Bellwether 2.0

Aayushi Agrawal
North Carolina State University
agrawa@ncsu.edu

Akshay Ravichandran
North Carolina State University
aravich6@ncsu.edu

Yudong RAO
North Carolina State University
yrao3@ncsu.edu

ABSTRACT

While most machine learning algorithms are designed to address single tasks, the development of algorithms that facilitate transfer learning is a topic of ongoing interest in the machine-learning community. Software analytics helps to build quality prediction models for software projects as these continue to be difficult to predict despite large volumes of data and many types of metrics. The central insight of this report is to discuss a methodology of selecting a suitable source i.e. the "bellwether"[1], and help improve the cost of building a model which can outperform transfer learning method known as "Bellwether"[1] which is taken as baseline for this report.

KEYWORDS

Transfer learning, Defect Prediction, Bad smells, Performance Optimization

1 INTRODUCTION

In the original work done by Krishna et. al[1] the *"bellwether effect"* is defined as follows:

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

The authors also showed that using the bellwether dataset, a baseline transfer learner called the *"Bellwether method"* can be constructed to benchmark other more complex transfer learners. Bellwethers offer a way to mitigate *conclusion instability* because conclusions about a community are stable as long as the bellwether continues to be the the best oracle.

The primary objective of this study is to reduce the cost of discovering the bellwether dataset. The original method proposed by Krishna et al[1] trains prediction models on every dataset in the community, and then uses those models to make quality predictions on all the remaining datasets. The dataset whose model returns the best prediction scores for majority of the other datasets is selected as the bellwether. Therefore, if there are n datasets and each dataset has a size r, then this results in an $\mathbf{O}((nr)^2)$ algorithm. This may be a limiting factor in using the bellwether method as a baseline for transfer learning, especially if n and r are large. This brings us to the research question targeted in this report:

RQ1: How many measurements are required to discover bellwether datasets?

In other words, we wanted to find out how we reduce the cost of discovering the bellwether, both in terms of amount of data used and time taken.

In this report, we present an alternate approach for discovering the bellwether dataset. The rest of the report is structured as follows. *Section 2* introduces transfer learning and gives a brief overview of some previous research work done on bellwethers. *Sections 3* and *4* describe the baseline criteria and software engineering domains that were targeted as a part of this study. *Section 5* introduces our proposed methodology to discover the bellwether dataset. *Sections 6* through *9* give details about our technology stack, experiments, evaluation strategies and results. Finally, *Sections 10* through *12* present our conclusions, the threats to validity of this study and future research scope.

2 LITERATURE REVIEW

Transfer learning reuses knowledge from past related tasks to ease the process of learning to perform a new task. The goal of transfer learning is to leverage previous learning and experience to more efficiently learn novel, but related, concepts, compared to what would be possible without this prior experience. The utility of transfer learning is typically measured by a reduction in the number of training examples required to achieve a target performance on a sequence of related learning problems, compared to the number required for unrelated problems: i.e., reduced sample complexity. When there is insufficient data to apply data miners to learn defect predictors, transfer learning can be used to transfer lessons learned from other source projects S to the target project T. Our project focuses on the following two methodological variants of transfer learning:

2.1 Bellwether method

The Bellwether Method[1] is a frame-work that assumes some software manager has a watching brief over N projects grouped together as community C . As part of those duties, the manager can access issue reports and static code attributes of the community. Using that data, this manager will apply the framework described in Figure 2 which comprises of three phases: *DISCOVER*, *TRANSFER* and *MONITOR*.

During the *DISCOVER* phase, the bellwether dataset is identified. In the *TRANSER* phase, a predictor is constructed using the bellwether dataset. After this, the *MONITOR* phase is entered. During this phase, the community of datasets evolve, and the predictor is tested to see if it continues to make good predictions. If its predictions start becoming poor, we go back to the *DISCOVER* phase and start over.

A benefit of this methodology is the ability to optionally replace the naive learner in the *TRANSFER* stage with any other transfer learner. The bellwether method is very simple in that it just uses the bellwether dataset to construct a prediction model (without any further complex data manipulation). Practitioners use bellwethers as an "off-the-shelf" transfer learner.

