Quality Technology &
Quantitative Management
Vol. 9, No. 4, pp. 421-433, 2012

QTQM
© ICAQM 2012

Experience in Predicting Fault-Prone Software Modules Using Complexity Metrics

Liguo Yu[1] and Alok Mishra[2]

[1]Computer Science and Informatics, Indiana University South Bend, IN, USA
[2]Department of Computer & Software Engineering, Atilim University, Ankara, Turkey

(*Received* November 2011, *accepted* May 2012)

Abstract: Complexity metrics have been intensively studied in predicting fault-prone software modules. However, little work is done in studying how to effectively use the complexity metrics and the prediction models under realistic conditions. In this paper, we present a study showing how to utilize the prediction models generated from existing projects to improve the fault detection on other projects. The binary logistic regression method is used in studying publicly available data of five commercial products. Our study shows (1) models generated using more datasets can improve the prediction accuracy but not the recall rate; (2) lowering the cut-off value can improve the recall rate, but the number of false positives will be increased, which will result in higher maintenance effort. We further suggest that in order to improve model prediction efficiency, the selection of source datasets and the determination of cut-off values should be based on specific properties of a project. So far, there are no general rules that have been found and reported to follow.

Keywords: Binary logistic regression, complexity metrics, fault-prone software module.

1. Introduction

Software quality analysis and prediction focuses on detecting high-risk fault prone program modules, allowing practitioners to allocate project resources strategically [9, 34]. Through allocating more testing resources on fault-prone modules, we can effectively and extensively test the product. Therefore, predicting fault-prone software modules is an important technique used in reducing test time and test effort and is becoming an important research topic in recent years [7, 15-16, 20, 28-31, 36].

In practice, software engineers employ various methods to identify and revise high-risk or low-quality program modules [6, 9]. Effectiveness of detecting such modules is affected by the software measurements used. In these predictive models, source code metrics could be used as input variables [32, 34]. One of the most commonly used methods in this area is binary logistic regression [2-3, 12-13, 19, 23, 37], which is proven to be a powerful technique in software development and maintenance.

However, most of the previous studies using binary regression analysis focused on identifying candidate complexity metrics and building relation models that are capable of identifying fault-prone software modules. Few efforts are spent on investigating the applicability and usability of prediction models. More specifically, the accuracy of the reported models is usually evaluated using goodness of fit of the model predictions to the same fault data generating the model. Few studies are performed to build cross-project prediction models that are evaluated on different projects with a similar development environment.

The idea of building a cross-project prediction model might seem promising. On the one hand, as the fundamental structures, management, and measurement of two projects could be different, it might not be feasible to build these models. On the other hand, software projects always share some common properties: they are building similar products, targeting similar problems, using similar methods, or following similar development processes. It is reasonable to believe that such cross-project models could be built under certain circumstances, especially if the two projects are within the same application domain. The objective of this study is to build cross-project prediction models and evaluate their predictability. The findings of this study will be helpful to understand if a universal cross project mode could be built.

In this paper, we present a case study of using complexity metrics to build binary regression models to predict fault-prone software modules. Comparing with previous studies, this paper makes the following contributions: (1) we build fault-proneness prediction models based on available projects and assess them on different projects; and (2) we investigate how to efficiently adjust prediction models to detect more fault-prone software modules. The study is performed using publicly available software fault data, which makes the study easy to be reproduced and improved.

The rest of this paper is organized as follows. Section 2 reviews related work of using software complexity metrics to build binary logistic regression models in order to predict fault-prone software modules. Section 3 describes the data source and the mathematical method. Section 4 presents the results and the analysis of this study. Conclusions are presented in Section 5.

2. Related Work

Binary logistic regression analysis has been intensively studied in software engineering field, especially in predicting fault-prone modules. For example, using Mozilla project, Gyimothy *et al.* [8] studied how to build a prediction model for error-prone software classes in open-source projects. Vokac [35] found some design patterns can be used to identify fault-prone classes. Subramanyam and Krishnan [25] performed a study that supports using object-oriented design complexity metrics (Chidamber and Kemerer's CK metrics) in determining software defects.

Most recent studies in this area are reported by Olague's group and Zhou's group. For example, Olague *et al.* [21-22] found that object-oriented complexity metrics, such as Chidamber and Kemerer's CK metrics, Michura's standard deviation method complexity metrics, Bansiya and Davis' quality metrics, and Etzkorn's average method complexity are good candidate predictors of fault-prone classes. Zhou and Leung [38] investigated the usefulness of object-oriented design metrics in predicting fault-proneness. Their study found that the prediction capabilities of the investigated metrics greatly depend on the severity of faults. In another study, Zhou *et al.* [39] described how to correctly interpret the performance of a prediction model based on odds ratio. In particular, they found that odds ratio associated with one standard deviation increase should be used to represent model effectiveness.

