Software Bug Location Algorithm on the basis of the Integration of Fuzzy Multi-objective Model

Gao Yinsheng

Xi'an University, 710065

Keywords: Bug Location; Fuzzy Multi-objective Model Analysis; Software Measurement; Software Testing

Abstract: Bug location is an important part of the software development process. Making full use of the structural features and behavioral features of the program is conducive to improving the bug location efficiency. In this paper, a kind of bug location framework on the basis of the integration of the fuzzy multi-objective model is put forward to carry out the bug location at the class method level on the new version of program during the evolution of software. Firstly, a set of indexes for the measurement of the structural features and the behavioral features are designed. Through the static analysis and the testing program, the feature datasets of the old versions of programs are collected and constructed. At the same time, the old versions of bug information are obtained from the bug tracking system. Secondly, on the basis of the resulting feature datasets and bug information, the univariate analysis is used to filter out the indexes from the measurement indexes that are significantly related to the bugs. Subsequently, the significant indexes that have been selected are used to carry out the multivariate analysis and perform training on the integrated fuzzy multi-objective model. Finally, the feature data set of the new version of program is collected and constructed on the basis of the selected significant indexes. And the fuzzy multi-objective model obtained is used to predict the error probability of each class method. The class method is further inspected in accordance with the error probability in a descending order to achieve the error location. The empirical study of the bug location is carried out on the basis of a set of open source programs. And the results show that the efficiency of bug location can be improved through the integration of the fuzzy multi-objective model.

1. Introduction

Software is a complex artificial product that has profoundly changed the production method and lifestyle of human beings. Due to the restriction of human intellectual activities, it is often inevitable that bugs are introduced into the new version of programs in the evolution process of the software. And these potential bugs often result in losses to the user [1-2]. Hence, elimination of the software bugs and improvement of the quality of software evolution is an important subject to be solved urgently in the field of software engineering. People often make use of the program analysis to identify bugs or expose the bugs in the program through the testing software [3]. Studies have shown that some of the program bugs are caused by the complex program structures, such as the recursive calls and the complex loops, which can easily lead to program errors [4-5]. On the one hand, the static information of the program can fully reflect the structural features of the program; on the other hand, the dynamic information in the program testing can reflect the behavior features of the program under certain conditions (test input) [6-7]. In accordance with whether to use the information during the running of the software as well as how to make use of such information, the existing software bug location algorithms can be divided into three categories. The first category is the statistical fault location method SFL on the basis of the program spectrum [8-9]. In this type of methods, the information during the program testing process (that is, the statement, the predicate, the execution path or the function and other coverage information) is collected for statistics. And their correlation with the program running results is analyzed. A certain kind of heuristic calculation formula for the degree of skepticism is given to estimate the possibility of error. And finally, the

  DOI: 10.25236/isrme.2019.027

program entity is inspected in the descending order of the degree of skepticism until the bug is located [10-11]. The second category is the software bug location algorithm based on the program slicing. The program slicing makes use of the dependency analysis to identify the sets of statements in the program that affect the specific program entity [12-13]. The slicing technique can effectively separate the sentences related to bugs, and thus has become an effective tool for assisting the location of bugs. The third category falls under the static analysis method [14-15]. This kind of method first analyzes the static structure information of the program that has been acquired, and then combines the syntax and semantic constraints of the program design language to detect the program violations or locate the software bugs by means of the symbolic execution, formal proof and other techniques [16].

On the basis of the above analysis, a kind of bug location framework that integrates the fuzzy multi-objective model is put forward in this paper, which is used to locate the bugs of the new version of the software at the class method level during the evolution of the software. Specifically: First of all, a set of measurement indexes that cover the structure features and behavior features of the software is designed. And the feature data sets are constructed by using the static analysis and the software test tracking. Secondly, univariate analysis is carried out in combination with the bug information in the old version of the software to screen for the feature indexes related to the significant bugs and construct the fuzzy multi-objective model analysis model. And the screened feature data and bug information is used to train the model parameters. Finally, the feature data set for the new version of the program is constructed. The integrated fuzzy multi-objective model obtained from the training is used for the bug prediction and location. And we carry out the empirical study of the bug location model put forward in this paper on a set of benchmarking programs.

