Testing and Code Review Based Effort-Aware Bug Prediction Model

K. Muthukumaran, N.L. Bhanu Murthy, G. Karthik Reddy and Prateek Talishetti

Abstract Incremental and iterative processes compel vendors to release numerous versions of the software and quality of each and every version is the top most priority for vendors. Bug prediction models attempt to rank or identify the bug prone files so that quality assurance team can optimally utilize their resources to ensure the quality delivery. Recently, Researchers proposed effort aware bug prediction models wherein files are ranked not only based on their bug proneness but also on the effort that is required to perform quality assurance activities like testing, code review etc. on bug prone files. And Cyclomatic Complexity is being considered as a metric for the effort required irrespective of the assurance activity that will be undertaken during post prediction phases. We propose effort aware bug prediction model that will consider the effort required to perform the specific quality assurance activity and empirically prove the supremacy of these models over the existing effort aware models.

Keywords Defect prediction · Evaluation · Effort estimation · Cost-benefits

K. Muthukumaran · N.L. Bhanu Murthy (✉) · P. Talishetti
BITS Pilani Hyderabad Campus, Hyderabad, India
e-mail: p2011415@hyderabad.bits-pilani.ac.in

K. Muthukumaran
e-mail: p2011415@hyderabad.bits-pilani.ac.in

P. Talishetti
e-mail: h2011013@hyderabad.bits-pilani.ac.in

G. Karthik Reddy
Stony Brook University, New York, USA
e-mail: kgreddy@cs.stonybrook.edu

R. Lee (ed.), *Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing*, Studies in Computational Intelligence 653, DOI 10.1007/978-3-319-33810-1_2

1 Introduction

Quality has always been a non-compromising priority task for all IT companies and post release bugs or bugs experienced by end customer hampers the quality heavily. Quality assurance activities, such as tests or code reviews, are an expensive, but vital part of the software development process. Any support that makes this phase more effective may thus improve software quality and reduce development costs. With time and manpower being limited or gets limited for quality assurance activities, it would be great if one can predict bug prone files in early stages of software development so that the project team can focus more on those bug prone files. Also, it helps the team to allocate or manage their critical resources well. It has been observed that the distribution of defects follows a Pareto-principle, that is, that most of the bugs are located in only few files [7]. Hence the research on bug prediction models have become significant and more than hundred research papers have been published.

The project team might perform quality assurance activities on bug prone files to catch bugs earlier to release of the software. The quality assurance activities could be peer code reviews, expert code reviews, testing etc. The team can choose one or more quality assurance activities that are apt for the project so that maximum number of defects will be uncovered. Also, project team has to allocate resources/revenues to complete these activities. Though it is ideal to get assurance activities done for all of the bug prone files, it might not be feasible due to cost implications. Hence project team would act on top x% of bug prone files based on the costs that can be borne by the team.

The top x% of files will be selected by sorting files in the descending order of number of predicted bugs and this approach has been adopted by researchers Ohlsson and Alberg [16], Khoshgoftaar and Allen [9]. We illustrate the limitations of this approach with the following example. Generally it is assumed that after performing the quality assurance activities like code review of files, all bugs in the file will be uncovered. Suppose project team would like to take up code review of top x% of files and there are two files File A, File B with the following characteristics.

File	LOC	# of predicted bugs	Effort per bug
File A	100	2	50
File B	1200	4	300

File B appears prior to File A as per the above mentioned sorted order and File A might have been ignored when project team selects top x% of files. The Effort per bug indicates the effort required to uncover one bug and is defined as the ratio of LOC and the number of bugs. The effort per bug for File A (50) is quite low as compared to File B (300) whereas the classical ordering ranks File B on top of File A. This issue has been addressed by Mende and Koschke [13].

Mende and Koschke [13] proposed a strategy to include the notion of effort into defect prediction models. They propose to rank files with respect to their effort per bug. They use McCabe's cyclomatic complexity as the surrogate measure for effort. They evaluate their new model against the naive model which just ranks files on the predicted number of bugs, by adopting to effort aware measures like CE [1]. They apply Demšar's non-parametric tests to conclude that their new model is significantly cost effective than the naive model.

Contributions: In this work we argue that the measure of effort should not be a generic measure such as cyclomatic complexity but instead it should be one that is specific to the kind of activity involved in the quality assurance process. We identify two most popular quality assurance activities namely code review and unit testing. We use lines of code to measure the effort involved in code reviews and the number of test cases to measure the effort in case of unit testing. We compare the cost effectiveness of our specific effort based models to generic effort based models.

