Software Bug Prediction Based on Semi-definite Logistic Regression Model

Tadashi Dohi, Jingchi Wu, and Hiroyuki Okamura

Graduate School of Advanced Science and Engineering, Hiroshima University
Higashi-Hiroshima 739-8527, Japan
email: {dohi, d220580, okamu}@hiroshima-u.ac.jp

***Abstract*—In software bug prediction to identify bug-prone modules, several machine learning techniques have been used in past. However, it has been known that almost all of them were not *explainable* and could not be applied to the program understanding, because the contributions of software metrics were unclear in such black box techniques. In this article, we aim at overcoming the problems in an explainable logistic regression model, called *multicollinearity* and *interaction*, and apply the semi-definite logistic regression model to identify software bug-prone modules. More specifically, we use three actual software development project data sets to evaluate the F-score as well as precision and recall, and compare our semi-definite logistic regression model with the classical logistic one, in terms of the predictive performance of software bug-prone modules. It is shown that our semi-definite logistic regression model involves the common logistic regression model as a special case and can improve the predictive performances on the F-score.**

Keywords-software bug prediction; bug-prone module; logistic regressions; semi-definite programming; discrimination problem; F-score.

I. INTRODUCTION

In testing and maintenance phases of software development, identification of software bug-prone modules containing bugs is crucial for both localizing software bugs on a computer program and optimizing the software test process. Since this problem is formulated as a typical discrimination problem to identify software bug-prone modules, several machine learning techniques have been used in past, where the underlying data are the binary data to denote whether each module is bug-prone (1) or not (0), and the features called software metrics to characterize the quality attributes of each module, such as the module's size and program complexity. Various discrimination and data mining techniques, including logistic regressions [2] [14], support vector machines [4], naive Bayes [15], Bayesian networks [3] [16], random forest [5], multilayer perceptron neural networks [6], convolutional neural networks [1], and spam filtering technique [13], among others [12] [17], have been directly used to identify software bug-prone modules. For the recent survey on software bug prediction, see Li et al. [11].

However, it has been known that almost all of them were not *explainable* and could not be applied to the program understanding, because the contributions of software metrics were unclear in such black box techniques. In fact, through a careful analysis on the contribution of each software metric in the bug prediction, it would be possible to improve the test efficiency by localizing software bugs from the code metrics such as the number of lines of code, cyclomatic numbers, the number of operators measured in each module development. In the view point of program understanding, it is quite important to investigate the relationship between bug-prone modules and explanatory variables (features), and to infer the presence/absence of software bugs in each module. If there is a clear causal relationship with the explanatory variables in an explainable method as logistic regression models, we may design the test cases efficiently according to the contribution of the metrics.

Unfortunately, it should be noted that the classical logistic regression model could not provide satisfactory bug-prediction results in terms of predictive performances [2] [14], compared to the typical deep machine learning techniques. Even for the explainable models, we need to carefully check not only the independence between explanatory variables but also *multi-collinearity* and *interaction* in the regression-based approach. Konno et al. [8] pointed out in the problem of estimating bankruptcy probability from financial metrics in companies that the logistic regression model used conventionally deals with the metrics that have a monotonic relationship, where the bankruptcy probability increases (decreases) as the values of the financial metrics increase (decrease). In the financial bankruptcy problem, it is implicitly assumed that there is no interaction on effects of each explanatory variable as a financial metric on the bankruptcy probability, but generally, the impact of explanatory variables on the bankruptcy probability may vary depending on the size of the explanatory variable.

Although the conventional logistic regression model, being simple and low in computational cost, is frequently applied to many real-world discrimination problems by devising the selection of explanatory variables and the classification of the dependent variable groups, there are theoretical and empirical limitations on the explainable logistic regression model. Konno et al. [8] proposed a semi-definite logistic regression model and attempted to solve large-scale semi-definite logistic regression problems in real-time [10] by applying a few optimization techniques such as the cutting-plane method [7] and the two-stage method [9].

In this article, we aim at overcoming the problems in a classical logistic regression model for software bug prediction, and apply the semi-definite logistic regression model to identify software bug-prone modules. More specifically, we use three actual software development project data sets, and evaluate the predictive performance on F-score, as well as precision and recall. We compare our semi-definite logistic regression model with the classical logistic regression model

in the context of software bug prediction. It is shown that our semi-definite logistic regression model involves the common logistic regression model as a special case and can improve the predictive performances on the F-score in the software bug prediction.

