Predicting Defective Components based on Code Complexity Measures

Hongyu Zhang
School of Software
Tsinghua University
Beijing 100084, China
hongyu@tsinghua.edu.cn

Xiuzhen Zhang
School of CS & IT
RMIT University
Melbourne 3001, Australia
zhang@cs.rmit.edu.au

Ming Gu
School of Software
Tsinghua University
Beijing 100084, China
guming@tsinghua.edu.cn

ABSTRACT

The ability to predict defective modules can help us allocate limited maintenance resources effectively and efficiently. In this paper, we propose a code complexity-based method for predicting defect-prone components. Our method takes three code-level complexity measures as input, namely Lines of Code, McCabe's Cyclomatic Complexity and Halstead's Volume, and classifies components as either defective or non-defective. We perform an extensive study of twelve classification models using the public NASA datasets. Cross-validation results show that our method can achieve good prediction accuracy. Our study confirms that static code complexity measures can be useful indicators of components' quality.

Keywords

Software defect prediction, complexity measures, software metrics, quality indicators

1. INTRODUCTION

Software maintenance and quality assurance are resource and time-consuming activities, which may include manual inspections of design and code, technical review meetings and intensive software testing. Large software systems are usually composed of many components. Applying equal effort to all components is very costly. Knowing which components are more likely to be defective can help us allocate limited resources effectively and efficiently.

It is widely believed that some internal properties of software have a relationship with software quality. Many defect prediction models have been proposed based on the measurement of such properties. For example, Gaffney [4] proposed a LOC (Lines of Code) based model to predict defects; Khoshgoftaar and Seliya [7] proposed prediction models based on 21 static product metrics; Subramanyam and Krishnan [12] provide empirical evidence supporting the role of object-oriented metrics, specifically the Chidamber and Kemerer (CK) metrics [3], in determining software defects.

In this paper, we propose a complexity-based method for predicting defective components. Software complexity is a key property that has been discussed widely in software literature. Lines of Code, McCabe's Cyclomatic Complexity, and Halstead's Volume are static code attributes that are commonly used for measuring code complexity ([15], [13], [6]). We use these three complexity metrics as defect predictors.

To obtain empirical results, we use public domain defect data - the NASA datasets. We perform an extensive study of model construction using twelve different classification techniques (such as Decision Tree, K-nearest Neighbor and Random Forest). Given the measurement data of software complexity, the models classify all components into two classes: defective (with one or more defects) or non-defective (with no defects). We use 10-fold cross-validation to evaluate the prediction performance. The results show that all twelve classification techniques can predict well.

The contributions of this paper are as follows: Firstly, we confirm that software complexity measures, especially the static code complexity measures, can be useful indicators of software quality. Secondly, we show that using classification techniques, we are able to predict defect-prone components based on their complexity with good accuracy.

2. SOFTWARE COMPLEXITY MEASURES

Software complexity has been a subject of considerable research. The term software complexity is often defined as “the difficulty of performing tasks such as coding, debugging, testing and modifying the software” [6]. Software researchers and engineers attempt to know the complexity of the software undertaken quantitatively and to find the relationships between the complexity and the difficulty of development/maintenance task.

Software complexity can be measured at design level and code level. Chidamber and Kemerer [3] proposed a metrics suite for object oriented design. They measure the structural properties of an OO design such as depth of inheritance, number of children, etc. Mahmond and Lai measure [11] the complexity of a UML component specification based on its interfaces, constraints, and interactions. Many researchers have shown that design level complexity measures can be early indicators of software quality ([2], [12]). In this paper, we focus on code complexity only, and show the relationship between code complexity and component quality.

Code-level software complexity can be measured based on program size, structure, or control/data flow ([15], [13], [6]). Many code-level complexity metrics have been proposed over the years [15]. In this study, we choose three commonly used metrics that are independent of programming languages and development methodology, namely Lines of Code, McCabe’s Cyclomatic Complexity, and Halstead’s Volume. Their measurement data can be easily collected using automated tools. Line of Code (LOC) is a size-based complexity metric. It counts each physical source line of code in a program, excluding blank lines and comments. Although some authors have pointed out the deficiencies of LOC, it is still the most commonly used size measure in practices because of its simplicity. Furthermore, LOC is the key input in many software cost and effort estimation models such as COCOMO.

