Using Hoeffding sampling and project elimination to make Bellwether discovery faster

Akshay Nalwaya
North Carolina State University
Raleigh, North Carolina
analway@ncsu.edu

Sanjana Kacholia
North Carolina State University
Raleigh, North Carolina
skachol@ncsu.edu

Shantanu Sharma
North Carolina State University
Raleigh, North Carolina
ssharm34@ncsu.edu

ABSTRACT

In software engineering, the project data is continuously updated and augmented. Prediction models build from these projects become increasingly varied as the number of projects increased and ultimately resulting in changing results. This problem of conclusion instability in software engineering, can be mitigated by using *"Bellwethers"*. It helps to build quality software prediction models. This problem was extensively researched in paper Bellwethers [1]. Bellwethers are used as baseline method for transfer learning and then this baseline is used for comparing future models.

So, in this paper we explore alternative methods to make the task of identification of bellwethers project in a group of projects faster for the defect prediction domain. An O(N^2) approach was presented in the Bellwethers paper and we try to explore the applicability of Hoeffdings bounds to sample the training set and experiment various combinations in the train and test sets. In addition to this, we also try to prune projects which are unlikely to be a candidate for bellwether project.

Keywords: Defect Prediction, Bellwether, Hoeffdings Bounds, Transfer Learning

1 INTRODUCTION

Bellwethers

The *bellwether effect* described in [1] states that when a community works on software, then there exists one exemplary project, called the bellwether, which can make predictions for the others.
The model built using bellwether project can serve as a baseline model for constructing different transfer learners in various domains of software engineering. Bellwethers uses the concept of transfer learning. When there is insufficient data to apply data miners to learn defect predictors, transfer learning can be used to transfer lessons learned from other source projects to new projects. Suppose, there is a new project, so we wouldn't have enough data for it, to predict the defects. This is when we can use data from other projects to make predictions for new projects.

Importance of bellwether identification

If there is insufficient data, *transfer learning* can be used by data miners and they could use lessons learned from one project and apply them on another project.

Since the probability of having a defective code and a non-defective code is not similar, the SE data is often imbalanced and difficult to get. In such cases, Bellwethers method presents a simple solution - instead of exploring all available data, find one data set that may offer stable conclusion over longer period of time. Bellwethers [**?**] shows existence of such projects in SE data sets and strives to find them. It is true that bellwethers, with such simplicity, are always better than other complex algorithms used for similar applications. At the same time, it has been shown that bellwethers are capable of outperforming some of the more complicated algorithms.

Defect Prediction

The programs written for software engineering have flaws. Every piece of code has to be tested before addition to the main repository. The software can crash or malfunction, if the code has defects. These part of codes that can lead to crashing of the application or malfunctioning is called defects. Now, a possible solution to not making defects is by testing the code before hand. However, testing introduces a lot of additional expenses and sometimes the time taken for testing is similar to the time taken for developing that piece of code. This regards testing as a possible solution but not a optimum solution. Exponential costs exhaust the resources available and testing could be made less expensive, if only a part i.e., a critical section has to be tested.

There are various approaches to find such critical section in the code. For example, the defect predictions can be made using static code attributes. So, once the miner learns which section of code is most likely to have a defect, it can make predictions. Another method, although more time consuming is manually going through the code, which is more accurate. So, these methods can be used to guide which sections of code might have defects. The defect prediction models prioritize code review as well as testing resources, hence making them easier to use. Additionally, defect predictors often find the location of 70% (or more) of the defects in code. One method of predicting these defects, could be using Bellwethers. Once we have the bellwethers data set, we can use it to predict defects for other projects. This will be extremely helpful not only when we there is no relevant amount of data of the new project but also, using just one data set and trained model we can predict defects for various projects.

Current Solution

Current methodology of bellwether identification is an N^2 algorithm that tends to evaluate each project against every other project. Though this is a simple process we believe that this approach needs investigation. In the current solution, data sets from a community is taken. Suppose we have 10 data sets, each data set is used for training a model. A model trained on a data set is tested on the remaining 9 data set. Relevant evaluation metric values are found. The model that makes the best predictions for most of the data set is found. The data set that was used to train this model is identified as Bellwethers. Evaluation metric generally used is G-score. We aim to find a method that identifies bellwether project in much lesser time without making the identification process much complex. Also, it

should be scalable when number of projects is much larger that the ones explored in the current approach. We can also sample each data set to reduce the training or testing data set. This should lead to significant reduction in the data set size, hence, reducing the time required to find the bellwethers.

