General models for defect prediction?

[progress report]

Lonnie Bowe
West Virginia University
lbowe@csee.wvu.edu

Aglika Gyaourova
West Virginia University
agyaouro@csee.wvu.edu

Zack Hutzell
West Virginia University
zhutzell@csee.wvu.edu

ABSTRACT

Keywords

defect predictors, static attributes, general model

1. INTRODUCTION

The software industry is like any other business. It must find a balance between the resources it invests in a project and the return gained from that project. Unfortunately, with the size and complexity of software systems, defects are often present. Of course, these defects are undesireable and should be fixed. However, this can cost a business more resources than they wish to spend. According to Brooks [3] testing a software system makes up fifty-percent of the cost of developing the system. Because of this, throughness in testing can be sacrificed to save resources. This can have a negative effect on the quality of a system. This can also have hidden costs. After studying economic elements of software Boehm [2] [1] concluded that the longer a defect remains in the system, the repair cost will rise at an exponential rate. In the face of this information, businesses are faced with a dilema in deciding how to spend their resources.

This dilema has motivated researchers to study automated methods to predict software defects. If these automated systems are successful, it could save businesses from spending precious resources and hopefully increase the quality of software at the same time. This field of study is not without its own conflicts, however. There is a great debate about the effectiveness and methodologies used by these systems. Because of Boehm's results, we know that it is better to detect defects as early as possible in a system to save costs. A major problem with this is that there may not be enough information in the early stages of a project to effectively use an automated defect predictor. To solve this problem, there has been investigation into whether one software system can predict another. This is a widely debated topic. The golden prize in this field of study would be to discover and prove a general model for defect prediction that will work on all software systems. This paper seeks to find evidence to prove or disprove if that general model exists.

The paper is structured as follows. The first section provides a survey of previous work and what we can learn from them. The second section explores our methodology and the data sets we are using in our experiment. The third section discusses the results we have encountered so far. The fourth section provides an outline for future work.

2. PREVIOUS WORK

Before diving into the material specifically related to automated defect prediction, we would like to present a study that will influence our research methodology. With advancements in the study of machine learning techniques Holte [6] observed that the complexity of the learners was increasing. To see if this was a good trend, he developed a simple rule learner, called 1R. 1R makes its classification decisions based on a single attribute. He compared this very simple learner to an advanced decision tree learner, C4 which was developed by Quinlan [11]. Holte found that 1R produced classification accuracy that was only a few percentage points less than the ones produced by C4. From these results, Holte proposed a "Simplicity First" methodology for researching machine learning systems. This methodology includes such things as using a small sample space with a simple hypothesis. Once that hypothesis is tested, increase the sample space and hypothesis slowly to fix any problems. This is the methodology that we will use in our investigation.

Many researchers have investigated the characteristics that make up software and what is important in making decisions about software. McCabe [8] investigated the complexity of a piece of software by seeing how many linear paths exist in the program. If a software has more branches to take, McCabe argues that it is more complex. His investigation resulted in the Cyclomatic Complexity. Boehm [1] extensively investigated the economic factors that influence software design. His model, COCOMO is used worldwide. Halstead [5] developed a complexity model based on a very simple system that makes calculations based on the number of operators and operands in a piece of software. These software metrics have been of value to the software engineering field. The Halstead and McCabe results have been used in defect prediction research, though there is some debate about their effectiveness. Both use static code attributes for their calculations.

Ostrand et al [10] also use static code attributes in developing an automated defect predictor. Their first predictor focused on predicting faults from one release of a system to another. Instead of focusing on attributes that exist in the software, they picked attributes that describe the software process. They hand-picked attributes such as the age of the file and whether the file has been changed in a release or not. They did include one attribute that exists in the software, lines of code. They then used a Negative Bionomial Regression Model to analyze the data from their software. Their system achieved a high percentage of accuracy. They then tested these attributes on a different software system and had similar results. This suggests that their model could work across different systems and perhaps become a general model. However, in their analysis they noticed that the lines of code closely modelled the results from their predictor. In a comparison test, they found that the lines of code achieved high accuracy near the begining of the software's development but was soon outperformed by the more complicated system. They concluded that lines of code could be used as a rough estimate in the begining of a project or if the trade-off between accuracy and complexity was beneficial to the investigator.

