisions
		NFIX	Number of revisions a file
		ADDED LOC	Lines added
		DELETED LOC	Lines deleted
		MODIFIED LOC	Lines modified

TABLE 1: List of software metrics used in this study.

and avoid outliers). Google Scholar reports that the original paper proposing BIRCH has been cited over 5,400 times. For this experiment we used defaults proposed by [48]; a branching factor of 20 and the "new cluster creation" threshold of 0.5.

3.3 Data Mining

The bellwether analysis of step3 requires a working data miner. Three requirements for that learner are:

- Since it will be called thousands of times, it must run quickly. Hence, we did not use any methods that require neural nets or ensembles.
- Since some projects have relatively few defects, before learning, some *over-sampling* is required to increase the number of defective examples in the training sets
- Since one of our research questions (**RQ5**) asks "what did we learn from all these projects", we needed a learning method that generate succinct models. According, we used *feature selection* to check which subset of Table 1 mattered the most.

According, this study used:

- The logistic regression learner (since it is relatively fast);
- The SMOTE class imbalance correction algorithm [49], which we run on the training data[2];
- and Hall's CFS feature selector [52][3].

We selected use these tools since in the domain of software analytics, the use of LR (logistic regression) and SMOTE is endorsed by recent ICSE papers [51], [53], [54]. As to CFS, we found that without it, our recalls were very low and we could not identify which metrics mattered the most. Also, extensive studies have found that CFS more useful than many other feature subset selection methods such as PCA or InfoGain or RELIEF [52].

3.4 Select the Best Model

As discussed below, the defect models assessed in these experiments

To find the bellwether, our method must compare many models and select the best one. As discussed below, we score model performance according to five goals:

- Maximize recall and precision and popt(20);
- While minimizing false alarms and ifa_auc.

(For definitions and details for these criteria, and why we selected them, see §4.5.)

In such *multi-objective* problems, one model is better than another if it satisfies a "domination predicate". We use the Zitler indicator dominance predictor [55] to select our bellwether (since this is known to select better models for

2. The SMOTE Synthetic Minority Over-Sampling Technique algorithms subsamples the majority class (i.e., deletes examples) while over-sampling the minority class until all classes have the same frequency. To over-sample, new examples are synthesized extrapolating between known examples (of the minority class) and its k nearest neighbors. While it is useful to artificially boost the number of target examples in the training data [5], [49], [50], it is a methodological error to also change the distributions in the test data [51]. Hence, for our work, we take care to *only* resample the training data.

3. CFS is based on the heuristic that "good feature subsets contain features highly correlated with the classification, yet uncorrelated to each other". Using this heuristic, CFS performs a best-first search to discover interesting sets of features. Each subset is scored via $merit_s = kr_{cf}/\sqrt{k + k(k-1)r_{ff}}$ where $merit_s$ is the value of some subset s of the features containing k features; r_{cf} is a score describing the connection of that feature set to the class; and r_{ff} is the mean score of the feature to feature connection between the items in s. Note that for this to be maximal, r_{cf} must be large and r_{ff} must be small. That is, features have to connect more to the class than each other.

5-goal optimization [56], [57]). This predicate favors model y over x model if x "losses" most:

$$\begin{array}{rcl} worse(x,y) & = & loss(x,y) > loss(y,x) \\ loss(x,y) & = & \sum_j^n -e^{\Delta(j,x,y,n)}/n \\ \Delta(j,x,y,n) & = & w_j(o_{j,x} - o_{j,y})/n \end{array} \quad (3)$$

where "n" is the number of objectives (for us, $n = 5$) and $w_j \in \{-1, 1\}$ depending on whether we seek to maximize goal x_j.

An alternative to the Zitler indicator is 'boolean domination " that says one thing is better than another it if it no worse on any criteria and better on at least one criteria. We prefer Equation 3 to boolean domination since we have a 5-goal optimization problem and it it is known that boolean domination often fails for 3 or more goals [56], [58].

4 Experimental Methods

4.1 Data Collection

To perform our experiments we choose to work with defect prediction datasets. We use the data collected by Zhang et al. [59]. This data has the features of Table 1. Originally, this data was collected by Mockus et al. [60] from SourceForge and GoogleCode. The dataset contains the full history of about 154,000 projects that are hosted on SourceForge and 81,000 projects that are hosted on GoogleCode to the date they were collected. In the original dataset each file contained the revision history and commit logs linked using a unique identifier. Although there were 235K,000 projects in the original database, many of there are trivially small or are about non-software development projects. Zhang et al. cleaned the dataset using the following criteria:

- **Avoid projects with a small number of commits:** Zhang et al. removed any projects with less than 32 commits (which is the 25 % quantile of the number of commits as the threshold).
- **Avoid projects with lifespan less than one year:** Zhang et al. filtered out any projects with a lifespan less than one year.
- **Avoid projects with limited defect data:** Zhang et al. in their study counted the number of fix-inducing and non-fixing commits from a one-year period and removed any projects with 75 % quantile of the number of fix-inducing and non-fixing commits.
- **Avoid projects without fix-inducing commits:** Zhang et al. filtered out projects that have no fix-inducing commits during six months as abnormal projects, as projects in defect prediction studies need to contain both defective and non-defective commits.