2 Related Work

Many approaches in the literature use the source code metrics and change metrics as predictors of bugs. In this paper we have used the CK metrics suite [5], object oriented metrics, lines of code and some change metrics as features to build the bug prediction model. Basili et al. [3], Tang et al. [19], Cartwright and Shepperd [4], Subramanyam and Krishnan [18] explored the relationship between CK metrics and and defect proneness of files. Basili et al. [3] found that WMC, RFC, CBO, DIT and NOC are correlated with faults while LCOM is not associated with faults based on experiments on eight student projects. Tang et al. [19] validated CK metrics using three industrial real time systems and suggest that WMC and RFC can be good indicators of bugs. Cartwright and Shepperd [4] found DIT and NOC as fault influencers. Subramanyam and Krishnan [18] investigated the relationship between a subset of CK metrics and the quality of software measured in terms of defects. Though they proved the association, they conclude that this observation is not consistent with different OO languages like C++ and Java.

Nagappan et al. used a catalog of source code metrics to predict post release defects at the module level on five Microsoft systems [15]. Ostrand et al. [17] conclude that lines of code is a significant influence on the faults. Also their simple model based only on lines of code achieves roughly 73 % bugs in the top 20 % of files which is only 10 % less than their full model. Krishnan et al. [10] studied the performance of a large set of change metrics on seven versions of multiple Eclipse products between the years 2002–2010. They conclude that, the performance of change metrics improved for each product as it evolved across releases. For each product, they identified a small, stable set of change metrics that remained prominent defect predictors as the products evolved. For all the products, across all the releases, the change metrics "maximum changeset", "number of revisions" and "number of authors" were the good predictors.

Ohlsson and Alberg [16], Ostrand et al. [17] advocate models which rank files based on their fault proneness. Khoshgoftaar and Allen [9] coined the term Module-Order-Model (MOM). It enables to select a fixed percentage of modules for further treatment which is a more realistic scenario for projects with a fixed quality assurance budget. MOMs can be evaluated by assessing the percentage of defects detected at fixed percentages of modules. Ostrand et al. [17] propose a model which found up to 83 % of the defects in 20 % of the files. One way to graphically evaluate MOMs are lift charts, sometimes known as Alberg diagrams [16]. Ostrand et al. validates Pareto principle in bug prediction and their results show that 20 % of files have 84 % of defects. Ostrand et al. [17] conclude that a prediction using defect densities is able to find more defects in a fixed percentage of code, but argue that the testing costs are, at least for system tests, not related to the size of a file.

Mende and Koschke [12] compare the performance of various classifiers over lines of code based lift charts. They conclude that performance measures should always take into account the percentage of source code predicted as defective. In their subsequent work [13], they propose two strategies to incorporate the treatment effort into defect prediction models. But both their strategies include cyclomatic complexity as the effort for quality assurance. In this work we propose that specific effort measures perform better than generic effort based classifiers.

3 Metrics and Datasets

We have considered a hybrid of source code metrics which include the Chidamber and Kemerer metric suite [5], few object oriented metrics and change metrics. Table 1 describes details about metrics considered. We need to have the number of test cases for each source file to achieve the objectives of this work. The benchmarked open source datasets, that are used for bug prediction research, do not have the number of test cases against each file. Hence we, acquired data from a private software company for three projects in which a record of all testing activities for every file is recorded. For the sake of convenience and anonymity, let us consider them as datasetA, datasetB and datasetC. The details about the number of files and number of defects in each dataset are shown in Table 2.

4 Experiment and Results

We build defect prediction models using the metrics mentioned in Table 1 for each of the three datasets in Table 2. We use stepwise linear regression algorithm to build our model and predict the number of faults for a file. Regression algorithm takes the number of faults into consideration while building the model whereas a classification algorithm such as Naive Bayes classifier only considers the label of the file. Since we intend to predict effort per bug of a file we use stepwise linear

