Fault Prediction using Early Lifecycle Data

Yue Jiang, Bojan Cukic, Tim Menzies
Lane Department of Computer Science and Electrical Engineering
West Virginia University, Morgantown, WV 26506, USA
{yue, cukic@csee.wvu.edu; tim@menzies.us}

Abstract

The prediction of fault-prone modules in a software project has been the topic of many studies. In this paper, we investigate whether metrics available early in the development lifecycle can be used to identify fault-prone software modules. More precisely, we build predictive models using the metrics that characterize textual requirements. We compare the performance of requirements-based models against the performance of code-based models and models that combine requirement and code metrics. Using a range of modeling techniques and the data from three NASA projects, our study indicates that the early lifecycle metrics can play an important role in project management, either by pointing to the need for increased quality monitoring during the development or by using the models to assign verification and validation activities.

1 Introduction

The prediction of fault-proneness has been studied extensively [4, 5, 7, 11–13, 15, 18, 22]. In a stable development environment, metrics can be used to predict modules that are likely to harbor faults. Some researchers endorse to use of product metrics, such as Halstead complexity [14], McCabe's cyclomatic complexity [17], and various code size measures to predict fault-prone modules [4, 5, 7, 11–13, 15, 18, 22], while others are skeptical of such simplistic approaches [10, 23].

V&V textbooks, [21] for example, recommend using static code metrics to decide whether modules are worthy of manual inspections. Our experience with NASA software Independent Verification and Validation facility and with several large government software contractors is that they won't review software modules *unless* tools like McCabe predict that they are fault prone. The use of such measures is controversial. Fenton offers an example where *the same* program functionality is achieved using *different* programming language constructs resulting in *different* static measurements for that module [12]. Fenton uses this example to argue the uselessness of static code attributes. Fenton & Pfleeger note that the main McCabe's attribute (cyclomatic complexity, or $v(g)$) is highly correlated with lines of code [12]. Also, Shepperd & Ince remark that "for a large class of software cyclomatic complexity is no more than a proxy for, and in many cases out performed by, lines of code" [23]. Therefore, they argue against the use of *single* features to predict for defects. Further, they reject other commonly used indicators since they are all highly correlated and, so they argue, just as uninformative.

When individual features fail, *combinations* can succeed. This paper argues that *combinations* of static features extracted from requirements and code can be exceptionally good predictors for modules that actually harbor defects. We do not intend to propose yet another classification algorithm. Our overall goal is to explore whether prediction of fault prone modules can be achieved using the information available in the early phases of software development.

In this paper, we investigate whether metrics available early in the development lifecycle can be used to identify fault-prone software modules. More precisely, we build predictive models using the metrics that characterize structured textual requirements. We compare the performance of requirements-based prediction models against the performance of code-based models. Finally, we develop a methodology that combines requirement and code/module metrics. Since such models cannot be developed early in the lifecycle, we evaluate whether such combination can increase the prediction accuracy. Using a range of modeling techniques and the data from three NASA projects, CM1, JM1, and PC1, our experiments indicate that the early lifecycle metrics can play an important role in project management, either by pointing to the need for increased quality monitoring during the development or by using the models to assign verification and validation activities.

The rest of paper is organized as follows. Section 2 introduces the measurement techniques used to evaluate predictive software classification models. Section 3 explains the experimental setup, while Section 4 presents the results.

Predicted	Real data		
		Yes	**No**
	Yes	TP	FP
	No	FN	TN

Figure 1. A defect level prediction sheet.

These results are discussed in Section 5. Section 6 summarizes our findings and points out possible directions for future work.

2 Measurement

In this study, we develop statistical models to predict defective software modules. Requirement metrics, module-based code metrics, and the fusion of requirement and module metrics serve as predictors. The predicted variable is whether one or more defects exists in the given module. Figure 1 describes prediction outcomes.

Throughout the paper, we use the following set of evaluation measures. The Probability of Detection (PD), also called recall or specificity in some literature [13, 18]), is defined as the probability of the correct classification of a module that contains a defect:

$$PD = \frac{TP}{TP + FN}$$

The Probability of False alarm (PF) is defined as the ratio of false positives to all non-defect modules:

$$PF = \frac{FP}{FP + TN}$$

Intuitively, we would like to maximize PD and at the same time minimize PF. Since we have a limited space available here, we refer readers to a recent publication [16] which provides a rather comprehensive overview of statistical methods relevant for evaluating predictive models in software engineering.