While some researchers were having success with these static code attributes, Menzies et al [9] decided to investigate the value of these static code attributes. Instead of hand-picking attributes, the researchers fed attribute information from several software projects into three different learners. These learners were: 1R, J48 and Naive Bayes. They then investigated which attributes that the learners picked were the "best" ones to pick. This study produced two interesting results. The first is that the Naive Bayes learner seemed to outperform the more complex J48 learner. However, 1R was outperformed by both Naive Bayes and J48. The researchers do note that the performance for Naive Bayes can be improved with preprocessing methods. The second results comes from the analysis of which attributes were the "best". If certain attributes were more important to defect prediction, then the results should have been skewed towards them. Instead the learners picked different attributes for the different data sets. The authors conclude that there were no clear winners. However, they develop two methods, which are quoted below.

- Do Not Seek "best" subsets of static code attributes.
- Rather, seek learning methods that can combine multiple partial defect indicators, like the statistical methods of naive Bayes.

We will also be taking these methods into consideration in our investigation. Another point worth mentioning is that Menzies et al used publicly available datasets, which allow other researchers to replicate, verify and expand on their work. These data sets are available at (Promise Website). We will be using some of the same data sets. To this end, Lessmann et al [7] decided to investigate the effect that a learner has on the results. Using the same data from the Menzies paper, Lessmann et al, investigated twenty-two different classifiers. They let the classifiers run over the data and then did a statistical analysis of the results. They found that the simple learners performed well, but not necessarily better than the more complicated ones. Out of the simple learners, Naive Bayes performed the best, supporting the conclusions from the Menzies study [9]. The conclusion that the authors come to is that the choice of classifier is not as important as previously thought in other research. From this, they state that other attributes of a classifier are more important than the accuracy and how it classifies the information. These attributes are "computational efficiency, ease of use and, especially comprehensibility".

While published before the Lessmann study [7], the Menzies study [9] provides a good argument for Naive Bayes using similar criteria, as it is efficient and easy to comprehend. This argument and the results from the Lessmann paper will be taken into heavy consideration during our investigation. Following and incorporating some of the the previous studies, Turhan et al, [12] return to the problem of using prediction information from one company to predict for another company. Using static code attributes and the Naive Bayes classifier, the authors use the previously mentioned public data sets with some new datasets from a Turkish white goods company. Since the publicly available data is from NASA projects, we can safely say that comparing it against the Turkish data is truely cross-company and cross-domain in nature. In their first experiment, they found that data from within a company is better than data from another company. To help alleviate this problem, they introduced a nearest-neighbor filtering method. This method increased the accuracy, but not enough to make data from another company equal to data from within a company.

So the question for Turhan et al [12], became how much data from within company does one need to build a good predictor? The surprising result from this investigation was that there is negligible change in the accuracy of Naive Bayes after one-hundred training examples. They then estimate that this much data can be generated in approximately three person months. At the end of the paper, the authors conclude that companies that want to use automated defect prediction should start out using data from another company while rapidly building their own training data. Once there is enough training data, then the company should switch to it instead.

Zimmermann et al [13], however, disagree that cross-company data is a viable method of defect prediction. They selected several large commerical programs from Microsoft and several large Open Source projects. While picking some of the same attributes as the other studies, they also include attributes such as the number of edits made and the number of developers that worked on the software. They then use a logistic regression model with random sampling to classify their data. Their results were not encouraging. Out of 622 combinations investigated, only 3.4 percent were accurate. They surmise that if one wants to use methods involving static attributes, one must carefully select the training data. This is a contradiction to the Turhan study [12]. The authors then develop a system for predicting what projects would be a good predictor for others. They give each project forty external attributes such as domain, type of user interface, intended audience, etc. They then build decision trees based on the factors and their effect on accuracy. The highest accuracy they achieved was 84 percent, though the average is much lower. They conclude that using data from another company to predict software defects is a very complicated task and that great care has to be taken when choosing the training data.

3. METHODS

Our goal is to find a general model for defect predictors. For this purpose, the experiments are divided into within

```
naive-bayes-classify(instance train)
  k = 1
  m = 2
  classification = null
  classification_score = inf
  for each class in classes(train)
    prior_probability=(k+get_class_count(train, class))/(length(train)+(k*(length(classes(train)))))
    sum = 0
    for each feature in row
      if(!class_column(feature) && !unknown_feature(feature))
        sum = sum + (class_frequency(feature, class, train)
                + (m * prior_probability)) / (length(get_class_count(train class)) + m)
    if(sum > classification_score)
      classification = class
  return classification
```

Figure 1: Naive Bayes classifier on categorical data.

company (WC) and cross-company (CC). For the WC experiments, 90% of one data set is used for training the classifier and the rest 10% for evaluating the classifier performance. This process is repeated 10 times, by randomly selecting the instances to be used for evaluation.