2.2 BEETLE

BEETLE[2] is a bellwether based-approach that finds the near-optimal software configuration using the knowledge in the bellwether environment. BEETLE can be separated into the following main steps: (i) *finding the bellwether environment*; and (ii) *using the bellwether environment to find the near-optimal configuration for target environments.*

For step (i), BEETLE uses a recursive algorithm to eliminate bad configurations that are not bellwethers. We have drawn inspiration from this approach for our own study. BEETLE's'objective is to find a bellwether among the source environments and use it to find the near optimal configuration for the target environments.

3 TARGET BASELINE CRITERIA

It is methodologically useful to have a baseline criteria. Such baselines let a developer quickly rule out any method that falls "below the floor". Following are the baseline requirements which were set for our methodology:

3.1 Reasonable

Methods are considered to be reasonable if they offer comparable performance to standard methods. This baseline is useful as it keeps the experimentation process to aim for a particular result/target or perform better. If the results are not compared to standard methods, the experimentation can either go in wrong direction which would be difficult to be indentified without having standard results in hand.

3.2 Cheap

The basic requirements optimization methods should comply with, is that an optimized program must have the same output and side effects as its non-optimized version. A program may be optimized so that it becomes a smaller size, consumes less memory, executes more rapidly, or performs fewer input/output operations. One doesn't just need to know how exactly optimization should be done, but also what particular part of the program should be optimized.

4 TARGET DOMAINS

This section describes the two domains in which our methodology has been tested: Defect Predication and Code Smells.

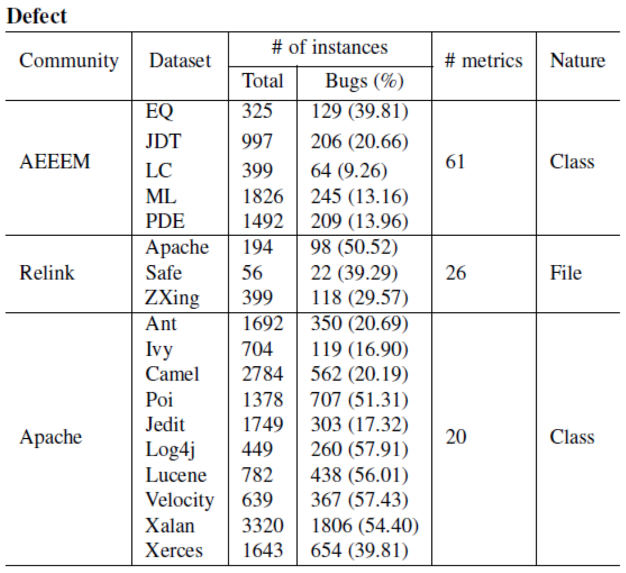
4.1 Defect Prediction

Defect prediction can precisely estimate the most defect-prone software components, and help software engineers allocate limited resources to those bits of the systems that are most likely to contain defects in testing and maintenance phases. Zimmermann et al.[3] stated that defect prediction performs well within projects as long as there is a sufficient amount of data available to train any models. However, it is not practical for new projects to collect such sufficient historical data. Thus, achieving high-accuracy defect prediction based on within project data is impossible in some cases.

We have used three defect prediction communities of datasets. The first community, *AEEEM*, was used by [11]. This was gathered by D'Amborse et al.[12]. 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 was obtained from work by Wu et al. [5] who used the *Understand tool* [11], to measure 26 metrics that calculate code complexity in order to improve the quality of defect prediction. This data is particularly interesting because the defect information in it has been manually verified and corrected. It has been widely used in defect prediction[11].

The *Apache* community of datasets was gathered by Jureczko et al. [6]. This community records the number of known defects for each class using a post-release bug tracking system. The classes are described in terms of 20 object-oriented 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. Details about defect predication dataset is listed in figure 1.