On top of that, we also applied two more filters:

- **Use mainstream programming Languages:** the tool we used (Understand [41]) only supported mainstream languages in widespread industrial use; specifically: object-oriented languages with file extension i.e *.c, *.cpp, *.cxx, *.cc, *.cs, *.java, and *.pas.
- **Avoid projects with less than 50 rows:** We removed any project with less than 50 rows as they are too small to build a meaningful predictor.
- **Avoid projects with too few errors:** We pruned projects which did not have enough fix-inducing vs non-fixing data points to create a stratified k=5 fold cross-validation an

These filters resulted in a training set of 697 projects[4]. Fig 4 and fig 3 shows the Distribution of projects depending on defect percentage, data set size, lines of code, number of files and project languages to confirm the projects selected comes from wide verity and representative of a software community. From these selected projects, the data was labeled using issue tracking system and commit messages. If a project used issue tracking system for maintaining issue/defect history the data was labeled using that. Like Zhang et al., we found that nearly half of the projects did not use an issue tracking system. For these projects, labels were created analyzing commit messages by tagging them as fix-inducing commit if commit message matches the following regular expression

(bug | fix | error | issue | crash | problem | fail | defect | patch)

4.2 Experimental Setup

Figure 5 illustrates our experimental rig. The following process was repeated 20 times, with different random seeds used each time.

- Projects were divided randomly into train_1 and test_1 as a 90:10 split.
- The projects in train_1 were used to find the bellwether .
- Each project in test_1 was then divide into train_2 and test_2 (using a 2:1 split).
- LR and feature selection and SMOTE were then used to build two models: one from the train_1 bellwether and one from the train_2 data.
- Both models were then applied to the test_2 data.

4.3 Learners

In this study, we applied the follow learners:

Self: (a.k.a. local learning). This is the standard method used in software analytics [61], [62]. In this approach, the local project data is divided into a 90% training set (which we call train_2) and a 10% test set (which we call test_2). After that, some some learner builds a model from the training data (using the methods of §4.2), which is then assessed on the test data.

As we shall see, this approach produces competent defect predictors. Recalling the motivation of this paper: we do not seek better predictor that is (say) more accurate than **self**. That is, hierarchical bellwethers can be recommended even if they *perform no better than* **self**. Rather:

- As listed in the motivations of §2.1, we seek ways to make conclusions across a wide number of projects.
- That is, our goal is to test if hierarchical bellwethers can quickly find a small set of *adequate conclusions* that hold across a large space of projects.

So, here, by "adequate", we mean conclusions that perform no worse than those found by other methods.

ZeroR: In his textbook on "Empirical AI", Cohen [63] recommends base-lining new methods against some simpler approach. For that purpose, we use **ZeroR** learner. This learner assigns labels every test instance according to the

4. http://tiny.cc/bellwether_data

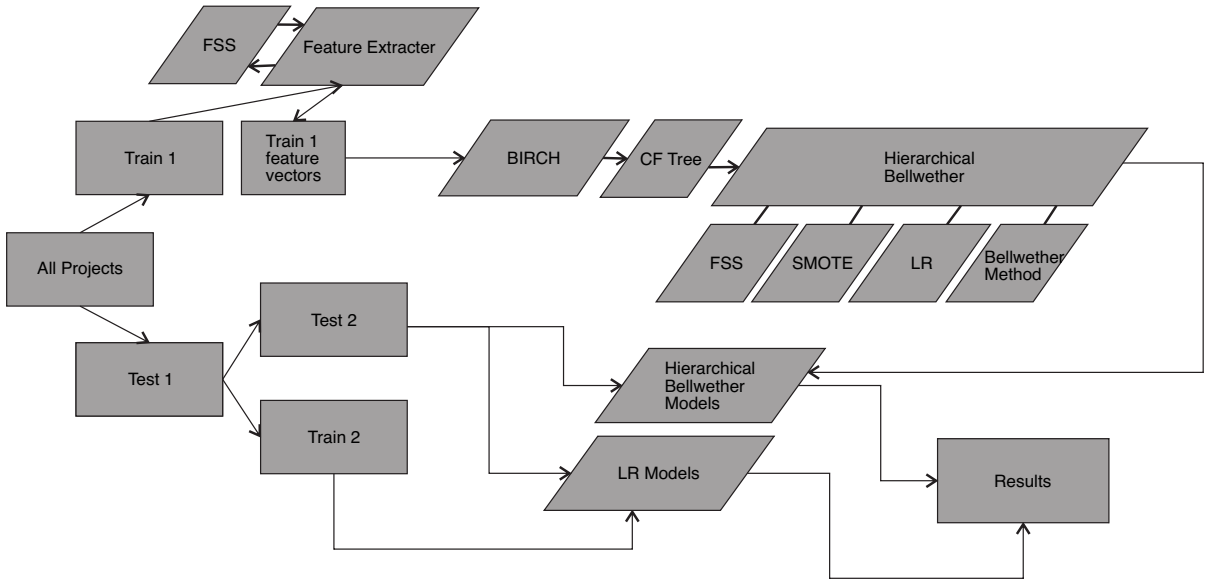

Fig. 5: Experimental rig for this paper. In this rig, bellwethers are learned and tested on *separate* projects. Within the test set (denoted "test1", above), the data is further divided into "train2, test2". To assess the bellwether found from "train1" against local learning, data from each project is divided into "train2,test2" then (a) local models are learned from "train2"; after which time, (b) the local models from "train2" and the bellwether model from "train1" are both applied to the same data from "test2". Note that this process is repeated 20 times, with different random number seeds, to generate 20 different sets of "train1, train2, test2".

majority class of the training data. Note that if anything we do performs worse that ZeroR, then there is no point to any of the learning technology explored in this paper.

Global: Another baseline, against which we compare our methods is a **Global** learner build using all the data train_1. Note that, if this learner performs best, then this would mean that we could replace GENERAL with a much simpler system.

Bellwether0: This learner is the $O(N^2)$ bellwether method proposed by Krishna et al. [2]. What will we show is that GENERAL does better than **Bellwether0** is three ways: (a) GENERAL is inherently more scalable; (b) GENERAL is (much) faster; and (c) GENERAL produced better predictions. That is, our new GENERAL method is a significant improvement over the prior state-of-the-art.

