An Investigation of the Effect of Module Size on Defect Prediction Using Static Measures

A. Güneş Koru and Hongfang Liu
Department of Information Systems,
University of Maryland, Baltimore County - UMBC
Baltimore, MD, 21250
{gkoru,hfliu}@umbc.edu

ABSTRACT

We used several machine learning algorithms to predict the defective modules in five NASA products, namely, CM1, JM1, KC1, KC2, and PC1. A set of static measures were employed as predictor variables. While doing so, we observed that a large portion of the modules were small, as measured by lines of code (LOC). When we experimented on the data subsets created by partitioning according to module size, we obtained higher prediction performance for the subsets that include larger modules. We also performed defect prediction using class-level data for KC1 rather than the method-level data. In this case, the use of class-level data resulted in improved prediction performance compared to using method-level data. These findings suggest that quality assurance activities can be guided even better if defect prediction is performed by using data that belong to larger modules.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*complexity measures, product metrics*; D.2.9 [**Software Engineering**]: Management—*Software quality assurance*

General Terms

Measurement, Management, Reliability.

Keywords

Software Quality Management, Defect Prediction, Prediction Models, Software Metrics, Static Measures.

1. INTRODUCTION

Defective modules pose considerable risk by decreasing customer satisfaction and by increasing the development and maintenance costs. Therefore, in software development life cycle, it is desirable to predict defective modules as early as possible. Effective risk prediction models can improve software developers' ability to identify the defect-prone modules and focus quality assurance activities such as testing and inspections on those modules.

Static measures obtained from source code, such as size, complexity, coupling, and cohesion measures, have been associated with the risk factors such as defects and change. Some researchers found a strong relationship between these measures and risk factors (e.g. [3]), whereas the other researchers found weak or non-supportive evidence as explained by Shepperd and Ince [8] in depth. To a certain extent, the differences in the research results can be due to the variation in the nature of software development because of the differences in development processes, skill levels or expertise of the programmers, problem difficulty, and so on. However, the differences in the results were wide enough to create two groups; one believing in the merits of static measures and prediction models, and the other remaining skeptical. The recently emerging software repositories allow us to bring additional empirical evidence by working on the same data sets and by having an open discussion about our methodologies and results.

For this purpose, we studied the publicly available data that belong to five NASA products, CM1, JM1, KC1, KC2, and PC1[1]. For each module, the related data sets included a set of static measures and a binary variable that takes true value if there are one or more defects reported for that module, and false otherwise. We built several prediction models using the machine learning algorithms available in the tool Weka [10]. The results were neither discouraging nor very promising. However, when we took a closer look at the data, our first observation was that the majority of the modules in these data sets can be categorized as small modules. Therefore, we decided to investigate the effect of module size on the performance of defect prediction models.

Recently, El Emam et al. mentioned the confounding effect of class size on the validity of the object oriented measures [2]. They found that after controlling for size, none of the object-oriented metrics they studied were associated with fault-proneness anymore. In our study, our focus was not on validating individual measures, but on applying data mining techniques and using all available data to build defect prediction models. However, while doing so, our observations were generally aligned with those made by El-Emam et al.

Size is an important characteristic because the static measures usually depend on size. A number of previous studies showed that defect count is positively correlated with size. Therefore, normally, machine learning algorithms can be expected to predict the large modules as defectives with a higher probability. However, to obtain better predictions, machine learning algorithms should detect more specific patterns for defective modules. When a data set is greatly populated by small modules, the following reasons can limit the ability of machine learning algorithms to find such patterns:

9[1]The data sets are available on-line at:
http://promise.site.uottawa.ca/SERepository/datasets-page.html

	n	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
cm1	498	1	8	17	29.64	31	423
jm1	10,885	1	11	23	42.02	46	3,442
kc1	2,109	1	3	9	20.37	24	288
kc2	522	1	4	13	36.89	45	1,275
pc1	1,109	0	7	13	23.38	26	602

Table 1: Summary of the Size Data as Measured by Lines of Code (LOC) in the Examined Products

- For small modules, the static measures other than LOC are also likely to be small, or in other words, closer to zero. Therefore, the characteristics of small modules might show little variation. This similarity can make it difficult for a machine learning algorithm to distinguish between small defective and small non-defective modules.

