13th IEEE International Symposium on Pacific Rim Dependable Computing

Predicting Defective Software Components from 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
Key Lab for ISS of MOE
Tsinghua University
Beijing 100084, China
guming@tsinghua.edu.cn

Abstract

The ability to predict defective modules can help us allocate limited quality assurance resources effectively and efficiently. In this paper, we propose a 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. This study confirms that static code complexity measures can be useful indicators of component quality.

1. Introduction

Software quality assurance is a resource and time-consuming activity, 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 there are relationships between external software characteristics (e.g., quality) and internal product attributes. Many defect prediction models have been proposed based on the measurement of static code attributes. However, the prediction results are not satisfactory. For example, Khoshgoftaar and Seliya [2] performed an extensive study on NASA datasets using 25 classification techniques with 21 code metrics. They observed low prediction performance, and they did not see much improvement by using different classification techniques. Menzies et al. also had similar results [4]. A recent work of Menzies [5] reported improved probability of detection using the Naïve Bayes classifier with 38 code metrics, but Zhangs [7] pointed out that their model is unsatisfactory when precision is concerned.

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 (LOC), McCabe's Cyclomatic Complexity (V(g)) [3], and Halstead's Volume (V) [1] are static code attributes that are commonly used for measuring code complexity [8]. LOC is a size-based complexity metric. It counts each physical source line of code in a program, excluding blank lines and comments. V is also a size-based complexity metric proposed by Maurice Halstead in his software science theory [1]. V(g) is a structural complexity metric based on program control flow. 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 modules at component level based on their complexity with good accuracy.

2. 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.), McCabe's metrics (such as cyclomatic complexity, essential complexity, design complexity, etc), and miscellaneous code attributes such as parameter_count and branch_count. 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. 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). The descriptive statistic analysis shows that the mean values for LOC, V(g) and V are 1462, 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

3. Predicting Defect-Prone Components

3.1 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.

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 3.

Table 3. The results of a prediction model

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

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:

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

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 [6]) 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.

3.2 Classifying defective components

In this project, we have chosen twelve commonly-used classification techniques as shown in Table 4. More details about these techniques can be found in [6]. All classifiers are supported by WEKA[1], a public domain data mining tool.

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.

Table 4. 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
Neural Network	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/

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 5 shows the 10-fold 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 5. Validation results

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

5. Comparisons and Conclusions

The original NASA data is collected from modules at function/method level. The related work on defect prediction over NASA datasets [2, 4, 5] also predict defective modules at the function/method level. The size and other software measures for a single function/method are usually small and 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 4.4% of total functions/methods, which causes the difficulty of classification learning in the presence of imbalanced class distribution. Therefore, the prediction performances reported by the related work are low.

In our research, we aggregate function-level data into component-level data, and predict the defect-prone components using three complexity measures. 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. The number of metrics required by our method is also much less than the number required by other defect prediction models. We tested our model on ten NASA datasets and the prediction accuracy is satisfactory.

Our results confirm that static code complexity measures can be useful indicators of component quality, and that using classification techniques, we are able to build effective defect prediction models based on the code complexity measures.

Acknowledgments

This work is supported by the China 973 Project 2004CB719406, NSFC 60635020 and the 6th Key Researcher Support Program of Tsinghua University.

References

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

[2] 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, 2004, pp. 119–130.

[3] T. McCabe, A complexity measure, *IEEE Trans. on Software Eng.*, 2(4):308–320, Dec 1976.

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

[5] 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.

[6] I.H. Witten and E. Frank, *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementation*, Morgan Kaufmann, 2005.

[7] H. Zhang and X. Zhang, Comments on "Data Mining Static Code Attributes to Learn Defect Predictors", *IEEE Trans. on Software Eng.*, vol. 33(9), Sep 2007.

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