GENERAL_level2: GENERAL finds bellwethers at various levels of the BIRCH cluster tree. GENERAL_level2 results show the performance of the model learned from the bellwether found in the leaves of the BIRCH cluster tree. That is these results come from a bellwether generated from 15 to 30 projects. For this process:

- First, we tag each leaf cluster with its associated bellwether;
- Second, we use the test procedure built into BIRCH; i.e. a test case is presented to the root of the cluster tree and BIRCH returns its *relevant leaf*; i.e. the cluster closest to that test case.
- We then apply the bellwether tagged at that leaf.

GENERAL_level1: GENERAL_level1 results show the performance of the model learned from the bellwether found between the root and the leaves of the BIRCH cluster tree. In practice, BIRCH divides our data only twice so there is only one GENERAL_level1 between root and leaves. For this process, we use the same procedure as GENERAL_level2 but this time, we use the bellwether tagged in the *parent* cluster of the relevant leaf. Note that these level1 results come from an analysis of between 50 to 200 projects (depneding on the shape of the cluster tree genrated via BIRCH).

GENERAL_level0: In the following, the GENERAL_level0 results show the performance of the model learned from the bellwether found at the root of the BIRCH cluster tree. Note that these results come from an analysis of over 600 projects.

4.4 Statistical Tests

When comparing the results different models in this study, we used a statistical significance test and an effect size test. Significance test is useful for detecting if two populations differ merely by random noise. Also, effect sizes are useful for checking that two populations differ by more than just a trivial amount. For the significance test, we use the Scott-Knott procedure recommended at TSE'13 [64] and ICSE'15 [54]. This technique recursively bi-clusters a sorted set of numbers. If any two clusters are statistically indistinguishable, Scott-Knott reports them both as one group. Scott-Knott first looks for a break in the sequence that maximizes the expected values in the difference in the means before and after the break. More specifically, it splits l values into sub-lists m and n in order to maximize the expected value of differences in the observed performances before and after divisions. For e.g., lists l, m and n of size ls, ms and ns where $l = m \cup n$, Scott-Knott divides the sequence at the break that maximizes:

$$E(\Delta) = \frac{ms}{ls} \times abs(m.\mu - l.\mu)^2 + \frac{ns}{ls} \times abs(n.\mu - l.\mu)^2 \quad (4)$$

Scott-Knott then applies some statistical hypothesis test H to check if m and n are significantly different. If so, Scott-Knott then recurses on each division. For this study, our hypothesis test H was a conjunction of the A12 effect size test (endorsed by [65]) and non-parametric bootstrap sampling [66], i.e., our Scott-Knott divided the data if *both* bootstrapping and an effect size test agreed that the division was statistically significant (90% confidence) and not a "small" effect ($A12 \geq 0.6$).

4.5 Performance Measures

In this section, we introduce the following 5 evaluation measures used in this study to evaluate the performance of machine learning models. Suppose we have a dataset with M changes and N defects. After inspecting 20% LOC, we inspected m changes and found n defects. Also, when we find the first defective change, we have inspected k changes. Using this data, we can define 5 evaluation measures as follows:

(1) **Recall:** This is the proportion of inspected defective changes among all the actual defective changes; i.e. n/N. Recall is used in many previous studies [67]–[71].

(2) **Precision:** This is the proportion of inspected defective changes among all the inspected changes; i.e. n/m. A low Precision indicates that developers would encounter more false alarms, which may have negative impact on developers' confidence on the prediction model.

(3) **pf:** This is the proportion of all suggested defective changes which are not actual defective changes among all the suggested defective changes. A high *pf* suggests developers will encounter more false alarms which may have negative impact on developers' confidence in the prediction model.

(4) **popt20:** This is the proportion number of suggested defective changes among all suggested defective changes, when when 20% LOC modified by all changes are inspected. A high *popt20* values mean that developers can find most bugs in a small percent of the code. To compute Popt20, we divided the test set into the modules predicted to be faulty (set1) and predicted to be bug-free (set2). Each set was then sorted in ascending order by lines of code. We then ran down set1, then set2, till 20% of the total lines of code were reached– at which point *popt20* is the percent of buggy modules seen up to that point.

(5) **ifa_auc:** Number of initial false alarms encountered before we find the first defect. Inspired by previous studies on fault localization [72]–[74], we caution that if the top-k changes recommended by the model are all false alarms, developers would be frustrated and are not likely to continue inspecting the other changes. For example, Parnin and Orso [72] found that developers would stop inspecting suspicious statements, and turn back to traditional debugging, if they could not get promising results within the first few statements they inspect. Using the nomenclature reported about *Ifa*$= k$. In this study we use a modified version of *ifa* called ifa_auc, which calculates *ifa* based on efforts spent on inspecting the code. We use gradually increment the efforts spent by increasing the total LOC inspected and calculate ifa on each iteration to get the area under the curve (auc), here the x-axis is the percentage of effort spent on inspection and y-axis is *ifa*.

5 Results

RQ1: Can hierarchical clustering tame the complexity of bellwether-based reasoning?

Figure 1 showed that, theoretically, GENERAL is an inherently faster approach than traditional bellwether methods. To test that theoretical conclusion, we ran the rig of Figure 5 on an four core machine running at 2.3GHz with 8GB of RAM.

Figure 6 shows the mean runtimes for one run of GENERAL versus traditional bellwether. For certification purposes, this had to be repeated 20 times. In that certification run:

- The $O(N^2)$ analysis of the traditional *bellwether0* approach needed 60 days of CPU time.
- The $O(m * (N/m)^2)$ analysis of GENERAL needed 30 hours. That is, in empirical result consistent with the theoretical predictions of Figure 1, GENERAL runs much faster than traditional bellwether.
- All the other methods required another 6 hours of computation.

If we were merely seeking conclusions from one project, then we would recommend ignoring bellwethers and just use results from each project. That said, we still endorse bellwether method since we seek lessons that hold across many projects.