Table 1 Metrics

Source code metrics	
WMC	Weighted method count
DIT	Depth of inheritance
NOC	Number of children
CBO	Coupling between object classes
RFC	Response for a class
LCOM	Lack of cohesion in methods
CE	Efferent coupling
NPM	Number of public methods
CC	Cyclomatic complexity is a popular procedural software metric equal to the number of decisions that can be taken in a procedure
NOF	Number of fields in the class
NOI	Number of interfaces implemented by the class
LOC	Number of lines of code in the file
NOM	Number of methods in the class
Fan-in	Number of other types this class is using
Fan-out	Number of other types using this class
PC	Percentage of the file commented
Change code metrics	
noOfBugs	Number of bugs found and fixed during development
noOfStories	Number of stories this file is part of
noOfSprints	Number of sprints this file is part of
revisionCount	Number of commits this file is involved in
noOfAuthors	Number of authors who worked on this file during the release
LocA	Number of lines of code added in total to this file during development
MaxLocA	Max number of lines of code added among all commits to this file during development
AvgLocA	Average number of lines of code added among all commits to this file during development
LocD	Total number of lines of code deleted among all commits to this file during development
MaxLocD	Max number of lines of code deleted among all commits to this file during development
AvgLocD	Average number of lines of code deleted among all commits to this file during development
LocAD	Total number of lines of code added-deleted among all commits to this file during development
MaxLocAD	Max number of lines of code added-deleted among all commits to this file during development
AvgLocAD	Average number of lines of code added-deleted among all commits to this file during development

Table 2 Datasets

Projects	Number of files	Post release defects
DatasetA	2257	252
DatasetB	2557	652
DatasetC	2151	349

regression algorithm. Nagappan and Ball [14] also used stepwise linear regression to predict file defect density. We adopt the method of building bug prediction models using 2/3rd data and testing the models with remaining 1/3rd data. The division of training and testing data is being done randomly. The data is split randomly fifty times and the average of values were recorded as final results in experiments conducted for this work.

Khoshgoftaar and Allen [9] proposed software quality model, MOM (Module Order Model) that is generally used to predict the rank-order of modules or files according to a quality factor [9]. We take this quality factor to be number of defects in our study. We apply stepwise linear regression algorithm and order files as per their predicted value of bugs. MOMs can be evaluated by assessing the percentage of defects detected against the fixed percentages of files. One way to graphically evaluate MOMs are lift charts, sometimes known as Alberg diagrams [16]. They are created by ordering files according to the score assigned by a prediction model and denoting for each fixed percentage of files on the x-axis, the cumulative percentage of defects identified, on the y-axis. Thus for any selected percentage of files, one can easily identify the percentage of predicted defects. We have drawn Alberg Diagrams [16] for two module-order models namely actual and simple prediction for each of the project and the plots are depicted in Fig. 1a–c.

The optimal or ideal or actual model (represented as red curve in the above plot) is drawn by arranging the modules in the decreasing order of actual defects. The simple prediction model(represented as blue curve in the plots) is drawn by arranging the files in the decreasing order of predicted number of faults. The predicted number of faults is value outputted by the stepwise linear regression algorithm for each file. The optimal or ideal model represents the best possible model that any defect prediction model can aspire for. The goodness of any prediction model depends on its closeness to the optimal curve, closer to the optimal curve better the prediction model is. And the major advantage of this method is that the model is flexible enough to predict bugs depending on the costs that can be borne by project team towards quality assurance activities.

The Pareto principle for this problem is '20 % of files should have 80 % of bugs' and we have evaluated the number of bugs that can be found in top 20 % of files for the prediction models across all the datasets and these values are tabulated in Table 3. The percentage of bugs found using these models across projects is found to be varying between 39.56 and 80 %. These results are depicted visually in Fig. 1a–c. Now, we shall explain the results of datasetB. The results, suggest that if we do quality assurance activities on the 20 % of files we can catch 80 % of the bugs. Although this is an excellent result, we shall now explore the amount of effort