3 Experimental Methodology

3.1 Random Forests

Random Forest is a tree-based classifier which has demonstrated its robustness in building software engineering models [13]. As implied from its name, it builds an ensemble, i.e., the "forest" of classification trees using the following strategy:

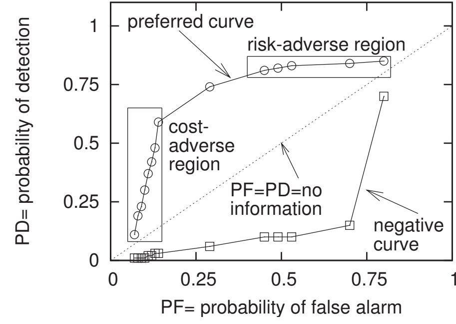

Figure 2. Regions of a typical ROC curve.

1. The root node of each tree contains a bootstrap sample data. Each tree has a different bootstrap sample.
2. At each node, a subset of variables are randomly selected to split the node.
3. Each tree is grown to the largest extent possible without pruning.
4. When all trees in the forest are built, test instances are fitted into all the trees and a voting process takes place. The forest selects the classification with the most votes as the prediction of new instances.

Random forest [6] as a machine learning classifier provides many advantages. One, it automatically generates the importance of each attribute in the process of classification. Two, by varying voting thresholds in step 4 of the algorithm, we can generate a $Receiver\ Operator\ Characteristic$ (ROC) curve that represents an entire range of achievable performance characteristics relative to PD and PF. In the experiments, we build 20 random forests for each data set, each with a different voting threshold ranging from 0.05 to 0.95.

3.2 ROC curve

In this study, we apply the same set of classification algorithms to the set of software engineering datasets. Therefore, we need an intuitive way to compare the ensuing classification performance. An ROC curve provides a visual comparison of the classification performance. It is a plot of PD as a function of PF across all the possible experimental settings. A typical ROC curve is shown in Figure 2. Typical ROC curve has a concave shape with (0,0) as the beginning and (1,1) as the end point.

Figure 2 provides an insight into the implications of the classification performance to software engineering experiments. A straight line connecting the (0,0) and (1,1) implies that the performance of a classifier is no better than random

guessing. The cost-adverse region in Figure 2 has low PD and PF. If a classifier falls into this region it will be useful for organizations with limited $V\&V$ budgets. The risk-adverse region indicates high PD and high PF. For safety critical systems, classifiers demonstrating such performance may be preferred because the identification of faults is more important than the cost to validate false alarms. The negative curve in Figure 2 is not necessarily a bad news, as it can be transposed into a preferred curve [18, 19]. In an ROC curve, software engineers need to identify the points that suit technical risks and budgets specific to their project.

3.3 Experimental Design

In this paper, we report the development and evaluation of models to predict fault-prone software modules using the following information from NASA MDP datasets [2]:

1. Experiment 1: Available metrics describing unstructured textual requirements;
2. Experiment 2: Available static code metrics;
3. Experiment 3: A combination of the requirement metric and static code metrics.

The goal of each of these experiments is different. In experiment 1, we try to assess how useful the early lifecycle data and the related metrics are in identifying potentially problematic modules. The second experiment is the least interesting from the research standpoint as the use of static code metrics in the prediction of faulty modules has been investigated extensively, even on the same datasets we use here. However, in our case, it sets the performance baseline for the third experiment. We will demonstrate that the use of combined requirements metrics and static code metrics performs extremely well in predicting fault-prone modules.

In order to enable these experiments, we had to be able to relate individual requirements with software modules. The generic structure of an entity relationship diagram that connects requirements with modules for a specific project entered in NASA MDP database is shown in Figure 3. All software requirements and modules are uniquely numbered. A requirement may be implemented in one or more modules. A module implementation may reflect one or more requirements. Further, a module may contain zero, one or more faults. For the purpose of our study, if a module contains any faults, it is considered fault-prone. Otherwise, it is defect free. Unfortunately, MDP datasets reveal anomalies too. Some requirements are associated with no software modules and some modules cannot be traced to any stated requirement. As our research group has not been involved with the data collection, we could only point out to such inconsistencies. The extent of such inconsistencies is described later.

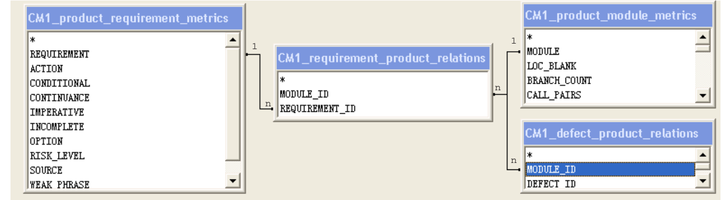

Figure 3. An entity-relationship diagram relates project requirements to modules and modules to defects

3.4 Combining requirement and module metrics

Combining requirement metrics and module metrics if there is a one-to-one relationship between the requirements and modules is trivial. But, due to many-to-many relationship, we need to utilize the $inner-join$ database operation. Inner-join creates an all-to-all association between the corresponding entries in two database tables. This is the most common type of the database join operation.