2. Preliminary Knowledge

The bug location in this paper is achieved by calculating the probability of a bug contained in the class method. For each class method in the system, it is assumed that whether the bug can be detected is an independent event (denoted as Y = 1). And the probability of this class method containing the bug is *prob (Y = 1)*. For a given class method, if *prob (Y = 1)* > ω, the class method contains a bug; otherwise, it contains no bug (ω stands for the specified threshold value).

Definition 1 Univariate analysis. Given the independent variable x and the dependent variable y, the fuzzy multi-objective model $y = f(x)$ is used to analyze the statistical correlation of the variables x and y.

In this paper, the variable x stands for the measurement on a certain index (such as the number of the lines of the code) of the class method, and y stands for the bug information of the class method tracked by the bug tracking system. The univariate analysis is used to identify the variables that are significantly related to the bug (with the statistical significance level $\alpha = 0.05$).

In order to determine the variables used in the fuzzy multi-objective model, it is necessary to first analyze the statistical correlations between the individual measurement and the bugs, so as to select those variables that are significantly related to the bugs for the subsequent fuzzy multi-objective model.

Definition 2 Multivariate analysis. The fuzzy multi-objective model $y = f(x_1, x_2, \ldots, x_m)$ is used to analyze the bug correlation of the measurement x_m for a set of independent variables $x_1, x_2, \ldots, x_m$ and the dependent variable y.

In this paper, the variable $x_1, x_2, \ldots, x_m$ are the variables that are significantly related to the bug screened out by the univariate analysis. And y stands for the bug information of the class method tracked by the bug tracking system. The $f(x)$ stands for the fuzzy multi-objective model selected. And its form is as the following:

$$\pi(x_1, x_2, \cdots, x_m) = \frac{e^{c_0 + c_1 x_1 + \cdots c_m x_m}}{1 + e^{c_0 + c_1 x_1 + \cdots c_m x_m}} \quad (1)$$

The fuzzy multi-objective model is a kind of standard technique on the basis of the maximum likelihood estimation. And the equation (1) stands for the multivariable fuzzy multi-objective model. In the bug location, the fuzzy multi-objective model is first trained by the known bug information and the measurement information of the class method to determine the model parameters (c_0, c_1, …, c_m). And then in the bug location, the dynamic information of the new version of the program and its class method structure measurement information are used as the model input to calculate the degree of skepticism of the class method.

Definition 3 Fuzzy multi-objective prediction. A set of measurement $x_1, x_2, \ldots, x_m$ is introduced into the constructed fuzzy multi-objective model $y = f(x_1, x_2, \ldots, x_m)$ to calculate a set of predicted values. In this paper, for the measurement of the selected variables $x_1, x_2, \ldots, x_m$ that are significantly related to the bugs, the data set in the new version of the software is constructed. And the fuzzy multi-objective model constructed by the training is used to calculate the prediction value of each class method. And from the equation (1), it can be inferred that the calculation formula for the error probability of each class method is as the equation (2) in the following

$$prob = \frac{e^{\pi_i}}{1+e^{\pi_i}}, 1 \le i \le n \tag{2}$$

In this paper, prob stands for the conditional probability that bug is present in the class method, that is, the probability of identifying a bug in the class method under a set of variables $x_1, x_2, \ldots, x_m$. For example, prob = 0.6 indicates that there is 60% probability that the corresponding class method has the presence of a bug. And during the location of bugs, it is possible to identify the highly suspicious class method on the basis of the given threshold value (such as $prob > 0.5$).

3. Software Bug Location Algorithm

3.1. Method Framework

Our method framework is shown in Figure 1. It mainly includes three phases: the pretreatment phase, the model construction phase and the fuzzy multi-objective model phase. Among them, mainly two tasks are accomplished in the pretreatment phase: (1) Run the program in accordance with the test case; (2) Obtain the structural information of the class method. For example, the number of the lines of the code in the class method, the complexity of the loop and so on. And the purpose of the model construction phase is to screen out the variables that are highly related to the bugs by means of the univariate analysis and to train the fuzzy multi-object model accordingly. The main purpose of the fuzzy multi-objective model phase is to predict the bug correlation of each class method by means of the multivariable fuzzy multi-objective model, that is, the degree of skepticism of the class method. In the subsequent section, the next two phases will be introduced in detail, followed by our entire algorithm flow for the bug location.