In all data sets, the distribution of the instances from the two classes, i.e. the defect and the non-defect, is imbalanced. The defect class is represented by a smaller number of instances ranging from 0.4% to 32% of all instances in different data set [12]. During training the number of instances of the non-defect class was reduced to the number of instances in the defect class by random deletion.

The Bayes classifier finds a decision that minimizes the Bayes error among classes. This is the theoretically optimal classifier when the attribute values of each class classes have a normal probability distribution. The Bayes classifier operates using the posterior probabilities of the classes as defined by the Bayes' theorem. In a two class problem having only a single attribute, the classifier decision is the attribute value at which the posterior probability of one of the classes becomes larger than the posterior probability of the other class. The naive Bayes classifier assumes independent attributes. Practice in software engineering has shown that this classifier performs well even when its assumptions are not met.

Applying naive Bayes on categorical (discrete) data is fast and easy. In this case the naive Bayes operates on the conditional frequencies of the categorical values 1. Since discretization of continuous attributed does not have significant effect on the performance [4], we mapped the attribute values into a discrete range before running the learner. A discretization method that accounts for the distribution of the values is the equal frequency discretization. This method splits a list of numbers into a predefined number of intervals, such that every interval contains the equal count of list elements, figure 2.

Another discretization method, `Nbins`, creates intervals whose value ranges have equal width, figure 3.

4. EXPERIMENTS

We performed experiments on cross-project, within company data using the sets `ar3`, `ar4`, and `ar5`. The learner was naive Bayes and data was preprocessed using equal frequency discretization. Deletion of rows was used to obtain

```
equal-frequency-discretizer(dataset, bins)
  for each column in dataset
    //sort the values in the column
    sort(dataset, column)
    for each value in the current column
      //change each value to its discretized replacement
      value=min(n-1, floor((value/(length(column)-1))*bins))
```

Figure 2: Equal frequency discretization.

```
equal-width-discretizer(set, bins)
    minval=min(set)
    maxval=max(set)
    for each value in set
      //find the interval in which the value belongs
      //merge the last two intervals
      value=min(nbins,
                floor((value-minval)/(maxval-minval)*nbins))
```

Figure 3: Equal width discretization (`Nbins`).

an equal number of instances in the two classes. Thus, the prior probabilities of the classes are the same. Equal priors may be an appropriate assumption and this is one way to achieve it for data having a large sample bias. Instances from the over-represented class were deleted at random before training the classifier.

4.1 Performance evaluation

To evaluate the performance of the learned model we used the accuracy, recall, and precision measures, defined by:

$$\begin{gathered} pd = recall = D/(B+D)\ , \\ pf = C/(A+C), \\ balance = 1 - \frac{\sqrt{(0-pf)^2 + (1-pd)^2}}{\sqrt{2}}\ , \end{gathered} \tag{1}$$

where A, B, C, and D are the true negatives, false negatives, false positives, and true positives respectively. The accuracy gives the portion of correct decision over all classes. The

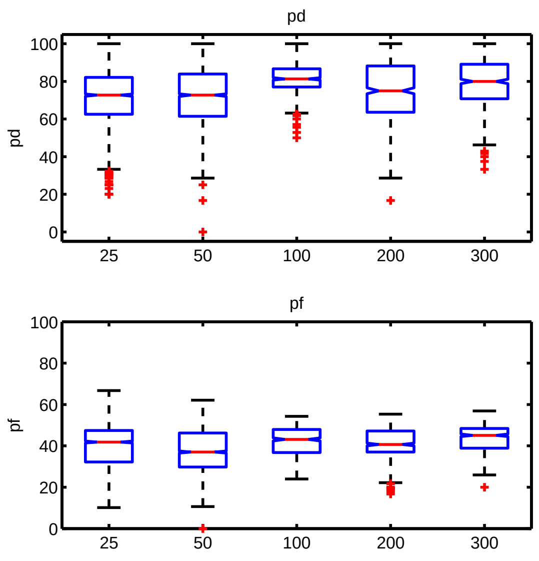

Figure 4: Quartile plots for the *pd* and *pf* on the NASA sets with micro-sampling.

precision is the portion of true positives from all instances predicted as a given class and the recall is the portion of true positives from all instances belonging to a given class. All these measures have a range between 0 and 1, with 1 being the best score.

4.2 Results

Micro-sampling does not change performance significantly, Figure 4 and Figure 5.