In summary, based on these results, we conclude that:

Theoretically and empirically, the hierarchical reasoning on GENERAL performs much faster than standard bellwether methods.

RQ2: Is this faster bellwether effective?

The speed improvements reported in **RQ1** are only useful of this faster method can also deliver adequate predictions (i.e. predictions that are not worse than those generated by other methods).

Figure 7 shows the distribution of performance score results seen in of Figure 5. These results are grouped together by the "rank" score of the left-hand-side column (and this rank was generated using the statistical methods of §4.4).

In these results, the *ifa_auc* and *precision* scores were mostly uninformative. With the exception of *ZeroR*, there was very little difference in these scores.

As to *ZeroR*, we cannot recommend that approach. While *ZeroR* makes few mistakes (low *ifa*s and low *pf*s), it scores badly on other measures (very low *recall*s and *popt(20)*.

Similarly, we cannot recommend the *global* approach. In this approach, quality predictors are learned from one data

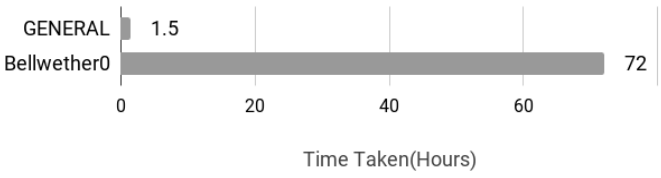

Fig. 6: Mean runtime for for one run of standard bellwether and GENERAL.

set that combines data from hundreds of projects. As seen in Figure 7 that approach generates an unacceptably large false alarm rate ($pf = 79\%$).

Another approach we would deprecate is the traditional bellwether approach. By all the measures of Figure 7, the *bellwether0* are in the middle of the pack. That is:

- That approach is in no way outstanding.
- So, compared to hierarchical bellwether, there is no evidence here of a performance benefit from using traditional bellwether.
- Given this lack luster performance, and the **RQ1** results (where traditional bellwether ran very slowly), we therefore deprecate the traditional bellwether approach.

As to GENERAL vs the local learning results of *self*, in many ways their performance in Figure 7, is indistinguishable:

- As mentioned above, measured in terms of *ifa_auc* and *precision*, there is no significant differences.
- In terms of *recall* there is no statistical difference in the rank of local learning with *self* and *GENERAL_level0* (a bellwether generated from the root of a BIRCH cluster tree) and
- In terms of *pf* (false alarms), some of the GENERAL results are ranked the same as *self* (and we will expand on this point, below).

Overall, we summarize the Figure 7 results as follows:

	rank	treatment	mean	sd
Recall	1	GENERAL_level2	38	32
	1	bellwether0	39	31
	1	ZeroR	40	49
	2	GENERAL_level1	48	33
	3	self	55	27
	3	GENERAL_level0	56	31
	4	global	78	40
Pf	1	GENERAL_level2	28	28
	1	bellwether0	28	25
	1	self	30	20
	1	GENERAL_level1	35	28
	1	ZeroR	39	49
	2	GENERAL_level0	47	31
	3	global	79	39
Precision	1	ZeroR	21	30
	2	global	35	28
	2	GENERAL_level2	39	34
	2	bellwether0	40	33
	2	GENERAL_level1	42	31
	2	GENERAL_level0	44	30
	2	self	50	30
Popt20	1	ZeroR	13	16
	2	GENERAL_level2	26	22
	2	global	26	13
	2	bellwether0	28	21
	2	GENERAL_level0	28	15
	2	GENERAL_level1	28	19
	3	self	35	19
ifa_auc	1	ZeroR	7	11
	2	global	19	16
	3	GENERAL_level2	22	14
	3	bellwether0	23	14
	3	self	23	12
	3	GENERAL_level1	23	14
	3	GENERAL_level0	25	13

Fig. 7: Statistical Results comparison. The "rank" column at left comes from the statistical analysis methods of §4.4. Note that for *pf*, and *ifa* rank=1 is the best rank while for all other performance measures, ranks $\in 3, 4$ are best.

Measured in terms of predictive performance, the effectiveness of hierarchical bellwethers is very similar to local learning (and these two methods are more effective than the other options explored here).

When two options have similar predictive performance, then other criteria can be used to select between them:

- If the goal is to quickly generate conclusions about one project, then we would recommend local learning since (as seen above), local learning is five times faster than hierarchical bellwether.
- But, as said at the start of §5, our goal is to quickly generalize across hundreds of projects.

RQ3: Does learning from too many projects have detrimental effect?

Returning now to Figure 2, this research question asks if there is such a thing as learning from too much data. What we will see is that answers to this question are much more complex than the simplistic picture of Figure 2. While for some goals it is possible to learn from too much data, there are other goals where it seems more is always better.

To answer **RQ3**, we first note that when GENERAL calls the BIRCH clustering algorithm, it generates the tree of clusters shown in Figure 8. In that tree:

- The bellwether found at level 0 of the tree (which we call *GENERAL_level0*) is learned from 627 projects.
- The bellwethers found at level 1 of the tree (which we call *GENERAL_level1*) is learned from four sub-groups of our projects.
- The bellwethers found at level 2 of the tree (which we call *GENERAL_level2*) is learned from 80 sub-sub groups of our projects.

That is, to answer **RQ3** we need only compare the predictive performance of models learned from these different levels. In that comparison, if the level_$(i+1)$ bellwethers generated better predictions that the level_(i) bellwethers, then we would conclude that it is best to learn lessons from smaller groups of projects.

Figure 7 lets us compare the performance of the bellwethers learned from different levels:

- The *ifa* and *Popt20* and *precision* results for the different levels are all ranked the same. Hence we say that, measured in terms of those measures, we cannot distinguish the performance at different levels.
- As to *recall*, the level2,1,0 bellwether results are respectively ranked worst, better, best.
- Slightly different results are offered in the *pf* false alarm results. Here, levels2,1,0 bellwether are respectively ranked best, best, worst.