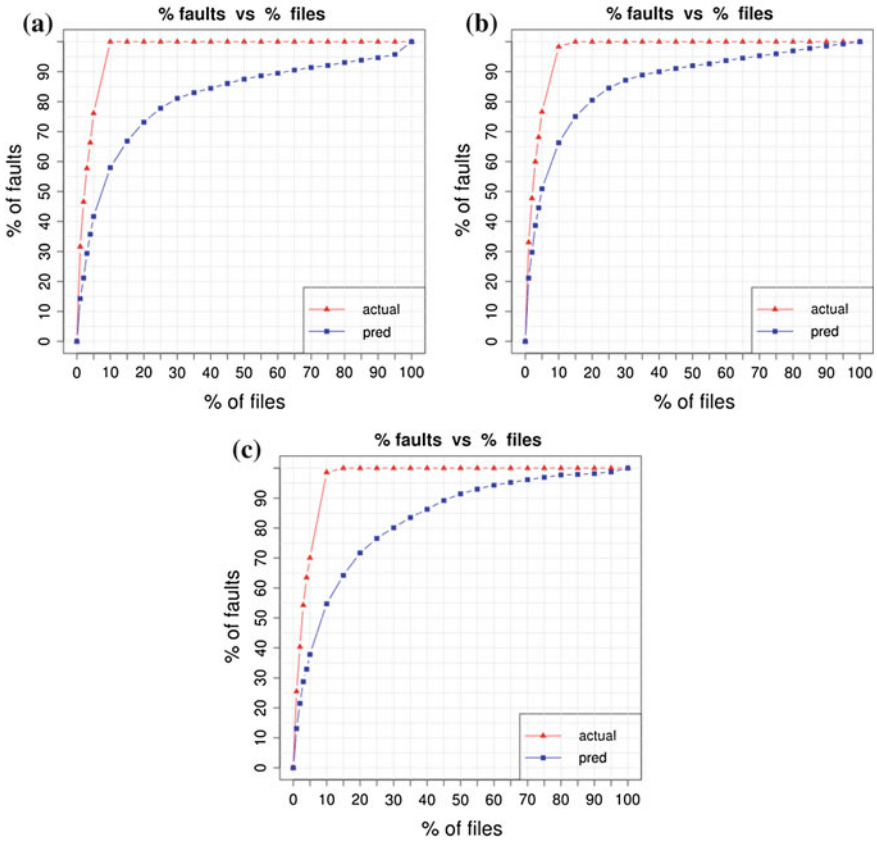

Fig. 1 Actual and simple prediction. **a** DatasetA. **b** DatasetB. **c** DatasetC

Table 3 Percentage of bugs in top 20 % of files

Projects	Actual	Predicted
DatasetA	100	73.18
DatasetB	100	80.48
DatasetC	100	71.72

it takes to review/test these files. In practice there are couple of quality assurance activities that can be performed soon after the bugs are predicted through the learning model. We consider two such popular quality assurance activities namely code review and unit testing.

We assume that effort to do code review is directly proportional to the lines of code one has to review and for unit testing, effort is proportional to the number of test cases one has to perform manually. The effort required to conduct the two quality assurance activities for the top 20 % files mentioned in Table 3 are tabulated in Table 4. From these results, we can infer the following: for datasetB, by

Table 4 Percentage of effort required for the top 20 % of files

Projects	Model	% Lines of code	% Testcases
DatasetA	Actual	39.75	36.67
	Predicted	66.15	78.06
DatasetB	Actual	47.76	56.97
	Predicted	64.72	76.91
DatasetC	Actual	41.66	40.05
	Predicted	63.81	66.4

reviewing the top 20 % of the predicted files, one can uncover 80 % of the bugs and the effort required for these files is reviewing 64.72 % of the total lines of code or performing 76.91 % of the testcases. This shows that, although the number of files predicted as bug prone is only 20 %, the amount of effort required to uncover the bugs from these files is very large which is only slightly better than a random inspection model which gives n% of bugs for n% of effort. This result is also observed in the remaining two datasets. In view of effort, the standard Pareto principle should be modified as follows: '20 % of effort should give 80 % of bugs'.

4.1 Effort Based Evaluation

In-order to minimize the effort involved to uncover the bugs in files, Mende and Koschke [13] propose a new ranking order. They rank the files based on the ratio: $\frac{\textit{predicted number of bugs}}{\textit{effort}}$ instead of predicted number of bugs alone. They use cyclomatic complexity as the surrogate measure for effort with the assumption that the lines of code as well as testcases correlate highly with cyclomatic complexity. In our datasets we have found the previous statement to be not true. Although the cyclomatic complexity is correlating highly with the lines of code, but it is not so with testcases. The Spearman correlations between cyclomatic complexity and lines of code, cyclomatic complexity and test cases is tabulated in Table 5.