The remaining part of this article is organized as follows. In Section II, we describe a software bug prediction problem by means of the logistic regression model. Section III formulates a semi-definite logistic regression and summarizes a variable selection method. Section IV is devoted to numerical experiments, where the underlying data sets are given and the predictive performances between two logistic regression models are compared. Finally the article is concluded with some remarks in Section V.

II. Software Bug Prediction

Suppose that there are N software modules in the module testing. Let $\boldsymbol{x}_i = (x_{i1}, x_{i2}, \ldots, x_{in})$ denote the feature vector of the i $(= 1, 2, ..., N)$-th software module, where n types of features, called software metrics, are available in the software development. Define the probability that the i-th module contains any software bug by $f(\boldsymbol{x}_i)$ as a function of the feature vector $\boldsymbol{x}_i$, where $f(\cdot)$ denotes a nonlinear function. In the nonlinear equation $y_i = f(\boldsymbol{x}_i)$ for a given y_i, it is not possible to directly observe the probability of a module containing software bugs in advance, so that the information about the presence/absence of software bugs in each module is used post-hoc. Define the binary random variable Y_i with the realization y_i in the following:

$$Y_i = \begin{cases} 1, & \text{if module } i \text{ contains software bugs,} \\ 0, & \text{if module } i \text{ does not contain software bugs.} \end{cases} \tag{1}$$

There are various methods to formulate the above discrimination problem. Among them, the logistic regression model is easy to understand in terms of the formulation and the low computation cost. In the logistic regression model, the regression function $f(\boldsymbol{x}_i)$ is given by

$$f(\boldsymbol{x}_i) = \frac{\exp(\boldsymbol{Z}_i)}{1 + \exp(\boldsymbol{Z}_i)}, \tag{2}$$

where $\exp(\boldsymbol{Z}_i) = \boldsymbol{\beta}^T \boldsymbol{x}_i + \beta_0$ denotes the random variables representing the tendency of bug presence, $\boldsymbol{\beta} = (\beta_1, \beta_2, \ldots, \beta_n)$ is the regression coefficient vector, β_0 is a scalar constant, and T is the transpose. Since the bug-prone probability $f(\boldsymbol{x}_i) = f(\boldsymbol{x}_i; \boldsymbol{\beta}, \beta_0)$ is given by a function of $\boldsymbol{x}_i$, it turns out that the dependent variable becomes a monotonic function with respect to each component of $\boldsymbol{x}_i$.

Once the binary data and the software metric data $(y_i, \boldsymbol{x}_i)$ $(i = 1, 2, \ldots, N)$ are given, the log likelihood function is obtained as

$$\ln L(\boldsymbol{\beta}, \beta_0) = \sum_{i=1}^{N} \Big\{ y_i \ln f(\boldsymbol{x}_i; \boldsymbol{\beta}, \beta_0) + (1 - y_i) \ln(1 - f(\boldsymbol{x}_i; \boldsymbol{\beta}, \beta_0)) \Big\}. \tag{3}$$

Then the problem is to seek the maximum likelihood estimate $(\tilde{\boldsymbol{\beta}}, \tilde{\beta}_0) = \operatorname{argmax} \ln L(\boldsymbol{\beta}, \beta_0)$.

As mentioned in Section I, it is known that the logistic regression has several limitations. First, it is assumed that the larger (smaller) the value of each explanatory variable, the larger (smaller) the predicted probability y_i becomes. Second, it is assumed that the elements of each explanatory variable are independent of each other, and there is no interaction effects of each explanatory variable on the bug-prone probability. In the actual software bug prediction, for instance, an increase in the number of comment lines on a program is implicitly assumed as one of the software metrics increases the bug-prone probability. However, this property may not always hold, because insertion of a certain type of detailed comments may increase the understandability of the program, and may reduce the bug-prone probability efficiently. Furthermore, among many software metrics, the relationship between the lines of code and the total number of operators on the program may be unlikely to be independent. This is because an increase in the lines of codes may naturally tend to increase the total number of operators.