- The measurements for large modules will have a better chance to show variation. However, the small percentage of large modules in a data set may not be enough to train the models to distinguish between defective and non-defective large modules.

To overcome the above problems, measurements, defect counting, and defect prediction could be done for larger modules. We should mention that, to do so, the system should be large enough to provide enough number of data points which would allow statistically meaningful conclusions. In this case, instead of using binary (true/false) values to indicate defectiveness for each module, defect count could be obtained. In such a scheme, high-defect modules, for example the top N% in defect ranking, could be labeled as "risky", and prediction models could be built to predict those risky modules.

In this paper, we report the results of our study which explored the effect of module size on defect prediction and the potential benefits of using larger modules for prediction. At this point, it is useful to give the definition of *defect* in this study because it might have an effect on the prediction models as mentioned by Nikora and Munson [7]. Defect is defined as "a change needed in a software module because of a problem or a combination of related problems in software". Next, we start with a preliminary analysis of the data sets.

2. PRELIMINARY DATA ANALYSIS

Considering the relationship between size and the effectiveness of the defect prediction models, we first examined the module size in the products that we studied. Table 1 shows a summary of the size data for all products. From the median and average values, it can be seen that the module size of these products can be categorized as small. For example, in KC1, the median value corresponds to 9 LOC. PC1 included relatively larger modules with the median of 23 and the mean of 42.02. However, Table 1 demonstrates that it is fair to say that most of the modules in these data sets are small modules.

Following that, for each product, we examined the LOC histogram of the product and that of the defective modules in the product. Figure 1 compares the LOC histogram of KC1 with the LOC histogram of the defective modules in KC1. It can be seen that the defective modules are shifted to the right of the whole set in terms of LOC distribution. The same situation occurred in all five products. We also examined the LOC rankings for the defective modules. Table 2 shows the average LOC rankings for the defective

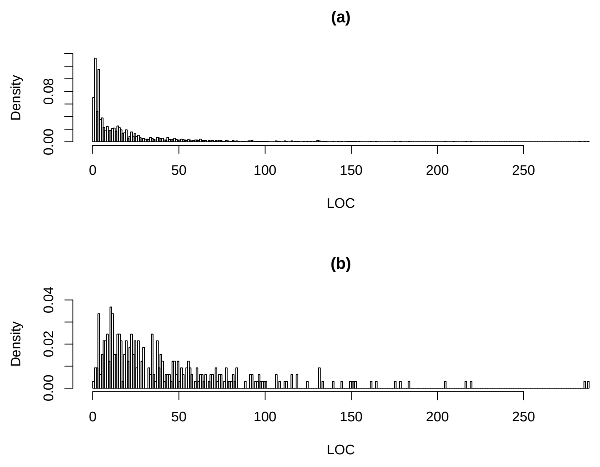

Figure 1: (a) LOC Histogram for KC1 (b) LOC Histogram for Defective Modules in KC1.

Product	Average LOC Rankings for Defectives	Average LOC Rank Percentiles for Defectives
CM1	236.92	73.23
JM1	7343.65	67.47
KC1	1569.19	74.40
KC2	404.39	77.47
PC1	768.07	69.26

Table 2: Results of Rank Analysis

modules and the average LOC ranking percentiles for each product (minimum LOC ranked as 1). The percentiles were obtained by dividing the average LOC ranking for defective modules by the maximum LOC ranking in each product.