Thus, instead of using a generic measure of effort such as cyclomatic complexity, we use lines of code and the number of testcases as our measures of effort. The predicted number of bugs term in the ratio is the value outputted by any learning algorithm which is in this case stepwise linear regression algorithm. The effort term in the ratio can be substituted with the lines of code, if quality assurance activity is code review or it can be number of testcases, if quality assurance activity is unit

Table 5 Spearman correlation

Projects	Cyclomatic complexity and lines of code	Cyclomatic complexity and number of testcases
DatasetA	0.9387	0.2703
DatasetB	0.9379	0.5487
DatasetC	0.9358	0.3687

Table 6 Percentage of defects caught by 20 % of effort (LOC)

Projects	Actual	Simple prediction	$\frac{predicted\ \#bugs}{LOC}$	$\frac{predicted\ \#bugs}{CC}$
DatasetA	70	22	33	30
DatasetB	65	35	42	40
DatasetC	60	25	40	38

Table 7 Percentage of defects caught by 20 % of effort (testcases)

Projects	Actual	Simple prediction	$\frac{predicted\ \#bugs}{TC}$	$\frac{predicted\ \#bugs}{CC}$
DatasetA	59	10	66	6
DatasetB	50	24	33	14
DatasetC	64	16	55	6

testing. This ratio gives a higher rank to files which have less effort per bug and a lower rank to files which have more effort per bug. Using our proposed ranking orders, the percentage of defects captured by 20 % of effort are recorded in Tables 6 and 7. The results are depicted visually by a plot of % of effort versus % defects in Figs. 2a, b, 3a, b and 4a, b. Let us consider the Fig. 3a, b which correspond to datasetB. In both figures, the plots of $\frac{predicted\ number\ of\ bugs}{lines\ of\ code}$ and $\frac{predicted\ number\ of\ bugs}{testcases}$ is always higher than the plot of simple prediction. This indicates that, to uncover a certain amount of bugs, our proposed ranking orders require less effort than the traditional simple prediction rank order. Although the simple prediction model is better than our proposed model when considering only the % of bugs caught versus the % of files (Fig. 1a), it fails to perform better when considering the effort required to uncover the bugs (Fig. 3a, b). This result is observed in all the three datasets. Also, common to all the three datasets is the observation that the specific effort based