The studies close to our work are those intended to improve the applicability of the prediction models. For example, based on cost-effectiveness analysis, Arisholm and Briand [1] proposed a simple and pragmatic method for assessing the effort of using the prediction models. Shatnawi and Li [26] applied prediction models to the Eclipse project and found that although some complexity metrics can predict fault-prone classes, the accuracy of the prediction decreased from release to release, which prevented them from building a model to identify evolving error-prone classes. Briand *et al.* [2] studied the applicability of using prediction models generated from one Java project to predict fault-prone classes in another Java project. Their study showed that a model built on one system can be accurately used to evaluate classes in another system according to their fault-proneness. Menzies *et al.* [17] studied the performance ceiling of prediction models, i.e., some inherent upper bound on the amount of information offered by complexity metrics in identifying fault-prone modules. They proposed using case-based reasoning when applying prediction models.

As described before, none of the reported research has focused on evaluating predicting models across projects. However, in reality, a software practitioner is interested in knowing, among a given set of software metrics, which ones to use and how to get optimal defect prediction results. This is the objective of the present study.

3. Data Description and Mathematical Method

The data used in this study is retrieved from online public repository PROMISE [4]. The original data is donated by Software Research Laboratory of Bogazici University, Istanbul, Turkey [27]. Five data sets are utilized (AR1, AR3, AR4, AR5, and AR6). These are embedded software products implemented in C. Each dataset contains the measurements of 29 static code attributes (complexity metrics) and 1 defect information (false/true) of tens to hundreds of modules. Module attributes were collected using Prest Metrics Extraction and Analysis Tool [24, 33].

Some of the 29 metrics, such as number of lines of blank code and number of lines of comment code, have no apparent relation with module faults. They are not included in this study. Accordingly, 24 metrics are initially chosen to be used in this study. They are described in Table 1. The detailed descriptions of these metrics can be found in [18].

Table 1. Software module attributes (complexity metrics) used in this study [18].

Line of Code (LOC) Metrics	Halstead Metrics	McCabe Metrics	Miscellaneous Metrics
• Total LOC • Executable LOC	• Length • Volume • Level • Difficulty • Programming effort • Error estimate • Programming time	• Cyclomatic complexity • Cyclomatic density • Decision density • Design complexity • Design density • Normalized cyclomatic complexity	• Unique operands • Unique operators • Total operands • Total operators • Branch count • Call pairs • Condition count • Multiple condition count • Formal parameters

Logistic regression analysis is a method to predicting a categorical variable from a set of predictor variables. In this study, it is used to analyze the relations between software module attributes (complexity metrics) and module defect information. More specifically, the predicted dependent variable has a binary value, true or false, where true indicates faults and false indicates no faults. The independent variables are module attributes (complexity metrics) which have continuous numerical values, such as those listed in Table 1.

Assume Y is the dependent variable and value 1 represents the corresponding module has fault and value 0 representing the corresponding module has no fault. Also assume $X_1, X_2, \ldots, X_n$ are independent variables and $\Pr(Y=1 \mid x_1, x_2, \ldots, x_n)$ represents the probability that $Y=1$ when $X_1=x_1$, $X_2=x_2$, …, $X_n=x_n$. Accordingly, binary logistic regression analysis can generate the following model [39].

$$\hat{Y}=\Pr(Y=1|x_1,x_2,\ldots,x_n)=\frac{e^{a+b_1x_1+\ldots+b_nx_n}}{1+e^{a+b_1x_1+\ldots+b_nx_n}}. \quad (1)$$

The right side expression of Equation (1) will generate a real number. If its value is greater than or equal to 0.5, $\hat{Y}=1$; If its value is less than 0.5, $\hat{Y}=0$. Here, 0.5 is called the *cut-off* value. Using different *cut-off* values can give us different prediction results. It can be seen that lowing the *cut-off* value can result in high value of $\hat{Y}$, increasing the *cut-off* value can result in low value of $\hat{Y}$. It has been a common practice to adjust *cut-off* values in order to achieve different prediction results [14].

In Equation (1), a, b_1, b_2, …, and b_n are coefficients of the independent variables. In statistics, the odds are expressed as the ratio of the probability an event that will happen over the probability the event won't happen. Therefore, we can derive the following odds model.

$$ODDS(Y=1|x_1,x_2,\ldots,x_n)=\frac{\hat{Y}}{1-\hat{Y}}=e^{a+b_1x_1+\ldots+b_nx_n}. \quad (2)$$

Our regression model will be predicting the *logit*, that is, the natural log of the odds a module having faults or not. That is,

$$\ln(ODDS)=a+b_1x_1+\ldots+b_nx_n. \quad (3)$$

4. Analysis and Results

Twenty-four module attributes (complexity metrics) are used in this study. Not all of them are necessary to yield acceptable predictability. Therefore, our analysis is divided into two steps. The first step is to determine the module attributes (complexity metrics) that can yield acceptable predictability. The second step is using the selected module attributes (complexity metrics) to build prediction models. These two steps are presented in the following two subsections.