That is, these results say that:

Assessed in terms of *recall*, it is better to learn bellwethers from more data rather than less. But assessed in terms of *false alarms*, while learning bellwethers from many projects is useful, it is possible to learn from too much data.

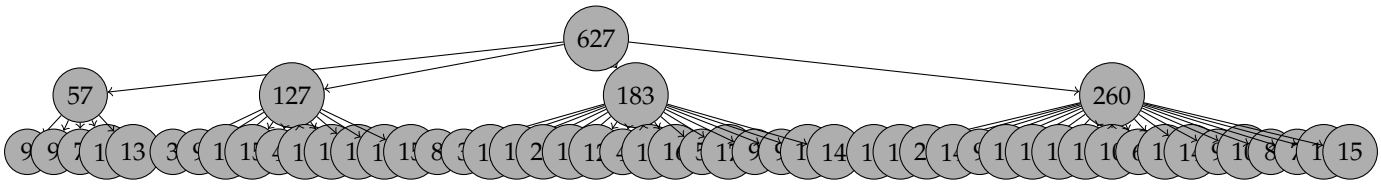

Fig. 8: Example of Hierarchical Clustering for 627 projects

Rank	Attr	coef	Odds ratio
1	avg_NPRM	2.23	9.26
2	avg_NPBM	-1.31	0.27
3	max_NPBM	-1.12	0.33
4	max_RFC	0.74	2.09
5	total_NPBM	-0.70	0.50
6	max_CBO	-0.64	0.53
7	total_ModifiedLOC	0.10	1.10
8	avg_WMC	0.07	1.07

TABLE 2: Importance of coefs on *log p* from logistic regression model of "Bellwether" shown in Fig 9. Here Odds ratio shows one increment in in respective variable increase in the log-odds of being defective.

To put that another way, the answer to "is is possible to learn from too much data", is "depends on what you value":

- For risk-adverse development of mission or safety critical systems, it is best to use all data to learn the bellwether since that finds most defects.
- On the other hand, for cost-adverse development of non-critical systems (where cutting development cost is more important than removing bugs), then there seems to be a "Goldilocks zone" where the bellwether is learned from just enough data (but not too much or too little).

RQ4: What exactly did we learn from all those projects?

Having demonstrated that we can quickly find bellwethers from hundreds of software projects, it is appropriate to ask what model was learned from all that data. This is an important question for this research sinceif we cannot show the lessons learned from our 627 projects, then all the above is wasted effort.

Table 2 shows the weights learned by logistic regression after feature selection using the bellwether project selected by *GENERAL_level0*. Note that:

- The number of features that appear in Table 2 is much smaller than the list of features shown in Table 1. That is, our bellwether is reporting that only a few features are most important for predicting software defects.
- Table 2 is sorted by the absolute value of the weights associated with those features. The last two features have near zero weights; i.e. they have negligible effect.

Apart from the negligible features, all that is left are NPRM, NPNM, RFC , and CBO. As shown in Table 1, these features all relate to class interface concepts; specifically:

- The number of public and private methods;
- The average number of methods that respond to an incoming message;
- Inter-class coupling.

Figure 9 shows what might be learned with and without the methods of this paper. Recall that the learners used in this research used feature selection and logistic regression.

- The gray bars in Figure 9 show how often the features of Table 1 were selected in the models learned from local data using *self*.
- The red bars in Figure 9 shows which features used in the local models that also appeared in the model learned from the bellwether. Note that only a very small subset of the features seen in the *self* models were found useful in the bellwether model of Table 2.

Just to say the obvious: when learning local models from very many projects, there is a wide range of features used in the model. It is far easier to definitively learn lessons from a much smaller range of features, such as those listed in Table 2. For example, based on these results we can say that for predicting defects, in this sample of features taken from 627 projects:

- Issues of inter-class interface are paramount;
- While many other issues are far less important such as file size, depth of inheritance tree, intra-method complexity, file size, revision history, and anything relating to the other features of Table 1 that are not listed in Table 2.

In summary:

The importance of many key features is not apparent at the local project level. To fully appreciate the critical impact on defects of class interface design, it is necessary to conduct a global analysis across hundreds of projects.

To say that another way,

> *Learning from many other projects can be better than learning just from your own local project data.*

6 Threats to Validity

As with any large scale empirical study, biases can affect the final results. Therefore, any conclusions made from this work must be considered with the following issues in mind:

(a) *Evaluation Bias*: In **RQ1, RQ2** and **RQ3** we have shown the performance of local model, hierarchical bellwether models, default bellwether model and compared them using statistical tests on their performance to make conclusion about presence of generality in SE datasets. While those results are true, that conclusion is scoped by the evaluation metrics we used to write this paper. It is possible that, using other measurements, there may well be a difference in these different kinds of projects. This is a matter that needs to be explored in future research.