Proposed Solution

In this paper we do various experiments with sampling the training and testing data sets. In addition to just sampling the data sets, we also compare this sampling method with idea of eliminating a project altogether without having to test it on all the projects. First and foremost we explored sampling methods available and zeroed on Hoeffdings bound method. We also explored project elimination strategy.

Challenges faced

The goal of the project was to reduce the time taken to suggest bellwether, with that in mind we explored various algorithms and techniques to achieve results. The challenge we faced was either focus on reducing number of projects (N) or sample the data sets with various techniques. In the end we decided to explore other in isolation and see which approach among the two is better.

Overview of Results

The results of our experiments are documented in the results section. We see that project elimination is gives us the test results with a scale of 6 with respect to the baseline. The sampling methods though gave good results in terms of data set required to reach the baseline scores (we needed 8%-10% data) but gain in terms of time was negligible. However, if we sample the data only once, instead of sampling for each data set, we reduced the time by 4 times. Overall, we were able to successfully find Bellwethers with a method taking less time than the baseline strategy.

2 RESEARCH QUESTIONS

Our study concentrated on finding answers to the following three research questions.

RQ1: Can we predict which data set is bellwether?

Motivation:

In many experiments that we perform we investigate if we can find the bellwether for given set of projects. Bellwether projects can be used to serve as "sanity checker"[1].

Approach:

For identifying a Bellwether we take the Jureczko repository for defect predictions, we built a Random Forest Classifier using the approach mentioned in the [1] paper and compare these results with different sampling and elimination techniques as discussed in the subsequent sections.

Results:

Based on the results obtained, we were able to find the Bellwether project and it was the same in all the different implementations. This also shows that there exists a project which can act as predictor for all the other projects.

RQ2: Can we reduce the time to find bellwether by reducing the size of data ?

Motivation:

A problem worth investigating is that if we can reduce the time to identify the bellwether project using sampling techniques and/or project elimination. The current approach takes N^2 time for identification which will be too large as the number of projects increase. So, if we can find a faster approach then it would reduce the runtime significantly.

Approach:

We try different algorithms that explore the possibility of not requiring to test current project against every other project or to reduce the data instances required for training/testing. For this, we have applied Hoeffding bounds and project elimination.

Results:

We found that Hoeffding bounds reduced the bellwether identification time significantly by reducing the amount of training data needed. Then we also implemented project elimination which also reduced the runtime by a fraction of *N*.

RQ3: Does sampling data based on Hoeffding sampling outperforms idea of project elimination?

Motivation:

Techniques like Hoeffding sampling reduce the data set for each of the projects but do not eliminated then completely. This surely does reduce the run-time but if we can devise some algorithm that helps to eliminate a project then that can also reduce the time complexity significantly.

Approach:

We studied the results obtained by baseline model and established a threshold value for G-score that each project should satisfy to be considered for being a bellwether. This cut-off helped to eliminated certain projects that performed poorly for other projects.

Results:

The time taken by this project elimination approach was even lower than sampling techniques. The final G-score values were almost similar to those obtained by baseline and sampling techniques showing that without any loss in identification accuracy, we are able to reduce the run-time significantly.

3 METHODOLOGY

In this section we describe various steps done and try to address why those were required. We cover data set description, generation of baseline and evaluation criteria.

3.1 Data Set Description

Since we limited out work to finding Bellwethers for defect measures we relied on data set gathered by Jureczko [3]. The data set contains defect measures from several Apache projects. The data set comprises of data from 10 different projects. This data set contains records the number of known defects for each class using a post-release bug tracking system. The classes are described in terms of 20 metrics. Each data set in the Apache community has several versions. We merged data set across different version to create bigger data set.

Project	Rows
ant	1692
camel	2784
ivy	704
jedit	1749
log4j	449
lucene	782
poi	1378
velocity	689
xalan	3320
xerces	1643

Table 1: Data Points for each project in Jureczko

Following are the details of the data set for each project. All projects had 20 features. Since the class variable was continuous we did some pre-processing to convert to binary. This was done because our objective was to find whether for a given instance, will it have a bug or not and not the number of bugs identified. So we mapped all the instances having at least 1 bug as positive class while those not having any bug as negative class.