4.1. Selecting Independent Variables

For each data set (RA1, RA3, RA4, RA5, and RA6), we performed score tests to determine whether each of the 24 attributes would be significant predictors in the binary regression model. In these tests, a single attribute model (which includes intercept of course), $\ln(ODDS)=a+b_1x_1$, is compared with a null model, $\ln(ODDS)=a$, which has no predictors but just the intercept. The result of the score tests can tell if adding a predictor (attribute, independent variable) to a null model can significantly change the fitness of the prediction model. The results of the score tests are summarized in Table 2, where it shows the significance of the predicator in a single attribute model in each data set.

In Table 2, the significance values that are at the 0.01 level are highlighted. In these 24 attributes, 4 attributes (*branch_count*, *condition_count*, *multiple_condition_count*, and *cyclomatic_complexity*) are significant predicators in all five datasets. They are from two categories in Table 1 (McCabe Metrics and Miscellaneous Metrics) and accordingly chosen as the independent variables. To make the dependent variables represent different properties of a software product, two other attributes from categories Line of Code Metrics and Halstead Metrics are determined to be chosen as the independent variables. Physical line of code (*executable_loc*) is one of the most commonly used metrics to measure software complexity, and estimated number of delivered bugs (*halstead_error*) reflects the combined effects of other Halstead metrics. Accordingly, they are also chosen as the additional independent variables.

Therefore, in this study, 6 module attributes (complexity metrics) are used as the independent variables, $X_1, X_2, \ldots, X_6$. They are bolded in Table 2. Function (3) can accordingly be modified as the following, where $x_1, x_2, \ldots,$ and x_6 are the measurements of the attributes of six metrics: $X_1, X_2, \ldots, X_6$ (*branch_count*, *condition_count*, *multiple_condition_count*, *cyclomatic_complexity*, *executable_loc*, and *halstead_error*).

$$\ln(ODDS) = a + b_1x_1 + b_2x_2 + \ldots + b_6x_6 \tag{4}$$

Equation (4) is the specific binary logistic model we are going to build in this study. It shows the relation between the odds a module having faults and the six measurements of a module's properties. In the following of this section, we will use existing data to build the models represented in Equation (4), i.e., to determine the model coefficients $(a, b_1, b_2, \ldots, b_6)$ and evaluate model predictability.

Table 2. Summary of score tests.

Metrics	AR1	AR3	AR4	AR5	AR6
total_loc	.034	.000	.000	.000	.059
executable_loc	.021	.000	.000	.000	.029
unique_operands	.103	.000	.000	.000	.004
unique_operators	.000	.001	.000	.000	.013
total_operands	.010	.000	.000	.000	.211
total_operators	.010	.000	.000	.000	.016
halstead_length	.010	.000	.000	.000	.110
halstead_volume	.011	.000	.000	.000	.081
halstead_level	.028	.066	.002	.022	.001
halstead_difficulty	.000	.000	.000	.007	.809
halstead_effort	.001	.000	.000	.003	.304
halstead_error	.013	.000	.000	.000	.001
halstead_time	.001	.000	.000	.003	.305
branch_count	.003	.000	.000	.000	.000
call_pairs	.191	.459	.001	.119	.018
condition_count	.003	.000	.000	.000	.001
multiple_condition_count	.002	.000	.001	.000	.000
cyclomatic_complexity	.003	.000	.000	.000	.001
cyclomatic_density	.388	.854	.714	.185	.000
decision_density	.038	.405	.026	.334	.392
design_complexity	.191	.459	.001	.119	.207
design_density	.959	.475	.638	.161	.219
normalized_cyclomatic_complexity	.820	.879	.329	.591	.900
formal_parameters	.988	.220	.515	.181	.126

4.2. Building and Assessing Prediction Models

There are two ways to build and evaluate fault prediction models: self-assessment and forward assessment. There are five datasets (AR1, AR3, AR4, AR5, and AR6) in this study. Figure 1(a) illustrates the self-assessment model, where one prediction model is built on each dataset and is evaluated on the same dataset. For example, the prediction model based on dataset AR1 is evaluated on dataset AR1; the prediction model based on dataset AR3 is evaluated on dataset AR3, and so on. Figure 1(b) illustrates the forward assessment model, where prediction models are built based on one or more datasets and evaluated on a different dataset. For example, the prediction model based on dataset AR1 is evaluated on dataset AR3; the prediction model based on datasets AR1 and AR3 is evaluated on dataset AR4, and so forth.