Halstead’s Volume is also a size-based complexity metric proposed by late Maurice Halstead in his software science theory [5]. Software science theory considers a computer program as a series of tokens that is classified as either “operators” or “operands”. The operators include keywords, arithmetic and Boolean operators, and delimiters. The operands include all program variables and constants. A family of software metrics (such as program length, volume, language level, programming effort, etc.) is derived from direct measurement of the number of operators and operands. Halstead’s Volume metric is defines as follows:

$$V = N \log_2 n$$

where N is the total number of tokens in the program (program length), and n is the total number of unique operators and operands (program vocabulary). Since its official publication in 1977, the software science theory has drawn widespread attention from the computer science community, and has grown from measuring the algorithms to measuring entire software systems. A large number of research reports and papers can be found in the literature (for example, a special issue of IEEE transactions on software engineering in March 1979).

McCabe’s cyclomatic complexity metric is based on control flow [9]. MaCabe considers a program as a direct graph (*g*) in which the edges are control flow paths and the nodes are sequential (non-branching) code segments. For most programs, the metric is:

$$V(g) = edges - nodes + 2$$

V(g) is also equal to the number of decisions plus one in a flow graph. McCabe’s complexity metric provides a quantitative measure of the logic complexity of a program. When used in white-box testing, it provides an upper bound on the number of test cases that will be required to guarantee branch coverage of all program statements.

3. DATA COLLECTION AND ANALYSIS

In this research, we use the data from the NASA IV&V Facility Metrics Data Program (MDP) repository. The data is collected from many NASA projects such as flight control, spacecraft instrument, storage management, and scientific data processing. The MDP repository is open for public (available at http://mdp.ivv.nasa.gov/).

We analyzed ten projects, which were developed in C, C++ and Java programming languages. For each project, NASA applied around 38 static product metrics including different size metrics (such as LOC, Lines of Comments, etc.), a whole set of Halstead’s metrics (such as Volume, Length, Effort, etc.), MaCabe’s metrics (such as cyclomatic complexity, essential complexity, design complexity, etc), and miscellaneous code attributes such as parameter_count and branch_count. Additional object-oriented metrics, such as the CK metrics, were collected for a few C++ projects (such as the KC1 project). In this research, we only choose three complexity metrics LOC, V(g) and V as predictors.

The NASA datasets contain software measurement data and associated defect data at the function/method level. We find that each dataset also includes a Product Hierarchy document which describes the function-component relationship in a project. A unique ID is assigned to each component and function. We develop a tool that analyzes the Product Hierarchy document and aggregates the function level data into component level data. We then use the component level data in our model construction. We remove the data points with missing values. Some NASA projects contain reused components, which cause duplicate records in the datasets; therefore we also preprocess the data to remove such duplication. Table 1 shows the component-level modules in NASA datasets. Total 182 data points are gathered, among them 44.51% (81) are defective components (having one or more defects).

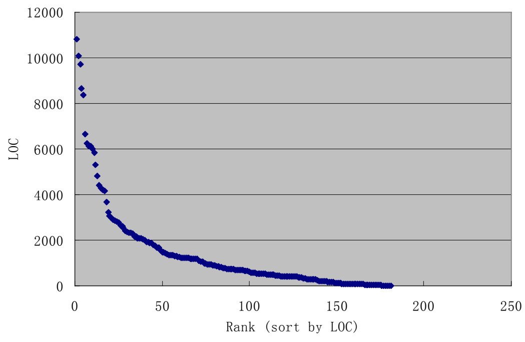
Figure 1 shows the obtained LOC, V(g) and V measures for all components. We can see that the distributions are highly skewed. Most components have small measurement values with a few have large values. The descriptive statistic analysis shows that the means for LOC, V(g) and V are 1462.3, 339 and 36159, respectively (Table 2).