From the results of the preliminary analysis, we can say that the larger modules have a higher chance to be defective which is in accordance with the observations made by the previous studies. However, the average and median values for module size in these data sets are very small. It should be noted that the average module size in the previous studies of defect prediction was much higher. For example, in [5], Khoshgoftaar et al. measured 13 million LOC for 7,000 modules, having an average module size of approximately 1,860 LOC. In [4], Khoshgoftaar et al. reported their results for a sample of 1.3 million LOC for 1,980 modules, having an average around 650 LOC. Tian and Nguyenta used tree-based models for Nortel Networks products which had 1,000 modules with the average of 1,000 LOC [9].

3. DEFECT PREDICTION ON SUBSETS

At this step, for each original data set, we obtained fifteen subsets by recursively partitioning in half after ranking the modules according to their size. After each partitioning, we mixed the data points in the subsets randomly to provide an unbiased learning scheme for the cross-validation runs. Figure 2 shows the naming scheme for these subsets. We found it appropriate to stop at the third level where the subsets still included enough number of data points for machine learning.

After that, we ran several machine learning algorithms on the subsets in an exploratory manner and made some observations about the prediction performance. The measures of prediction perfor-

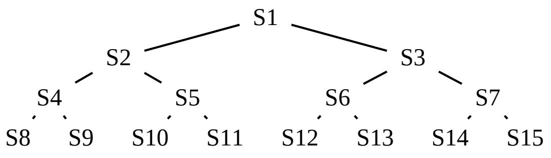

Figure 2: Naming Scheme for Subsets of Data

CONFUSION MATRIX

		Predicted	
		NODEF	DEF
Actual	NODEF	a	b
	DEF	c	d

Precision = d / (b + d) Recall= d / (c+d)
F-Measure = 2 * Recall * Precision / (Recall + Precision)

Figure 3: Measures of Prediction Performance on a Typical Confusion Matrix (NODEF: Not Defective DEF: Defective)

mance, *precision*, *recall*, and *F-measure*, are explained in Figure 3. Normally, the choice of the machine learning algorithm and adjustment of the model parameters affect the performance of the models. Among many machine learning algorithms implemented in Weka, we chose to report our results for two algorithms J48 and KStar. We generally obtained better prediction performances with J48 and KStar. However, it should be noted that comparing the performances of different algorithms was not the focal point of this study.

J48 is a decision tree learner and has been also used for defect detection by Menzies et al. on the same original data sets [6]. The model generated by a decision tree learner is a simple tree structure where non-terminal nodes represent tests on one or more attributes and terminal nodes reflect decision outcomes. Besides, a nice feature of J48 is that tree-based models are easy to interpret directly by human experts. The foundations of tree-based models are explained in detail in [1]. KStar is an instance-based learner. The classification is done for an instance based on the majority classes of K closest instances where the closeness is measured through some similarity measure (here, entropic distance measure). More information on both algorithms can be found in [10].

In some other algorithms available in the tool Weka, the models are generated based on the statistical information about the overall effects of all features and all instances. These models are known to give unsatisfactory performance on very skewed data which is the case in our data sets. Some examples are Bayes Network, Neural Network, and Support Vector Machine. In our exploratory analyses, we found that the prediction performance was poor using these algorithms. For instance, when Neural Networks (or Multilayer Perceptrons) is run for CM1 by its default parameters, precision, recall, and F-measure for defective modules were 0.067, 0.02, and 0.031, respectively.

We developed a set of PERL programs and shell scripts which call the related classes of the tool Weka to execute J48 and KStar using a comprehensive set of parameter-value combinations. They recorded the best precision, recall, and F-Measure results and the parameters corresponding to those best results. When presenting the results, we use F-Measure because it takes both precision and recall into account. F-Measure is used popularly when reporting the performance of prediction models. The resulting plots can be seen in Figure 4 and Figure 5 for J48 and KStar, respectively. The plots are divided by dashed lines vertically to show the different levels of the partitioning scheme shown in Figure 2. It can be seen

