Predicting software defects from cross company data

CS573, Fall 2009

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

Prediction of software defects from models build from other company data has shown contradictory results. We investigate how simple data pre-processing helps in improving the performance of such models. Using a Naive Bayes classifier we show that the prediction performance of models built from other company is not significantly different from the performance of models built from data collected within the company. Our experiments indicate that the use of loss function in the Bayes classifier allows to achieve classification that is more suitable to the goals of defect prediction. An attempt to evaluate the generality of the models shows that a small number of training examples from randomly chosen projects is enough to achieve reliable prediction on other projects. Finally, we show that the size of the training set can be reduced down to 50 examples from each class without significant change in the ability to predict defects.

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 undesirable and should be fixed. However, this can cost a business more resources than they wish to spend. According to Brooks [4] testing a software system makes up fifty-percent of the cost of developing the system. Because of this, thoroughness in testing is sometimes sacrificed to save resources. However, this can have a negative effect on the quality of a system, and also have hidden costs. After studying economic elements of software, Boehm [3, 2] concluded that the repair cost for a defect rises at an exponential rate the longer it remains in the system. In the face of this information, businesses are faced with a dilemma in deciding how to spend their resources.

This dilemma 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 data from one software system can be used to predict defects for another. This is a widely debated topic. The ultimate goal of 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.

In the next section we provide a brief survey of previous work. The methods used to pre-process the data and Naive Bayes classifier is discussed in section 4. Section 6 describes our experiments and evaluation methodology. Conclusions and suggested directions for future work are presented in section 7.

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. Holte [7] observed that with the advancements in machine learning, the complexity of 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 [12]. 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 of

software and what is important in making decisions about software. McCabe [9] investigated the complexity of a piece of software by seeing how many linear paths exist in the program. If a program has more branches to take, McCabe argues that it is more complex. His investigation resulted in the concept of Cyclomatic Complexity. Boehm [2] extensively investigated the economic factors that influence software design; his model, COCOMO is used worldwide. Halstead [6] 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. [11] also used 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 Binomial 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 modeled the results from their predictor. In a comparison test, they found that the lines of code achieved high accuracy near the beginning 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 beginning 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. [10] 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 tried to determine 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 result 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.

We will also be taking these methods into consideration in our investigation. Another point worth mentioning is that Menzies et al. [10] used publicly available data sets, which allow other researchers to replicate, verify and expand on their work. These data sets are available at (Promise Website [1]). To this end, Lessmann et al. [8] 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 [10]. The authors concluded that the choice of classifier is not as important as previously thought. 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 [8], the Menzies study [10] provides a good argument for Naive Bayes using similar criteria, as it is efficient and easy to comprehend. Following and incorporating some of the the previous studies, Turhan et al. [13] returned 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 used the previously mentioned public data sets with some new data sets 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 truly 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 for the training set. This method increased the accuracy, but not enough to make data from another company as useful as data from within a company.

So the question for Turhan et al. [13], became how much data from within company does one need to build a good predictor? The surprising result from this investigation was that after about 100 training examples, the accuracy of a Naive Bayes classifier did not improve with additional training data. 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 within-company training data, then the company should switch to it instead.

However, Zimmermann et al. [14] do not agree that cross-company data is a viable method of defect prediction. They selected several large commercial 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 [13]. The authors then developed a system for predicting what projects would be a good predictor for others. They gave each project forty external attributes such as domain, type of user interface, intended audience, etc. They then built

decision trees based on these factors and their effect on accuracy. The highest accuracy they achieved was 84 percent, though the average was much lower. They conclude that using data from another company to predict software defects is a very complicated task and training data must be chosen carefully.

3. NAIVE BAYES CLASSIFIER

The Bayes classifier uses the posterior probabilities of the classes to find a decision that minimizes the Bayes error among them. It is the theoretically optimal classifier when the attributes have normal probability distribution. The naive Bayes classifier assumes independent attributes.

In all experiments we use the naive Bayes classifier. 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, 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. When working with discrete attributes, the naive Bayes classifier operates on the class-conditional frequencies of the categorical values.

4. DATA PRE-PROCESSING

The software defects data often contains few unusually large values in some attributes. To equalize the variance of the data we apply logarithmic transformation.