III. Semi-definite Logistic Regression Approach

The semi-definite logistic regression model was proposed in the reference [8]. For the real symmetric matrix $\boldsymbol{B} \in \mathcal{R}^{n \times n}$, the bug-prone probability $f(\boldsymbol{x}_i) = f(\boldsymbol{x}_i; \boldsymbol{B}, \beta_0)$ $(i = 1, 2, \ldots, N)$ is also defined by Eq.(2), where

$$\boldsymbol{Z}_i = \boldsymbol{x}_i^T \boldsymbol{B} \boldsymbol{x}_i + \boldsymbol{\beta}^T \boldsymbol{x}_i + \beta_0, \tag{4}$$

$$\boldsymbol{B} = \boldsymbol{B}^T. \tag{5}$$

In our semi-definite logistic regression model, it should be noted that $\mathbf{Z}_i$ is a quadratic form of $\boldsymbol{x}_i$. Hence, it is possible to incorporate non-monotonicity and interaction effects between explanatory variables.

Several optimization algorithms have already been proposed to solve the semi-definite logistic regression problems [7] [8] [9] [10]. As the problem size in dealing with the software bug prediction increases, it tends to be difficult to solve the maximum likelihood estimation problem. Fortunately, since our problem size is relatively small comparing with the financial bankruptcy problem, we can handle the maximum likelihood estimation for the semi-definite logistic regression model, by applying the quadratic programming algorithm implemented in the statistical software, R, without using the cutting-plane method [7] and the two-stage method [9]. On the other hand, it should be emphasized that all the explanatory variables may not always be useful for discriminating the software bug-prone modules. Generally, it is important to select a small number of useful explanatory features from all available ones. In this article, we use the well-known Akaike information criterion (AIC):

$$\text{AIC} = -2 \ln L(\tilde{\boldsymbol{B}}, \tilde{\beta}_0) + 2 \times (\text{number of free parameters in the model}) \tag{6}$$

TABLE I
DATA SETS.

	No. modules (N)	Bug inclusion rate	No. metrics (n)
jm1	7782	21.05%	21
pc1	705	8.65%	37
cm1	327	12.80%	37

for the maximum likelihood estimate $(\tilde{\boldsymbol{B}}, \tilde{\beta}_0)$ to determine the explanatory variables employed in the analysis. A smaller AIC indicates better model fit.

In general, there are two variable selection methods; the variable reduction method and variable increase method under the AIC criterion. We apply the variable reduction method to the semi-definite logistic regression model. The main reason why the variable increase method is not used here is that the number of arbitrary parameters in the semi-definite logistic regression model becomes large in the order of squares, and results in enormous computation cost, so our semi-definite logistic regression model penalizes by applying the variable increase method. Concretely, we estimate the parameters (regression coefficients) using all software metrics first, and then remove unnecessary feature one by one so as to minimize the AIC. We repeat this procedure again and again until the AIC value can be reduced no longer. In summary, we describe the procedure for our variable reduction method as follows.

Step 1: For $(y_i, \boldsymbol{x}_i)$ $(i = 1, 2, \ldots, m)$, where m is the number of training data, apply a semi-definite quadratic programming to the semi-definite logistic regression model, derive the maximum likelihood estimate $(\tilde{\boldsymbol{\beta}}, \tilde{\beta}_0)$, and calculate the AIC.

Step 2: Apply the variable reduction method, remove one unnecessary software metric with minimum regression coefficient and set $m - 1 \to m$.

Step 3: Go to Step 1 and calculate the AIC' with the updated m.

Step 4: If AIC $>$ AIC', then Go to Step 1, otherwise, and set $m \to m - 1$ and Stop the procedure.

IV. NUMERICAL EXPERIMENTS

A. Data Sets

Data sets used here are from NASA's software development projects, namely, jm1, pc1, and cm1. The number of modules, defect density, and the number of software metrics used in the experiments for each data set are shown in Tables I to III.[1]

B. Predictive Performances

We compare the predictive performances of four bug prediction models; the standard logistic regression model, the semi-definite logistic regression model, and respective models refined by variable reduction. Hereafter, we denote the standard logistic regression as Logit-11, the standard logistic regression with variable reduction method Logit-12, the semi-definite logistic regression model Logit-21, and the semi-definite logistic regression model with variable reduction method Logit-22. In our experiments, the data sets are randomly split into the training data and the validation data. Especially, three cases with different training data sizes; 25%, 50%, and 75% of the whole data, are considered. In each case, the remaining 75%, 50% and 25% data sets are used for validation/prediction.