3.2. Model Construction

The first step in the model construction phase is to select the variables that are significantly related to the bug by means of the univariate analysis. And the second step is to train the fused fuzzy multi-objective model using the filtered variables in combination with the bug tracking information.

By means of the static analysis and the program operation tracking, we can obtain a few measurements of the program structure features and the behavior features, as shown in Table 1 in the following.

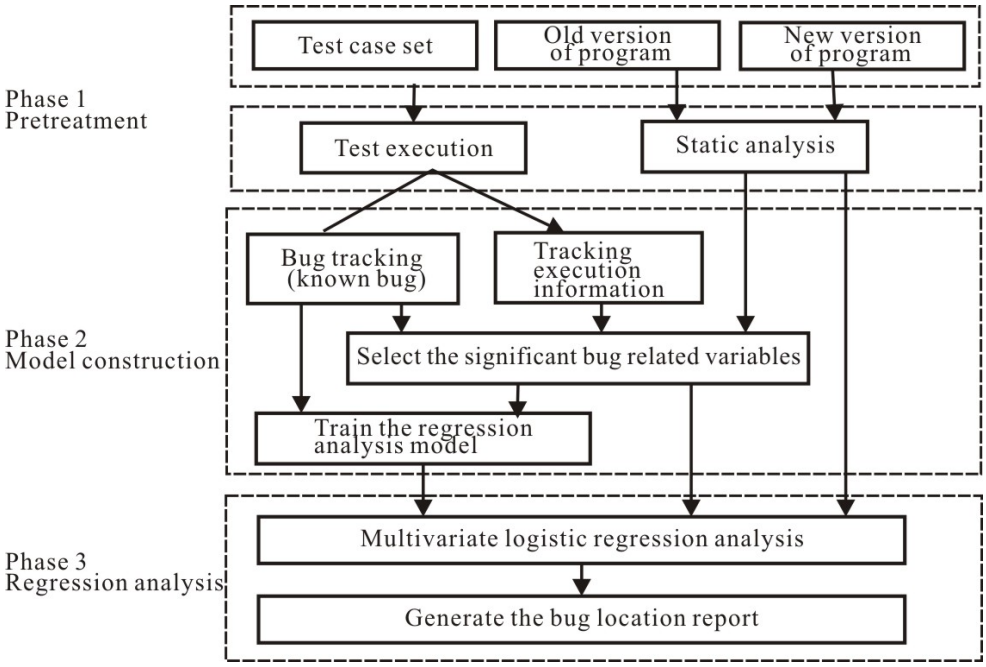


Figure 1 Framework of the software bug location method

Table 1 feature measuring and bug tracking of legacy software

Metbod number	Static features			Static features			#fault(y)
	X1	X2	....	xv	....	xm	
1	e11	e12	....	e1k	....	e1m	f1
2	e21	e22	....	e2k	....	e2m	f2
.	.	.		.		.	.
.	.	.		.		.	.
.	.	.		.		.	.
n	en1	En2	....	eck		enm	fn

Table 1 shows the information of a program feature measurement with n methods. Among them, the first column shows the numbers of all the program class methods; the last column shows the number of known bugs $f_i(i \in [1,n])$ contained in each method obtained by the bug tracking system; the middle $e_{ij}(i \in [1,n]),\ j \in [1,m]$ shows the information of the variables obtained from the program structure feature measurement and the dynamic feature tracking. And these variable variables correspond to the structural features and the dynamic behavior features associated with the class method in the program, respectively, which are denoted by x_1, x_2, …, x_m, respectively. As the existing program complexity measurement oriented towards the subject is mainly targeted for the class measurement, its measurement index cannot be directly applied in this method. We refer to the design method of the above-mentioned measurement set and combine the commonly used measurement in the existing bug location, the number of lines of code in the class method and the circle complexity are used measure the static features of the class. And the number of times that the class method is used to cover all the successful or failed testing is used to measure the dynamic features.

The $y = f(x)$ model is used carry out the univariate analysis on each variable $x_j(j \in [1,m])$ and test the correlation of the variable with the bug. Here the vector y stands for the number of bugs contained in each method, that is, the last column in Table 1. The variable x_j corresponds to the static information and the dynamic information sub column in Table 1. The linear fuzzy multi-objective model is selected for the $f(x)$ here. And in the univariate analysis, the fuzzy multi-objective model is carried out on the above x and y successively. And the variable x that is significantly related to the bug (significant level) is selected for the subsequent fuzzy multi-

objective model.