Another way to equalize the variance is by converting the numeric data to categorical data. Estimation of the posterior probabilities of categorical data is fast and easy because there is finite small number of possible values. Discretization of numerical attributes has a further benefit to correct small errors made during data collection. When the number of bins is several times smaller than the number of examples, small inaccuracies made during data recording are voided.

In equal frequency discretization Figure 1, an ordered list of numbers is split into a predefined number of bins, such that every bin contains an equal count of elements from the list. This discretization method takes into account the non-uniformity of the data by putting together values that occur rare and preserving values that occur often.

In the publicly available data sets of software defects, the distribution of the examples from the two classes, i.e. the defect and the non-defect, is highly 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 [13]. Too many training examples from one class may lead to overestimation of the posterior probability for this class. We reduce the number of non-defect examples by random sub-sampling until the two classes have the same number of examples. From classification perspective, this sub-sampling approach results in equal priors for the two classes. Such model may be desirable because although the defect modules occur less often, the importance of the detecting them is higher. One problem with our approach to delete rows in the data is that when the number of defect examples is very small the size of data can be reduced tremendously and thus become insufficient for learning. Another way to assign different importance to the two classes is by introducing a loss parameter in the Bayes decision rule [5].

```
equal-frequency-discretizer(data, nbins)
  sorted = sort(data)
  N = number of elements in "data"
  bin = 1
  discrete[1]=bin
  for i = 2 to N
    if i > N * (bin / nbins)
      //do not put the same value in multiple bins
      if sorted[i-1] != sorted[i]
        bin++
    discrete[i] = bin
```

Figure 1: Equal frequency discretization. Initially the elements are sorted and each bin is assigned the same number of elements. Next, if two successive bins contain elements of the same numeric value, then these elements are moved into the earlier bin. A number of optimizations of this rule are possible.

dataset	project description
kc1	storage management for ground data
pc1	flight software for earth orbiting satellite
kc2	storage management for ground data
cm1	spacecraft instrument
ar4	embedded controller for dishwasher
ar3	embedded controller for washing machine
ar5	embedded controller for refrigerator

Table 1: Datasets used in the experiments and the description of the underlying software project.

When learning from a dataset collected at a different company or for a different project it may be reasonable to filter the training set in order to make it relevant to the test data. This should be done in an unsupervised manner so that the training set is kept independent of the test set. For this purpose, we used the nearest neighbor filtering method described in [13]. The 10 nearest neighbors from the train set are selected for every instance in the test set, disregarding its class belonging. Duplicate entries in the resulted train set are deleted. We refer to this filtering method by the name Burak filter.

5. DATASETS

We used 4 datasets collected for projects developed by different NASA contractors and at different sites and 3 datasets from a Turkish manufacturer of white goods, Softlab. These datasets are available at the repository created for the Promise Conference [1]. The NASA projects were written in C++, the Softlab projects in C. Other specifics of the datasets are shown in Table 1 and Table 2. There were 17 numeric attributes, which were present in all 7 datasets, Table 3.

6. EXPERIMENTS AND RESULTS

To test whether cross-company data can be used for predicting software defects, we ran two types of experiments:

dataset	total number of examples	number defective	percent defective
kc1	2,109	326	15.46
pc1	1,109	77	6.94
kc2	522	107	20.50
cm1	498	49	9.84
ar4	107	20	18.69
ar3	63	8	12.70
ar5	36	8	22.22

Table 2: Total number of examples and frequency of the defective modules.

#	feature	#	feature
1	total lines of code (loc)	10	cyclomatic complexity
2	comment loc	11	design complexity
3	code & comment loc	12	Halstead volume
4	executable loc	13	Halstead difficulty
5	unique operands	14	Halstead effort
6	unique operators	15	Halstead error
7	total operands	16	Halstead time
8	total operators	17	Halstead level
9	branch count		(aka program length)

Table 3: Static code attributes present in the datasets.

- Within company (WC) experiments, in which we build models and evaluate their performance using only data collected for a single project.
- Cross-company (CC) experiments, in which we build the models using data from one project and test its performance using data from a different project and from a different company.

Then we compare the performance of these experiments to evaluate the merit of cross company prediction.