[1]The data sets were reported in NASA/WVU IV &V Facility, Metrics Data Program. http://mdp.ivv.nasa.gov/.

TABLE II
SOFTWARE METRICS IN JM1.

	Software metrics
x_{i1}	LOC BLANK
x_{i2}	BRANCH COUNT
x_{i3}	LOC CODE AND COMMENT
x_{i4}	LOC COMMENTS
x_{i5}	CYCLOMATIC COMPLEXITY
x_{i6}	DESIGN COMPLEXITY
x_{i7}	ESSENTIAL COMPLEXITY
x_{i8}	LOC EXECUTABLE
x_{i9}	HALSTEAD CONTENT
x_{i10}	HALSTEAD DIFFICULTY
x_{i11}	HALSTEAD EFFORT
x_{i12}	HALSTEAD ERROR EST
x_{i13}	HALSTEAD LENGTH
x_{i14}	HALSTEAD LEVEL
x_{i15}	HALSTEAD PROG TIME
x_{i16}	HALSTEAD VOLUME
x_{i17}	NUM OPERANDS
x_{i18}	NUM OPERATORS
x_{i19}	NUM UNIQUE OPERANDS
x_{i20}	NUM UNIQUE OPERATORS
x_{i21}	LOC TOTAL
x_{i22}	DEFECTIVE

To evaluate the bug-prone probability, we apply the F-score which is a harmonic mean of *precision* and *recall*, where precision indicates a proportion of correct results in the prediction, and recall is a proportion of how much of the correct answers could be predicted. That is, we have

$$\begin{aligned} \text{Precision} &= \frac{\text{True Positive}}{\text{True Positive} + \text{False Positive}}, \\ \text{Recall} &= \frac{\text{True Positive}}{\text{True Positive} + \text{False Negative}}, \end{aligned} \tag{7}$$

where True Positive is the number of data that could be correctly predicted to contain software bugs, False Positive is the number of data incorrectly predicted to contain software bugs, and False Negative is the number of data incorrectly predicted to contain no software bugs. Then F-score is defined by

$$F\text{-score} = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}. \tag{8}$$

Note that F-score is given by a real number between 0 and 1, and can be interpreted such that the higher the value of F-score the higher the predictive performance of the model.

In order to predict the bug-proneness of each software module, it is necessary to choose a *threshold* for judgement of bug proneness. In our experiments we set five threshold values; 0.3, 0.4, 0.5, 0.6, and 0.7. If the bug-prone probability is greater than a given threshold, then the resulting software

TABLE III
SOFTWARE METRICS IN PC1 AND CM1.

	Software metrics
x_{i1}	LOC BLANK
x_{i2}	BRANCH COUNT
x_{i3}	CALL PAIRS
x_{i4}	LOC CODE AND COMMENT
x_{i5}	LOC COMMENTS
x_{i6}	CONDITION COUNT
x_{i7}	CYCLOMATIC COMPLEXITY
x_{i8}	CYCLOMATIC DENSITY
x_{i9}	DECISION COUNT
x_{i10}	DECISION DENSITY
x_{i11}	DESIGN COMPLEXITY
x_{i12}	DESIGN DENSITY
x_{i13}	EDGE COUNT
x_{i14}	ESSENTIAL COMPLEXITY
x_{i15}	ESSENTIAL DENSITY
x_{i16}	LOC EXECUTABLE
x_{i17}	PARAMETER COUNT
x_{i18}	HALSTEAD CONTENT
x_{i19}	HALSTEAD DIFFICULTY
x_{i20}	HALSTEAD EFFORT
x_{i21}	HALSTEAD ERROR EST
x_{i22}	HALSTEAD LENGTH
x_{i23}	HALSTEAD LEVEL
x_{i24}	HALSTEAD PROG TIME
x_{i25}	HALSTEAD VOLUME
x_{i26}	MAINTENANCE SEVERITY
x_{i27}	MODIFIED CONDITION COUNT
x_{i28}	MULTIPLE CONDITION COUNT
x_{i29}	NODE COUNT
x_{i30}	NORMALIZED CYLOMATIC COMPLEXITY
x_{i31}	NUM OPERANDS
x_{i32}	NUM OPERATORS
x_{i33}	NUM UNIQUE OPERANDS
x_{i34}	NUM UNIQUE OPERATORS
x_{i35}	NUMBER OF LINES
x_{i36}	PERCENT COMMENTS
x_{i37}	LOC TOTAL
x_{i38}	DEFECTIVE