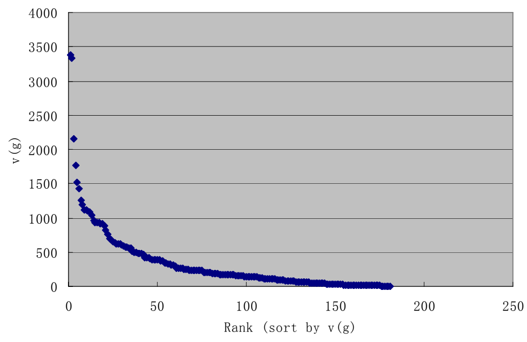
Table 1. The components in NASA MDP datasets

Project	Total LOC	Functions/ Methods	Defective Functions/Methods	Components	Defective Components	Language
CM1	17K	505	48	20	9	C
PC1	26K	1107	76	29	17	C
PC2	25K	5589	23	10	8	C
PC3	36K	1563	160	29	17	C
PC4	30K	1458	178	33	1	C
KC1	43K	2107	325	18	18	C++
MC1	66K	9466	68	36	6	C++
MC2	6K	161	52	1	1	C++
KC3	8K	458	43	5	3	Java
MW1	8K	403	31	1	1	C
Total	**265K**	**22817**	**1004**	**182**	**81**	

Table 2. Descriptive statistics of the data

	#data points	Min	Max	Mode	Mean	Std. Dev.	Skewness	Skewness Std. Error
LOC	182	6	10833	88	1462	2015	2.553	0.18
V(g)	182	2	3374	4	339	495	3.384	0.18
V	182	18	293527	32	36159	51442	2.434	0.18





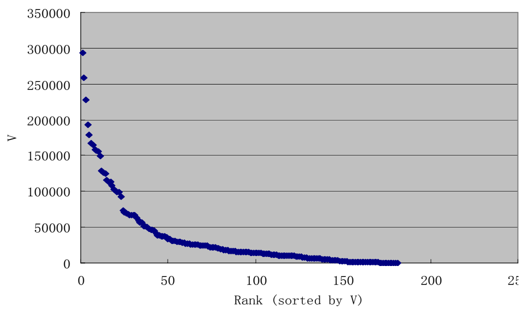


Figure 1: The LOC, V(g) and V measures at component-level

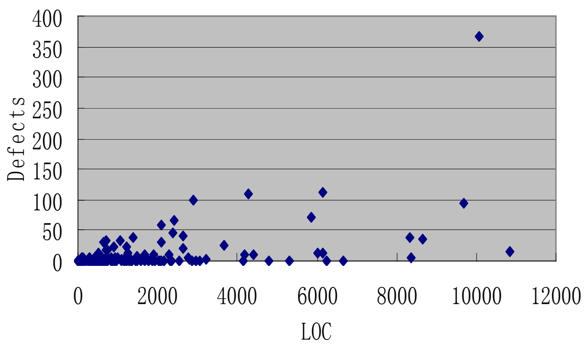


Figure 2: Scatter plot of defects against LOC

Table 3. The correlations among variables (Pearson *r* and Spearman *p* correlations)

	LOC	V(g)	V	#of Defects
LOC	r=1 p=1	-	-	-
V(g)	r=0.878 p=0.926	r=1 p=1	-	-
V	r=0.915 p=0.967	r=0.825 p=0.923	r=1 p=1	-
#of Defects	r=0.521 p=0.494	r=0.296 p=0.351	r=0.502 p=0.445	r=1 p=1

* All correlations are significant at the 0.01 level (2-tailed)

We observe that there is no linear relationship between the complexity metrics and the number of defects. As an example, Figure 2 shows the scatter plot of defects against LOC. Furthermore, from Figure 2 we can see that there is no simple "threshold" size, above which the defect proneness of components increases rapidly. Similar results are also observed between defects and V(g) and V. To statistically verify our observations, we compute the Pearson correlation and the Spearman rank order correlations among variables. The results are shown in Table 3. We can see that the three complexity measures are positively correlated with each other, but none of them is strongly correlated with the number of defects. The absence of simple relationship between complexity measures and defects motivated us to use more advanced data mining techniques to predict defective components.

4. PREDICTING DEFECT-PRONE COMPONENTS

4.1 Accuracy measures

Prediction of defective components can be cast as a classification problem in machine learning. A classification model can be learnt from the training samples of components with labels Defective and Non-defective, the model is then used to classify unknown components.

Table 4. The results of a prediction model

Actual	Predicted: Defective	Predicted: Non-defective
Defective	TP	FN
Non-defective	FP	TN

We denote the defective components as the Positive (P) class and the Non-defective components as the Negative (N) class. A defect prediction model has four results: true positives (TP), false positives (FP), true negatives (TN) and false negatives (FN), as shown in Table 4.