All experiments are performed using 4-fold cross validation repeated 25 times. Thus, 75% of the data set is used for training the classifier and the rest 25% for evaluating the classifier performance. This results in $4 * 25 * 7 = 700$ test runs for each WC data pre-processing scenario and in $(4 * 25) * (7 * 6) = 4,200$ test runs for each CC scenario. When comparing WC versus CC results the WC we reduce the output rows from the CC experiments to 700 (same as the output rows of the WC experiments).

6.1 Performance Evaluation

The performance of classification is measured using probability of detection (*pd*), probability of false alarm (*pf*) and the *g*-measure:

$$pd = recall = D/(B + D) \ ,$$
$$pf = C/(A + C),$$
$$g = \frac{2 * pd * (1 - pf)}{pd + (1 - pf)} \ ,$$

where A, B, C, and D are the true negatives, false negatives, false positives, and true positives respectively. Probability of detection (or *recall*) is the portion of true positives from all instances belonging to the given class. Probability of false alarm is the portion of instances that are wrongly classified as belonging to the target class from all instances that do not belong to the target class. These measures have a range between 0 and 1, with 1 being the best score for *pd* and *g* and 0 being the best score for *pf*.

symbol	pre-processing technique
d (or D)	equal frequency discretization
log	logarithmic transform
BK	Burak filter (see section 4)
EP	Naive Bayes using equal priors
SS	row sub-sampling
raw	no pre-processing

Table 4: Naming convention for the experiments.

To check the effect of variable in the experiments, we used the Kruskal-Wallis nonparametric Anova test followed by a significance test to evaluate the differences in the distributions. We also used the Mann-Whitney *U* test to rank the experiments according to the chosen performance measure. These tests we performed in Matlab using the functions "ranksum", "kruskalwallis", and "multcompare". We use the standard errors of the ranks to decide whether two experiments are significantly different, which is exact only if sample sizes of the two experiments are the same. Whenever needed, we make the sample sizes the same by randomly deleting samples (output rows) from the larger experiment.

6.2 Results

The naming convention used in the experiments is shown in Table 4.

6.2.1 Within-company training

The basic pre-processing methods as described in section 4 were first tested using WC training. From all parameters, discretization led to the best performance as measured by *g*, Figure 2. Also, the three experiments that use discretization do not have significantly different performance - the intervals according to standard errors of their samples do not overlap. The logarithmic transform led to the worst performance. It greatly increased the possibility of false alarm compared to any other experiment, Figure 3. This results might be due to either the values of some attributes (many small values) or due to the nature of their distributions and the use of the naive Bayes classifier (the assumption of normal distribution might be grossly violated). When using the Burak filter, both the *pd* and the *pf* rates vary a lot depending on the pre-processing technique. Perhaps, a larger number of the nearest neighbors should be selected. The experiments when using cross-company training showed the same trends as the experiments when using within-company training. However, by using CC training we can perform a larger number of tests and therefore, the standard errors can be estimated more accurately. For this reason, further experimental scenarios were performed using CC training.

6.2.2 Cross-company training

The WC results show that the challenge of defect prediction lies in the low detection rate. Two possible ways to improve the detection rate are: (i) using datasets with a larger percentage of defect modules; and (ii) introducing a loss (or cost) parameter in the Bayes decision rule. Creating new databases is a problem on its own and therefore we experimented with the second approach. For ease of im-

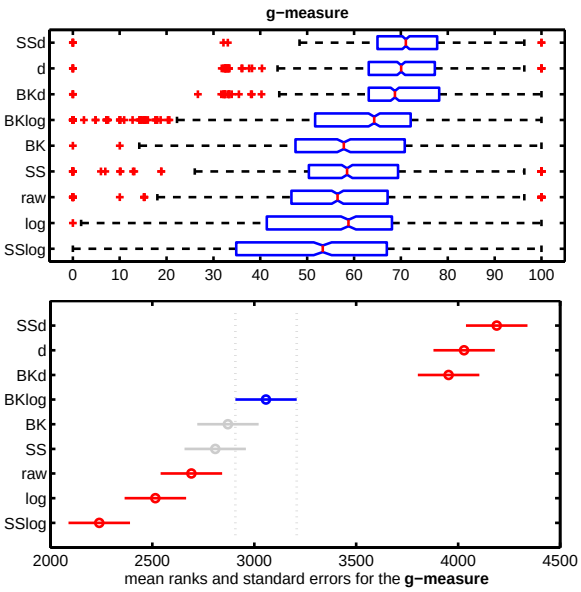

Figure 2: WC training - performance according to *g*. The experiments are displayed in descending order of their ranks computed using the Mann-Whitney test.