(b) *Construct Validity*: At various places in this report, we made engineering decisions about (e.g.) choice of machine learning models, hierarchical clustering algorithm, selecting feature vectors for each project. While those decisions were

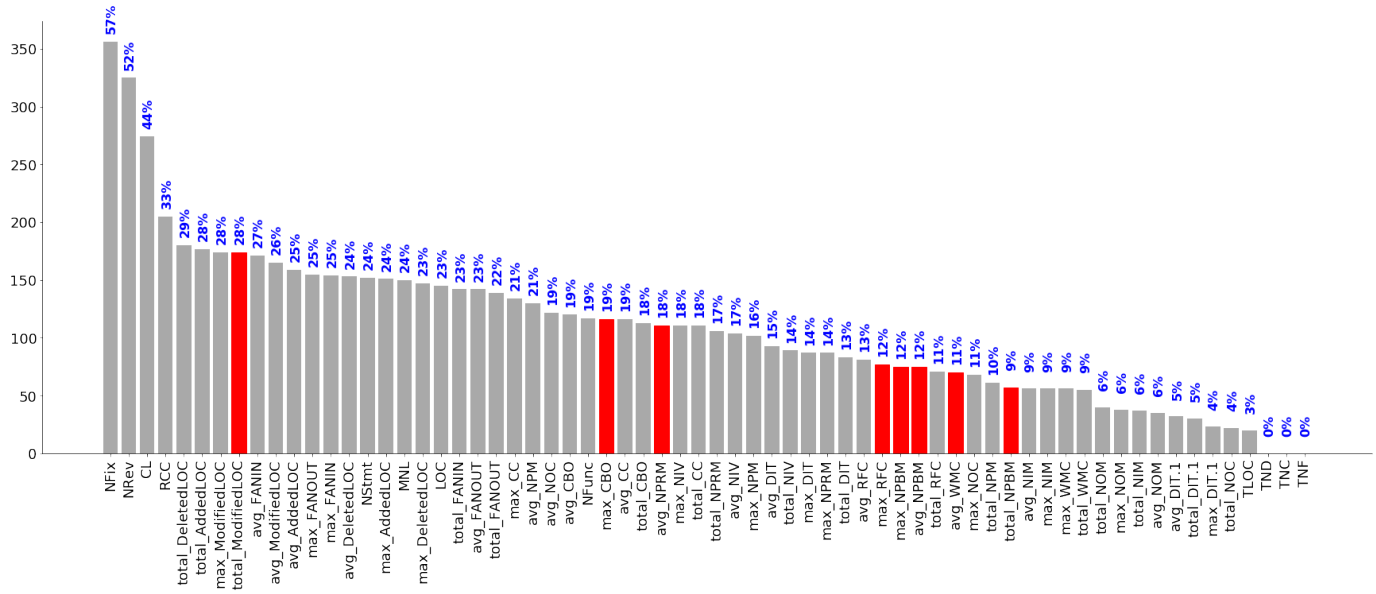

Fig. 9: Distribution of features selected using self model and "Bellwether" model.

made using advice from the literature, we acknowledge that other constructs might lead to different conclusions.

(c) *External Validity*: For this study we have relied on data collected by Zhang et al. [59] for their studies. The metrics collected for each project were done using an commercialized tool called "Understand". There is a possibility that calculation of metrics or labeling of defective vs non-defective using other tools or methods may result in different outcome. That said, the "Understand" is a commercialized tool which has detailed documentation about the metrics calculations and Zhang et al. has shared their scripts and process to convert the metrics to usable format and has described the approach to label defects.

We have relied on issues marked as a 'bug' or 'enhancement' to count bugs or enhancements, and bug or enhancement resolution times. In Github, a bug or enhancement might not be marked in an issue but in commits. There is also a possibility that the team of that project might be using different tag identifiers for bugs and enhancements. To reduce the impact of this problem, we did take precautionary step to (e.g.,) include various tag identifiers from Cabot et al. [75]. We also took precaution to remove any pull merge requests from the commits to remove any extra contributions added to the hero programmer.

(d) *Statistical Validity*: To increase the validity of our results, we applied two statistical tests, bootstrap and the a12. Hence, anytime in this paper we reported that "X was different from Y" then that report was based on both an effect size and a statistical significance test.

(e) *Sampling Bias*: Our conclusions are based on the 697 projects collected by Zhang et al. [59] for their studies. It is possible that different initial projects would have lead to different conclusions. That said, this sample is very large so we have some confidence that this sample represents an interesting range of projects. As evidence of that, we note that our sampling bias is less pronounced than other "Bellwether" studies since we explored.

7 Conclusion

In this paper, we have proposed a new transfer learning bellwether method called GENERAL. While GENERAL only reflects on a small percent of the projects, its hierarchical methods find projects which yield models whose performance is comparable to anything else we studied in this analysis. Using GENERAL, we have shown that issues of class interface design were the most critical issue within a sample of 628 projects.

One reason we recommend GENERAL is its scalabiity. Pre-existing bellwether methods are very slow. Here, we show that a new method based on hierarchical reasoning is both must faster (empirically) and can scale to much larger sets of projects (theoretically). Such scalability is vital to our research since, now that we have shown we can reach general conclusions from 100s of projects, our next goal is to analyze 1000s to 10,000s of projects.

Finally, we warn that much of the prior work on homogeneous transfer learning many have complicated the homogeneous transfer learning process with needlessly complicated methods. We strongly recommend that when building increasingly complex and expensive 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 GENERAL as a baseline method against which they can test more complex methods.

8 Acknowledgements

This work was partially funded by NSF Grant #1908762.

References

[1] R. Krishna and T. Menzies, "Bellwethers: A baseline method for transfer learning," *IEEE Transactions on Software Engineering*, 2018.

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

[3] S. Mensah, J. Keung, S. G. MacDonell, M. F. Bosu, and K. E. Bennin, "Investigating the significance of the bellwether effect to improve software effort prediction: Further empirical study," *IEEE Transactions on Reliability*, vol. 67, no. 3, pp. 1176–1198, Sept 2018.

[4] S. Mensah, J. Keung, M. F. Bosu, K. E. Bennin, and P. K. Kudjo, "A stratification and sampling model for bellwether moving window." in *SEKE*, 2017, pp. 481–486.

[5] S. Mensah, J. Keung, S. G. MacDonell, M. F. Bosu, and K. E. Bennin, "Investigating the significance of bellwether effect to improve software effort estimation," in *Software Quality, Reliability and Security (QRS), 2017 IEEE International Conference on*. IEEE, 2017, pp. 340–351.

[6] 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.

[7] 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.