To evaluate the predication model, we use Recall and Precision, which are the accuracy measures widely used in Information Retrieval area. They are defined as follows:

$$\mathrm{Re}\,call = \frac{TP}{TP+FN}, \quad \mathrm{Pr}\,ecision = \frac{TP}{TP+FP}$$

The Recall defines the rate of true defective components in comparison to the total number of defective components, and the Precision relates the number of true defective components to the number of components predicted as defective. A good prediction model should achieve high Recall and high Precision. However, high Recall often comes at the cost of low Precision, and vice versa. Therefore, F-measure is often used to combine Recall and Precision. It is defined as the harmonic mean of Precision and Recall as follows:

$$F-measure = \frac{2\times \mathrm{Re}\,call \times \mathrm{Pr}\,ecision}{\mathrm{Re}\,call + \mathrm{Pr}\,ecision}$$

The values of Recall, Precision and F-measure are between 0 and 1, the higher the better.

We also use the Accuracy metric (*Acc*, or *success rate* as termed in [14]) to complement F-measure to measure the overall accuracy of the prediction. The *Acc* is defined as follows:

$$Acc = \frac{TP+TN}{TP+TN+FP+FN}$$

The *Acc* measure relates the number of correct classifications (true positives and true negatives) to the total number of instances.

4.2 Classifying defective components

In this project, we have chosen twelve commonly used classification techniques as shown in Table 5. All classifiers are supported by WEKA[1], a public domain data mining tool.

Table 5. The classifiers used in this project

Technique	Classifier in WEKA	Description
Bayesian Network	BayesNet	A Bayesian Network based classifier
Bagging	Bagging	A meta-learning algorithm that bags a classifier to reduce variance
k-nearest Neighbor	Ibk	A typical instance-based learning classifier
Random Forest	RandomForest	An ensemble of decision trees, each tree is trained on a bootstrap sample of the given training dataset
Multilayer Perceptron	Multilayer Perceptron	A backpropagation neural network
Logistic Regression	Logistic	A linear logistic regression based classification
RBF Network	RBFNetwork	A type of feedforward network that is based on radial basis functions
Support Vector Machine	SMO	A sequential minimal optimization algorithm for support vector classification
Naive Bayes	NaiveBayes	A standard probabilistic Naive Bayes model
Decision Tree (C4.5)	J48	A C4.5 decision tree learner
K-Star	KStar	An instance based learning algorithm that uses entropy as the distance measure
Boost	AdaBoostM1	The AdaBoost.M1 boosting algorithm

[1] WEKA data mining tool is available at: http://www.cs.waikato.ac.nz/ml/weka/

Before training the prediction models, we firstly use logarithmic filter to transform data *n* into their natural logarithms *ln(n)*, the transformation makes the data range narrower and make it easy for classifiers to learn. We then construct the classification models using measurement data on LOC, V(g) and V.

We use the 10-fold cross-validation to evaluate classification models. The whole training data is partitioned into 10 folds (segments) randomly. Each fold in turn is held as the test data and a classification model is trained on the rest nine folds.

Table 6 shows the 10-folds cross validation results of the defect prediction models constructed using three inputs (LOC, V(g) and V). We can see that all classification techniques obtain good results with Recall ranging from 64.2% to 91.4%, Precision ranging from 58% to 74.6%, F-measure ranging from 0.64 to 0.73, and Acc ranging from 65.9% to 74.7%. The K-Star technique achieves the better overall performance. The models constructed can be used to predict defect-prone components in new projects developed in similar environment.

Table 6. Validation results using LOC, V(g) and V (three inputs)

Classifier	Recall (%)	Precision (%)	F-measure	Acc (%)
Bayesian Network	85.2	62.2	0.72	70.3
Bagging	71.6	63	0.67	68.7
k-nearest Neighbour	72.8	65.5	0.69	70.9
Random Forest	66.7	62.1	0.64	67.0
Multilayer Perceptron	65.4	74.6	0.70	74.7
Logistic Regression	72.8	71.1	0.72	74.7
RBF Network	65.4	63.1	0.64	67.6
Support Vector Machine	64.2	65.0	0.65	68.7
Naive Bayes	85.2	58	0.69	65.9
Decision Tree	91.4	59.7	0.72	68.7
K-Star	79.0	68.1	0.73	74.2
Boost	86.4	58.8	0.70	67.0

Table 7. Validation results using LOC (one input)