Something must be wrong in our handling of the performance metrics. The results seem biased towards the larger class (non-defect), Figure 6 and Figure 7.

5. CONCLUSIONS

Our goals for the next project are:

- Implement attribute selection.
- Implementing knn to be used in the cross-company experiments.

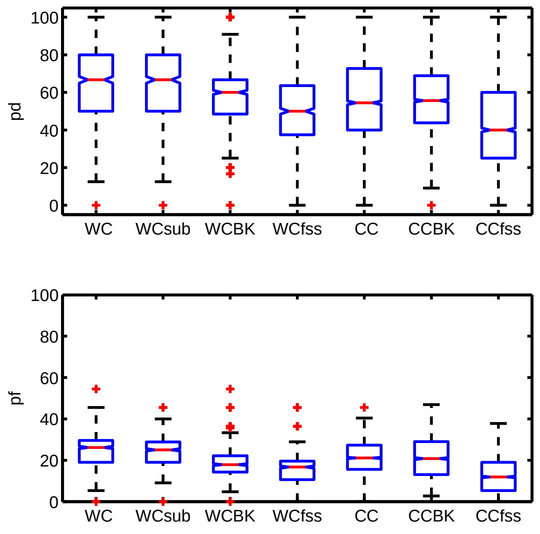

Figure 6: Quartile plots for the *balance* on the NASA sets.

- Run cross-company experiments.
- Report results using cross-validation.
- Try different subsampling, e.g. the Burak or the super-Burak filters.
- Try different classifiers, such as WHICH and Regression methods.

6. REFERENCES

[1] B. Boehm. *Software Engineering Economics.* Prentice-Hall, 1981.

[2] B. Boehm. Software engineering economics. *IEEE Transactions on Software Engineering*, 14(10):1462–1477, 1988.

[3] F. Brooks. *Mythical Man Month (anniversary edition).* Addison-Wesley Longman Publishing Co., Inc., 1995.

[4] J. Dougherty, R. Kohavi, and M. Sahami. Supervised and unsupervised discretization of continuous features.

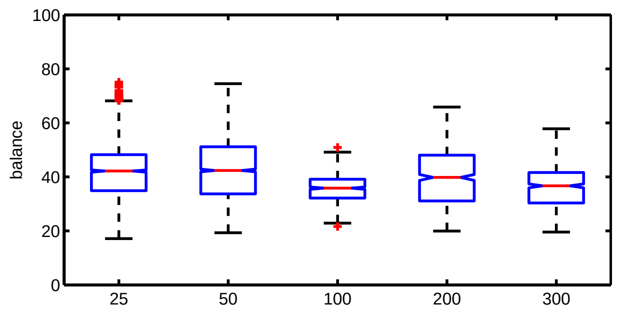

Figure 5: Quartile plots for the *balance* on the NASA sets with micro-sampling.

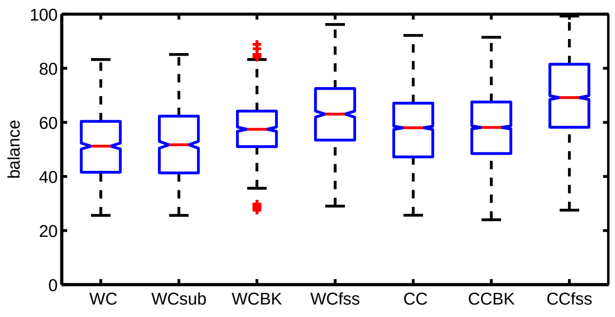

Figure 7: Quartile plots for the *balance* on the NASA sets.

In *International Conference on Machine Learning*, pages 194–202, 1995.

[5] M. Halstead. *Elements of Software Science*. Elsevier, 1977.

[6] R. Holte. Very simple classification rules perform well on most commonly used datasets. *Machine Learning*, 11:63, 1993.

[7] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch. Benchmarking classification models for software defect prediction: A proposed framework and novel findings. *IEEE Transactions on Software Engineering*, 34(4), 2008.

[8] T. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, 1976.

[9] T. Menzies, J. Greenwald, and A. Frank. Data mining static code attributes to learn defect predictors. *IEEE Transactions on Software Engineering*, January 2007.

[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*, pages 86–96, 2004.

[11] J. R. Quinlan. C4.5: Programs for machine learning, 1993.

[12] 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 Softw. Engg.*, 14(5):540–578, 2009.

[13] 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 *ESEC/FSE '09: Proc. of the 7th joint European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 91–100, New York, NY, USA, 2009. ACM.