Defect

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

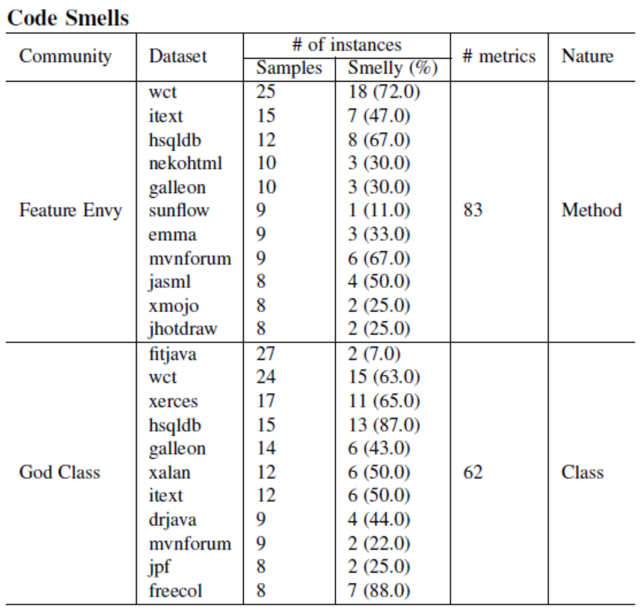
Figure 1: Defect Predication Dataset

4.2 Code Smells

Code smells are symptoms of poor design and implementation choices. Such symptoms may originate from activities performed by developers during emergencies, poor design or coding solutions, by making bad decisions, or employing so called anti-patterns. Most detection tools for code smells make use of so called *detection rules*. Usually, these detection rules are based only on the computation of a set of metrics, e.g., well-known object-oriented metrics. These metrics are then used to set some thresholds for the detection of a code smell.

Fontana et al. [11] in their study of several code smells, considered 74 systems for their analysis and their validation and they experimented with 16 different machine learning algorithms. They made available their dataset, which we have adapted for our applications in this study. The code smells repository we use contains of 22 datasets. For two different code smells: *Feature envy* and *God*

Class. The God Class code smell refers to classes that tend to centralize the intelligence of the system. This tends to be: (1) complex; (2) have too much code; (3) use large amounts of data from other classes; and (4) implement several different functionalities. Feature Envy tends to: (1) use many attributes of other classes (considering also attributes accessed through accessor methods); (2) use more attributes from other classes than from its own class; and (3) use many attributes from few different classes. The God Class is a class level smell and Feature Envy is a method level design smell. The number of samples in these datasets are particularly small. Datasets details are as follows in Figure 2.



Code Smells

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

Figure 2: Code Smells Dataset

5 OUR METHODOLOGY

To discover the bellwether dataset in different data communities, Krishna et al. [1] categorized datasets into two different types: datasets with discrete class variables, like code smells and defect prediction and datasets with continuous class variables like effort estimation. Binary classifiers are used for discrete classes while regression algortihms are applied to continuous classes. More specifically, Random Forests, an ensemble learner is used in the datasets with discrete class variables.

By definition, a bellwether dataset is the dataset with the best performance in a data community. Therefore, a proper evaluation strategy is needed. With different kinds of class variables, different evaluation strategies should be applied.

For code smells and defect predication data, "G-Score" is used for evaluating the bellwether. G-Score is measured as follows:

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

Where Pd is the probability of true positives and (1-Pf) is the probability of true negatives. The larger the G-Score, the better the performance of the dataset.

5.1 Baseline method to discover bellwether

The algorithm proposed by Krishna et al is simple. Detailed steps are as follows:

(1) For every community, select a project p_i and use it as the training set to construct a prediction model.
(2) Use every other project in the community as the test set, and predict for quality using the prediction model from step 1
(3) Record the performance scores
(4) Repeat step 1 to 3 for all projects in the community.
(5) Select the project with the best performance scores for majority of the other projects as the bellwether.

The pseudo-code is shown in Figure 3 and flowchart is shown in Figure 4.

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

  "Return data with best prediction score"
```

Figure 3: Pseudo-code for Baseline Bellwether

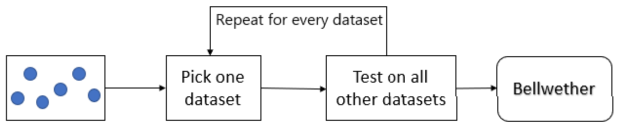


Figure 4: Flowchart for Baseline Bellwether

Suppose that in a community we have *n* projects and the average number of rows in each project is *r*. The time complexity of this baseline algorithm will be:

$$O((nr)^2) = O(n^2 r^2),$$

which is not very fast. So we propose Bellwether 2.0 as an alternate method which would be faster.

5.2 Bellwether 2.0

The inspiration for our approach comes from *recursive feature elimination* suggested by Guyon et al. [14]. The idea of recursive feature elimination is to eliminate features that do not meet the expectation recursively when we have too many features to do the training and prediction.

Instead of eliminating features, our Bellwether 2.0 approach eliminates datasets that do not perform well. And instead of using all rows of data in every dataset, we use a certain percentage of rows of data. In this way, we can find the bellwether dataset using lesser data and with less time complexity. The pseudo-code of our proposed algorithm is shown in Figure 5.