Classifier	Recall (%)	Precision (%)	F-measure	Acc (%)
Bayesian Network	88.9	60.0	0.72	68.7
Bagging	71.6	58.0	0.64	64.3
k-nearest Neighbour	54.3	53.7	0.54	58.8
Random Forest	54.3	53.0	0.54	58.2
Multilayer Perceptron	66.7	64.3	0.66	68.7
Logistic Regression	64.2	65.8	0.65	69.2
RBF Network	66.7	65.9	0.66	69.8
Support Vector Machine	65.4	64.6	0.65	68.7
Naive Bayes	79.0	62.7	0.70	69.8
Decision Tree	90.1	60.3	0.72	69.2
K-Star	77.8	64.3	0.70	70.9
Boost	87.7	59.7	0.71	68.1

Table 8. Validation results using V(g) (one input)

Classifier	Recall (%)	Precision (%)	F-measure	Acc (%)
Bayesian Network	80.2	62.4	0.62	56.6
Bagging	54.3	52.4	0.53	57.7
k-nearest Neighbour	48.1	51.3	0.50	56.6
Random Forest	48.1	51.3	0.50	56.6
Multilayer Perceptron	82.7	54.5	0.66	61.5
Logistic Regression	48.1	56.5	0.52	60.4
RBF Network	60.5	52.7	0.56	58.2
Support Vector Machine	45.7	57.8	0.51	61.0
Naive Bayes	67.9	52.9	0.60	58.8
Decision Tree	90.1	52.1	0.66	58.8
K-Star	63.0	55.4	0.59	61.0
Boost	55.6	51.1	0.53	56.6

Table 9. Validation results using V (one input)

Classifier	Recall (%)	Precision (%)	F-measure	Acc (%)
Bayesian Network	85.2	60.0	0.70	68.1
Bagging	66.7	54.5	0.60	60.4
k-nearest Neighbour	48.1	49.4	0.49	54.9
Random Forest	50.6	39.6	0.50	56.0
Multilayer Perceptron	65.4	60.9	0.63	65.9
Logistic Regression	60.5	62.8	0.62	66.5
RBF Network	65.4	62.4	0.64	67.0
Support Vector Machine	66.7	60.0	0.63	65.4
Naive Bayes	82.7	59.8	0.69	67.6
Decision Tree	86.4	59.8	0.71	68.1
K-Star	74.1	59.4	0.66	65.9
Boost	85.2	60.0	0.70	68.1

To compare the performance of the prediction models constructed using different inputs, we also build models for each of the complexity metrics separately. The validation results are shown in Tables 7-9. In average, the models built using one input can predict reasonably well but are less accurate than the three-inputs model.

4.3 Tradeoffs between Recall and Precision

In previous sections, we assume that Recall and Precision are equally important. In practice, however, managers and QA engineers often have preferences between Recall and Precision. For example, when cost is not an issue, Recall is often more important than Precision for mission-critical systems. For less critical systems with very limited budget and tight schedule, Precision is often preferable over Recall.

As introduced in Section 4.1, F-measure is often used to combine Recall and Precision. The equation

$$F-measure = \frac{2 \times \mathrm{Re}call \times \mathrm{Pr}ecision}{\mathrm{Re}call + \mathrm{Pr}ecision}$$

makes an assumption that the Recall and Precision are evenly weighted. A more general form of F-measure is defined as a weighted harmonic mean of Precision and Recall [22]. Let Recall have a weight of *a* (*a* >0) and Precision have a weight of 1, then the F-measure becomes:

$$F-measure_a = \frac{(1+a) \times (\mathrm{Re}call \times \mathrm{Pr}ecision)}{(\mathrm{Re}call + a \times \mathrm{Pr}ecision)}$$

Commonly used *a* values are 0.5 (weighting Precision twice as much as Recall), 1 (weighting Recall and Precision evenly) and 2 (weighting Recall twice as much as Precision).

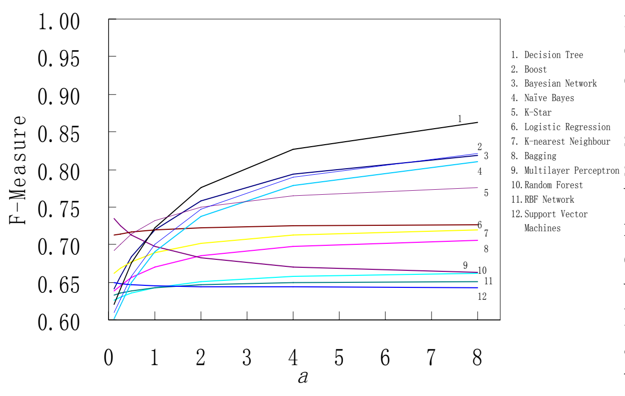


Figure 3: The relations between *a* and F-measure.

