Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 93 (2016) 924 – 930

Procedia Computer Science

6th International Conference On Advances In Computing & Communications, ICACC 2016, 6-8 September 2016, Cochin, India

A Novel Machine Learning Approach For Bug Prediction

Shruthi Puranik[a], Pranav Deshpande[a], K Chandrasekaran[a,*]

[a]*National Institute of Technology, Karnataka, Surathkal-575025, India*

Abstract

With the growing complexities of the software, the number of potential bugs is also increasing rapidly. These bugs hinder the rapid software development cycle. Bugs, if left unresolved, might cause problems in the long run. Also, without any prior knowledge about the location and the number of bugs, managers may not be able to allocate resources in an efficient way. In order to overcome this problem, researchers have devised numerous bug prediction approaches so far. The problem with the existing models is that the researchers have not been able to arrive at an optimized set of metrics. So, in this paper, we make an attempt to select the minimal number of best performing metrics, thereby keeping the model both simple and accurate at the same time. Most of the bug prediction models use regression for prediction and since regression is a technique to best approximate the training data set, the approximations don't always fit well with the test data set. Keeping this in mind, we propose an algorithm to predict the bug proneness index using marginal R square values. Though regressions are performed as intermediary steps in this algorithm, the underlying logic is different in nature when compared with the models using regressions alone.

Keywords: Bug prediction metrics, Multiple regression, Marginal R square, F-measure.

1. Introduction

Since the inception of software development, bug fixing has been considered as one of the tedious tasks, mainly because of its inherent uncertainty. In addition to this, the process of fixing bugs is also time consuming. In fact, according to Rogue Wave Software, half the developer's time is spent in debugging. To add to these problems, some bugs can't be detected at an early phase as it might take time for their effects to manifest. The severity of bugs increases when fixing them might allow other unanticipated bugs to creep in. Considering the aforementioned reasons, it can be said that the process of bug-fixing has a major stake in the software development time. In order to alleviate the issue of bug fixing, bug prediction has been studied extensively by the researchers. Many machine learning driven prediction models have been built and tested on various grounds.

Given the historical data of a project (data extracted from CVS, Bugzilla, etc), the task is to predict the number of bugs in the future releases. Since indicating an exact number of bugs expected would be difficult, we resort to determining the probability of the occurrence of bugs. Filtering the input further, more specific prediction models can

[*] Corresponding author. Tel.: +91-824-247-4000 Extn. 3400, 3044.
E-mail address: kch@nitk.ac.in

be built to cater the requirements. For instance, a model can be built to predict the occurrence of most critical bugs. And, these prediction models can be evaluated on a variety of criteria, depending on the nature of the results required.

The prediction models can be incorporated in the development environments so that the developers can get feedback as and when they write codes. Although models can be built near to perfection, it is unreasonable to expect the prediction to be 100% accurate. Not all the reasons for the occurrence of bugs can be related to the history of the software. Although few in number, false predictions are inevitable. And, these false predictions can be of two types, one in which a clean code is classified as buggy and the other one is which a buggy code is classified as clean. In order to make developers trust the model, it is necessary to get an optimal model, balancing the false predictions. Despite the lack of standard benchmarks for the comparison of the models, efforts have been made to study the models in terms of the level of accuracy and complexity.

The selection of metrics being the first and the foremost stage in the bug prediction, has a great influence in the accuracy and the complexity of the model. More the number of metrics in the model, more complex is the process[17]. Inclusion of irrelevant metrics can cause the accuracy to drop considerably. Moreover, the impact of these metrics can be counter-intuitive. It is possible that the seemingly important metrics can have less equity in the bug prediction, attributing to some subtle, unforeseen factors. It turns out that the selection of appropriate metrics is possible only by empirical verifications. Keeping in mind the severe consequences that the poor metrics selection can engender, we devote extra efforts to carefully select the determinant metrics.

Bug prediction can be considered as a function of metrics and the nature of this function remains unknown. In the past, researchers have designed prediction models using determinants like past bugs[19], code churn[15], number of developers, file length, code refactoring[20] and so on. After making a comprehensive study of the developed models and approaches, we devise a novel approach to estimate the bug proneness of source code at the class level.

The rest of the paper is organized as follows. Section 2 enumerates the past research done in the area of bug prediction and likes. Section 3 gives a detailed description of the algorithm proposed for bug proneness index determination. And, we present the results obtained when the proposed model is implemented in the section 4. Also, a comparison is made between the proposed model and few other representative models in the same section. In section 5, we discuss the threats to the validity of the model. The paper ends with the conclusion drawn from this study (section 6). Furthermore, a small discussion on the possibility of future work in this area is also presented (section 6).