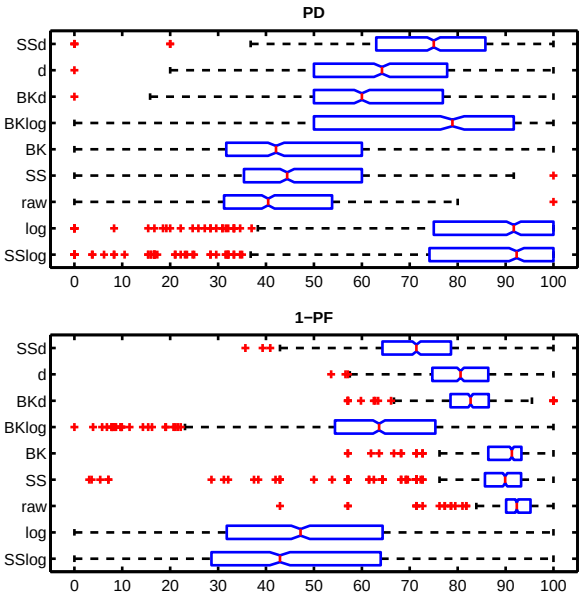

Figure 3: WC training - distributions of *pd* and 1-*pf*. Experiments are sorted by their ranks using *g*.

plementation and to avoid estimating a proper loss value, we performed a straightforward experiment by setting the prior probabilities of the two classes to the same value. This experiment was ranked high according to the *g* measure, Figure 4, but not as high as the sub-sampling method (which by design also uses equal priors). The reason for that was its high false alarm rate, Figure 5. However, the detection rate of the experiment using equal priors was better, with its 75th percentile reaching 100%, compared to 86% for the sub-sampling method.

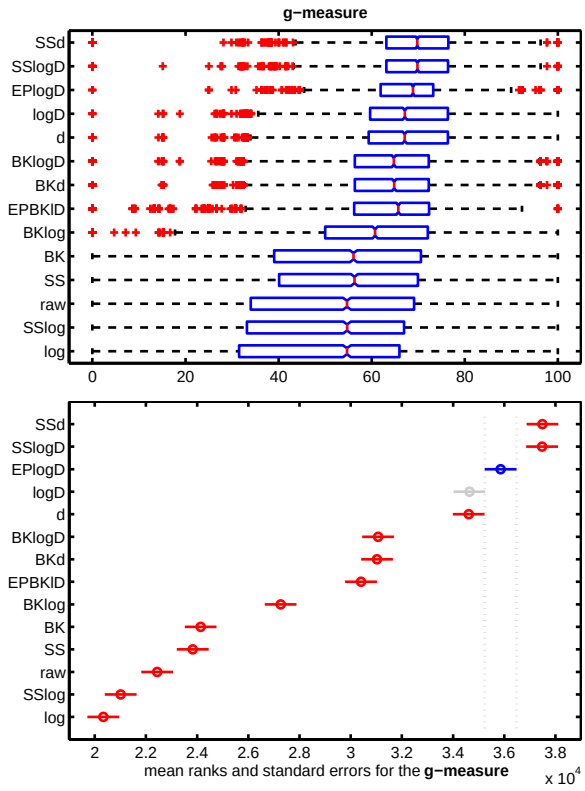

Figure 4: CC training - performance according to *g*. The experiments are displayed in a descending order of their ranks computed using the Mann-Whitney test.

Combing the logarithmic transform and the discretization did not lead to significant improvement of the performance compared to applying discretization only, Figure 4. A relative robustness in classification performance was achieved only after discretization. By robustness, we mean that most of the values in the distributions of both the *pf* and the *pd* rates are within a finite interval (i.e., the whiskers of the box plot do not extend to 0% and to 100%). When using numeric attributes, the distributions of either one of these rates or both of them covered the full range of [0, 100], Figure 5.

Finally, because there is a larger number of test runs for the CC experiments, it is possible to see that the performance of the logarithmic transform was significantly worse than the performance when using the raw data.