module is judged to contain software bugs. The prediction results obtained in the experiments are shown in Tables VI to XII. In these tables, the largest value in each table is denoted by a double underline, and the largest value in each threshold level is singly underlined.

In the data set, jm1, our semi-definite logistic model with variable reduction could show the highest F-scores when estimating the parameters using 75% of the training data. Furthermore, in Tables IV to VI, the standard logistic regression showed the highest F-score with 25% of the training data, but our semi-definite logistic regression model with variable reduction could give the higher performances on F-score when estimating parameters using 50% and 75% of the training data.

In the data set, pc1, it is observed that our semi-definite logistic regression model with variable reduction provided the highest F-score with 75% of the training data. Also, in all cases of models in Tables VII to IX, when estimating the model parameters with 25%, 50% and 75% of the training data, the semi-definite logistic regression model with variable reduction could give the highest F-score evidently.

Finally, in the data set, m1, it can be seen that our semi-definite logistic regression model with variable reduction gave

TABLE IV
JM1 (25% TRAINING DATA).

Threshold	Model Type	Precision	Recall	F-score
0.3	Logit-11	0.445	0.307	0.363
	Logit-12	0.451	0.298	0.357
	Logit-21	0.305	0.296	0.291
	Logit-22	0.443	0.304	0.360
0.4	Logit-11	0.508	0.186	0.271
	Logit-12	0.514	0.176	0.261
	Logit-21	0.318	0.288	0.292
	Logit-22	0.510	0.182	0.267
0.5	Logit-11	0.550	0.111	0.184
	Logit-12	0.546	0.108	0.179
	Logit-21	0.330	0.287	0.292
	Logit-22	0.539	0.113	0.187
0.6	Logit-11	0.592	0.070	0.125
	Logit-12	0.587	0.072	0.127
	Logit-21	0.325	0.288	0.290
	Logit-22	0.605	0.075	0.132
0.7	Logit-11	0.648	0.046	0.086
	Logit-12	0.640	0.046	0.086
	Logit-21	0.317	0.286	0.285
	Logit-22	0.652	0.050	0.093

TABLE V
JM1 (50% TRAINING DATA).

Threshold	Model Type	Precision	Recall	F-score
0.3	Logit-11	0.453	0.309	0.367
	Logit-12	0.462	0.307	0.368
	Logit-21	0.330	0.319	0.311
	Logit-22	0.432	0.326	0.371
0.4	Logit-11	0.513	0.182	0.268
	Logit-12	0.518	0.176	0.262
	Logit-21	0.336	0.279	0.289
	Logit-22	0.509	0.183	0.268
0.5	Logit-11	0.559	0.105	0.176
	Logit-12	0.558	0.105	0.176
	Logit-21	0.325	0.288	0.281
	Logit-22	0.557	0.115	0.190
0.6	Logit-11	0.609	0.069	0.123
	Logit-12	0.610	0.066	0.119
	Logit-21	0.374	0.240	0.261
	Logit-22	0.631	0.071	0.128
0.7	Logit-11	0.685	0.043	0.082
	Logit-12	0.682	0.044	0.083
	Logit-21	0.356	0.276	0.272
	Logit-22	0.646	0.046	0.086

the highest F-score with 25% of the training data. In Tables X to XII, the standard logistic regression model gave the highest F-score with 50% of the training data, but our semi-definite logistic regression model with variable reduction could show the highest predictive performances when estimating parameters using 25% and 75% of the training data.