2. Literature Survey

In the past, there have been many works in the areas of bug prediction and likes[2]. Different measures have been used to predict the incidence of bugs in the source codes. Nagappan et. al found that fault prone software entities are statistically correlated to the the complexity of the code[4]. Hassan et al. proposes that the code change complexity is a good determinant of the occurrence of bugs[2]. There have been works in which the history information was used to study about refactorings. In this regard, van Rysselberghe has given approaches to recognize move and inheritance change operations and refactorings as well[5]. The requirements of refactorings have also been discussed, i.e. the error proneness of refactorings has been addressed[6]. However, this approach remained inconclusive. Using classification trees, Khoshgoftaar et al.[12] predicted error prone modules. Nagappan et al. showed that there is a high positive correlation between the density of defects obtained by performing a static analysis and the pre-release defect density[10]. The types of changes between releases of software have been studied by Fluri et al.[9]. Kim et al. also used software history to predict error proneness of software entities[11]. The relationship between distributed development and software quality has been discussed using a case study of Windows Vista[8]. A recency weighting approach has been proposed in which the historical bug data has been analysed both locally and globally[19]. Kim. et al. proposes noise detection and elimination algorithm to remove the FP (false postive) and FN (false negative) noises present in the defect data[13].

3. Proposed Model

Regression is one of the machine learning methods to derive the relationship between variables of interest, i.e. regression enables the determination of best fit curve to the data in hand. Many regression methods are known and depending on the nature and the number of both independent and dependent variables, a suitable method can be chosen. The regression works to minimize the sum of the squares of the errors (least squares). This amounts to saying that the regression techniques provide the best fit curve globally, though there may be some discrepancies locally. But, the performance of regression models on the test data may not chime with that of training data.

It has been shown that the linear model is the close approximation for the bug predictions[18]. Moreover, the linear model is the simplest of all the models known. So, we resort to a linear model on the carefully chosen metrics and their weights determined using the proposed algorithm. But instead of using the regression coefficients directly in the model, we use the differences in the coefficients of determination (R square) as the weights. The steps in the proposed model are described below according to the algorithm 1.

3.1. *Selection of metrics*

The input to any regression model is a good set of metrics (independent variables) which can determine the dependent variable to the highest accuracy achievable. We choose metrics after a careful evaluation of their coefficients of determination (R square) in simple linear regressions. Taking the actual bug count as the dependent variable and the metric in hand as the independent variable, we perform simple linear regression to determine the coefficient of determination. Once the coefficients of determination of every metric are obtained through linear regressions, we sort the metrics in the decreasing order of their coefficients of determination. Finally, we select just the top 'n' metrics and these metrics will be used in the next step. Here, 'n' is the number of metrics that will be used in the final prediction model. And this number can vary from one project to another, depending on the tolerance for the complexity of the prediction model.

3.2. *Perform multiple regression*

Linear multiple regression is a good machine learning method to obtain the best fit curve for the training data set. Taking the 'n' chosen metrics as the independent variables and the actual bug count as the dependent variable, we perform a linear multiple regression. The coefficient of determination obtained will be used in the further steps to determine the marginal R square values, which in turn will be used to determine the weights of the metrics in the final model.

3.3. *Compute marginal R square values*

In order to find the weights of the 'n' chosen metrics, we perform 'n' linear multiple regressions taking all the 'n' chosen metrics except one (a total of (n-1)) as the independent variables and the actual bug count as the dependent variable. The differences in the coefficients of determination (marginal R square) obtained in the current regressions and that obtained in the previous step are determined and these values when normalized in the range of 0 to 1 give the weights of the metrics left out in the current regressions. Higher marginal R square value of a metric indicates that the error in the regression is more because this metric was not considered. This, in turn, means that the contribution of the metric in the bug prediction is high. So, we use these normalized marginal squares as the weights of the metrics.

3.4. *Compute bug proneness index*

Once the weights of the 'n' chosen metrics are obtained, the next step is to normalize the 'n' metric values of the test set in the range of 0 to 1. The sum of the products of the normalized marginal R squares obtained in the previous step and the corresponding normalized metric values gives the bug proneness index in the range of 0 to 1 as shown in the method 2. More the index close to 1, more buggy is the class.