6.2.3 *Cross-company vs Within-company models*

The best performing CC model is not significantly different from the best performing WC model. The relative performance of classification based on WC and CC training shows that the top ranked CC model is not significantly different from the top ranked WC model, Figure 6. Overall, most of the CC models do not have significantly different performance from their respective WC model. Therefore, CC training is a viable option for predicting defective modules.

6.2.4 *NASA vs. Softlab cross-project prediction*

The performance of defect prediction from cross-project

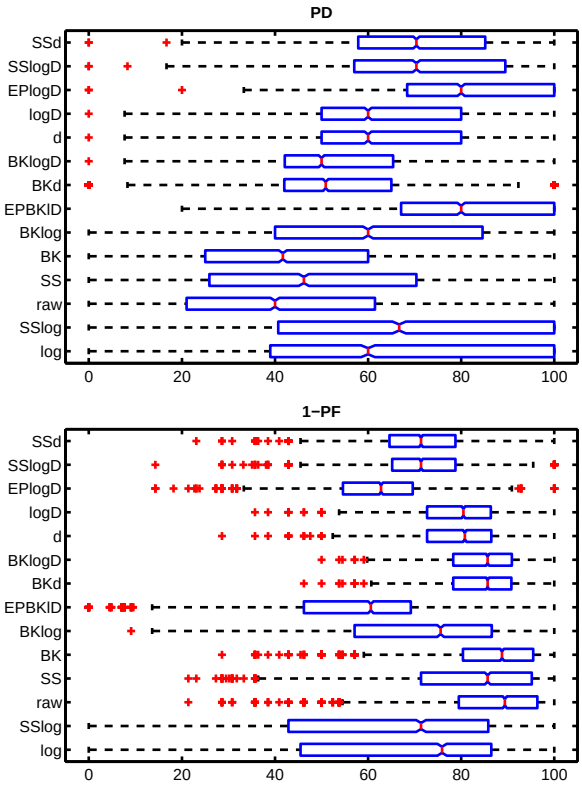

Figure 5: CC training - distributions of *pd* and 1-*pf*. Experiments are sorted by their ranks using *g*.

training is lower for the NASA datasets compared to the Softlab datasets, Figure 8. The median *pd* and *pf* values of some of these experiments are shown in Table 5. Also note,

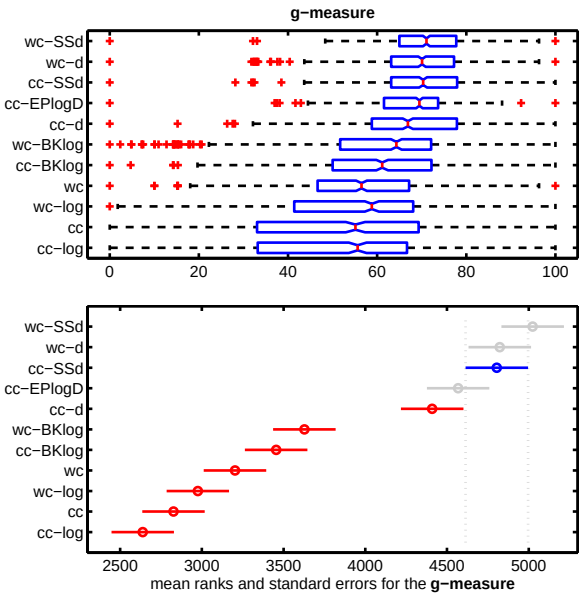

Figure 6: Relative performance of CC and WC models. The experiments are displayed in a descending order of their ranks.

experiment	NASA (*pd*, *pf*)	Softlab (*pd*, *pf*)
SSd	(71.8, 32)	(80, 22.8)
D	(63.2, 25)	(50,14.3)
BKd	(56, 20.2)	(50, 14.3)
log	(66.7, 27)	(100, 18.2)
BKlog	(66.7, 31)	(100, 21.4)
raw	(32, 10.6)	(50, 9.1)

Table 5: Median values of *pd* and *pf* for cross-project experiments.

that their ranking is slightly different compared to when all datasets are used together. The 75th percentile of *pd* never reaches 100% for the NASA datasets. However, comparing *pd* values may be unfair because the Softlab datasets have a small number of defect examples. In fact, for these datasets only 2 defect examples are present in each test set when performing cross-validation. On average, the performance as measured by *pf* is also worse for the NASA compared to the Softlab datasets. These results support the argument to consider the NASA datasets as cross- company.