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

[9] 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

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

[11] M. Kim, T. Zimmermann, R. DeLine, and A. Begel, "The emerging role of data scientists on software development teams," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 96–107. [Online]. Available: http://doi.acm.org/10.1145/2884781.2884783

[12] S.-Y. Tan and T. Chan, "Defining and conceptualizing actionable insight: a conceptual framework for decision-centric analytics," *arXiv preprint arXiv:1606.03510*, 2016.

[13] R. Sawyer, "Bi's impact on analyses and decision making depends on the development of less complex applications," in *Principles and Applications of Business Intelligence Research*. IGI Global, 2013, pp. 83–95.

[14] C. Bird, T. Menzies, and T. Zimmermann, *The Art and Science of Analyzing Software Data*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2015.

[15] K. Petersen and C. Wohlin, "Context in industrial software engineering research," in *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 401–404. [Online]. Available: http://dx.doi.org/10.1109/ESEM.2009.5316010

[16] N. Bettenburg, M. Nagappan, and A. E. Hassan, "Think locally, act globally: Improving defect and effort prediction models," in *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*. IEEE, 2012, pp. 60–69.

[17] T. Menzies, A. Butcher, D. Cok, A. Marcus, L. Layman, F. Shull, B. Turhan, and T. Zimmermann, "Local versus global lessons for defect prediction and effort estimation," *IEEE Transactions on software engineering*, vol. 39, no. 6, pp. 822–834, 2012.

[18] D. Posnett, V. Filkov, and P. Devanbu, "Ecological inference in empirical software engineering," in *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 2011, pp. 362–371.

[19] S. Herbold, A. Trautsch, and J. Grabowski, "Global vs. local models for cross-project defect prediction," *Empirical software engineering*, vol. 22, no. 4, pp. 1866–1902, 2017.

[20] a. f. C. L. Webpage, ""learning has just started" – an interview with prof. vladimir vapnik," 2014, url: http://www.learningtheory.org/learning-has-just-started-an-interview-with-prof-vladimir-vapnik/.

[21] T. Menzies, "Guest editorial for the special section on best papers from the 2011 conference on predictive models in software engineering (promise)," *Information and Software Technology*, vol. 55, no. 8, pp. 1477–1478, 2013.

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

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

[24] 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.

[25] N. E. Fenton and N. Ohlsson, "Quantitative analysis of faults and failures in a complex software system," *IEEE Transactions on Software engineering*, vol. 26, no. 8, pp. 797–814, 2000.

[26] 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.

[27] M. Nagappan, R. Robbes, Y. Kamei, É. Tanter, S. McIntosh, A. Mockus, and A. E. Hassan, "An empirical study of goto in c code from github repositories," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 2015, pp. 404–414.

[28] B. Ray, D. Posnett, V. Filkov, and P. Devanbu, "A large scale study of programming languages and code quality in github," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 155–165.

[29] D. Fucci, H. Erdogmus, B. Turhan, M. Oivo, and N. Juristo, "A dissection of the test-driven development process: does it really matter to test-first or to test-last?" *IEEE Transactions on Software Engineering*, vol. 43, no. 7, pp. 597–614, 2017.

[30] T. Menzies, W. Nichols, F. Shull, and L. Layman, "Are delayed issues harder to resolve? revisiting cost-to-fix of defects throughout the lifecycle," *Empirical Software Engineering*, vol. 22, no. 4, pp. 1903–1935, 2017.

[31] M. Rees-Jones, M. Martin, and T. Menzies, "Better predictors for issue lifetime," *Journal of Software and Systems, under review*, 2018.

[32] B. Clark and R. Madachy, *Software Cost Estimation Metrics Manual for Defense Systems, Software*. Metrics Inc., Haymarket, VA, 2015.

[33] 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.

[34] 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

[35] 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.

[36] F. Zhang, A. Mockus, I. Keivanloo, and Y. Zou, "Towards building a universal defect prediction model with rank transformed predictors," *Empirical Softw. Engg.*, vol. 21, no. 5, pp. 2107–2145, Oct. 2016. [Online]. Available: http://dx.doi.org/10.1007/s10664-015-9396-2

[37] 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," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 2015, pp. 496–507.

[38] R. Krishna and T. Menzies, "Simpler transfer learning (using" bellwethers")," *arXiv preprint arXiv:1703.06218. Under review, IEEE TSE*, 2017.

[39] R. Krishna, T. Menzies, and W. Fu, "Too much automation? the bellwether effect and its implications for transfer learning," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering - ASE 2016*. New York, New York, USA: ACM Press, 2016, pp. 122–131. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2970276.2970339

[40] M. Xenos, "Software metrics and measurements," *Encyclopedia of E-Commerce, E-Government and Mobile Commerce*, pp. 1029–1036, 01 2006.

[41] "Understand™ static code analysis tool." [Online]. Available: https://scitools.com/

[42] A. Gizas, S. Christodoulou, and T. Papatheodorou, "Comparative evaluation of javascript frameworks," in *Proceedings of the 21st International Conference on World Wide Web*. ACM, 2012, pp. 513–514.

[43] F. A. Fontana, E. Mariani, A. Mornioli, R. Sormani, and A. Tonello, "An experience report on using code smells detection tools," in *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*. IEEE, 2011, pp. 450–457.

[44] M. Orrú, E. D. Tempero, M. Marchesi, R. Tonelli, and G. Destefanis, "A curated benchmark collection of python systems for empirical studies on software engineering." in *PROMISE*, 2015, pp. 2–1.

[45] D. S. Pattison, C. A. Bird, and P. T. Devanbu, "Talk and work: a preliminary report," in *Proceedings of the 2008 international working conference on Mining software repositories*. ACM, 2008, pp. 113–116.