For clarity, we provide an example of the inner-join. Table 1 shows a few entries from the table which relates requirements and modules for CM1 project.

Table 1. CM1_requirement_product_relation table.

MODULE_ID	REQUIREMENT_ID
25321	100
25333	100
25325	101
25321	102
25333	102
...	...

Referring to Figure 3, the inner-join takes all the records from CM1_product_requirement_metrics table and finds the matching record(s) in table CM1_requirement_product_relations based on the common predicate — Requirement_ID. The result is written into a temporary table. Inner-join further takes all the records from the temporary table and looks for the matching records in CM1_product_module_metrics via the join predicate Module_ID. Table 2 shows partial results of this operation.

3.5 Machine learning algorithms

We construct predictive models using machine learners from Weka package [20] as shown in Table 3. All these

Table 2. The result of inner join on CM1 requirement and module metrics.

REQUIREMENT	ACTION	CONTINUANCE	MODULE	LOC_BLANK	BRANCH_COUNT	···
100	1	0	25321	82	43	···
100	1	0	25333	34	17	···
101	3	1	25325	14	7	···
102	1	0	25321	82	43	···
102	1	0	25325	14	7	···
102	1	0	25333	34	17	···
···	···	···	···	···	···	

machine learning algorithms are used with their default parameters. For the Random Forest algorithm we used the implementation from the statistical package R [3]. We would like to stress that Weka and R are publicly available. Also publicly available are NASA MDP datasets. Therefore, our results should be easily checked and reproduced.

Table 3. Machine learners used in experiments

learners	abbreviation
OneR	OneR
NaiveBayes with kernel	nbaye
VotedPerceptron	VP
Logistic	Log
J48	J48
VFI	VFI
IBk	IBk
Random Forest	RF

Cross-validation is the statistical practice of partitioning a sample of data into two subsets: one is training subset and the other is testing subset. The training subset trains the predictors and the testing subset validates the predictors. We use 80% of data as training subset and 20% of data as validation subset in all the experiments. The data is randomly divided into 5 fixed bins with equal size. We leave one bin to act as test data, and the other 4 bins is used to train the learners. Our experiments use the same training set and testing set 10 times, and then one of the training bins becomes the test bin. We conduct 50 experiments (5×10) and predict the mean prediction result. This methodology is called the five-fold (5×10 way) cross validation.

3.6 Datasets

The datasets used in this study come from NASA Metrics Data Program (MDP) data repository [2]. The repository provides metrics that describe the software artifacts from 13 NASA projects.

Although MDP data repository contains 13 projects, only 3 of them offer requirement metrics. These three projects are CM1, JM1, and PC1. CM1 project is a NASA spacecraft instrument, JM1 is a realtime ground system, PC1 is an earth orbiting satellite system. There are 10 attributes that describe requirements. One of them is the unique requirement identifier. The remaining 9 attributes are the metrics shown in Table 4. We use them as attributes when building defect prediction models.

All the MDP requirement metrics follow the definitions from Wilson [24, 25]. According to these references, a tool "searches the requirements document for terms [the research at NASA-Goddard Software Assurance Research Center has] identified as quality indicators". ARM (*Automated Requirement Measurement*) tool [24] was an experiment in lightweight parsing of requirements documents. Rather than to tackle the complexities of full-blown natural language, the ARM research explored what could be easily automatically identified. The ARM work resulted in an automatic parser and some threshold guidelines regarding when to be concerned about a document. Since our classification algorithms can automatically generate such thresholds, we ignored the ARM thresholds and just used the results of the parser.

3.7 Static Requirements Features

The ARM parser reports information at the level of individual requirement specifications. Specification statements are evaluated along the following dimensions:

- *Imperatives* (something that must be provided) contain the phrases "shall", "must" or "must not", "is required to", "are applicable", "responsible for", "will", or "should".
- *Continuances* (connections between statements) contain the phrases "as follows", "following", "listed", "in particular", or"support".
- *Directives* (to supporting illustrations) contain the phrases "figure", "table", "for example", "note".
- *Options* (giving the developer latitude in satisfying the specification statement) contain the phrases "can", "may", "optionally".