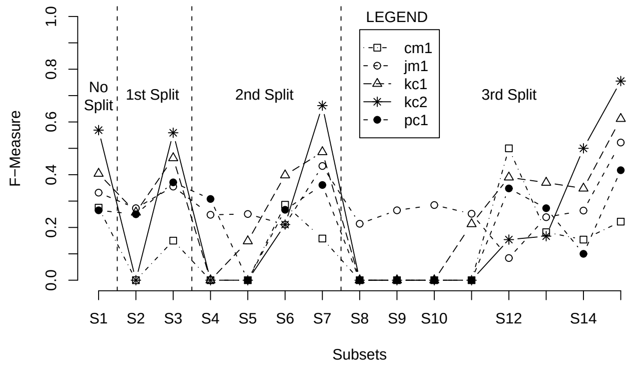

Figure 4: F-measures of Subsets Using J48

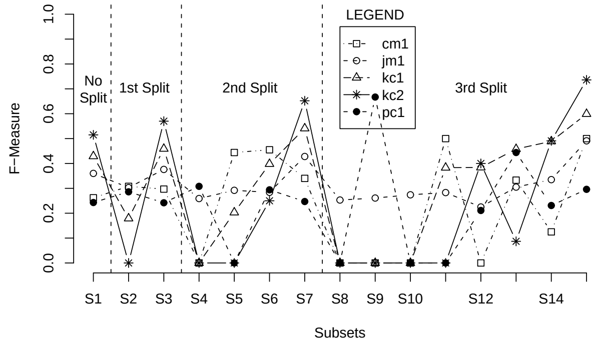

Figure 5: F-measures of Subsets Using KStar

that, at each level, F-Measure values are higher for the subsets that contain larger modules and, generally, there is an upward trend of F-Measure. We should note that, in PC1, there were some small modules with the same measurement values that caused many machine learning algorithms to predict them as defective modules but no other modules. We considered it as a rather unusual case and removed those data points before the analysis.

4. DEFECT PREDICTION USING CLASS LEVEL DATA FROM KC1

From our analyses on the subsets of the available data sets, we concluded that the module size has an effect on the performance of defect prediction. In the datasets that include small modules, the prediction performance was poor. Therefore, we looked for additional data that belong to the same products but collected for modules that are at a higher abstraction level.

We found that KC1 has both class and method level measures posted on-line at the NASA Metrics Program web site[2]. In addition to using the already available class level measures, for each class, we aggregated the method level measures to class level. During this process, we took the minimum, maximum, sum, and average values. There was also a defect file, in which, each defect entry was associated with a method. We obtained the defect data at class level too. We made the resulting data sets and the R function used

9[2]http://mdp.ivv.nasa.gov

Statistics	Size (LOC)	Defect Count
Minimum	1.0	0
1st Quarter	44.0	0
Median	162.0	0
Mean	296.3	4.61
3rd Quarter	315.0	4.00
Maximum	2,883.0	101.00
Total	42,963	669

Table 3: Summary of Size (LOC) and Defect Count for Class Level KC1 Data

	DF	*Top5%DF*
Number of Modules	60	8
LOC Covered	30,728	10,940
Percentage of Total LOC	72%	25%
Minimum LOC	64	157
Median LOC	286	1,195
Average LOC	512	1,367.5
Maximum LOC	2,883	2,883
Total Defect Count	669	293
Percentage of Total Defects	100%	44%
Minimum Defect Count	1	22
Median Defect Count	6	25
Average Defect Count	11.15	36.63
Maximum Defect Count	101	101

Table 4: Summary of *DF* and *Top*5%*DF* for KC1

to create them on-line and publicly available[3]. The class level data for KC1 included 145 data points. The summary of the size and defect data for this data set can be seen in Table 3.