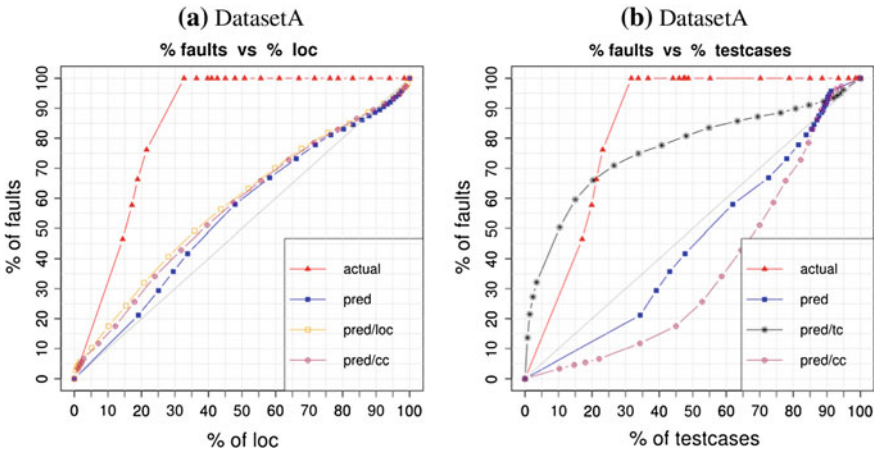

Fig. 2 Effort-aware prediction: DatasetA. **a** DatasetA. **b** DatasetA

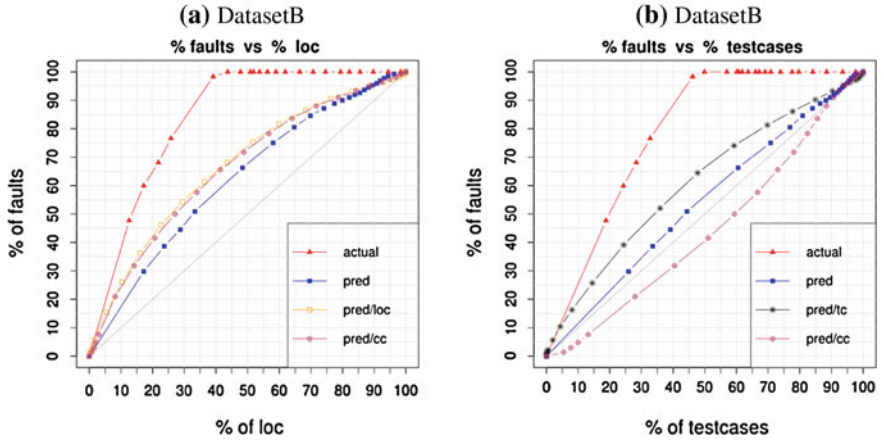

Fig. 3 Effort-aware Prediction: DatasetB. **a** DatasetB. **b** DatasetB

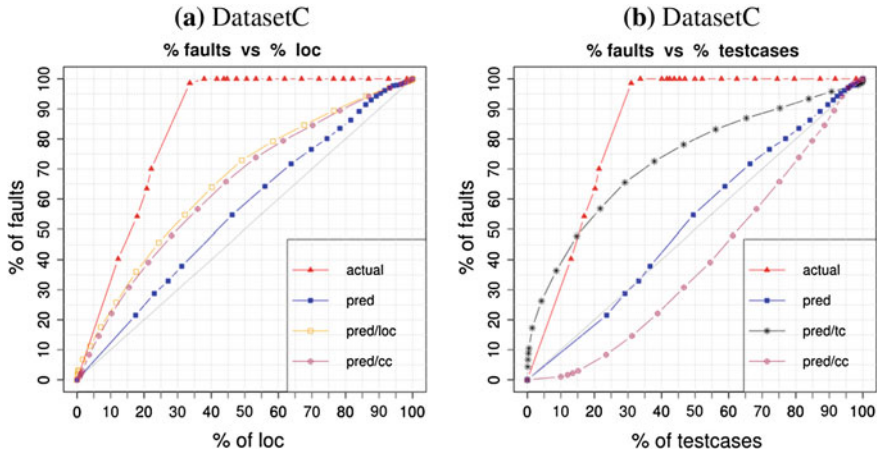

Fig. 4 Effort-aware prediction: DatasetC. **a** DatasetC. **b** DatasetC

models i.e. $\frac{predicted\ number\ of\ bugs}{lines\ of\ code}$ and $\frac{predicted\ number\ of\ bugs}{testcases}$ are performing better than the $\frac{predicted\ number\ of\ bugs}{cyclomatic\ complexity}$. In order to quantify the performance of each classifier we consider an evaluation scheme described in the next section.

4.2 Percentage of Cost Effectiveness

Arisholm et al. [2] proposed an effort evaluation measure CE which stands for Cost Effectiveness. They calculate a cost effectiveness estimation based on the assumption that a random selection of n% of source lines contains n% of the defects.

A defect prediction model is cost effective only when the files predicted as defective contain a larger percentage of defects than their percentage of lines of code. Their performance measure CE can be obtained by calculating the area under the prediction model's curve which lies above a line of slope one. Mende and Koschke [12] defined the measure p_{opt} which measures how close a model is to the ideal curve. Closer the curve to the ideal curve, the higher the value of p_{opt}. This measure takes into consideration the effort as well as the actual distribution of faults by benchmarking against a theoretically optimal model.

We consider the evaluation measure POA, Percentage of Area. This measure attempts to combine the aspects of both CE and p_{opt}, which has information about the lower bound of cost effectiveness, the random model and the upper bound, the theoretically optimal value. POA is the ratio of the CE of a model to the CE of the theoretically optimal model. This value gives the fraction of cost effectiveness a model achieves out of the theoretically maximum possible cost effectiveness.

$$POA = \frac{CE\ of\ model}{CE\ of\ the\ ideal\ model}$$

We present the POA values for the models when effort is lines of code in Table 8 and when effort is testcase in Table 9. From the results it is clear that specific effort based models outperform generic effort based measure cyclomatic complexity.