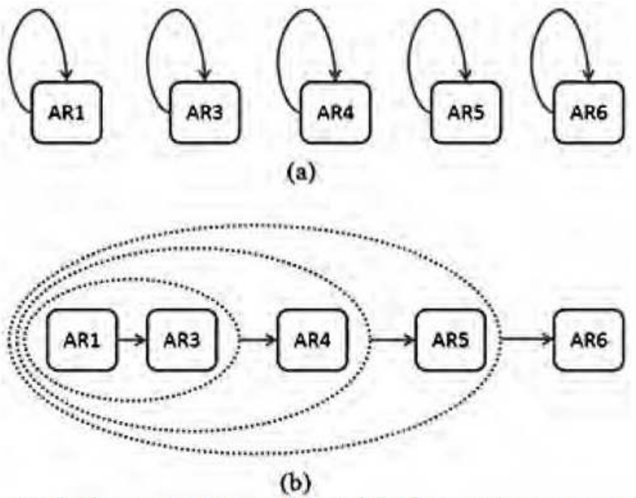


Figure 1. (a) Self-assessment model and (b) forward assessment model.

4.2.1. *Self-Assessment*

In self-assessment, each dataset (AR1, AR3, AR4, AR5, and AR6) is used as the source data to build a prediction model. Accordingly, five logistic models represented in Equation (4) are built. Table 3 summarizes the model parameters, where parameters a, b_1, b_2, ..., and b_6 are *constant*, *branch_count*, *condition_count*, *multiple_condition_count*, *cyclomatic_complexity*, *executable_loc*, and *halstead_error*, *respectively*.

Table 3. The parameters of logistic self-assessment models.

Model number	Data source	Model parameters						
		a	b_1	b_2	b_3	b_4	b_5	b_6
S1	AR1	-4.027	-0.726	0.380	1.162	1.458	-0.182	21.111
S2	AR3	-3.245	0.367	-0.134	-0.643	-0.580	-0.028	6.530
S3	AR4	-3.913	0.557	-1.306	0.091	0.015	0.062	3.892
S4	AR5	-10.602	-2.890	0.255	5.574	5.631	0.027	5.788
S5	AR6	-2.383	-0.268	1.108	-0.902	-0.013	-0.050	0.016

These five modes are then evaluated using the same dataset generating the model. Using a model to evaluate each module, we can obtain two measurements of dependent variable Y : predicted and observed. In binary logistic regression, dependent variable Y has two values (in both observation and prediction): positive (1), which indicates defect, and negative (0), which indicates defect-free. The cross-analysis of predictions against observations can divide data into four categories, as shown in Table 4 [10].

Table 4. Evaluation of prediction against observation [10].

		Predicted	
		Positive (1)	**Negative (0)**
Observed	**Positive (1)**	**tp** (true positive)	**fn** (false negative)
	Negative (0)	**fp** (false positive)	**tn** (true negative)

Predication accuracy, *prediction precision*, and *recall rate* are commonly used metrics to evaluate the binary prediction models. They are defined in Equations (5), (6), and (7) [10]. *Prediction accuracy* describes the general prediction power of a model; *prediction precision* is used to evaluate the correctness of positive signal predictions; and *recall rate* is used to evaluate the prediction power for positive signals. In software engineering field, detecting fault is considered most important, therefore, *recall rate* has been considered to be the most important metric.

$$Prediction\ accuracy = \frac{tp + tn}{tp + tn + fn + fp}. \tag{5}$$

$$Prediction\ accuracy = \frac{tp}{tp + fp}. \tag{6}$$

$$Recall\ rate = \frac{tp}{tp + fn}. \tag{7}$$

Table 5 summarizes the results of self-assessment of the prediction models. It can be seen, except AR1, the other four models yield near 50% recall rate. However, as we discussed in Section 1, these models are built and evaluated using the same dataset. They might not be feasible for real applications. Therefore, there is a need to perform forward assessment of the prediction models.

Table 5. Self-assessment of prediction models.

Model number	S1	S2	S3	S4	S5
Dataset (source and evaluating)	AR1	AR3	AR4	AR5	AR6
Number of datasets	121	63	107	36	101
Prediction accuracy	93%	95%	88%	94%	90%
Prediction precision	50%	86%	77%	100%	77%
Recall rate	11%	75%	50%	75%	47%

4.2.2. Forward Assessment

Following the scheme described in Figure 1(b), we build four forward-assessment models. Accordingly, four logistic models represented in Equation (4) are built. Table 6 summarizes the model parameters, where parameters a, b_1, b_2, …, and b_6 are *constant*, *branch_count*, *condition_count*, *multiple_condition_count*, *cyclomatic_complexity*, *executable_loc*, and *halstead_error*, respectively. It should be noted that Model F1 is same as model S1, because they are generated from the same dataset (AR1). Different from self-assessment models (S1 to S5), forward-assessment models (F1 through F4) are generated using aggregate datasets.

Table 6. The parameters of logistic forward-assessment models.