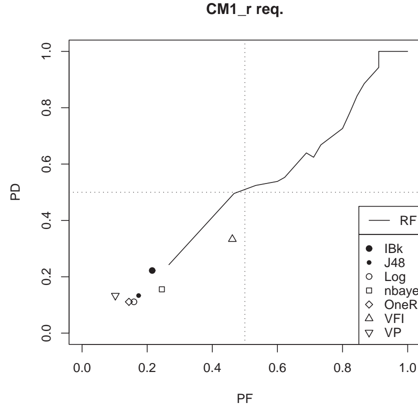

Figure 4. CM1_r prediction using requirements metrics only.

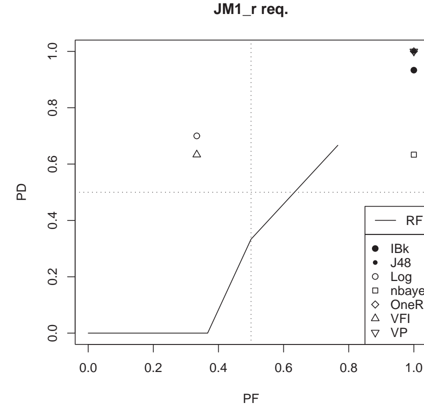

Figure 5. JM1_r prediction using requirements metrics only.

- *Weak Phrases* (causing uncertainty, leave room for multiple interpretations) contain the phrases "adequate", "as a minimum", "as appropriate", "be able to", "be capable", "but not limited to", "capability of", "capability to", " effective", "if practical", "normal", " provide for", "timely", and "tbd".

4 Experimental Results

As mentioned earlier, partial requirement metrics are available for the three MDP datasets: CM1, JM1 and PC1. In addition, our analysis uncovered several data discrepancies. Table 5 summarizes the available data. In projects CM1 and PC1, only 22% and 18% of all modules, respectively, have their requirements identified. The extreme case is the dataset JM1 in which only 1% of modules are associated with any requirement. But, due to the fact that JM1 is the largest project, the number of modules with identified requirements (97) is similar to the corresponding value for PC1 (109). In experiments, we will consider only the subset of modules in CM1, JM1 and PC1 which have their requirements identified. We will call these datasets CM1_r, JM1_r and PC1_r, to separate them from their usage in existing literature [7, 13, 18].

4.1 Prediction from Requirements Metrics

4.1.1 CM1_r dataset

CM1_r dataset describes software artifacts of a NASA spacecraft instrument. CM1_r has 160 requirements, but only 114 of them have associated program modules identified. Among these 114 requirements, 69 (60.53%) of them are related to a module which contains at least one defect. This is a very significant percentage. On the other hand, only the modules with identified requirements (109 of them out of 505) were used in the training/testing datasets.

As described in the Experimental Design section, we first developed models that predict defective modules using the requirements metrics only. The performance of different machine learning algorithms is depicted in the ROC curve shown in Figure 4. No model appears to provide particularly useful information for project managers, which can probably be attributed to the fact that more than 60% of the requirements are related to defective modules.

4.1.2 JM1_r dataset

JM1 metrics represent a realtime ground system that uses simulations to generate flight predictions. JM1_r has 74 requirements available (see Table 5). Only 17 of them are related to program modules. Three of these 17 requirements are associated to defective modules defect (17.65%). Figure 5 compares the performance of different models in an ROC curve. Although the performance of most models fails to provide useful information, the models built using Logistic and VFI algorithms appear to do better than the others. Although JM1 is a very large project ($> 10,000$ modules overall), JM1_r is very small as the information that relates requirements and program modules is largely missing.

4.1.3 PC1_r dataset

PC1 project refers to an Earth orbiting satellite software system. PC1_r has 320 available requirements (Table 5) and, in contrast to CM1_r and JM1_r, all of them are associated

Table 4. Requirement Metrics.

Requirement	Definitions
action	Represents the number of actions the requirement needs to be capable of performing.
conditional	Represents whether the requirement will be addressing more than one condition.
continuance	Phrases such as the following: that follow an imperative and precede the definition of lower level requirement specification.
imperative	Those words and phrases that command that something must be provided.
incomplete	Phrases such as 'TBD' or 'TBR'. They are used when a requirement has yet to be determined.
option	Those words that give the developer latitude in the implementation of the specification that contains them.
risk_level	A calculated risk level metric based on weighted averages from metrics collected for each requirement.
source	Represents the number of sources the requirement will interface with or receive data from.
weak_phrase	Clauses that are apt to cause uncertainty and leave room for multiple interpretations.