For the validation results shown in Table 6, we depict the relationships between the values of *a* and F-measures (Figure 3), from which we can select an optimal classifier when tradeoffs between Recall and Precision have to be made. If Recall is much more important than Precision, Decision Tree techniques can be considered. If Precision is much more important than Recall, then Multilayer Perceptron Network is a choice.

5. DISCUSSIONS AND COMPARISONS

5.1 Work on defect prediction over NASA datasets

Many researchers have constructed defect prediction models based on the NASA MDP data. For example, Menzies et al. [10] performed experiments on function/method level defect prediction on five NASA datasets. They obtained mean probability of detection (i.e. Recall) 36%. Khoshgoftaar and Seliya [7] performed an extensive study on JM1 and KC2 datasets using 25 classification techniques with 21 static code metrics. They also observed low prediction performance, and they did not see much improvement by using different classification techniques. They thus suggested "instead of focusing on searching for another classification technique for improving prediction accuracy, the quality of the software measurement data should be addressed".

We believe that besides the quality of measurement data, another cause of low prediction performance is the large number of small modules. The original NASA data is collected from modules at function/method level, the size and other complexity measures for a single function/method are usually small. The measurements of small modules may show little variations, which makes it difficult for a machine learning technique to distinguish between defective modules and non-defective modules. Also, from Table 1 we can see that the number of defective functions/methods only accounts for 0.44% of total functions/methods, which causes the difficulty of classification learning in the presence of imbalanced class distribution.

Koru and Liu [8] also observed these problems and suggested stratifying datasets according to modules size. For each original dataset, they obtained 15 subsets by recursively partitioning in half after ranking the modules according to their size. They then mixed the data points in each subset randomly for the cross-validation runs. For the five datasets they studied, the F-measures ranged from 0.22 to 0.61. Koru and Liu also proposed to use class-level data whenever possible, the F-measure for predicting defective classes in KC1 dataset was 0.65.

Supervised-learning approaches are difficult if there is no historical defect data available, Zhong et al. [16] proposed a cluster analysis based approach to handle such situations. They analyzed the NASA JM1 and

KC2 datasets and used clustering algorithms to group modules according to 13 software metrics. A human expert then inspected each cluster and labeled it defect-prone or not defect-prone. The reported classification errors (false-positive rate and false-negative rate) were about 20.71% to 33.75%.

In our research, we aggregate function level data to component level, and predict the defect-prone components. We believe that making prediction about components is more meaningful to project managers than about functions, because components are cohesive logical units and QA activities are often organized around components. We use the same three complexity metrics as predictors for all datasets. The number of metrics required is much lesser than the number (typically 21) required by other defect prediction models constructed from the NASA datasets. We tested our model on ten NASA datasets and the prediction accuracy is satisfactory (with F-measures ranging from 0.64 to 0.73).

5.2 Work on quality prediction using static code attributes

It is commonly believed that there are relationships between external software characteristics (e.g., quality) and internal product attributes. Discovering such relationships has become one of the objectives of software metrics [17]. Many researchers believe in the merits of static code metrics and have proposed many quality prediction models based on them (e.g., [4, 5, 7, 8, 10, 18]), whereas some others remain skeptical [19]. Some studies support the common intuition that highly complex modules lead to high number of defects [20], while others disagree [21].

In our experiment, using 12 classification techniques we are able to predict defective components based on static code complexity measures with good accuracy. Therefore, we tend to believe that there is a relationship between complexity and defects, although it may not be a linear relationship.

6. CONCLUSIONS

In this paper, we have presented a method for predicting defect-prone components using three software complexity measures. We have constructed twelve prediction models from the public NASA datasets using twelve different classification techniques. Cross-validation results show that all classification techniques can predict well. Our results confirm that static code complexity measures can be useful indicators of component quality, and that we are able to build effective defect prediction models based on the code complexity measures.