Model number	Data source	Model parameters						
		a	b_1	b_2	b_3	b_4	b_5	b_6
F1	AR1	-4.027	-0.726	0.380	1.162	1.458	-0.182	21.111
F2	AR1, AR3	-2.857	0.159	-0.001	-0.376	-0.226	-0.013	1.846
F3	AR1, AR3, AR4	-3.497	-0.154	-0.191	0.449	0.487	0.025	1.418
F4	AR1, AR3, AR4, AR5	-3.687	-0.268	-0.107	0.601	0.642	0.022	1.958

These four forward-assessment models are then evaluated on Datasets AR3, AR4, and AR5, and AR6, respectively. Table 7 summarizes the evaluation of four prediction models. In Table 7, each latest model is supposed to outperform the previous models, because of the increase of the size of source dataset. From Table 7, we can see that the *prediction accuracy* increases from Model F1 to Model F4. However the *prediction precision* and the *recall rate* are not in increasing trend.

Table 7. Forward assessment of prediction models (*cut-off* value is 0.5).

Model number	F1 (S1)	F2	F3	F4
Source dataset	AR1	AR1, AR3	AR1, AR3, AR4	AR1, AR3, AR4, AR5
Evaluating dataset	AR3	AR4	AR5	AR6
Prediction accuracy	76%	82%	83%	87%
Prediction precision	32%	100%	75%	100%
Recall rate	75%	5%	38%	13%

It should be noted here that as more data are included in the source dataset, we should expect the models yield better predictions. However, in Table 7, only the *prediction accuracy* is improved through adding more data; the *prediction precision* and the *recall rate* show no clear trend. The reason for this behavior of these forward-assessment models could be due to the differences among these projects. These differences could be design difference, measurement difference, management difference, or something else. On the other hand, these projects must share some common properties, as the *prediction accuracy* improves with more datasets being added.

Furthermore, if we compare Table 6 and Table 7, we can see that if models are self-evaluated on the source dataset generating the model, the model performance (*prediction accuracy*, *prediction precision*, and *recall rate*) is better than forward-evaluation. However in practice, self-assessment is less useful than forward assessment. Therefore, our goal is to improve the performance of forward prediction models. Specifically, we would like to improve the *recall rate*, which can help us identify more error-prone modules.

As described before, in Equation (1), $\hat{Y}$ has a value between 0 and 1 and represents the probability a fault can occur. When this model is used, a *cut-off value* should be assigned. If $\hat{Y}$ is less than the *cut-off value*, it will be considered as 0; if $\hat{Y}$ is greater than the *cut-off value*, it will be considered as 1. Conventionally, the *cut-off value* is set to be 0.5, which means, if $\hat{Y}$ is less than 0.5, we will predict no fault in the software module and if $\hat{Y}$ is greater than 0.5, we will predict fault in the software module.

If we would like to identify more true positive modules, we need to use a lower *cut-off value*. Following this scheme, *cut-off values* of 0.2 and 0.1 are utilized to evaluate models F1 through F4. The results are summarized in Table 8 and Table 9.

Table 8. Forward assessment of prediction models (*cut-off* value is 0.2).

Model number	F1	F2	F3	F4
Source dataset	AR1	AR1, AR3	AR1, AR3, AR4	AR1, AR3, AR4, AR5
Evaluating dataset	AR3	AR4	AR5	AR6
Accuracy	73%	83%	81%	87%
Precision	30%	75%	54%	100%
Recall	88%	15%	88%	13%

Table 9. Forward assessment of prediction models (*cut-off* value is 0.1).

Model number	F1	F2	F3	F4
Source dataset	AR1	AR1, AR3	AR1, AR3, AR4	AR1, AR3, AR4, AR5
Evaluating dataset	AR3	AR4	AR5	AR6
Prediction accuracy	65%	84%	72%	78%
Prediction precision	25%	62%	43%	23%
Recall rate	88%	40%	88%	20%

From Table 8 and Table 9, we can see that through lowering the cut-off value, we can improve the recall rate. For example, in Model F2, the recall rate is improved from 5% to 15% and 40%, respectively when the cut-off value is lowered from 0.5 to 0.2 and 0.1. It should be noted that although lowering the cut-off value can improve the recall rate, it will reduce the prediction accuracy and the prediction precision, which will increase the testing effort. However in some fields, such as safety-critical applications and mission critical applications, detecting errors is more important than reducing the testing cost, it is therefore reasonable to use lower cut-off values in order to detect all error-prone modules.

Based on previous analysis, we can see that balancing of prediction precision and recall rate is the balancing of effort and risk. A higher prediction precision means lower testing effort in software development and a higher recall rate means lower risk of faults in delivered products. Therefore, the selection of cut-off values is not just a technical issue, but a management issue. It depends on the budget, the available resource, and the quality requirement of the product.