From Table 3, it can be seen that when we accept classes as modules, the median and average module size is a bit closer to those appeared in the literature. This time instead of only having a binary value to indicate defect-proneness, we also had defect count for each class. We built our models both to predict the defective classes (DF), which were associated with at least one defect record, and to predict the top 5 percent (TOP5%DF) classes in the defect ranking. Table 4 gives a summary of the size and defect count in these groups of modules.

We used J48 to predict the modules that fall into these top defect module groups. The specific program call was:

$$weka.classifiers.trees.J48\ -C0.25\ -M8\ .$$

We set the minimum number of instances at any leaf node to 8. Again, 10-fold cross validation was used. To obtain even better estimates, we found the averages for these values over 10 different runs with different randomization seed values used for cross-validation shuffling in each run. Table 5 shows the resulting precision, recall, and F-Measure values.

9[3]http://promise.site.uottawa.ca/SERepository/datasets-page.html

		Precision	Recall	F-Measure
DF	HR	.62	.68	.65
	LMR	.76	.71	.73
TOP5%DF	HR	.50	.63	.56
	LMR	.98	.96	.97
KC1 Data at PROMISE Site	DEF	.51	.24	.33
	NODEF	.87	.96	.91

Table 5: Precision, Recall, and F-Measure (HR: High Risk, LMR: Low and Marginal Risk, DEF: Defective, NODEF: Not Defective)

5. DISCUSSION OF KC1 RESULTS

As mentioned before, in the KC1 data set posted at the PROMISE workshop web site, modules are labeled as defective or not defective. In Table 5, it can be seen that using this data set, recall was .24.

The precision and recall when predicting the defective classes (DF) is high, however, since there are 60 modules with at least one or more defects covering 72% of the total source code, predicting the defective classes is perhaps not an ambitious purpose.

In TOP5%DF, precision was .5 and recall was equal to .63. This means that, out of the 8 modules in TOP5%DF defect ranking, 5 of them were predicted correctly. Weka did not allow us to identify exactly which instances (data points or modules) were correctly classified because cross-validation in Weka shuffles the instances and changes their order in the output. However, considering that the average defect count for TOP5%DF was around 37, we can say 185 defects from 669 would be caught.

Another advantage of using large modules is that the managers and programmers can easily recognize them and make their judgements to further investigate them or not. When the inspectors or testers are given larger modules that have a very high probability to contain a large number of defects, they can better locate the problems and see overall opportunities to eliminate the defects and improve code quality because the surrounding code will also provide some context information.

The use of tree-based algorithms such as J48 can help managers and developers to both predict and characterize the top defect modules. For example, the decision tree that we obtained for TOP5%DF using the whole data set was a very simple 1-level tree that can be defined by the rule:

```
sumNUM_UNIQUE_OPERATORS <= 285: LMR
sumNUM_UNIQUE_OPERATORS > 285: HR
```

In this tree-based model, the whole set of data points in KC1 located in the root node are divided into two parts using the cut-off value of 285 of the metric sumNUM_UNIQUE_OPERATORS. The classes with greater values are predicted as HR (in this case TOP5%DF) and those with smaller values are predicted to be LMR. Such models can be easily understood and interpreted by human experts. Of course, the model can be extended to have more levels and to fit the data for better characterization of the data set, however the prediction performance on cross validation and on new data sets will drop in that case.

6. CONCLUSIONS

Based on our experiments on the data subsets obtained by partitioning according to size, we observed that predictability was poor for the subsets that included small components. In KC1, we observed that using class level data was a preferable alternative. Considering these results, it can be preferred to perform defect prediction for large components rather than small components. Another action could be, for small components, shifting the level of static measures and defect count to a higher abstraction level similar to what we have done in KC1 by shifting the static measures and defect data from method level to class level.

7. REFERENCES

[1] Leo Breiman, Jerome H. Friedman, Richard A. Olshen, and Charles J. Stone. *Classification and Regression Trees*. Wadsworth & Brooks, 1984.