3.2 Baseline

There are many binary classifiers to predict defects, the Bellwethers [1] cites studies on defect prediction and follows the use of Random Forests for defect prediction over several other methods. For sake of simplicity and effective comparison (if required) we decided to use the Random Forest Classifier.

Baseline calculation is a straight forward task - for each project in the community *train* the model, and *test* it against every other project and compute G-Scores for each of the test iteration. The project with the best median value of the G-Score is declared the Bellwether.

Algorithm 1 Baseline

```
start time = current time
for project in projects
    read data
    train random classifier
    for testproj in projects
        if testproj not equal to project
        make predictions on testproj
        calculate gscore
        append g to a table
end time = current time
runtime = end time - start time
```

3.3 Evaluation

The data set under consideration has binary class labels, with the records belonging to either positive or negative class. The instances of projects having defects (one or more) are assigned positive class while those without defects are assigned negative class implying no defect was found in that instance.

There are various metrics that can be derived from the confusion matrix obtained as a result of testing the classifier on each of the projects. Different measures of model evaluation are summarized below:

Standard Measures of Evaluation

Accuracy: It is the percentage of instances of the data set that have been classified correctly by the model. It emphasizes on correct classification of both positive and negative classes equally. The mathematical formula for accuracy is,

$$accuracy = \frac{true\,positive + true\,negative}{total number of instances}$$

Precision: It talks about how precise your model is, meaning it shows what fraction of instances that are predicted positive, are actually positive. So a model with low precision would imply that either a there was a large number of false positives in the model or the number of true positives was very low.

$$precision = \frac{true\,positive}{true\,positive + false\,positive}$$

Recall: It calculates how many of the actual positive instances have been correctly captured by the model (true positives). It is also denoted by *pd* or the *probability of detection.*

$$recall = \frac{true\,positive}{true\,positive + false\,negative}$$

False Alarm: As the name suggests, this metric gives the percentage of negative instances that were erroneously predicted as positive instances. It is also denoted by *pf.*

$$pf = \frac{false\,positive}{false\,positive + true\,negative}$$

Each of the metrics we discussed above are used for model evaluation depending on the application and the type of data. For instance, if one aims to increase the recall for a model, then it might also increase the false alarm (pf) of the model. Similarly, there is a kind of inverse relationship present in between precision and recall. If one tries to increase the precision of a model, then the recall might have to be compromised with.

Class Imbalance in classification problems is a scenario where classes are not represented equally. Most classification data sets do not have an equal representation of the classes and often such class imbalance needs a careful handling. Slight variations in the class distributions can be ignored but a significant variation needs the be taken into account. There are several ways of handling the class imbalance and most common among them are

- Collect more data
- Change performance metric
- Re-sampling data set

Why Not Accuracy

In the cases discussed above, accuracy can often be misleading. At times it may be desirable to select a model with a lower accuracy because better predictive power on the problem. For example, in a problem where there is a large class imbalance, a model can just predict the value of the majority class for all predictions and achieve a high classification accuracy, the problem is that this model is not

useful in the problem domain.

G-Score
We use the **G-Score** [4] [5] as a metric for evaluating performance of classifier in this case of class imbalance. It combines recall (pd) and false alarm rate (pf). The Bellwethers [1] cites studies which suggest that such a measure is justifiably better than other measures when the data set has imbalanced distribution in terms of classes. Hence we are using G-Score in this paper as well.
G-Score is measured as follows:

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

It is clear from this formula that models having higher G-Scores are better.