In comparison between the conventional logistic regression model and the semi-definite regression model, it is found that our novel approach could not always outperform the classical one. However, our experimental results showed that in most cases across all data sets, the semi-definite logistic regression models could exhibit higher F-score than the conventional logistic regression models. As shown in Tables IV to XII, even though the predictive performances of our semi-definite

TABLE VI
JM1 (75% TRAINING DATA).

Threshold	Model Type	Precision	Recall	F-score
0.3	Logit-11	0.459	0.313	0.372
	Logit-12	0.457	0.310	0.369
	Logit-21	0.343	0.325	0.315
	Logit-22	0.453	0.331	0.382
0.4	Logit-11	0.514	0.179	0.265
	Logit-12	0.520	0.177	0.264
	Logit-21	0.337	0.300	0.294
	Logit-22	0.515	0.187	0.274
0.5	Logit-11	0.563	0.104	0.175
	Logit-12	0.570	0.104	0.175
	Logit-21	0.346	0.288	0.284
	Logit-22	0.572	0.103	0.175
0.6	Logit-11	0.632	0.066	0.119
	Logit-12	0.630	0.067	0.120
	Logit-21	0.364	0.294	0.284
	Logit-22	0.655	0.066	0.120
0.7	Logit-11	0.686	0.042	0.080
	Logit-12	0.691	0.043	0.081
	Logit-21	0.351	0.260	0.260
	Logit-22	0.731	0.044	0.082

TABLE VII
PC1 (25% TRAINING DATA).

Threshold	Model Type	Precision	Recall	F-score
0.3	Logit-11	0.226	0.348	0.269
	Logit-12	0.239	0.342	0.278
	Logit-21	0.113	0.494	0.183
	Logit-22	0.286	0.462	0.351
0.4	Logit-11	0.232	0.349	0.273
	Logit-12	0.243	0.327	0.271
	Logit-21	0.106	0.489	0.173
	Logit-22	0.293	0.320	0.298
0.5	Logit-11	0.235	0.357	0.277
	Logit-12	0.248	0.334	0.277
	Logit-21	0.110	0.477	0.178
	Logit-22	0.309	0.313	0.305
0.6	Logit-11	0.237	0.351	0.276
	Logit-12	0.246	0.320	0.268
	Logit-21	0.112	0.518	0.182
	Logit-22	0.253	0.217	0.221
0.7	Logit-11	0.232	0.336	0.269
	Logit-12	0.238	0.325	0.268
	Logit-21	0.106	0.485	0.173
	Logit-22	0.319	0.242	0.264

TABLE VIII
PC1 (50% TRAINING DATA).

Threshold	Model Type	Precision	Recall	F-score
0.3	Logit-11	0.317	0.355	0.327
	Logit-12	0.303	0.338	0.312
	Logit-21	0.088	0.490	0.149
	Logit-22	0.328	0.298	0.304
0.4	Logit-11	0.337	0.325	0.323
	Logit-12	0.326	0.280	0.294
	Logit-21	0.086	0.486	0.146
	Logit-22	0.350	0.234	0.270
0.5	Logit-11	0.364	0.265	0.299
	Logit-12	0.369	0.243	0.280
	Logit-21	0.088	0.500	0.149
	Logit-22	0.489	0.267	0.334
0.6	Logit-11	0.366	0.248	0.286
	Logit-12	0.374	0.220	0.266
	Logit-21	0.088	0.490	0.149
	Logit-22	0.366	0.288	0.305
0.7	Logit-11	0.391	0.198	0.253
	Logit-12	0.421	0.207	0.266
	Logit-21	0.092	0.499	0.154
	Logit-22	0.461	0.187	0.239

TABLE IX
PC1 (75% TRAINING DATA).

Threshold	Model Type	Precision	Recall	F-score
0.3	Logit-11	0.346	0.356	0.342
	Logit-12	0.312	0.317	0.306
	Logit-21	0.853	0.483	0.144
	Logit-22	0.391	0.349	0.357
0.4	Logit-11	0.395	0.293	0.325
	Logit-12	0.388	0.267	0.305
	Logit-21	0.091	0.501	0.153
	Logit-22	0.412	0.329	0.353
0.5	Logit-11	0.439	0.243	0.301
	Logit-12	0.404	0.196	0.260
	Logit-21	0.092	0.534	0.156
	Logit-22	0.385	0.405	0.374
0.6	Logit-11	0.435	0.211	0.272
	Logit-12	0.422	0.167	0.242
	Logit-21	0.088	0.500	0.148
	Logit-22	0.417	0.245	0.294
0.7	Logit-11	0.479	0.190	0.260
	Logit-12	0.486	0.160	0.241
	Logit-21	0.094	0.52	0.158
	Logit-22	0.521	0.250	0.301