Another way to systematically evaluate the effect of different cut-off values is to draw receiver operating characteristic curves [11]. Accordingly, the metric fall out is defined.

$$Fall\ out = \frac{fp}{fp + tn}. \tag{8}$$

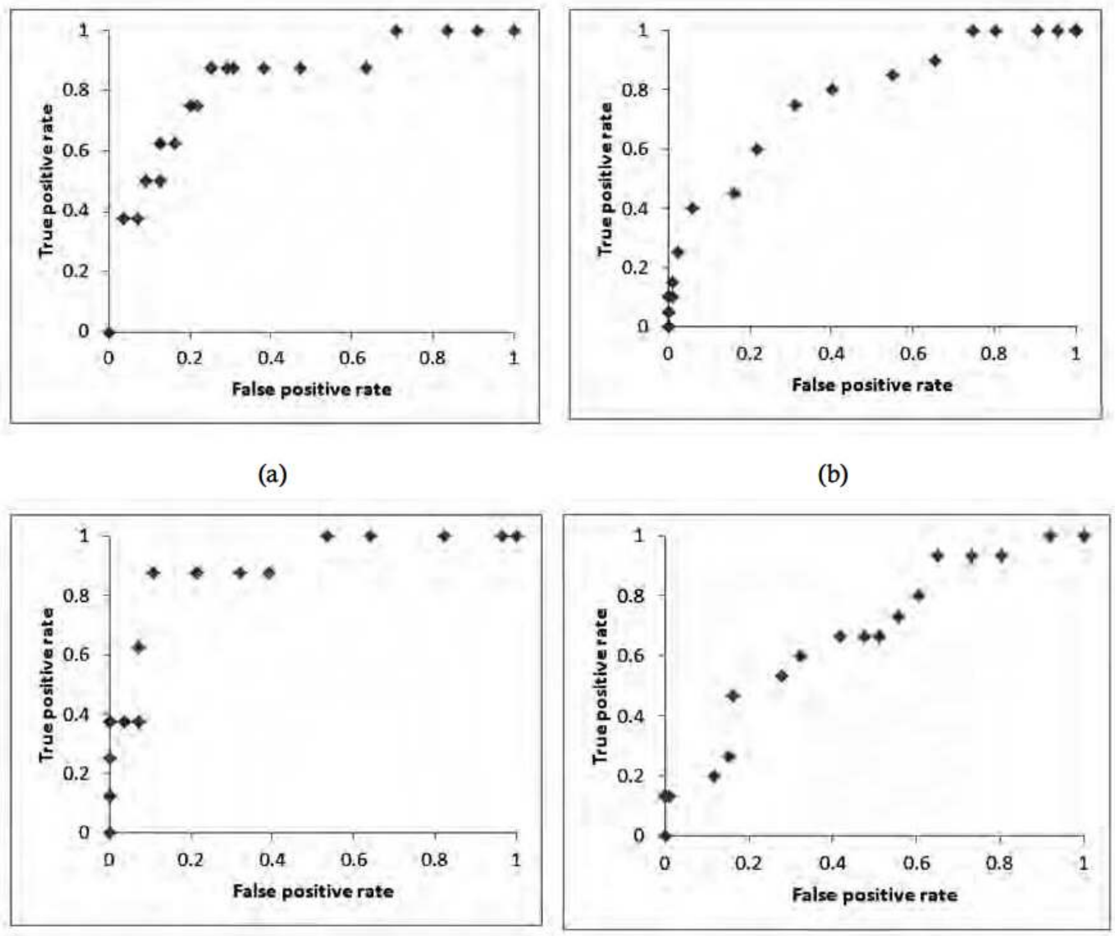
In Equation (8), fp and tn are defined in Table 4. *Fall out* is also called false positive rate (*FPR*). Correspondingly, *recall rate* is also called true positive rate (*TPR*). A receiver operating characteristic (ROC) curve, or simply a ROC curve, is a graphical plot of *true positive rate* (*TPR*), vs. *false positive rate* (*FPR*), for a binary classifier system as its *cut-off value* is varied.

Figure 2 shows the receiver operating characteristic (ROC) curves of Models F1 through F4. It can be seen that these curves have a normal behavior: the *true positive rate* has a positive relation with the *false positive rate*. In other words, increasing of *true positive rate* could result in the increasing of *false positive rate*. An optimum *cut-off value* could be obtained by examining the *precision accuracy*, which balances the *true positive rate* and the *false positive rate*. Table 10 shows the optimum *cut-off values* of Models F1 through F4.

In Table 10, the cut-off values are those that can generate the largest prediction accuracies. In all four models, through using different optimum cut-off values, we can achieve about 88% prediction accuracies. Also, it is interesting to see that the optimum cut-off values vary dramatically from one model to another model. Therefore, in real-world applications, the optimum cut-off values should be calibrated for each different project.

Table 10. Optimum *cut-off* values of forward assessment models.

Model number	F1	F2	F3	F4
Optimum cut-off value	0.9999	0.1000	0.2500	0.2000
False positive rate	0.0364	0.0574	0.1071	0.0
True positive rate	0.3750	0.4000	0.8750	0.1333
Largest prediction accuracy	0.8889	0.8411	0.8889	0.8713



(a)

(b)

(c)

(d)

Figure 2. ROC curves of forward-assessment Model (a) F1; (b) F2; (c) F3; and (d) F4.