Table 5. The associations between modules and requirements in CM1, JM1 and PC1.

	total modules	modules with defects	modules have req.	defect modules with req.	total # of req.	req. related to module(s)	# of req. related to defects
CM1_r	505	81(16.03%)	109(22%)	58(53.21%)	160	114	69(60.53%)
JM1_r	10,878	2102 (19.32%)	97(1%)	4(4.12%)	74	17	3(17.65%)
PC1_r	1107	73(6.59%)	203(18%)	44(21.67%)	320	320	109(34.06%)

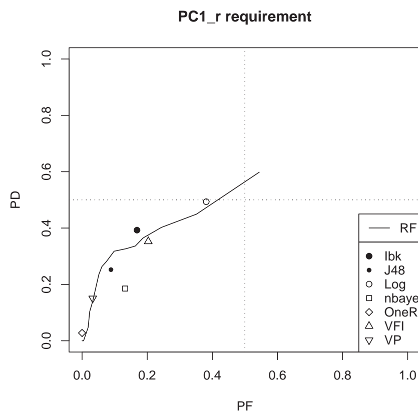

Figure 6. PC1_r prediction using requirements metrics only.

with program modules. 109 (34.06%) requirements, or at least some aspect of these requirements, are implemented by modules which contain one or more defects. On the other hand, all these requirements point to only 203 out of 1107 modules indicating, again, missing information. Consistent with our experimental design, we developed PC1_r models using the 320 requirements and 203 modules.

The performance of PC1_r is shown in Figure 6. Unlike in the other two experiments, the useful portion of the ROC curve is in the cost-adverse region with low false alarm rate ($PF < 0.5$), but coupled with the low probability of detection ($PD < 0.5$). If the data represented a stable development environment in which this type of a model was built from an earlier project, the fact that the V&V team could start to develop information about modules in which defects can be expected seems very encouraging. However, given that these are the first-of-the-kind, preliminary experiments with defect prediction models build from requirement metrics, we can only cautiously suggest that our results warrant further research in this area.

4.2 Prediction from Static Code Metrics

As discussed, an extensive body of work describes defect prediction from module code metrics. Therefore, the purpose of this section is to establish the baseline performance to be used for performance comparison throughout this paper. When building models, we use 37 static code

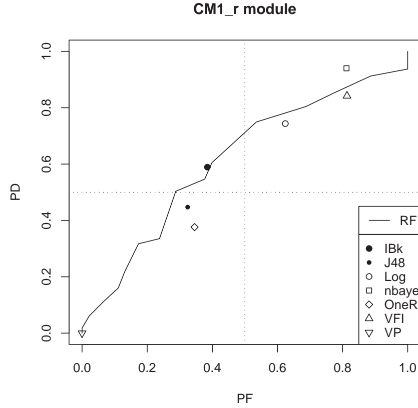

Figure 7. CM1_r using module metrics.

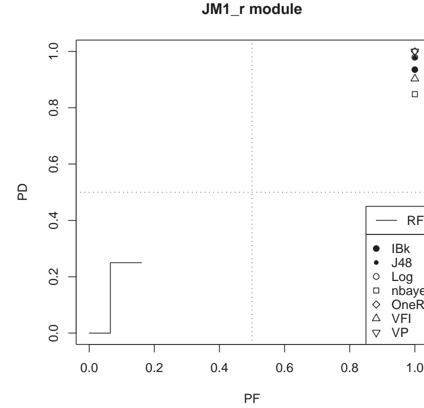

Figure 8. JM1_r using module metrics.

attributes of CM1_r and PC_r datasets. JM1_r dataset offers 21 code metrics. The predicted class is always the presence of defects in a module (defect-free or defective). A detailed description of module attributes can be found in [13, 18]. Several recent studies describe the defect prediction results for the NASA MDP datasets [7, 13, 18]. These studies provide insights into the effects of feature (attribute) selection to model performance, as well as the variations caused by the use of a wide spectrum of machine learning algorithms. The observation has been that the choice of learning methods have a much stronger impact on performance than feature selection [18].

Although CM1, JM1 and PC1 have static code metrics data available for all modules (505, $10,878$ and $1,107$, respectively), to make performance comparisons fair, we only use modules that explicitly correspond to requirements (109, 97, 203, respectively). Therefore, the performance results embodied by the ROC curves are not the same as reported in related literature [7, 13, 18], but similar.