3.3. Fuzzy Multi-objective Model

The intuitive meaning of the bug location in the fusion fuzzy multi-objective model can be expressed as the following: For each class method, the static structure measurement information and the dynamic behavior information are collected in turn. And the degree of skepticism is calculated based on the integration of the trained fuzzy multi-objective model. For a new version of the program, we first analyze the static structure information of the program; at the same time, the program test cases are run to collect the program dynamic information. The results collected are recorded as the sub columns of the static features and the dynamic features as shown in Table 2.

Table 2 Features measuring of evolved software

Metbod number	Static features			Static features			Faull(prob)
	X1	X2	....	xv	....	xw	
1	e11	e12	....	e1v	....	e1w	P1
2	e21	e22	....	e2v	....	e2w	P2
.	.	.		.		.	.
.	.	.		.		.	.
.	.	.		.		.	.
n	en1	en2	....	env		enw	pn

The $x_j (j \in [1, \omega])$ in Table 2 stand for the variables that are screened out from x_1, x_2, …, x_m in Table 1 that are significantly related to the bugs. Different from the known error in the program directly represented by the last column in Table 1, the last column in Table 2 stands for the possibility of the occurrence of error in the corresponding method.

The calculation method is to make use of the model established in phase II, input the information obtained from the tracking analysis of the new version, and further carry out the Logistic analysis and prediction to obtain the predicted value for each method. And for the error probability corresponding to each method P_i, it can be calculated in accordance with the equation (2). In the next section, the class method is inspected in a descending order of error probability to find out the bugs in the software.

3.4. Process of Bug Location

After the analysis of the above two key treatment processes, Algorithm 1 gives the bug location process on the basis of the fuzzy multi-objective model.

In the algorithm 1, the bug location input is carried out based on the integration of the fuzzy multi-objective model as the following: X, /* Measurement index set */

$P, P^{'}$, /*stand for the old version of the program and the new version of the program, respectively */

$T, T^{'}$. /* stand for the set of test cases for the old version of the program and the new version of the program, respectively*/

Output: prob. /* Error probability of the new version program class method */

1) $F = Construct(X, P, T)$;/* Construct the feature set of the old version of the program */
2) $y = faultTrace()$;/* Obtain the bug information through the bug tracking */
3) $X^{'} = \varnothing$;
4) For each do /* Select the indexes significantly related to the bugs */
5) $y = f(x_i)$ is used to carry out the univariate analysis; /*y stands for the bug information*/
6) If x_i and y are significantly related then/*p-value < 0.05*/
7) $X^{'} = X^{'} \cup \{x_i\}$;
8) End if
9) End for

10) Log_model=Train (F, X'); /*Train the fuzzy multi-objective model */

11) F'=Construct ((X'， P'， T'); /* Construct the new version program feature set */

12) pred=predict（F'， X'， Log_model）;

13) $prob = e^{pred} / (1 + e^{pred})$;

14) b return prob.

The algorithm is described as the following: In Line 1 to 2, the feature set F and the bug information y are constructed as shown in Table 1. Among them, the Construct function indicates that the feature set is constructed through the static analysis and the tracking program testing operation. In Line 4 to 9, the measurement index X' that is significantly related to the bugs is screened out through the univariate analysis. In Line 10, the data in the feature set F that are corresponding to the measurement index significantly related to the bugs are used to train the integrated fused fuzzy multi-objective model in combination with the bug information 2 obtained in Line 2. In Line 11 to 13, the construction of the feature set n for the new version of the program is implemented, and the trained model is used to carry out the bug location prediction.

4. Empirical Study

4.1. Experimental Subjects and Environment

We have selected four Java programs as the experimental objects. And the features of the programs are shown in Table 3 as the following.