4. Results

We chose Eclipse JDT Core[18] for the testing of our model and this dataset is freely available at *bug.inf.usi.ch*. This data set is designed to perform bug prediction at the class level. This approach can be extended to file, package, project and other levels in a straight forward manner. The results obtained when each of the above mentioned 4 steps are acted upon this data set are shown below.

Algorithm 1: Proposed algorithm to find the weights of metrics

```
Data: The training data set D of m metrics and actual bug counts, desired number of metrics n
Result: Weights of n metrics
1  Normalize all the m metrics and the actual bug counts in the range of 0 to 1.
2  for each metric i in m normalized metrics do
3      Perform the simple linear regression of i (independent variable) vs actual bug count, b (dependent variable)
         and determine R square value.
4  end
5  Sort the m metrics in the decreasing order of R square values and select top 'n' of them for the further process.
6  Perform multiple linear regression of the top 'n' normalized metrics (independent variables) vs actual bug count,
     b (dependent variable) and determine the R square value. Let it be $r_u$.
7  for each metric i in the set of top 'n' normalized metrics do
8      Perform multiple linear regression of remaining n-1 metrics (independent variables) vs normalized actual
         bug count, b (dependent variable) and determine the R square value. Let it be $r_{-i}$.
9      Calculate the difference $d_i = r_u - r_{-i}$.
10 end
11 Normalize all the 'n' $d_i$s in the range of 0 to 1. Let the normalized values be $d_{ni}$ for i=1 to n. These $d_{ni}$s form the
     weights of n metrics
```

Algorithm 2: Computation of bug proneness index

```
Data: The test data T=$v_1$, $v_2$,...,$v_n$ of n normalized metrics, and weights $d_{ni}$s of 'n' metrics
Result: Bug proneness index I in the range of 0 to 1
1  $I = \sum_{i=1}^{n}(d_{ni} * v_i)$
```

4.1. Running the proposed algorithm

4.1.1. Selection of metrics

The data set had many metrics like number of past bugs, lines of code, depth of inheritance and so on. After varying the number of chosen metrics 'n', we saw that the optimal value for 'n' (project dependent) is 4. When simple linear regressions were performed on the available metrics, the top 4 metrics in the decreasing order of the coefficients of determination were 'number of bugs found until', 'number of versions until', 'LOC' (lines of code) and 'entropy'. The obtained R square values are shown in table 1.

Table 1. Simple linear regression results.

Metric	R square
number of bugs found until	0.4491
number of versions until	0.3609
LOC	0.3553
entropy	0.1512

4.1.2. Perform multiple regression

When multiple regression is performed on the above mentioned 4 metrics, the obtained coefficient of determination is 0.4762.

4.1.3. *Compute marginal R square values*

When multiple regressions are done on different sets of n-1 metrics, the results obtained are as shown in the table 2. Each row indicates the results obtained when the regression was performed on all the metrics except the one mentioned in the first column. The final weights in the range of 0 to 1 are also shown in the last column of table 2.

Table 2. Weights of the metrics.

Metric	R square	marginal R square	Weight
number of bugs found until	0.4364	0.0397	0.5910
number of versions until	0.4751	0.0011	0.0159
LOC	0.4521	0.0241	0.3581
entropy	0.4738	0.0023	0.0350

4.1.4. *Compute bug proneness index*

We choose two classes of this software for the verification of the model. The metric values and the actual bug counts are normalized in the range of 0 to 1. The results are shown in the table 3. From the table, we can see that the actual normalized bug count and the predicted bug proneness index are very close to each other in their values.

Table 3. Test data results.

Number of bugs found until	Number of versions until	LOC	Entropy	Actual Bug count	Bug proneness index
0.0187	0.0028	0.0118	0.0260	0	0.0133
0.0888	0.0706	0.0473	0.6173	0.1111	0.1333

4.2. *Comparison of different models*

We chose two existing models for comparison, the simple linear regression based model and the multiple regression model.

The different training and test instance results can be divided into 4 classes as shown in table 4. The different parameters for the measurement of the quality of the model are accuracy, precision, recall and F-measure. Accuracy is the proximity of measurement results to the true value (Eq. 1). And, precision can be seen as a measure of exactness or quality (Eq. 2), whereas recall is a measure of completeness or quantity (Eq. 3). Finally, F-measure is the harmonic mean of precision and recall (Eq. 4).

Table 4. Classification of data instances.