[46] B. A. Malloy, S. A. Linde, E. B. Duffy, and J. F. Power, "Testing c++ compilers for iso language conformance," *Dr. Dobbs Journal*, no. 337, pp. 71–78, 2002.

[47] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[48] T. Zhang, R. Ramakrishnan, and M. Livny, "Birch: an efficient data clustering method for very large databases," in *ACM Sigmod Record*, vol. 25, no. 2. ACM, 1996, pp. 103–114.

[49] 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.

[50] 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

[51] A. Agrawal and T. Menzies, ""better data" is better than "better data miners"(benefits of tuning smote for defect prediction)," *ICSE*, 2018.

[52] M. A. Hall, "Correlation-based feature selection for machine learning," 1999.

[53] F. Rahman and P. Devanbu, "How, and why, process metrics are better," in *2013 35th International Conference on Software Engineering (ICSE)*, May 2013, pp. 432–441.

[54] 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.

[55] E. Zitzler, M. Laumanns, and L. Thiele, "Spea2: Improving the strength pareto evolutionary algorithm for multiobjective optimization," in *Evolutionary Methods for Design, Optimisation, and Control*. CIMNE, Barcelona, Spain, 2002, pp. 95–100.

[56] A. S. Sayyad, T. Menzies, and H. Ammar, "On the value of user preferences in search-based software engineering: A case study in software product lines," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 492–501. [Online]. Available: http://dl.acm.org/citation.cfm?id=2486788.2486853

[57] A. S. Sayyad, J. Ingram, T. Menzies, and H. Ammar, "Scalable product line configuration: A straw to break the camel's back," in *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE'13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 465–474. [Online]. Available: https://doi.org/10.1109/ASE.2013.6693104

[58] T. Wagner, N. Beume, and B. Naujoks, "Pareto-, aggregation-, and indicator-based methods in many-objective optimization," in *Proceedings of the 4th International Conference on Evolutionary Multi-criterion Optimization*, ser. EMO'07. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 742–756. [Online]. Available: http://dl.acm.org/citation.cfm?id=1762545.1762608

[59] 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

[60] A. Mockus, "Amassing and indexing a large sample of version control systems: Towards the census of public source code history," in *2009 6th IEEE International Working Conference on Mining Software Repositories*. IEEE, 2009, pp. 11–20.

[61] T. Menzies and T. Zimmermann, "Software analytics: so what?" *IEEE Software*, no. 4, pp. 31–37, 2013.

[62] D. Zhang, S. Han, Y. Dang, J.-G. Lou, H. Zhang, and T. Xie, "Software analytics in practice," *IEEE software*, vol. 30, no. 5, pp. 30–37, 2013.

[63] P. R. Cohen, *Empirical Methods for Artificial Intelligence*. Cambridge, MA, USA: MIT Press, 1995.

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

[65] A. Arcuri and L. Briand, "A practical guide for using statistical tests to assess randomized algorithms in software engineering," in *Software Engineering (ICSE), 2011 33rd International Conference on*. IEEE, 2011, pp. 1–10.

[66] B. Efron and R. J. Tibshirani, *An introduction to the bootstrap*, ser. Mono. Stat. Appl. Probab. London: Chapman and Hall, 1994.

[67] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi, "A large-scale empirical study of just-in-time quality assurance," *IEEE Transactions on Software Engineering*, vol. 39, no. 6, pp. 757–773, 2012.

[68] Y. Yang, Y. Zhou, J. Liu, Y. Zhao, H. Lu, L. Xu, B. Xu, and H. Leung, "Effort-aware just-in-time defect prediction: simple unsupervised models could be better than supervised models," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2016, pp. 157–168.

[69] X. Yang, D. Lo, X. Xia, and J. Sun, "Tlel: A two-layer ensemble learning approach for just-in-time defect prediction," *Information and Software Technology*, vol. 87, pp. 206–220, 2017.

[70] X. Xia, D. Lo, X. Wang, and X. Yang, "Collective personalized change classification with multiobjective search," *IEEE Transactions on Reliability*, vol. 65, no. 4, pp. 1810–1829, 2016.

[71] X. Yang, D. Lo, X. Xia, Y. Zhang, and J. Sun, "Deep learning for just-in-time defect prediction," in *2015 IEEE International Conference on Software Quality, Reliability and Security*. IEEE, 2015, pp. 17–26.

[72] C. Parnin and A. Orso, "Are automated debugging techniques actually helping programmers?" in *Proceedings of the 2011 international symposium on software testing and analysis*. ACM, 2011, pp. 199–209.

[73] P. S. Kochhar, X. Xia, D. Lo, and S. Li, "Practitioners' expectations on automated fault localization," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, 2016, pp. 165–176.

[74] X. Xia, L. Bao, D. Lo, and S. Li, ""automated debugging considered harmful" considered harmful: A user study revisiting the usefulness of spectra-based fault localization techniques with professionals using real bugs from large systems," in *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2016, pp. 267–278.

[75] J. Cabot, J. L. C. Izquierdo, V. Cosentino, and B. Rolandi, "Exploring the use of labels to categorize issues in open-source software projects," in *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*. IEEE, 2015, pp. 550–554.

Suvodeep Majumder is a second year Ph.D. student in Computer Science at North Carolina State University. His research interests include using large scale data mining and application of big data and artificial intelligence methods to solve problems in software engineering. https://www.suvodeepmajumder.us.

Rahul Krishna , (Ph.D. 2019, North Carolina State University) is a post-doc at Columbia University. His research explores ways in which machine learning and AI can be used for software testing. During his Ph.D. he worked on actionable analytics for software engineering to develop algorithms that go beyond prediction to generate insights that can assist decision making. http://rkrsn.us

Tim Menzies (IEEE Fellow) is a Professor in CS at NcState His research interests include software engineering (SE), data mining, artificial intelligence, search-based SE, and open access science. http://menzies.us