```
discoverBellwether(datasets, n_rep):
    repeat n_rep times:
        Initialize lives, x, step_size
        while lives > 0:
            for dataset_1, dataset_2 in datasets:
                data1 = sample x% of dataset_1
                data2 = sample x% of dataset_2
                predictor = train(data1)
                performance[dataset_1, dataset_2] = score(test(predictor, data2))
            rank and eliminate datasets based on performance
            if no dataset to eliminate:
                lives  = lives - 1
            x = x + step_size
        for each dataset not eliminated:
            increment its wins
    return dataset with most wins
```

Figure 5: Pseudocode for Bellwhether 2.0

The performance score we use is the *median G-score*. That is, for every dataset, the performance score is the median of all G-scores obtained when the model trained using that dataset is applied on every other dataset. The outermost loop in the pseudo-code depicts a single *run* in our algorithm. At the end of every run, all data sets that have not been eliminated are said to have *won* the run. At the end of all runs, the dataset with the most *wins* is declared as the bellwether.

We have experimented with two strategies for eliminating datasets:

(1) Throw away bottom $1/3^{rd}$ of datasets sorted with performance score in each iteration
(2) Throw away datasets with performance worse than threshold in each iteration

The *lives* variable is used to prevent our algorithm from getting stuck infinitely if no datasets can be eliminated after a few iterations. Such a situation may arise when we use the threshold strategy and set a very low threshold value. Figure 6 is a flowchart of our proposed algorithm. The motivation behind using the parameter lives is to detect convergence of the search process.

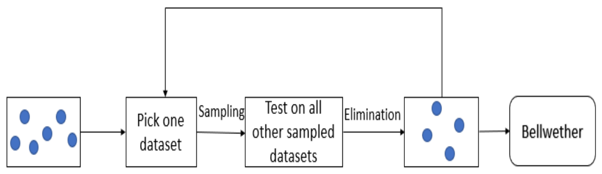


Figure 6: Flowchart of Bellwhether 2.0

As we only use a part of the datasets for training models and making predictions, and as we eliminate bad datasets in the process, the run time complexity of our algorithm would be lesser than the baseline method.

6 TECHNOLOGY STACK

We use **Python 3.7** as the programming language for the implementation of algorithms. Libraries we used include **pandas**, **scikit-learn**, **pathlib** and **numpy**. **pandas** is used to transform data into dataframe data structure for better processing. **scikit-learn** is used for training models, testing data and making predictions. **pathlib** is used for processing file paths, and **numpy** is used for array objects processing.

7 EXPERIMENTS CONDUCTED

As mentioned earlier, we used two strategies for eliminating datasets in our proposed algorithm. Given below are the various experiments we conducted.

(1) Leave one-third out strategy on the Apache dataset
(2) Leave one-third out strategy on the Relink dataset
(3) Leave one-third out strategy on the AEEEM dataset
(4) Leave one-third out strategy on the God Class dataset
(5) Threshold strategy on the Apache dataset
(6) Threshold strategy on the Relink dataset
(7) Threshold strategy on the AEEEM dataset
(8) Threshold strategy on the God Class dataset

The parameters we set for the above experiments are:

Initial sample size **x**	=	**0.2**
Sample size increment **step_size**	=	**0.05**
Number of **lives**	=	**10**
Threshold G-score value	=	**52**
Number of repetitions **n_rep**	=	**30**

We arrived at these numbers after plugging in different values, and comparing the resulting predictions with the expected results.