Representation	Meaning
$n_{b->b}$	number of buggy instances classified as buggy
$n_{b->c}$	number of buggy instances classified as clean
$n_{c->b}$	number of clean instances classified as buggy
$n_{c->c}$	number of clean instances classified as clean

$$Accuracy = \frac{n_{b->b} + n_{c->c}}{n_{b->b} + n_{b->c} + n_{c->b} + n_{c->c}} \tag{1}$$

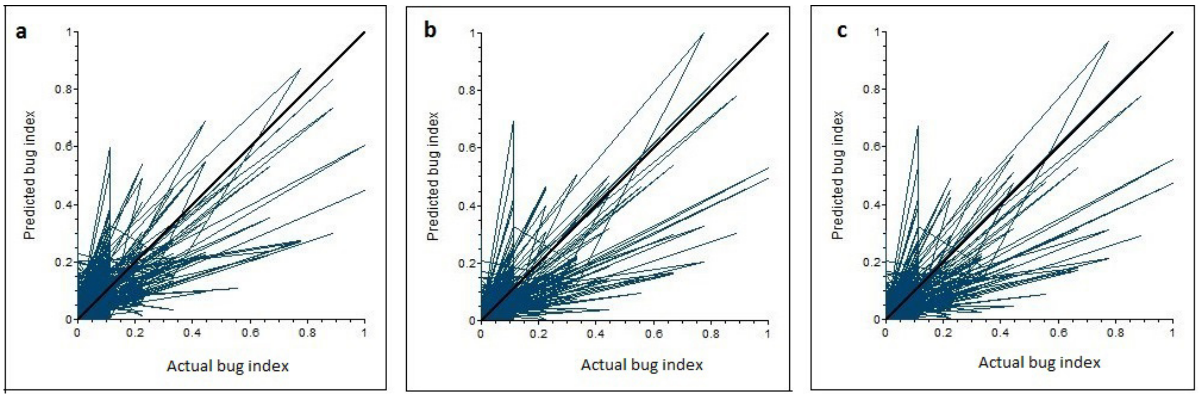

Fig. 1. Performance measurement in bug prediction models in the order: (a) Simple prediction model (b) Multiple regression model (c) Proposed model.

$$Precision\ (bug) = \frac{n_{b->b}}{n_{b->b} + n_{c->b}} \tag{2}$$

$$Recall\ (bug) = \frac{n_{b->b}}{n_{b->b} + n_{b->c}} \tag{3}$$

$$F - measure\ (bug) = \frac{2 * P(b) * R(b)}{P(b) + R(b)} \tag{4}$$

The results are shown separately for training and test data sets. The table 5 indicates the model having the highest value for the parameter in the first column and these results are shown for both the training and test data sets. 1 indicates the simple regression model, 2 being the multiple regression model and 3 the proposed model. The graphs shown below (Fig. 1) compare the actual bug counts with the predicted bug proneness index for different models including the proposed one. The graph of the proposed model (Fig. 1 (c)) indicates the closeness of the results to the actual bug counts. In fact, the graph coincides with the line y=x (Actual normalized bug count = Predicted bug proneness index) at some points. It is evident that the proposed model performs the best for both the training and test data sets when the composite measure of precision and recall (F-measure) is considered. Though other models might perform better when other measures are considered, F-measure is a composite, comprehensive measure and a good value of F-measure is an indication of high quality of the model. Though we see that the multiple regression model performs well when training data set is considered it is the test data results that are more important because the very purpose of the model is to predict the bugs in the future.

Table 5. Comparison of various prediction models.

Measure	Training	Test	Overall
Accuracy	2	3	2
Precision	2	3	3
Recall	1	1	1
F-measure	3	3	3

5. Threats to validity

In the first place, the data set chosen might be biased or incomplete. Depending on the proficiency of the authors and other factors, the nature and the number of bugs reported might vary. Also, since the experiment was performed on just few data sets, there is a possibility that the proposed model may not perform similarly with other data sets. In addition to this, when metrics are selected for the model, it must be ensured that they are independent of each other. If the predictor metrics are highly correlated, then the resulting linear equation might have redundant weights and hence false results. However, this collinearity between the metrics can be eliminated to a fair extent by using suitable methods like Partial Least Squares Regression (PLS), Principal Components Analysis[22] and so on.