Figures 7, 8 and 9 depict the performance of defect prediction models based on code metrics. Machine learning algorithms provide models in the different regions of the ROC space. While the performance on neither of the project datasets appears impressive, consistent with the trends observed in related studies, JM1_r dataset proves extremely difficult. Among those who use NASA MDP datasets, JM_1 has been suspected of containing noisy (inaccurate) information. But while its large size usually makes it challenging for defect prediction, in our experiment JM1_r contains only 97 modules, with 4 of them being defective. Therefore it comes as no surprise that most machine learning algorithms (all except the random forest) generate theories that classify all the modules as defect-free.

The machine learning algorithms which typically perform better than others are random forests, naive bayes, and

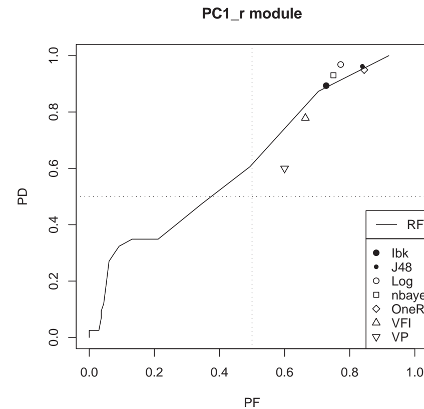

Figure 9. PC1_r using module metrics.

IBk, although the usefulness of their predictions in the context of software engineering projects appears rather limited.

4.3 Combining Requirement and Module Metrics

We combine requirement and module metrics using the inner-join method described earlier. Recall that, there are 109, 97 and 203 modules associated to requirements in CM1, JM1 and PC1 datasets, respectively. When many-to-many relationship exists between modules and requirements, the inner join creates multiple entries in the resulting table, one for each unique Requirement_ID and Module_ID pair. Consequently, the datasets used in this experiment are larger. CM1_RM has 266 records, while PC1_RM has 477 records. JM1_RM maintained the same number of records, 97 because each module is related to a single requirement. The number of defective modules in each dataset remains

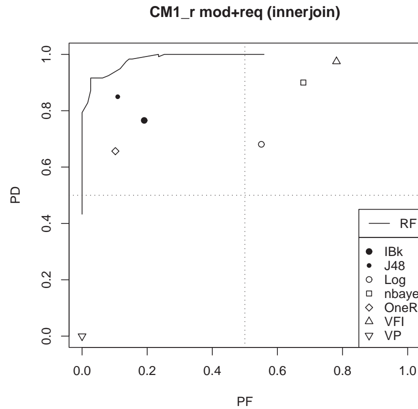

Figure 10. CM1_RM model uses requirements and code metrics.

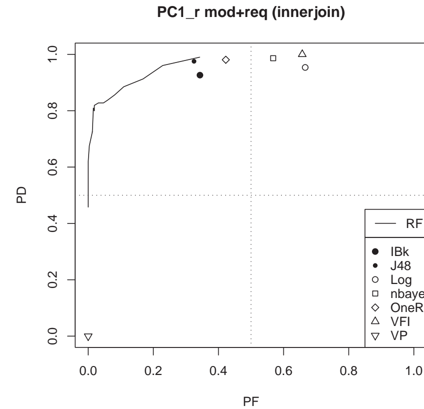

Figure 11. PC1_RM model uses requirements and code metrics.

the same as in the previous experiments; 147 defect records for CM1_RM, only 4 for JM1_RM, and 111 for PC1_RM, but multiple defect records may exist for a single module.

Following the preparation of data set, we ran the same set of experiments as before. The prediction results for CM1_RM are shown in Figure 10 and for PC1_RM in Figure 11. The presence of additional requirement metrics attributes in JM1_RM did not improve the prediction model which used code metrics only. Both JM1_r and JM1_RM have the same number of entries, 97, which seems to justify the lack of performance gains.

5 Discussion

Upon seeing the results of the experiments that combine requirements and static code metrics, we were simply astonished. We have been building models for defect prediction in NASA MDP datasets for several years. A recent publication [18] reports that the best model performance based on code metrics only is $(PD = 71\%, PF = 27\%)$ for CM1 dataset and is $(PD = 48\%, PF = 17\%)$ for PC1. Our own experiments with static code metric models reported in this paper reflect the use of subsets of CM1, JM1 and PC1 project data and are somewhat different. But the performance improvement gained by adding requirements metrics has surpassed all our expectations. Figures 12, 13 and 14 depict ROC curves for the three datasets, CM1, JM1 and PC1, respectively. Depicted models have been developed using the random forests algorithm. Random forests have demonstrated the most consistent performance in all our experiments and we believe it is appropriate to use them for performance comparisons. Each of these three figures contains three lines, depicting the performance of requirements-only based model, the code-only based model and, finally, the model that combines requirements and code metrics.