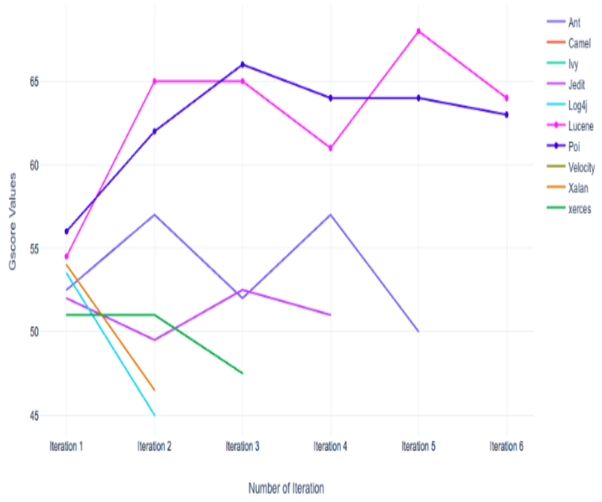


Figure 7: Graph shows the G-scores of the projects for each iteration when they are eliminated where x axis is Number of iterations and y axis is G-scores

The line graph in figure 7 represents one run of our algorithm on the Apache dataset. From the graph, it is clear how our algorithm works. At every iteration, the datasets with bad median G-scores are eliminated as indicated by their line stopping at the iteration.

8 EVALUATION STRATEGY

The strategy we used for evaluation was to compare the bellwether dataset predictions made by our proposed method, with those made by the original "Bellwether" method proposed by Krishna et al[1].

Our algorithm selects a dataset as the bellwether if it has the most number of *wins* at the end of all *runs*. As long as our algorithm predicted the same dataset(s) to be the bellwether(s), it should be a viable alternative approach to the original method.

As mentioned earlier, we used *median G-score* as our comparison metric to eliminate datasets. This is also the same metric used by Krishna et al in their proposed method.

In order to ensure that the results of our experiments, and those generated by the original method are comparable, we ran them on all the same machine - *Windows 10, Intel i5 processor, 16GB RAM*. We also used 30 repetitions to reduce random bias.

9 RESULTS

Our results are structured as answers to the following research questions:

RQ1: How many measurements are required to discover bellwether datasets?

Purpose: From the literature review, it is known that bellwethers are prevalent in software engineering datasets. So the purpose of this research question is to establish how many performance measurements need to be made in the to discover bellwether datasets with our methodology in comparison to the baseline method.

Approach: Two recursive methods based on elimination strategy were used to find the bellwether. Only a certain percentage of rows of data is used at every step of the recursion. Elimination of data can be in two ways:

- Remove bottom $1/3^{rd}$ of datasets from decreasingly sorted list according to median G-score in each iteration
- Throw away datasets with median G-score less than the threshold fixed, in each iteration

Summary: Our results are encouraging as they demonstrate that how the bellwether datasets can be discovered very fast with elimination of datasets. Since fewer datasest would take less number of comparisons and is cheaper, we can assert that discovering bellwether datasets can be very economical through our methodology.

Figures 8, 9, 10 and 11 show the bellwethers discovered by our approach. The x-axis represents the number of projects used for testing the methodology and the y-axis represents the number of wins, i.e the number of times those datasets were selected as bellwethers. We used a total of 30 runs on all experiments.

9.1 Defect Prediction

9.1.1 ***Apache Community.*** Figure 8 depicts the results for the *Apache* community of datasets. As shown, both the elimination strategies yield comparable results. Datasets *poi* and *lucene* have been selected as bellwethers the most number of times. The original method also gave the highest median G-scores to *poi* **(63.12)** and *lucene* **(62.81)**. Thus, the results we obtained are consistent with that of the original method.

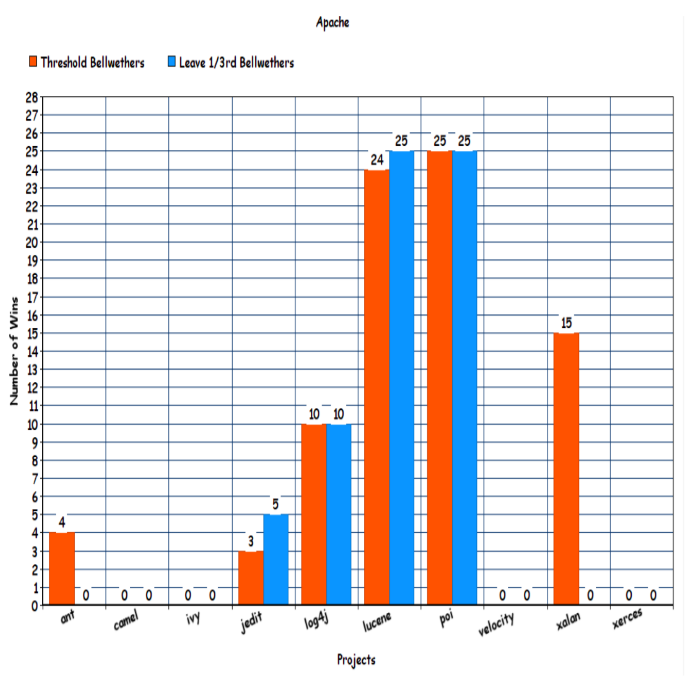


Figure 8: Number of wins for every dataset in Apache community