Table 3 characteristics of subjects

Name of the program	Executable lines of code	Number of class methods	Number of test cases	Number of errors
Print-tokens	478	25	4 130	5
jtcas	181	9	1 608	29
nanoxm1 v1	3 497	118	214	7
nanoxm1 v2	4 009	173	214	6

The first program in Table 3 is from the JAVA version of the Siemens program package, and the rest of the programs are from the SIR (Software-artifact Infrastructure Repository) [15]. The number of the lines of the codes in all the programs ranges from 181 to 4009, and the number of class methods ranges from 9 to 173. These are widely applied as the benchmark programs in the studies of the software bug location. We remove the v4 and v6 bug versions of the Print_tokens (these two bug version programs are data file errors). And retain 29 bug versions of the program in Jtcas (excluding the cases where the class attribute member definition error and so on do not fall under the category of the class method error). In addition, we select two distribution versions of the Nanoxml and regard them as the independent research objects (They differ relatively greatly in the number of class methods, which is 118 and 173, respectively), and select 13 bug versions from them. In this way, we have studied a total of 47 bug versions of the program.

With the Dell server (Intel(R) Xeon(R) 3.07GHz CPU, 16GB of memory), we use the OpenJDK 1.7, Python 2.7, R2.31 as well as the bug location prototype tool MLM-FL developed by our team to establish the software environment for the experimental running on the basis of the Ubuntu 64-bit operating system (version 12.04).

4.2. Evaluation Index

The experimental evaluation indexes adopt the Expense index put forward by Renieris M and ReissSP. This index has been widely by many researchers. In this paper, we use Expense to represent the percentage of the number of class methods that the debugging personnel need to identify the class methods that contain the bug in all the class methods. And the calculation method is shown in the equation (3) as the following.

$$\exp ense(f) = 100 \bullet \frac{rank(f)}{|M|} \tag{3}$$

4.3. Experimental Design

In this paper, five methods based on the measurement method for the degree of skepticism of the program spectrum are selected for the comparison experiment. They are Naish1, Naish2, Wong Russel & RaoBinary equations which are the optimal formulas screened out by Xie Xiaoyuan et al. [22] through the theoretical analysis of 30 skepticism degree calculation formulas, which are as the following in turn:

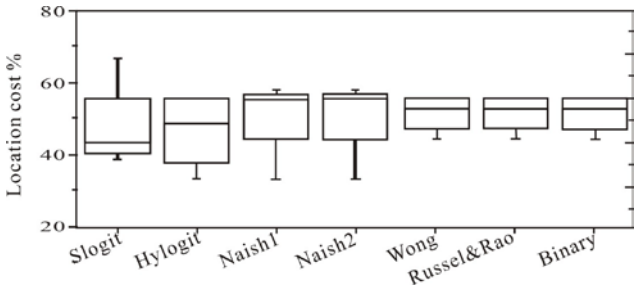
$$Naish1(f) = \begin{cases} -1, N_{CF}(f) < N_F \\ N_s - N_{CS}(f) = N_F \end{cases} \tag{4}$$

$$Naish2(f) = N_{CF} - \frac{N_{CS}(f)}{N_{CS}(f) + N_{CS}(f) + 1} \tag{5}$$

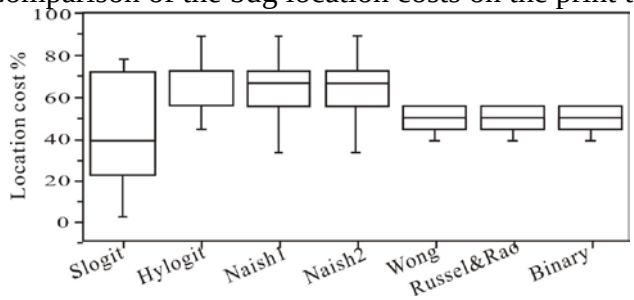
$$Wong(f) = N_{CF}(f) \tag{6}$$

$$Russel\ \&\ Rao(f) = \frac{N_{CF}(f)}{N_{CF}(f) + N_{UF}(f) + N_{CS}(f) + N_{US}(f)} \tag{7}$$

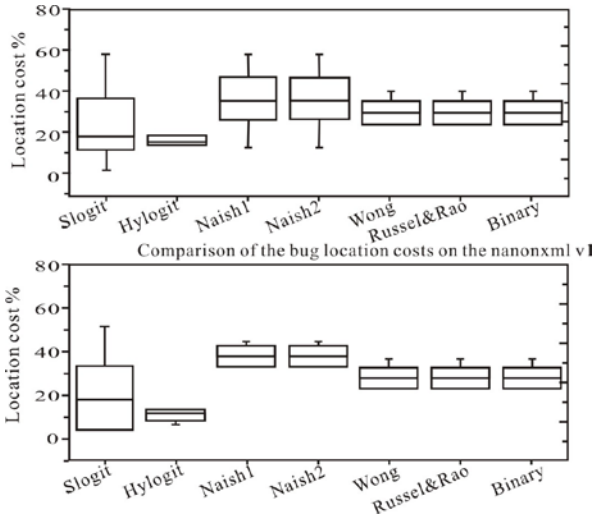
$$Binary(f) = \begin{cases} 0, & \text{if } N_{CF}(f) < N_F \\ 1, & \text{if } N_{CF}(f) = N_F \end{cases} \tag{8}$$