These results demand a careful analysis. Our first observation is specific and it relates to the inner workings of machine learning algorithms we used for modeling. After the inner-join, the datasets have more records. In other words, the datasets became oversampled. Oversampling is known to be an effective method in signal processing. Recent studies show that tree-based classifiers do not increase the performance as the result of oversampling in the training data [8, 9]. Random forests is a tree based ensemble-forming algorithm. So our results appear to contradict the results of Drummond and Holte in [8, 9]. Why random forests perform so well on the inner-join data? This phenomenon certainly is worth exploring in the future. However, nearly all the learners we used except VotedPerceptron (VP) achieve better performance when requirements and code metrics are combined. Therefore, the observed performance improvement cannot be explained through better understanding of the machine learning algorithms only.

The obvious speculation here is that the training/test datasets after the inner-join represent real world software engineering situations better. In the Introduction, we offered a glimpse of the debate on the usefulness of static code metrics for defect prediction. Our experiments provide initial evidence that combining metrics that describe different yet related software artifacts may significantly increase the effectiveness of defect prediction models. Although each metric, regardless whether it describes requirement or code features, appears highly abstract and seemingly unrelated to software defects, when combined they support what appear to be superior defect prediction models.

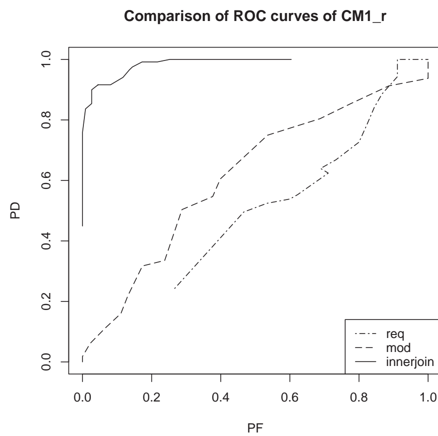

Figure 12. ROC curves for CM1 project.

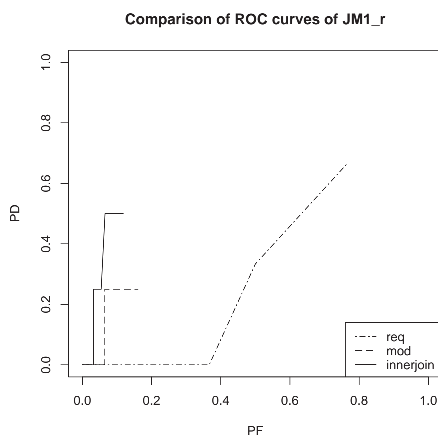

Figure 13. ROC curves for JM1 project.

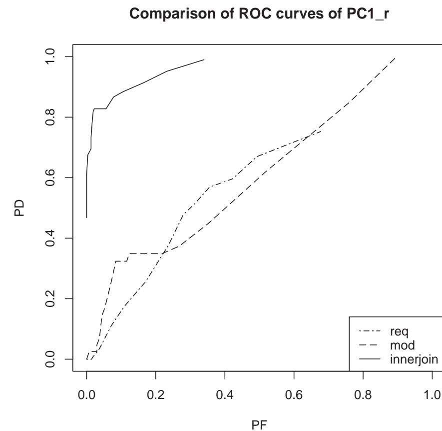

Figure 14. ROC curves for PC1 project.

We need to add a few caveats. NASA MDP is a continually growing dataset of software engineering data. Therefore, the data we were able to obtain for the three projects do not appear to be highly accurate. As we mentioned, there are many requirements with no corresponding modules and, even more concerning, many modules with no corresponding requirements. We believe to have alleviated this problem by limiting our study to only include modules which have identified links to requirements. But as a result, we have models designed for a subset of project data. Further, machine learning algorithms generally perform better on smaller datasets. Therefore, our results may be overly optimistic.

6 Conclusion

In this study, we build defect prediction models using metrics extracted from unstructured textual requirements, the product/module code metrics, and the fused requirement and module metrics. The models were developed and evaluated using metrics data from three NASA MDP projects: CM1, JM1, and PC1. Model comparison shows that requirement metrics can be highly useful for defect prediction. Although the requirement metrics do not predict defects well by themselves, they significantly improve the performance of the prediction models that combine requirement and module metrics together using the inner-join method. Our experiments suggest that the early lifecycle metrics can play an important role in project management, either by pointing to the need for increased quality monitoring during the development or by using the models to assign verification and validation activities.