6. Conclusion and Future Work

After making a comprehensive study of the different existing prediction models, we realized that there should be a deterministic strategy to select the metrics for predicting the bug count. Also, when regression was used as it is for bug prediction, though the sum of the squares of the errors was the least for training set of data, the test data results were not very convincing. Considering this, we developed a model that performs acceptably well on both the training and the test data sets. This model can be run on any dataset that has the relevant metric values in it. We have used the datasets[18] that are simple and readily available. The results show that the composite measure of precision and recall (F-measure) is the highest in the proposed model for both the training and the test data sets, in comparison with other existing models. This model, when integrated with the software development environments, will lead to reasonably accurate predictions. Furthermore, research has been made in adapting the fault prediction model to allow inter language reuse[16]. In the future, we will attempt to include this to make our solution more encompassing.

References

1. A. E. Hassan and R. C. Holt, *The top ten list: dynamic fault prediction*, Proceedings of the 21st International Conference on Software Maintenance, Budapest, Hungary, September 2005, pp. 263272.
2. A. Hassan, *Predicting faults using the complexity of code changes*, ICSE pages 78-88, 2009.
3. T. H. D. Nguyen, B. Adams, and A. E. Hassan, *Studying the impact of dependency network measures on software quality*, In ICSM 10, pp. 110.
4. N. Nagappan, T.Ball and A. Zeller, *Mining Metrics to Predict Component Failures*, In Proceedings of the 28th international conference on Software engineering, 2006.
5. F. Van Rysselberghe, *Studying Historic Change Operations: Techniques and Observations*, PhD thesis, Universiteit Antwerpen, 2008.
6. P. Weigerber and S. Diehl, *Are refactorings less error-prone than other changes?*, In Proceedings of the International Workshop on Mining Software Repositories (MSR 06). ACM, 2006.
7. B. Fluri, M. Wursch, M. Pinzger, and H. C. Gall, *Change distillingtree differencing for fine-grained source code change extraction*, Transactions on Software Engineering, 33(11):725743, November 2007.
8. C. Bird, N. Nagappan, P. Devanbu, H. Gall and B. Murphy, *Does distributed development affect software quality? An empirical case study of Windows Vista*, In Proceedings of the 31st International Conference on Software Engineering, 2009.
9. B. Fluri, H. C. Gall, and M. Pinzger, *Fine-grained analysis of change couplings*, In Proceedings of the Fifth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM05), 2005
10. N. Nagappan and T. Ball, *Static analysis tools as early indicators of pre-release defect density*. In Proceedings of the International Conference on Software Engineering, May 2005.
11. S. Kim, T. Zimmermann, E. J. Whitehead, Jr., and A. Zeller, *Predicting faults from cached history*. In Proceedings of the International Conference on Software Engineering, May 2007.
12. T. M. Khoshgoftaar, X. Yuan, E. B. Allen, W. D. Jones, and J. P. Hudepohl, *Uncertain classification of fault-prone software modules*, Empirical Software Engineering, 7(4):297318, December 2002.
13. S. Kim, H. Zhang, R. Wu and L. Gong, *Dealing with noise in defect prediction*, Proceedings of the 33rd International Conference on Software Engineering, 2011.
14. N. E. Fenton and M. Neil, *A critique of software defect prediction models*, IEEE Transactions on Software Engineering, 25(5):675689, September 1999.
15. N. Nagappan & T. Ball, *Use of Relative Code Churn Measures to Predict System Defect Density*, ICSE 2005, USA.
16. S. Watanabe, H. Kaiya & K. Kaijiri, *Adapting a fault prediction model to allow inter language use*, PROMISE'08, Leipzig, Germany.
17. Shivkumar Shivaji, E. James Whitehead, Jr. Ram Akella and Sunghun Kim, *Reducing features to improve bug prediction*, 2009 IEEE/ACM International Conference on Automated Software Engineering.
18. M. D'Ambros, M. Lanza & R. Robbes, *An Extensive Comparison of Bug Prediction Approaches*, MSR 2010, 7th IEEE Working Conference on Mining Software Repositories, pp. 31 - 41. IEEE CS Press.
19. H. Joshi, C. Zhang, S. Ramaswamy & C. Bayrak, *Local and global recency weighting approach to bug prediction*, Fourth International Workshop on Mining Software Repositories, 2007.
20. J. Ratzinger, T. Sigmund & H.C. Gall, *On the relation of refactoring and software defects*, MSR-2008, Leipzig, Germany.
21. T. Mende and R. Koschke, *Effort-aware defect prediction models*, In CSMR 10, pp. 107116, 2010.
22. *Handling Multicollinearity in Regression Analysis*, April 2013, Retrieved from http://blog.minitab.com/blog/understanding-statistics/handling-multicollinearity-in-regression-analysis.