6.2.5 *Size of the training set*

Reducing the size of the training set by random sub-sampling, which sometimes results in a very small training set, was ranked high in performance. Therefore, one may ask the question how important is the size of the training set. According to the *g*-measure the random sub- sampling performs better compared to the Burak filter and compared to teh experiment of equal priors. Therefore, the small size of the training set is perhaps important because this is what makes the sub-sampling method different from the rest. (In fact, perhaps it is the smaller percentage of non-defect examples in the training set and not the small size of the training set but this experiment is left for future work.) To test this

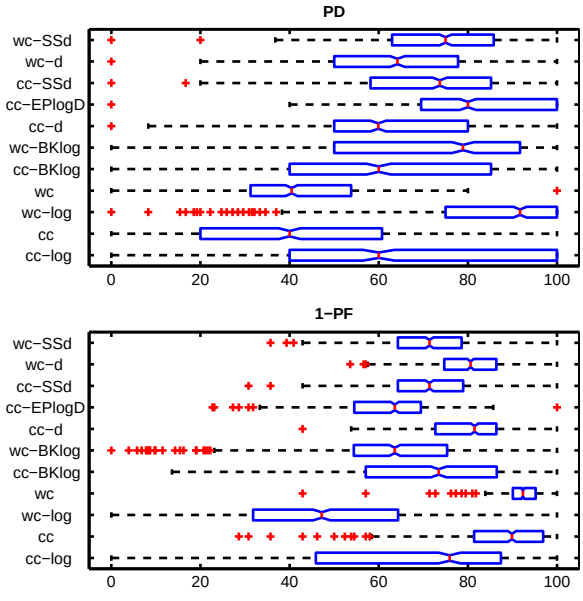

Figure 7: WC vs CC - distributions of *pd* and 1-*pf* for representative WC and CC models. Experiments are sorted by their ranks using *g*.

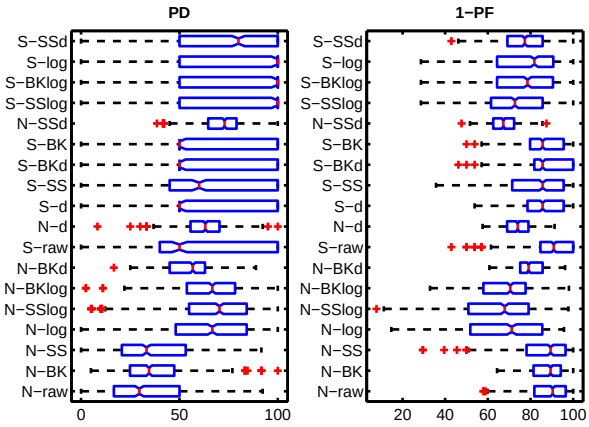

Figure 8: Cross-project prediction for NASA and Softlab datasets. Experiments are sorted by their ranks using *g*.

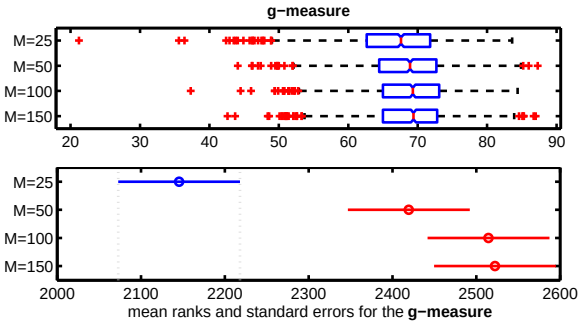

Figure 9: Reduced size of the training set - performance according to *g*. In each experiment the training set contains *M* defect and, whenever possible, *M* non-defect examples.

hypothesis, we performed experiments using training sets of different sizes.

The size of the training set was reduced to M defect examples and M non-defect examples by random sub-sampling. Only three datasets were used in this experiment, **kc1**, **kc2** and **pc1**, because they contain a sufficient number of defect examples. Even using these datasets the number of defect examples was sometimes smaller than M. In these cases all defect examples were used and the number of non-defect examples was reduced to M. Data was pre-processed using logarithmic transform followed by discretization. Performance was not significantly different for $M = 150$ and $M = 50$ but was lower for $M = 25$, Figure 9. Reduction from $M = 150$ to $M = 50$ did not result in a significant change of performance. However, a reduction to $M = 25$ resulted in a significantly lower performance. Therefore, our hypothesis for very small training sets was not supported. However, very large datasets do not help for better prediction. A balance between the total size of the training set and relative numbers of defect and non-defect examples is needed for good prediction performance.