4.3. Discussions

As described in the Introduction, cross-project prediction models can improve software product quality and software process quality. Our study shows that it is possible to apply a model generated using the data from one project to predict faults in another project. This is reflected in the increasing of prediction accuracy of our models if more datasets are employed. However, our study also found prediction precision and recall rate are not improved under larger dataset. Although these observations might be specific to this study, the non-monotonicity of the prediction accuracy, prediction precision, and recall rate in these experiments seem to indicate a fundamental lack of universality of the cross-project prediction models. To further prove our observations, more studies are needed in the future.

5. Conclusions

In this paper, we presented our experience of using complexity metrics to predict error-prone software modules. We found (1) forward assessment of prediction models usually has low performance compared to self-assessment, although it is a more realistic measure of the prediction power of a binary logistic regression model; and (2) lowering cut-off values could be a practical technique in revealing more true positive modules, although it might generate additional false positive signals, which will result in lower prediction accuracy, lower prediction precision, and higher testing effort.

Based on our study, we suggest that in order to improve the model prediction efficiencies, the selection of source dataset and the determination of cut-off values should be based on specific properties of a project. For example, if reducing cost of testing is the major objective of a project, a regular cut-off value (0.5) or a higher cut-off value could be used to improve the prediction accuracy; if uncovering the faulty modules is the major objective of the project, a lower cut-off value should be used to improve the recall rate. Currently, there exists no such a general rule that we can follow to guarantee detecting most of the fault-prone modules while reducing the test effort.

Acknowledgements

This study is partially supported by the Faculty Research Grant of Indiana University South Bend.

References

1. Arisholm, E. and Briand, L. C. (2006). Predicting fault-prone components in a Java legacy system. *Proceedings of the 5thACM-IEEE International Symposium on Empirical Software Engineering*, 8-17. Rio de Janeiro, Brazil.

2. Briand, L. C., Wust, J. and Lounis, H. (2001). Replicated case studies for investigating quality factors in object-oriented designs. *Empirical Software Engineering,* 6(1), 11-58.

3. Basili, V. R., Briand, L. C. and Melo, W. L. (1996). A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering,* 22(10), 751–761.

4. Boetticher, G., Menzies, T. and Ostrand, T. (2007). *PROMISE Repository of empirical software engineering data,* http://promisedata.org/ repository, Department of Computer Science, West Virginia University.

5. Briand, L. C., Melo, W. L. and Wust, J. (2002). Assessing the applicability of fault-proneness models across object-oriented software projects. *IEEE Transactions on Software Engineering,* 28 (7), 706-720.

6. D'Ambros, M., Lanza, M. and Robbes, R. (2011). Evaluating defect prediction approaches: a benchmark and an extensive comparison. *Empirical Software Engineering*, 17(4-5), 531-577.

7. Graves, T. L., Karr, A. F., Marron, J. S. and Siy, H. (2000). Predicting fault incidence using software change history. *IEEE Transactions on Software Engineering*, 26(7), 653-661.

8. Gyimothy, T., Ferenc, R. and Siket, L. (2005). Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Transactions on Software Engineering,* 31(10), 897-910.

9. Gao, K., Khoshgoftaar, T. M. and Seliya, N. (2011). Predicting high-risk program modules by selecting the right software measurements. *Software Quality Journal*, 20(1), 3-42.

10. http://en.wikipedia.org/wiki/Precision_and_recall

11. http://en.wikipedia.org/wiki/Receiver_operating_characteristic

12. Janes, A., Scotto, M., Pedrycz, W., Russo, B., Stefanovic, M. and Succi, G. (2006). Identification of defect-prone classes in telecommunication software systems using design metrics. *Information Sciences*, 176(24), 3711-3734.

13. Kanmani, S., Uthariaraj, V. R., Sankaranarayanan, V. and Thambidurai, P. (2007). Object oriented software fault prediction using neural networks. *Information and Software Technology*, 49(5), 483-492.

14. Kleinbaum, D. and Klein, M. (2002). *Logistic Regression-A Self-Learning Text*, Second Edition, Springer-Verlag New York, Inc.

15. Kastro, Y. and Bener, A. (2008). A defect prediction method for software versioning. *Software Quality Journal*, 16 (4), 543-562.

16. Livshits, B. and Zimmermann, T. (2005). DynaMine: finding common error patterns by mining software revision histories. *ACM SIGSOFT Software Engineering Notes*, 30(5), 296-305.

17. Menzies, T., Turhan, B., Bener, A., Gay, G., Cukic, B. and Jiang, Y. (2008). Implications of ceiling effects in defect predictors. *Proceedings of the 4th International Workshop on Predictor Models in Software Engineering*, 47-54. Leipzig, Germany.