Our defect prediction model is constructed from the NASA MDP data. As Basili et al. [1] argued, the conclusions from NASA data are relevant to the general software engineering industry. This is because NASA and its contactors all follow (and are influenced) by current industrial practices. An important advantage of using NASA MDP data is that it is reproducible. Many researchers have developed prediction models based on organizations' internal data and signed non-disclosure and confidentiality agreements with the organization. Therefore, their results cannot be verified and compared. By using the open NASA data, we hope that other researchers could repeat our experiments and improve our method.

In future, we will further evaluate our defect prediction method in large-scale industrial projects. We will also analyze the impact of component size on defects. We hope our work could contribute to the development of trustworthy software.

ACKNOWLEDGMENTS

We thank Prof. Richard Lai for his comments on this study.

REFERENCES

[1] Basili, V., McGarry, F., Pajerski, R. and Zelkowitz, M., Lessons Learned from 25 Years of Process Improvement: The Rise and Fall of the NASA Software Engineering Laboratory, *Proc. 24th Int'l Conf. Software Eng.* (ICSE), (2002).

[2] Basili, V., Briand, L. and Melo, W., A Validation of Object-Oriented Design Metrics as Quality Indicators, *IEEE Trans. Software Eng.*, vol. 22, no. 10, pp. 751-761, (1996).

[3] S.R. Chidamber and C.F. Kemerer, A Metrics Suite for Object-Oriented Design, *IEEE Trans. Software Eng.*, vol. 20, pp. 476-493, (1994).

[4] Gaffney, J.R., Estimating the Number of Faults in Code, *IEEE Trans. Software Eng.*, vol. 10, no. 4, (1984).

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

[6] Kearney J. et al., Software Complexity Measurement, *Communications of the ACM*, vol. 29, no. 11, (1986).

[7] T.M. Khoshgoftaar and N. Seliya, The Necessity of Assuring Quality in Software Measurement Data, *Proc. 10th Int'l Symp. Software Metrics* (METRICS'04), IEEE CS Press, pp. 119–130, (2004).

[8] A. Koru and Hongfang Liu, Building Effective Defect-Prediction Models in Practice, *IEEE Software*, vol. 22, no.6, pp. 23-29, (2005).

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

[10] T. Menzies, J. DiStefano, A. Orrego, and R. Chapman, Assessing Predictors of Software Defects, *Proc. Workshop Predictive Software Models*, (2004).

[11] Mahmood, S. and Lai, R., A Complexity Measure for UML Component–based System Specification, *Software Practice and Experience*, 36:1-18, Wiley, (2006).

[12] Subramanyam, R. and Krishnan, M.S., Empirical Analysis of CK Metrics for Object-Oriented Design Complexity: Implications for Software Defects, *IEEE Trans. Software Eng.*, 29, 4， pp. 52-74, (2003).

[13] Van Vliet, H., Software Engineering: Principles and Practice, Wiley, New York, (2000).

[14] I.H. Witten and E. Frank, Data Mining: Practical Machine Learning Tools and Techniques with Java Implementation, second ed. Morgan Kaufmann, (2005).

[15] Zuse, H., *Software Complexity: Measures and Methods*, Walter de Gruyter, Berlin, (1990).

[16] Zhong, S., Khoshgoftaar, T., and Seliya, N., 2004. Analyzing Software Measurement Data with Clustering Techniques, *IEEE Intelligent Systems*, March/April 2004, pp. 20-27.

[17] N. Fenton and S. Pfleeger, *Software Metrics: A Rigorous and Practical Approach*, PWS Publishing, (1997).

[18] T. Menzies, J. Greenwald and A. Frank, Data Mining Static Code Attributes to Learn Defect Predictors, *IEEE Trans. Software Eng.*, vol. 32, no. 11, (2007).

[19] M. Shepperd and D. Ince, *Derivation and Validation of Software Metrics, Oxford University Press*, (1993).

[20] J.C. Munson and T.M. Khoshgoftaar, The detection of fault-prone programs, *IEEE Trans. Software Eng.*, 18 (5), pp. 423-433, (1992).

[21] N. Fenton and N. Ohlsson, Quantitative Analysis of Faults and Failures in a Complex Software System, *IEEE Trans. Software Eng.*, 26 (8), pp. 797-814, (2000).

[22] C.J. van Rijsbergen, *Information Retrieval,* Butterworths, London, (1979).