4 EXPERIMENTS

4.1 Hoeffdings Bounds

Let us say that we have N points with which to test a given model. If we were to test a model on all of them, then we would have an average error that we will call E_{true}. However, if we only tested the model on ten points, then we only have an estimate of the true average error. We call the average after only n points (n<N) E_{est} since it is an estimate of E_{true}. The more points we test on (the bigger n gets), the closer our estimate gets to the true error. How close is E_{est} to E_{true} after n points? Hoeffding bound lets us answer that question when the n points are picked with an identical independent distribution from the set of N original test points. In this case, we can say that the probability of E_{est} being more thanaway from E_{true}

$$Pr(|E_{true} - E_{est}| > \epsilon) < 2e^{-2n\epsilon^2/B^2}$$

where B bounds the greatest possible error that a model can make [2]. This bound does not make any assumptions other than the independence of the samples. The hoffding racing algorithm was used for the test set originally.
Hoeffding racing algorithm has three major steps. It start with each model known as learning box. Learning box can be complex or time consuming, we consider just the input and output of the model. The error is one of the most important metric. Testing set is taken one data point at a time and data points are added. At each addition of data point, error is calculated. So, firstly, compute the leave-one-out cross validation error at each point of the test data set. The average error rate is updated at each iteration, as data points are added to the test data set. Using Hoeffding bound, we calculate how close is the average error rate to the original error rate. We can eliminate those learning boxes whose best possible error is still greater than the worst error of the best learning box. The point at which hoeffding bound is hit, we can break the loop. Racing algorithms basically, learn from the good models and eliminate the bad ones.
Since, racing never performs more queries than brute force, we formulated a strategy to use this in order to find Bellwethers. Also, the overhead involved in this process is negligible.

4.2 Sampling

S No	Sampled Training Data	Different % of Training Data for each Testing Data
0	No	No
1	Yes	Yes
2	Yes	No

4.2.1 ***Approach 1.*** In this approach, our motive was to decrease the time required to find Bellwethers by decreasing the training data set. Each data set was taken, and the samples were added to the training data set. We start with 5% of the data set and at each iteration, 1% of the data is added. Model is trained on this data at each iteration and tested on the other projects. The point at which we are 95% confident that our estimate of the running g score is within the epsilon of baseline g score, is noted. The loop breaks for that test project and runs for all the remaining test projects. Once, we hit hoeffding bound for each test project, a similar exercise is run for the remaining projects. This helped us reduce the training data considerably. However, the fraction of data being sampled for each test project was different. Hence, the time taken to run the process was not reduced. The fact that sampling the training data set for each test set was increasing the time of execution led us to devise another approach.

Algorithm 2 Approach 1 Algorithm

```
start time = current time
for project in projects
    read data
    for testproj in projects
        if testproj not equal to project
            sample data upto hoeffding bound
            train random classifier
            make predictions on testproj
            calculate gscore
            append g to a table
    end time = current time
runtime = end time - start time
```

Algorithm 3 Approach 2 Algorithm

```
start time = current time
for project in projects
    read data
    frac = get max % of training data
    sample data for frac
    train random classifier
    for testproj in projects
        if testproj not equal to project
            make predictions on testproj
            calculate gscore
            append g to a table
    end time = current time
runtime = end time - start time
```

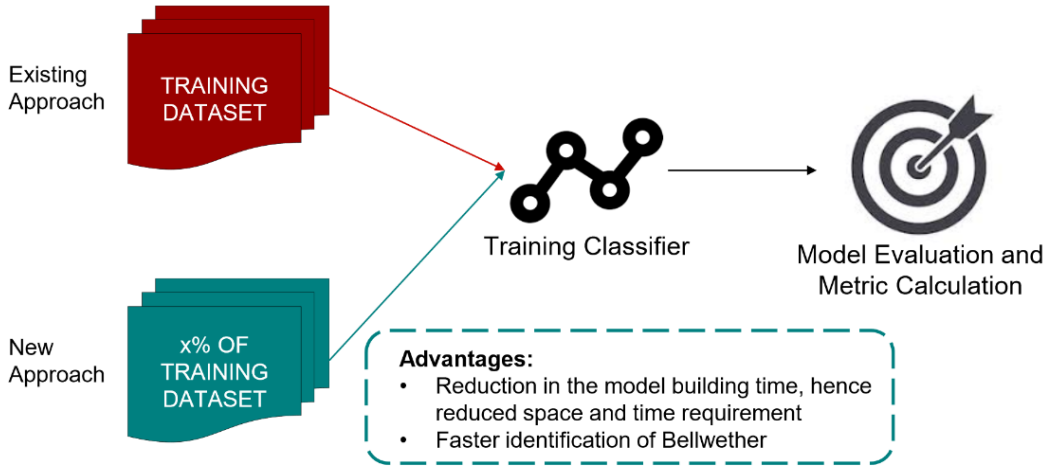

Figure 1: High level process flow of current process and comparing it with our Hoeffding Bounds process