logistic regression models without variable reduction method were rather low, applying the variable reduction could improve the predictive performances. In a few cases, it can be found that the conventional logistic regression models showed better predictive performances than the semi-definite logistic regression models. However, since the semi-definite logistic regression model includes the logistic regression model as a special case, the predictive performances of the semi-definite logistic regression models are never inferior to those of the common logistic regression models.

V. CONCLUSIONS

In this article, we have proposed a novel and explainable software bug-prediction model based on the semi-definite logistic model and compared the applicability in predicting the bug-prone module. In the past literature, almost all works have focused on only the predictive performances including F-score and attempted to apply several machine learning techniques in the software bug-prediction. However, since almost all machine learning techniques did not provide the feedback information on the dependence between the software metrics employed in the analysis and the bug-proneness, more sophisticated explainable bug-prediction methods have been demanded. In our numerical experiments, we have shown that our semi-definite logistic model could show the potential applicability in software bug prediction. By checking the regression coefficients with respect to a combination of software metrics, it would be possible to analyze the dependence in software bug-prone probability.

TABLE X
CM1 (25% TRAINING DATA).

Threshold	Model Type	Precision	Recall	F-score
0.3	Logit-11	0.200	0.360	0.253
	Logit-12	0.234	0.336	0.269
	Logit-21	0.137	0.467	0.211
	Logit-22	0.167	0.227	0.187
0.4	Logit-11	0.206	0.367	0.260
	Logit-12	0.226	0.327	0.260
	Logit-21	0.133	0.466	0.205
	Logit-22	0.296	0.456	0.354
0.5	Logit-11	0.204	0.355	0.254
	Logit-12	0.223	0.314	0.253
	Logit-21	0.139	0.488	0.215
	Logit-22	0.252	0.321	0.275
0.6	Logit-11	0.211	0.352	0.260
	Logit-12	0.220	0.316	0.252
	Logit-21	0.141	0.505	0.220
	Logit-22	0.247	0.260	0.247
0.7	Logit-11	0.202	0.359	0.254
	Logit-12	0.214	0.311	0.247
	Logit-21	0.136	0.452	0.207
	Logit-22	0.254	0.220	0.225

TABLE XI
CM1 (50% TRAINING DATA).

Threshold	Model Type	Precision	Recall	F-score
0.3	Logit-11	0.269	0.349	0.295
	Logit-12	0.261	0.343	0.288
	Logit-21	0.135	0.479	0.209
	Logit-22	0.258	0.309	0.274
0.4	Logit-11	0.268	0.327	0.288
	Logit-12	0.273	0.306	0.280
	Logit-21	0.139	0.502	0.216
	Logit-22	0.392	0.244	0.292
0.5	Logit-11	0.264	0.307	0.276
	Logit-12	0.273	0.263	0.258
	Logit-21	0.140	0.489	0.216
	Logit-22	0.277	0.236	0.246
0.6	Logit-11	0.269	0.251	0.251
	Logit-12	0.284	0.224	0.239
	Logit-21	0.133	0.464	0.206
	Logit-22	0.315	0.250	0.265
0.7	Logit-11	0.283	0.254	0.258
	Logit-12	0.287	0.221	0.239
	Logit-21	0.133	0.471	0.206
	Logit-22	0.377	0.186	0.242

TABLE XII
CM1 (75% TRAINING DATA).