(a) Comparison of the bug location costs on the print tokens



(b) Comparison of the bug location costs on the jtcas



(c) Comparison of the bug location costs on the nanonxml

Figure 2 Comparison of the location costs of the seven methods on the four programs

Among them, f stands for the class method, N_{CF} and N_{CS} stand for the number of times that the class method is covered by the failed test cases and successful test cases, and N_{UF} and N_{US} stand for the number of times that the class method is not covered by the failed test cases and successful test cases. Then the $N_S = N_{CS} + N_{US}$ and $N_F = N_{CF} + N_{UF}$ is established.

In order to carry out the fuzzy multi-objective model analysis, we have designed the static indexes LOC and VG of the measurement method, which stand for the number of the lines in the code and the circle complexity in the class method, respectively. N_{CS}, N_{US}, N_{CF} and N_{UF} are used to measure the dynamic behavior feature of the class method during testing process. In this way, we can develop the fuzzy multi-objective model on the variable set X = {LOC, VG, N_{CS}, N_{US}, N_{CF}, N_{UF}}. For the purpose of discussing the RQ1, we have developed two types of fuzzy multi-objective models: (1) The fuzzy multi-objective model that makes use of only the static information LOC and VG, which is denoted as Slogit; (2) The mixed variable fuzzy multi-objective model that is added with the dynamic information development on the basis of the variable set X, which is denoted as Hylogit.

4.4. Results Analysis and Discussion

Next, the results of the experiments on the four object programs are displayed and analyzed in turn (as shown in Figure 2). Figure 2 (a) shows the cost of the bug location using the seven methods in the Print_tokens. The mean values of the seven methods in the figure are compared. And it is easy to see that the mean cost of the Slogit is the smallest, followed by the mean cost of Hylogit, which shows that using the fuzzy multi-object model method can result in significantly lower cost than the other five optimal formulas for location on the basis of the program spectrum. However, we have also found that the cost of bug location of Slogit is significantly lower than that of Hylogit. The reason is that Hylogit has increased the variable that represents the dynamic information, and the fitting degree of the fuzzy multi-objective model constructed by N_{CS}, N_{US}, N_{CF} and N_{UF} is reduced, which is not as good as the model fitting of the Slogit method. In addition, the location costs of Naish1 and Naish2 are exactly the same.

In Figure 2, the location costs of the seven methods on the four programs, i.e. Wong, Russel & Rao, and Binary are compared, which are totally consistent. This has exactly confirmed the conclusion of Xie Xiaoyuan et al.: The efficiency of the first two formulas falls under the same optimal group, and the last three formulas fall under another optimal group.

Figure 2 (b) shows the comparison of the costs for the location of 29 bugs in the Jtcas. From Figure 2 (b), it can be seen that Slogit has the lowest mean cost, followed by Wong, Russel & Rao and Binary. However, we noticed that Slogit has a very large range of fluctuations, which is 72.43% and 22.23% at the upper and lower quarter points. After careful investigation, we have found out

that: When the class method with a very small number of lines of code contains a bug, the location cost of the Slogit method will increase significantly; otherwise, the location cost will decrease significantly. This finding is verified in the experiments of the other three experimental subjects in this paper. It is found that this rule is obviously present in the Print_tokens experiment. However, the performance in the two experiments of the Nanoxml program is not very obvious. And only a small amount of data is consistent with this rule. After analysis, we believe that the reason for this phenomenon may be that both the Jtcas and Print_tokens have extremely small size of the codes, and that the established model is affected by the LOC relatively significantly, which leads to the relatively large fluctuation in the results.