Algorithm 4 Project Elimination Algorithm

```
for each project do
    load X_train, y_train
    train classifier
    set threshold g-score
    for all other projects
        load X_test,y_test
        predict
        compute g-score
        If eliminate-count > 2:
            break
    If g-score < threshold and
            num projects tested >=3:
            g-score=0
    append results, g-score
return results
```

4.2.2 ***Approach 2.*** In order to eliminate the time taken for sampling the training data differently for each data set, we took the maximum of the percentage of data required by any test project. This did not just decrease the training data set but impacted the time taken to find bellwethers. The time taken to find Bellwethers was reduced by 4 times.

In this approach, the sampling of data was done just once, and with the reduced training data set, predictions are made the steps are as following

Firstly, each data set was taken, and the samples were added to the training data set. We start with 5% of the data set and at each iteration, 1% of the data is added. Model is trained on this data at each iteration and tested on the other projects. The point at which we are 95% confident that our estimate of the running g score is within the epsilon of baseline g score, is noted. The loop breaks for that test project and runs for all the remaining test projects. Once, we hit hoeffding bound for each test project, a similar exercise is run for the remaining projects. Then, for each project we consider the maximum amount of data required by any other test project. The training data is sampled according to the results from previous steps.

Bellwether prediction is done using the G-scores calculated according to the predictions made for each test data set. This helped us reduce the training data considerably as well as reduce the time by 4 times. G-scores calculated is almost similar to the values calculated through baseline method.

4.3 Project Elimination

In the above experiments, we focused on using Hoeffding Bounds for efficiently sampling the data sets for all the projects. Depending on the approach, we experimented with sampling only the training data, or testing data, and then sampling both: training and testing data. The end goal was to try and reduce the number of records to be used for training and testing purposes. This doesn't reduce the number of projects but it reduces the constant term in run-time complexity analysis, i.e., in cN^2 the term c is reduced.

In this experiment, our objective was to explore if we can eliminate some projects and hence try to reduce the value of N in the complexity expression. We studied the baseline results for all the projects and found that some projects which consistently had low values for G-score for prediction on other projects continued this trend for all the projects. This accounted for a large amount of time being spent on training and testing of projects that were not the candidates for bellwether. So, such projects should be eliminated without spending time in using such projects for testing for all other projects.

Based on the baseline G-score values we decided a threshold value for the G-score. This threshold value should be satisfied for all the projects in order to be considered as a candidate bellwether project. The central value of G-score distribution for all the projects was chosen as the threshold value. Mean, median and mode are the most commonly used measures of central tendency. We chose

median as the representative value of the central tendency for bellwethers since mean is prone to be influenced by the presence of outliers in the data (G-score of projects in this case). One project with very low G-score could bring down the threshold G-score and lead to increased processing time. On the other hand, median is not affected by some outliers in data, and hence is a better measure for this case.

Once this threshold was decided, we started with training a Random Forest Classifier on each of the projects. This trained random forest classifier is used for testing on all the other projects in a sequential manner. The G-score values for each iteration is recorded for each of the trained classifier model. The current project is eliminated if the following two conditions are satisfied:

- ***Condition 1:*** Project is tested on at least $1/3^{rd}$ of the projects
- ***Condition 2:*** Mean of G-score value is less than the specified threshold value

All the projects satisfying these conditions are tested for the other projects until they violate these conditions or it has been tested on all available projects. The projects violating these conditions are pruned and also removed from the list of candidate bellwether projects. This serves as the early stopping rule to avoid testing on all projects and efficiently reducing the number of projects used for testing.

The G-score values are then aggregated for projects that have not been eliminated. Median value of G-score is taken and reported as the G-score for each project and the project with highest mean value of G-score for testing on other projects is termed as the bellwether project for the given set of projects.

5 RESULTS

RQ1: Can we predict which data set is bellwether?
We can predict bellwethers using baseline method, and this method generates the G-score values given in Figure 3. It should be noted that the value of G-score for all the approaches discussed in this paper are very similar. The bellwether data set can be predicted using the G-score and poi, proves to be the data set with the best G-score. However, *xalan* and *lucene* have median G-score comparable to that of *poi*. We implemented three different approaches to predict Bellwethers and for all the three approaches, *poi* has the highest G-score throughout.