Threshold	Model Type	Precision	Recall	F-score
0.3	Logit-11	0.287	0.331	0.295
	Logit-12	0.276	0.326	0.295
	Logit-21	0.127	0.472	0.196
	Logit-22	0.312	0.396	0.337
0.4	Logit-11	0.338	0.305	0.299
	Logit-12	0.311	0.272	0.286
	Logit-21	0.136	0.534	0.213
	Logit-22	0.348	0.322	0.336
0.5	Logit-11	0.341	0.279	0.288
	Logit-12	0.301	0.188	0.235
	Logit-21	0.135	0.513	0.210
	Logit-22	0.418	0.212	0.277
0.6	Logit-11	0.358	0.200	0.239
	Logit-12	0.322	0.160	0.243
	Logit-21	0.133	0.492	0.205
	Logit-22	0.364	0.190	0.263
0.7	Logit-11	0.329	0.152	0.194
	Logit-12	0.342	0.091	0.215
	Logit-21	0.136	0.504	0.212
	Logit-22	0.347	0.123	0.220

In future, we will compare the semi-definite logistic regression approach with several deep learning methods in large-scaled experiments and explore the potential to use it in software bug prediction problems.

REFERENCES

[1] S. Balasubramaniam, and S. G. Gollagi, "Software defect prediction via optimal trained convolutional neural network," *Advances in Engineering Software*, vol. 169, p. 103138, 2022.

[2] L. C. Briand, W. L. Melo, and J. Wust, "Assessing the applicability of fault-proneness models across object-oriented software projects," *IEEE Transactions on Software Engineering*, vol. 28, pp. 706-–720, 2002.

[3] G. Denaro, and M. Pezze, "An empirical evaluation of fault-proneness models" *Proceedings of The 24th International Conference on Software Engineering (ICSE-2002)*, pp. 241–251, 2002.

[4] K. O. Elish, and M. O. Elish, "Predicting defect-prone software modules using support vector machines," *Journal of Systems and Software*, vol. 81, pp. 649–660, 2008.

[5] L. Guo, Y. Ma, B. Cukic, H. Singh, "Robust prediction of fault-proneness by random forests," *Proceedings The 15th International Symposium on Software Reliability Engineering (ISSRE-2004)*, pp. 417–428, 2004.

[6] C. Jin, and S. W. Jin, "Prediction approach of software fault-proneness based on hybrid artificial neural network and quantum particle swarm optimization," *Applied Soft Computing*, vol. 35, pp. 717–725, 2015.

[7] H. Konno, N. Kawadai, and H. Yuy, "Cutting plane algorithms for nonlinear semi-definite programming problems with applications," *Journal of Global Optimization*, vol. 25, pp. 141–155, 2003.

[8] H. Konno, N. Kawadai, and D. Wu, "Estimation of failure probability using semi-definite logit model," *Computational Management Science*, vol. 1, pp. 59—73, 2003.

[9] H. Konno, N. Kawadai, and H. Shimode, "A two step algorithm for solving a large scale semi-definite logit model," *Optimization Letters*, vol. 1, pp 329–340, 2007.

[10] H. Konno, S. Kameda, and N. Kawadai, "Solving a large scale semi-definite logit model," *Computational Management Science*, vol. 7, pp. 111—120, 2010.

[11] Z. Li, X.-Y. Jing, and X. Zhu, "Progress on approaches to software defect prediction," *IET Software*, vol. 12, 161–175, 2018.

[12] T. Menzies, J. Greenwald, and A. Frank, "Data mining static code attributes to learn defect predictors," *IEEE Transactions on Software Engineering*, vol. 33, pp. 2—13, 2007.

[13] O. Mizuno, and T. Kikuno, "Training on errors experiment to detect fault-prone software modules by spam filter," *Proceedings of The 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE2007)*, pp. 405–414, 2007.

[14] N. Ohlsson, and H. Alberg, "Predicting fault-prone software modules in telephone switches," *IEEE Transactions on Software Engineering*, vol. 22, pp. 886–894, 1996.

[15] S. K. Pandey, R. B. Mishra, and A. K. Triphathi, "Software bug prediction prototype using Bayesian network classifier: A comprehensive model," *Procedia Computer Science*, vol.1 32, pp. 1412–1421, 2018.

[16] E. Perez-Minana, and J. J. Gras, "Improving fault prediction using Bayesian networks for the development of embedded software applications," *Software Testing, Verification and Reliability*, vol. 16, pp. 157–174, 2006.

[17] H. Tong, B. Liu, and S. Wang, "Software defect prediction using stacked denoising autoencoders and two-stage ensemble learning," *Information and Software Technology*, vol. 96, pp. 94–111, 2018.