9.1.2 ***AEEEM Community.*** Figure 9 shows the results for the *AEEEM* community of datasets. Once again, we can see that both elimination strategies yield comparable results. Datasets *ML* and LC have been picked as bellwethers most times. The original method also resulted in the highest median G-scores being assigned to *ML* **(67.43)** and *LC* **(65.2)**, suggesting that our results are consistent with theirs.

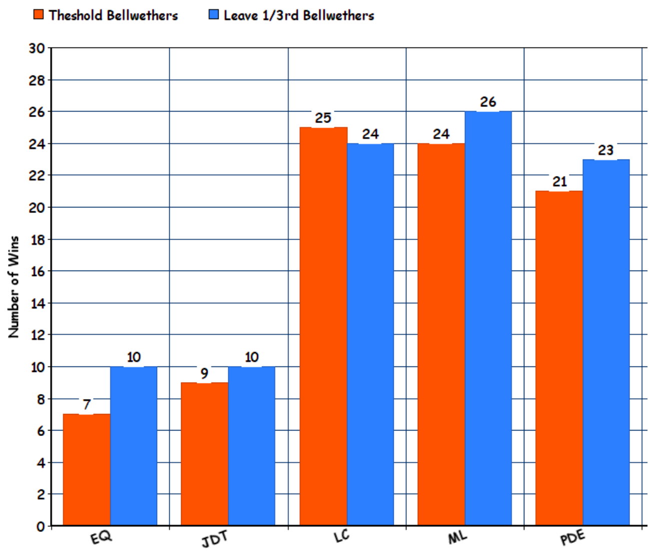


Figure 9: Number of wins for every dataset in AEEEM community

9.1.3 *RELINK Community*.

Figure 10 shows the results for the *RELINK* community of datasets. In this, we can see that the elimination strategies yield different results. The threshold strategy suggests that *Zxing* should be the bellwether, whereas the leave $1/3^{rd}$ strategy suggests that *Apache* must be the bellwether. Unfortunately, the original method suggests that *Safe* must be the bellwether dataset, with a median G-score of **63.75**. However, it should also be noted that the original method assigned the other datasets median G-scores which are very close to *Safe* - *Apache* has **61.75** and *Zxing* has **62.22**. We believe that the very nature of this community (small size and similarity in prediction scores) is the reason that our method did not yield good results.

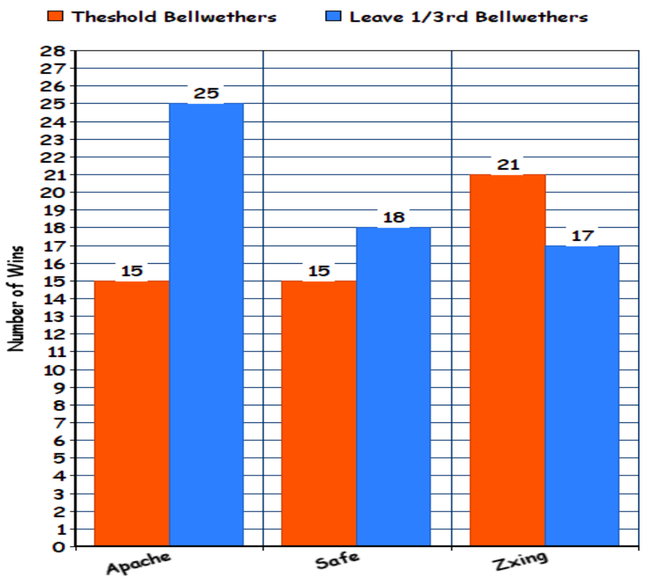


Figure 10: Number of wins for every dataset in RELINK community

	Leave 1/3rd Bellw
Apache	45.20%
RELINK	20.03%
AEEEM	45.17%

Table 1: Percentage of data used, compared to Original Bellw

	Bellw	Threshold Bellw	Leave 1/3rd Bellw
Apache	620	615	502
RELINK	13	11	10
AEEEM	110	124	64

Table 2: Execution Time in seconds

Cells highlighted, in Table 1, in gray indicate methods having less comparisons for different datasets as compared to other methods. Out of the three datasets studied here, we note that in all three cases the performance based on number of comparisons of Leave 1/3rd bellwether method was superior to all other methods for Apache dataset.

The running time of an algorithm for a specific algorithm depends on the number of operations executed. The greater the number of operations, the longer the running time of an algorithm. We have used the accuracy technique for calculating the execution time. Accuracy is the closeness of the measured value using a given method of measuring, as compared to the actual time if a perfect measurement was obtained. The time command is useful when using a UNIX-based system. Execution time measurement is activated by prefixing time to a command line. This command not only measures the time between beginning and end of the program, but it also computes the execution time used by the specific program, taking into consideration preemption, I/O, and other activities that cause the process to give-up the CPU. Leave 1/3rd Bellwethers uses less computation time in comparison to other methods which can be seen in Table 2.

9.2 Code Smells

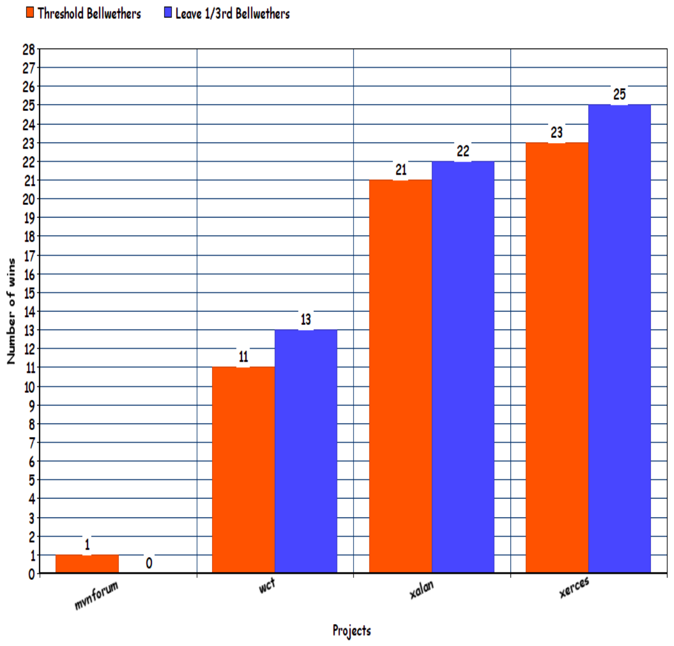


Figure 11: Number of wins for every dataset in God Class community

9.2.1 *God Class Community*.

This community belongs to code smell dataset. Figure 11 depicts the results for the *God class* community of datasets *(Note: The 7 remaining datasets which all had 0 wins are not depicted in the graph)*. As shown, both the elimination strategies yield comparable results. Datasets *xalan* and *xerces* have been selected as bellwethers the most number of times. The original method also gave the highest median G-scores to *xalan* **(88)** and *xerces* **(87)**. Thus, the results we obtained are consistent with that of the original method.