[2] Khaled El Emam, Saïda Benlarbi, Nishith Goel, and Shesh N. Rai. The confounding effect of class size on the validity of object-oriented metrics. *IEEE Trans. on Software Engineering*, 27(7):630–650, July 2001.

[3] Sallie Henry and Dennis Kafura. Software structure metrics based on information flow. *IEEE Trans. on Software Engineering*, 7(5):510–518, September 1981.

[4] Taghi M. Khoshgoftaar, Edward B. Allen, Kalai S. Kalaichelvan, and Nishith Goel. Early quality prediction: A case study in telecommunications. *IEEE Software*, 13(1):65–71, January 1996.

[5] Taghi M. Khoshgoftaar, Abhijit S. Pandya, and David L. Lanning. Application of neural networks for predicting program faults. *Annals of Software Engineering*, 1:141–154, 1995.

[6] Tim Menzies, Justin S. Di Stefano, Chris Cunanan, and Robert (Mike) Chapman. Mining repositories to assist in project planning and resource allocation. In *International Workshop on Mining Software Repositories*, May 2004.

[7] Allen P. Nikora and John C. Munson. The effects of fault counting methods on fault model quality. In *COMPSAC '04: The 28th International Computer Software and Application Conference*, pages 192–201. IEEE Press, September 2004.

[8] Martin Shepperd and Darrel Ince. *Derivation and Validation of Software Metrics*. Clarendon Press - Oxford, Oxford University Press, Walton Street, Oxford OX2 6DP, 1993.

[9] Jeff Tian, Anthony Nguyen, Curt Allen, and Ravi Appan. Experience with identifying and characterizing problem prone modules in telecommunication software systems. *Journal of Systems and Software*, 57(3):207–215, July 2001.

[10] Ian H. Witten and Eibe Frank. *Data Mining: Practical machine learning tools with Java implementations*. Morgan Kaufmann, San Francisco, 2000.

APPENDIX

A. STATIC MEASURES FOR KC1

For KC1, we used already available object-oriented measures. These are given in Table 6. There were also a set of measures at module level given in Table 7. We transformed them to class level using minimum, maximum, average, and sum operations. Therefore, for each measure in Table 7, we obtained four class level measures. For example, for NUM_UNIQUE_OPERATORS, we obtained:

- minNUM_UNIQUE_OPERATORS
- maxNUM_UNIQUE_OPERATORS
- avgNUM_UNIQUE_OPERATORS
- sumNUM_UNIQUE_OPERATORS

MEASURE NAME
PERCENT_PUB_DATA
ACCESS_TO_PUB_DATA
COUPLING_BETWEEN_OBJECTS
DEPTH
LACK_OF_COHESION_OF_METHODS
NUM_OF_CHILDREN
DEP_ON_CHILD
FAN_IN
RESPONSE_FOR_CLASS
WEIGHTED_METHODS_PER_CLASS

Table 6: Object Oriented Static Measures.

MEASURE NAME
LOC_BLANK
BRANCH_COUNT
LOC_CODE_AND_COMMENT
LOC_COMMENTS
CYCLOMATIC_COMPLEXITY
DESIGN_COMPLEXITY
ESSENTIAL_COMPLEXITY
LOC_EXECUTABLE
HALSTEAD_CONTENT
HALSTEAD_DIFFICULTY
HALSTEAD_EFFORT
HALSTEAD_ERROR_EST
HALSTEAD_LENGTH
HALSTEAD_LEVEL
HALSTEAD_PROG_TIME
HALSTEAD_VOLUME
NUM_OPERANDS
NUM_OPERATORS
NUM_UNIQUE_OPERANDS
NUM_UNIQUE_OPERATORS
LOC_TOTAL

Table 7: Method Level Measures Transformed to Class Level

The long names of the static measures in Table 6 and Table 7 can be found in the preamble of the arff files posted at http://promise.site.uottawa.ca/SERepository/datasets-page.html and in the glossary posted at http://mdp.ivv.nasa.gov/mdp_glossary.html.