6.2.6 *Learning a general model for defect prediction*

Logarithmic transform followed by discretization was the best performing model for CC training. To test how general is that model for various test data we performed the following experiment. All datasets were split into several subsets (or slices), such that a meaningful number of defect examples (at least 15) were present in each slice. The defect to non-defect ratio was preserved for each slice. This resulted into slices varying from 36 to 221 total number of examples. Next, on slice was chosen at random for a training set. Repeatedly, one slice of the rest was chosen as a test set and used for classification using 4-fold cross-validation and repeating 25 times. Then, this test set was added to the train set either if its mean performance was lower than 60% (which corresponds to the 25th percentile of the CC performance) or if there was more than 10% drop in the mean performance compared to the preceding test slice. The g measure was used in this experiment.

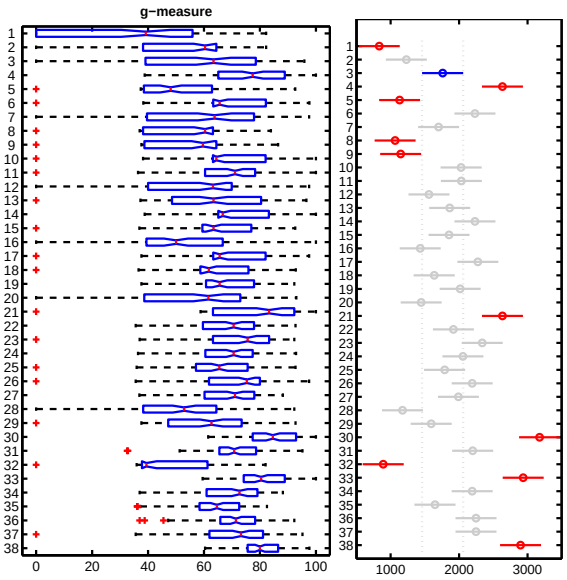

Figure 10: Incremental learning - distributions and rankings of *g*. On the top is the first test slice, on the bottom is the last. The test sets from slices 1, 5, 12, 16, 22, 28, and 32 were added to the training set.

Seven slices were added to the original train set (1, 5, 12, 16, 22, 28, and 32). The majority of the slices used later in the experiment show performance that is not significantly different from subset number 3, Figure 10. At this slice the size of the train set was only 163 examples which belonged to **ar3** and **kc1** datasets. In addition, there are more slice with better performance towards the end of the experiment and more slices with worse performance at the initial stage. This experiment was repeated several times, observing similar trends. No statistical evaluation was performed to combine the results of the different runs.

7. DISCUSSION AND FUTURE WORK

Our experiments lead to the following conclusions:

- Data pre-processing improves the performance of the naive Bayes classifier.
- Cross-company data can be used for prediction of soft-

ware defects.

- The performance of classification from CC training can improved by appropriate selection of the examples used for training.
- Different costs of misclassification can be used to adjust the performance to the application needs.
- There is no need of very large training sets.

As in every statistical testing it is important to take into account the properties of the data. The main problem in performing cross validation using the datasets from Softlab is the small number of defective modules. Results may become skewed if the datasets are split into more than 4 subsets.

The performed experiments suggest that the ratio of defective to non-defective modules in the train set may be an important factor for good prediction performance. Since the percentage of defective modules is small by nature it may be beneficial to conduct more experiments on reducing the number of non-defective modules. Perhaps a certain ratio is needed for good performance.

Different values for the number of neighbors in the Burak filter and the number bins in the discretization technique may improve the results.

8. REFERENCES

[1] Repository of the International conference on predictive models in software engineering, http://promisedata.org/.

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

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

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

[5] R. O. Duda, P. E. Hart, and D. G. Stork. *Pattern Classification, 2nd Edition*. Wiley-Interscience, 2000.

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

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

[8] 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.

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

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

[11] 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.

[12] J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers, 1993.

[13] 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. Eng.*, 14(5):540–578, 2009.

[14] 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.