	Leave 1/3rd Bellw
God Class	40.3%

Table 3: Percentage of data used, compared to Original Bellw

	Bellw	Threshold Bellw	Leave 1/3rd Bellw
God Class	501	430	399

Table 4: Execution Time in seconds

Tables 3 and 4 show the percentage of data used and the execution time of our proposed algorithm, when compared to the baseline method. As we can see, there is a reduction in both the amount of data used and the execution time.

9.2.2 ***Feature Envy Community.*** Due to time constraints and debugging issues, we were unable to obtain good results for the feature envy community of datasets.

Result: Our experiments demonstrate that our methodology using recursion and elimination outperforms original bellwether method in terms of cost, and makes accurate predictions for most communities.

10 CONCLUSION

In this paper, we have undertaken a detailed study of Bellwethers and Beetle. According to Bellwethers,

There exists a bellwether dataset that can be used to train relatively accurate quality prediction models and these bellwethers do not require elaborate data mining methods to discover (just a for-loop around the data sets) and can be found very early in a project's life cycle.

So Bellwethers are very useful as a baseline in transfer learning. However, one issue with their use is the cost of their discovery. The results we obtained through this study suggests that the Bellwether 2.0 approach is a step in the right direction towards reducing that cost. Despite the threats to validity and generalization of our experiments, the results have given us the confidence to say that there is definitely scope for improvement of the baseline bellwether discovery method.

11 THREATS TO VALIDITY

11.1 Parameter Bias

The *threshold* G-score value, initial sample size *x*, and the sample size increment *step_size* that we selected for our experiments cannot be used for all communities of SE datasets. We fixed those values after repeated experimentation.

11.2 Learner Bias

For our experiments, the prediction model used at every iteration was a Random Forest Classifier. Naturally, there is a lot of randomness involved. We ran our experiments for 30 repetitions to reduce this. It might also be useful to employ other kinds of classifiers for different communities.

11.3 Sampling Bias

At every iteration of our proposed algorithm, we sample data before training our prediction model. When this is coupled with Random Forest Classifiers, the results we get might not be consistent. In order to alleviate this, we ran our experiments for 30 repetitions.

11.4 Performance for small communities

For communities that have very few datasets (*RELINK*), our proposed algorithm does not seem to give accurate predictions. Our recommendation is to use the original proposed algorithm for small communities.

12 FUTURE WORK

Due to time constraints all aspects of the project could not be covered. Hence following is the list which can be considered in future scope:

12.1 Evaluation for other communities

This study was conducted on defect prediction and code smell data communities. In the future, our methodology can be tested on estimation dataset communities. Detailed implementation will be different as effort estimation is a continuous class variable, but the idea will be the same. Continuous classes mean that the class variable takes on continuous values for which regression model is used.