18. Metrics Data program, NASA IV&V Facility, http://mdp.ivv.nasa.gov/repository.html

19. Nagappan, N., Ball, T. and Zeller, A. (2006). Mining metrics to predict component failures. *Proceedings of the 28th International Conference on Software Engineering*, 452-461. Shanghai, China.

20. Ostrand, T. J., Weyuker, E. J. and Bell, R. M. (2005). Predicting the location and number of faults in large software systems. *IEEE Transactions on Software Engineering*, 31(4), 340-355.

21. Olague, H. M., Etzkorn, L. H., Messimer, S. L. and Delugach, H. S. (2008). An empirical validation of object-oriented class complexity metrics and their ability to predict error-prone classes in highly iterative, or agile, software: a case study. *Journal of Software Maintenance and Evolution: Research and Practice*, 20(3), 171-197.

22. Olague, H. M., Etzkorn, L. H., Gholston, S. and Quattlebaum, S. (2007). Empirical validation of three software metrics suites to predict fault-proneness of object-oriented classes developed using highly iterative or agile software development processes. *IEEE Transactions on Software Engineering,* 33(6), 402-419.

23. Pai, G. J. and Dugan, J. B. (2007). Empirical analysis of software fault content and fault proneness using Bayesian methods. *IEEE Transactions on Software Engineering,* 33(10), 675-686.

24. Prest Metrics Extraction and Analysis Tool, http://softlab.boun.edu.tr/?q=resources&i=tools.

25. Subramanyam, R. and Krishnan, M.S. (2003). Empirical analysis of CK metrics for object oriented design complexity: implications for software defects. *IEEE Transactions on Software Engineering*, 29(4), 297-310.

26. Shatnawi, R. and Li, W. (2008). The effectiveness of software metrics in identifying

error-prone classes in post-release software evolution process. *Journal of Systems and Software,* 81(11), 1868-1882.

27. Software Research Laboratory, Bogazici University, http://softlab.boun.edu.tr.

28. Turhan, B. and Bener, A. (2009). Analysis of Naive Bayes' assumptions on software fault data: an empirical study. *Data and Knowledge Engineering Journal*, 68 (2), 278-290.

29. Turhan, B., Bener, A. and Kocak, G. (2009). Data mining source code for locating software bugs: a case study in telecommunication industry. *Expert Systems with Applications*, 36 (6), 9986-9990.

30. Tosun, A., Turhan, B. and Bener, A. (2010). Ensemble of software defect predictors: a case study. *Proceedings of the 2nd International Symposium on Empirical Software Engineering and Measurement*, 318-320. Bolzano/Bozen, Italy.

31. Turhan, B. and Bener, A. (2007). A multivariate analysis of static code attributes for defect prediction. *Proceedings of the 7th International Conference on Quality Software*, 231-237. Portland, Oregon, USA.

32. Turhan, B. (2011). On the dataset shift problem in software engineering prediction models. *Empirical Software Engineering*, 2011.

33. Tosun, A., Bener, A., Turhan, B. and Menzies, T. (2010). Practical considerations in deploying statistical methods for defect prediction: A case study within the Turkish telecommunications industry. *Information and Software Technology,* 52(11), 1242-1257.

34. Vivanco, R., Kamei, Y., Monden, A., Matsumoto, K. and Jin, D. (2010). Using Search-Based Metric Selection and Oversampling to Predict Fault Prone Modules. *The 23rd Canadian Conference Electrical and Computer Engineering*, 1-6.

35. Vokac, M. (2004). Defect frequency and design patterns: an empirical study of industrial code. *IEEE Transactions on Software Engineering,* 30(12), 904-917.

36. Williams, C. C. and Hollingsworth, J. K. (2005). Automatic mining of source code repositories to improve bug finding techniques. *IEEE Transactions on Software Engineering*, 31(6), 466-480.

37. Yu, P., Systa, T. and Muller, H. (2002). Predicting fault-proneness using OO metrics: an industrial case study. *Proceedings of the 6th European Conference on Software Maintenance and Reengineering*, 99-107. Budapest, Hungary.

38. Zhou, Y and Leung, H. (2006). Empirical analysis of object-oriented design metrics for predicting high and low severity faults. *IEEE Transactions on Software Engineering,* 32(10), 771-789.

39. Zhou, Y., Xu, B. and Leung, H. (2010). On the ability of complexity metrics to predict fault-prone classes in object-oriented systems. *Journal of Systems and Software,* 83 (4), 660-674.

Author's Biographies:

Liguo Yu received the PhD degree in computer science from Vanderbilt University. He is an Associate Professor of computer science at Indiana University South Bend. His research concentrates on software measurement, software dependency, and open-source software development.

Alok Mishra is a Professor of Department of Computer & Software Engineering, Atilim University, Turkey. His research area includes Software Process, Software Quality Assurance, Agile Methodologies, and Software Engineering Education.