Another aspect that sets this work apart from other studies is *reproducibility*. Reproducibility is an important

methodological principle in other disciplines since it allows a community to confirm, refute, or even improve prior results. Every experiment we report in this paper can be reproduced. The datasets are publicly available, as well as all the modeling tools. We strongly encourage software engineering researchers to share data with publicly available repositories such as [1], define challenges, and to take the time to reproduce and demonstrate how to improve the results of others.

References

[1] Promise Data Repository, `http://promisedata.org/repository`.

[2] Nasa iv&v facility. metric data program. Available from `http://MDP.ivv.nasa.gov/`.

[3] The r project for statistical computing. available `http://www.r-project.org/`.

[4] S. H. Aljahdali, A. Sheta, and D. Rine. Prediction of software reliability: a comparison between regression and neural network non-parametric models. In *ACS/IEEE International Conference on Computer Systems and Applications*, pages 25–29, June 2001.

[5] N. N. T. Ball and B. Murphy. Using historical data and product metrics for early estimation of software failures. In *Proc. ISSRE , Raleigh, NC*, 2006.

[6] L. Breiman. Random forests. *Machine Learning*, 45:5–32, 2001.

[7] V. U. Challagulla, F. B. Bastani, and I.-L. Yen. A unified framework for defect data analysis using the mbr technique. In *Proc. of the IEEE International Conference on Tools with Artificial Intelligence (ICTAI'06)*, 2006.

[8] C. Drummond and R. C. Holte. C4.5, class imbalance, and cost sensitivity: why under-sampling beats over-sampling. In *Workshop on Learning from Imbalanced Datasets II, Washington DC*, 2003.

[9] C. Drummond and R. C. Holte. Severe class imbalance: Why better algorithms aren't the answer? In *Proc. of the 16th European Conference of Machine Learning,Porto,Portugal*, Oct. 2005.

[10] N. Fenton and N. Ohlsson. Quantitative analysis of faults and failures in a complex software system. *IEEE Transactions on Software Engineering*, pages 797–814, August 2000.

[11] N. E. Fenton and M. Neil. A critique of software defect prediction models. *IEEE Transactions on Software Engineering*, 25(5):675–689, 1999. Available from `http://citeseer.nj.nec.com/fenton99critique.html`.

[12] N. E. Fenton and S. Pfleeger. *Software Metrics: A Rigorous & Practical Approach*. PWS Publishing Company,International Thompson Press, 1997.

[13] L. Guo, Y. Ma, B. Cukic, and H.Singh. Robust prediction of fault-proneness by random forests. In *Proc. of the 15th International Symposium on Software Relaibility Engineering ISSRE'04*, 2004.

[14] M. H. Halstead. *Elements of Software Science*. Elsevier, North-Holland, 1975.

[15] T. Khoshgoftaar. An application of zero-inflated poisson regression for software fault prediction. In *Proceedings of the 12th International Symposium on Software Reliability Engineering, Hong Kong*, pages 66–73, Nov 2001.

[16] Y. Ma and B. Cukic. Adequate and precise evaluation of predictive models in software engineering studies. In *3rd Intl. Workshop on Predictor Models in SE (PROMISE 2007),Minneapolis, MN*, 2007.

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

[18] T. Menzies, J. Greenwald, and A. Frank. Data mining static code attributes to learn defect predictors. *IEEE Transactions on Software Engineering*, January 2007. Available from `http://menzies.us/pdf/06learnPredict.pdf`.

[19] T. Menzies, J. D. Stefano, K. Ammar, K. McGill, P. Callis, R. Chapman, and D. J. When can we test less? In *IEEE Metrics'03*, 2003. Available from `http://menzies.us/pdf/03metrics.pdf`.

[20] U. of Waikato. Weka software package. The University of Waikato, available `http://www.cs.waikato.ac.nz/ml/weka/`.

[21] S. Rakitin. *Software Verification and Validation for Practitioners and Managers, Second Edition*. Artech House, 2001.

[22] N. F. Schneidewind. Investigation of logistic regression as a discriminant of software quality. In *Proceedings of the 7th International Software Metrics Symposium, London*, pages 328–337, April 2001.

[23] M. Shepperd and D. Ince. A critique of three metrics. *The Journal of Systems and Software*, 26(3):197–210, September 1994.

[24] W. Wilson, L. Rosenberg, and L. Hyatt. Automated analysis of requirement specifications. In *ICSE '97*, pages 161–171, May 1997.

[25] W. M. Wilson, L. H. Rosenberg, and L. E.Hyatt. Automated quality analysis of natural language requirement specifications. availabel `http://satc.gsfc.nasa.gov/support/PNSQC_OCT96/pnq.html`.