12.2 Predicting threshold values for optimal bellwethers

In the elimination stage of our algorithm, one strategy is to set a threshold for deciding whether a dataset is good or bad. Right now we set the threshold by inspecting the performance of different threshold values and choose the threshold value that works the best. The problem with this strategy is that it can take much effort without fair rewards. Sometimes it can be no better than a random guess.

In the future, it would be the best if we can find a method to decide the threshold value automatically. Ideally, if we can propose an algorithm for predicting threshold values for optimal bellwethers, our whole Bellwether 2.0 algorithm will satisfy the baseline criteria of no magic.

12.3 Analysis of trade-off between different parameters

There exist multiple parameters in Bellwether 2.0 algorithm like *stepsize* and *lives*. At the moment these parameters are also decided by repeated experimentation. Like the threshold values we mentioned above, we also want to develop an automatic mechanism for finding the best combination of these parameters, dive deeper into the relationship between them, and analyze the trade-off between different parameters.

12.4 Combining our approach with Hoeffding Racing

Maron et al.[15] proposed a method called Hoeffding Racing in 1993. It is a technique used to find a good model for the data by quickly discarding bad models. To further reduce the time complexity of bellwether algorithm, Hoeffding Racing algorithm can be combined with recursive elimination method. As both of them individually reduce the complexity of bellwethers method, it would be an interesting study of thier combination.

REFERENCES

[1] Krishna, Rahul, and Tim Menzies. "Bellwethers: A Baseline Method For Transfer Learning." IEEE Transactions on Software Engineering (2018).

[2] Nair, Vivek, et al. "Transfer Learning with Bellwethers to find Good Configurations." arXiv preprint arXiv:1803.03900 (2018).

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

[4] 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

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

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

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

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

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

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

[11] F. Arcelli Fontana, M. V. M"antyl"a, M. Zanoni, and A. Marino, "Comparing and experimenting machine learning techniques for code smell detection," Empir. Softw. Eng., vol. 21, no. 3, pp. 1143-1191, jun 2016. [Online].

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

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

[14] Guyon, I., Weston, J., Barnhill, S., & Vapnik, V., "Gene selection for cancer classification using support vector machines", Mach. Learn., 46(1-3), 389-422, 2002.

[15] Maron, Oded. et al. "Hoeffding Races: Accelerating Model Selection Search for Classification and Function Approximation", Advances in Neural Information Processing Systems 6 (NIPS 1993)

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

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

[18] 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.

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

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

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