RQ2: Can we reduce the time to find bellwether by reducing the size of data ? For research question 2, we found the point when hoeffding bound is hit. Instead of the whole training data set, only a percentage of data set sample can be taken. The results are shown in the table below. All results were calculated after running for 30 iterations. The data for training can be reduced a lot but sampling the data again and again consumes a lot of time. Hence, we took the maximum amount of training data required to hit the hoeffding bound for any test project and sampled training data acoordingly. The sampling of data for each training set is just done once, considerably reducing run time. The results show that the run time has reduced more than 4 times. Tables below show the percentage of data used for training, their updated G-score and the runtimes for each approach.

S No	Experiment Name	Time Taken(secs)
1	Baseline	199.32
2	Hoeffding 1	211.57
3	Hoeffding 2	44.16
4	Project Elimination	34.63

Table 2: Comparing run-times of all the approaches for finding Bellwethers.

RQ3: Does sampling data based on Hoeffding bounds outperforms idea of project elimination?

All results were calculated after running for 30 iterations. Based on the values of average run times in the figure 5, it is evident that eliminating projects definitely reduces the time required for finding bellwether project when compared with the baseline run time obtained by round robin training of all projects against every other project. In addition to beating the run time of baseline method, project elimination proved to be even better than sampling data using Hoeffding bounds.

6 FUTURE WORK

Based on the results achieved in this project work, there are some more algorithms, and methods which can be explored to get even much better results. Some of the questions for which more exploration can be done are discussed below.

- *Explore alternative sampling methods*
  Can we implement more sampling methods instead of just Hoeffdings bounds and compare them? There is another sampling method named Bayesian Races which assume that data is normally distributed. This approach might also prove to be useful for reducing the time complexity associated with bellwether discovery.
- *Racing between number of projects and sampling*
  Is it possible to determine which approach needs to be taken sampling or elimination while just looking at dataset. Are there any threshold values which help us determine we should either go for project elimination or sampling.
- *Exploring project elimination with sampling*
  In this project we focused on evaluating sampling algorithms and eliminating projects in isolation. It would be interesting to see if we are able to achieve even better results by combining the idea of project elimination and data sampling.
- *Exploring with different repositories*
  For the scope of this project we limited ourselves to the use of Jureczko repository, but we can explore more data set repositories and use them to explore our research questions. It would help us to see how well does our work apply to other repositories.
- *Adding Parallelism to code*
  Another important aspect of computation is parallelism built in languages and platforms. Our current work does not explore this but adding parallelism to the code should lead to lower latency.

Trained on	Tested on									
	ant	camel	ivy	jedit	log4j	lucene	poi	velocity	xalan	xerces
ant	-	5	6	7	8	9	10	11	12	13
camel	5	-	6	7	8	9	10	11	12	13
ivy	5	6	-	7	8	9	10	11.5	12	12.5
jedit	5	6	7	-	8	9	10	8	12.5	12
log4j	5	6	7	8	-	9	10	11	12	13
lucene	5	6	7	8	9	-	10	11	12	13
poi	5	6	7	8	9	10	-	11	12	13
velocity	5	6	7	9	7	10	11	-	12	13
xalan	5	6	7	8	8.5	10	11	12	-	13
xerces	5	7.5	7	8	8	9.5	11	12	13	-

Figure 2: Used Hoeffding bounds to find the percentage of data required for training on a single project using random sampling and testing on all the other projects.

Project	Baseline	Hoeffding Bound	Sampled Training Data
ant	0.18	0.18	0.19
camel	0.24	0.25	0.24
ivy	0.09	0.12	0.12
jedit	0.04	0.03	0.04
log4j	0.34	0.34	0.32
lucene	0.52	0.52	0.51
poi	0.61	0.62	0.61
velocity	0.49	0.49	0.49
xalan	0.56	0.58	0.57
xerces	0.42	0.43	0.43