Figures 2 (c) and Figure 2 (d) show the comparison of the experimental results of the two versions of the Nanoxml, respectively. These two versions of the program have relatively large scale (up to 3 ~ 4 KLOC) and relatively large number of methods (which reach 118 and 173, respectively). Therefore, the experimental results are representative to a certain extent. It can be seen from the figure that the cost of the bug location by using the fuzzy multi-objective model is lower than that by using the five optimal formulas on the basis of the program spectrum, and the cost of the Slogit method is higher than that of the Hylogit method. We have found that under the premise of the significant increase in the size of the program and the number of class methods, the introduction of the dynamic information during the program testing process will be conducive to constructing the multivariable Lo-gistic model (the fitness of the model is improved). Hence, the efficiency of bug location based on the established model will also be improved accordingly.

5. Conclusions

In this paper, a kind of bug location framework that integrates the fuzzy multi-objective model is put forward. The framework filters a set of static features that describe the level of the method class and the indexes of the real-time running information, applies the multivariate Logistic fuzzy multi-objective modeling, uses it to locate the software bugs. The experiments show that the application of the multivariable Logistic fuzzy multi-objective and the introduction of the program running information can significantly improve the efficiency of the bug location.

References

[1] Smidts, C., Shi, Y., Prediction, S. R., Test, M., & Prediction, D. L. (2015). Advances in software engineering. Computer, 29(10), 47-58.

[2] Ekanayake, J., Tappolet, J., Gall, H. C., & Bernstein, A. (2012). Time variance and bug prediction in software projects. Empirical Software Engineering, 17(4-5), 348-389.

[3] Harter, D. E., Kemerer, C. F., & Slaughter, S. A. (2012). Does software process improvement reduce the severity of bugs? a longitudinal field study. IEEE Transactions on Software Engineering, 38(4), 810-827..

[4] Mora-Gutiérrez, R. A., Rincón-García, E. A., Ponsich, A., Ramírez-Rodríguez, J., & Méndez-Gurrola, I. I. (2016). Influence of social network on method musical composition. Artificial Intelligence Review, 46(2), 225-266.

[5] Subramanyam, R., & Krishnan, M. S. (2015). Empirical analysis of ck metrics for object-oriented design complexity: implications for software bugs. IEEE Transactions on Software Engineering, 29(4), 297-310.

[6] Bella, E. D., Fronza, I., Phaphoom, N., Sillitti, A., Succi, G., & Vlasenko, J. (2013). Pair programming and software bugs--a large, industrial case study. IEEE Transactions on Software Engineering, 39(7), 930-953.

[7] Harter, D. E., Kemerer, C. F., & Slaughter, S. A. (2012). Does software process improvement reduce the severity of bugs? a longitudinal field study. IEEE Transactions on Software Engineering, 38(4), 810-827.

[8] Bowes, D., Hall, T., & Petrić, J. (2017). Software bug prediction: do different classifiers find the same bugs?. Software Quality Journal(1), 1-28.

[9] Ljubomir Lazić, & Stevan Milinković. (2015). Reducing software bugs removal cost via design of experiments using taguchi approach. Software Quality Journal, 23(2), 267-295.

[10] Ullah, N. (2015). A method for predicting open source software residual bugs. Software Quality Journal, 23(1), 55-76.

[11] Sharafi, S. M. (2012). Shadd: a scenario-based approach to software architectural bugs detection. Advances in Engineering Software, 45(1), 341-348.

[12] Monden, A., Tsunoda, M., Barker, M., & Matsumoto, K. (2017). Examining software engineering beliefs about system testing bugs. It Professional, 19(2), 58-64.

[13] Ullah, N., Morisio, M., & Vetrò, A. (2015). Selecting the best reliability model to predict residual bugs in open source software. Computer, 48(6), 50-58.

[14] Mora-Gutiérrez, R. A., Rincón-García, E. A., Ponsich, A., Ramírez-Rodríguez, J., & Méndez-Gurrola, I. I. (2016). Influence of social network on method musical composition. Artificial Intelligence Review, 46(2), 225-266.

[15] Wang, S., & Yao, X. (2013). Using class imbalance learning for software bug prediction. IEEE Transactions on Reliability, 62(2), 434-443.

[16] Shepperd, M., Song, Q., Sun, Z., & Mair, C. (2013). Data quality: some comments on the nasa software bug datasets. IEEE Transactions on Software Engineering, 39(9), 1208-1215.