Figure 3: Median G Scores for Bellwether Prediction

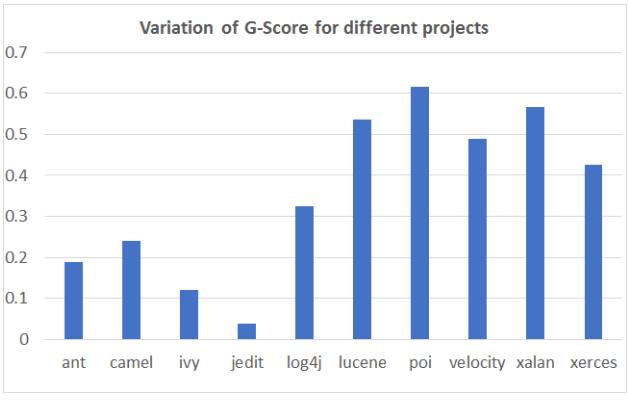

Figure 4: The median g-score for *poi* was highest amongst all the projects used for defect prediction.

Projects	% data for Training
ant	13
camel	13
ivy	12.5
jedit	12.5
log4j	13
lucene	13
poi	13
velocity	13
xalan	13
xerces	13

Figure 5: Approach 2 - Percentage of data required for Training for Bellwether Prediction

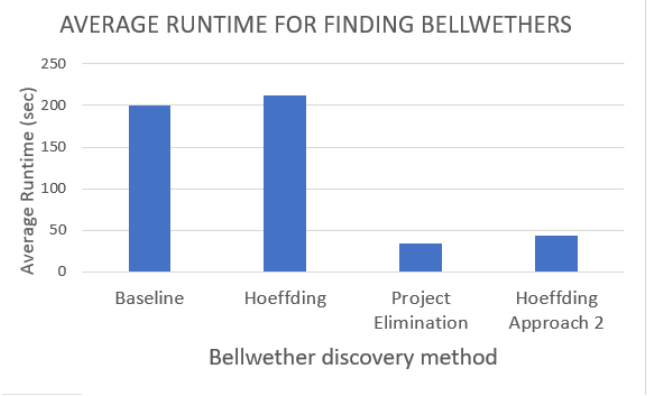

Figure 6: Average Runtime values for different approaches of finding Bellwether project

Projects Eliminated				
ant	camel	ivy	jedit	log4j

7 CONCLUSION

In this paper, we have performed a thorough study of Bellwether discovery process and their importance in various software engineering domains. Our results show that we can make the process of identification of bellwether project in a repository much faster than the current round robin $O(N^2)$ approach.

Projects	G score
lucene	0.54
poi	0.61
velocity	0.49
xalan	0.57
xerces	0.07

Figure 7: G-scores using Project elimination to predict Bellwethers

We have showed that sampling the training and/or testing data sets helps in reducing the amount of time spent for bellwether discovery process. Data sampling not only helps to reduce the run time of the code but also shows that efficiently selecting data from a project can help minimize redundancies and ensure that the classifier is trained only on the optimum percentage of training data. This comes at almost negligible loss of the model performance measured using G-score.

We also proved that sampling is not always the best method for reducing the time for bellwether discovery. We can prune projects to avoid the time spent on training projects which are highly unlikely to be prospective bellwethers. This helped to focus only on the projects that are capable of being the representative project for the whole repository which essentially reduced the runtime by a some fraction of N, with N being the number of projects in the repository.

8 ACKNOWLEDGEMENTS

We would like to thank Professor Timothy Menzies for giving this opportunity to work and explore Bellwethers phenomenon in SE and data mining for Software Engineering in general. We would also like to express our gratitude to Rahul Krishna for helping us at various times when we were stuck and providing us guidance when needed.

REFERENCES

[1] Tim Menzies and Rahul Krishna *Bellwethers: A Baseline Method For Transfer Learning* . IEEE Transactions on Software Engineering, April 2018
[2] Oden Maron and Andrew W. Moore *The Racing Algorithm: Model Selection for Lazy Learners.* Artificial Intelligence Review, February 1997, Volume 11
[3] M. Jureczko and L. Madeyski, *Towards identifying software project clusters with regard to defect prediction.* Proc. 6th Int. Conf. Predict. Model. Softw. Eng. PROMISE 2010
[4] Tim 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, Sepember 2007
[5] M. Kubat, S. Matwin et al., *Addressing the curse of imbalanced training sets: one-sided selection.* ICML, vol. 97. Nashville, USA, 1997