4.3 Statistical Significance

We now test whether the differences between the three models are statistically significant using the non parametric tests mentioned by Demšar [6]. Demšar uses

Table 8 POA—lines of code

Model	DatasetA	DatasetB	DatasetC
Ideal	100	100	100
Simple prediction	14.23	35.59	16.03
$\frac{Predicted\ Number\ of\ Bugs}{Lines\ of\ Code}$	22.61	52.2	46.55
$\frac{Predicted\ Number\ of\ Bugs}{Cyclomatic\ Complexity}$	19.56	49.84	41.73
Random model	0	0	0

Table 9 POA—testcases

Model	DatasetA	DatasetB	DatasetC
Ideal	100	100	100
Simple prediction	0.75	13.18	7.06
$\frac{Predicted\ Number\ of\ Bugs}{Testcases}$	75.13	38.65	67.22
$\frac{Predicted\ Number\ of\ Bugs}{Cyclomatic\ Complexity}$	0.73	0.3	0.19
Random model	0	0	0

the Friedman test [8] to check whether the null hypothesis, i.e. that all models perform equal on the datasets, can be rejected. It is calculated as follows:

$$\chi_F^2 = \frac{12N}{k(k+1)}\left(\sum_j R_j^2 - \frac{k(k+1)^2}{4}\right)$$

$$F_F = \frac{(N-1)\chi_F^2}{N(k-1)-\chi_F^2}$$

where k denotes the number of models, N the number of datasets, and R_j the average rank of model j on all data sets. F_F is distributed according to the F-Distribution with $k-1$ and $(k-1)(N-1)$ degrees of freedom. Once computed, we can check F_F against critical values for the F-Distribution and then accept or reject the null hypothesis. When the Friedman test rejects the null hypothesis, we can use the Nemenyi post hoc test to check whether the difference of performance between two models is statistically significant. The test uses the average ranks of each model and checks for each pair of models whether the difference in their average ranks is greater than the critical difference CD.

$$CD = q_\alpha\sqrt{\frac{k(k+1)}{6N}}$$

where k and N are number of models and datasets respectively. q_α is a critical value which are based on the Studentized range statistic divided by $\sqrt{2}$. The Studentized range statistic depends on the alpha value which in this cases we take as 0.05 and the number of models k. For our setup, we used k = 3, $\alpha = 0.05$ and $q_\alpha = 2.85$.

We use Lessmann et al. [11] modified version of Demšar's significance diagrams to depict the results of Nemenyi's post hoc test: For each classifier on the y-axis, the average rank across the datasets is plotted on the x-axis, along with a line segment whose length encodes CD. All classifiers that do overlap in this plot do not perform significantly different and those that do not overlap, perform significantly different. The Nemenyi's post hoc significance plots are presented in Fig. 5a, b.

Consider Fig. 5a: $\frac{predicted\ no.\ of\ bugs}{lines\ of\ code}$ ranks the best, followed by $\frac{predicted\ no.\ of\ bugs}{cyclomatic\ complexity}$ and *predicted no. of bugs*. Although, the $\frac{predicted\ no.\ of\ bugs}{lines\ of\ code}$ performs better, the difference between it and $\frac{predicted\ no.\ of\ bugs}{cyclomatic\ complexity}$ is not significant.

In Fig. 5b: $\frac{predicted\ no.\ of\ bugs}{test\ cases}$ ranks the best, followed by *predicted no. of bugs* and $\frac{predicted\ no.\ of\ bugs}{cyclomatic\ complexity}$. $\frac{predicted\ no.\ of\ bugs}{lines\ of\ code}$ model performs significantly better than $\frac{predicted\ no.\ of\ bugs}{cyclomatic\ complexity}$.

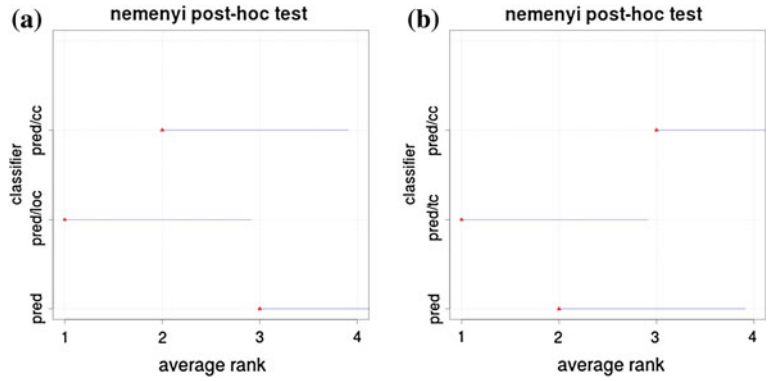

Fig. 5 Nemenyi diagram for LOC and testcases. **a** Nemenyi Plot - Lines of Code. **b** Nemenyi Plot - Testcases

4.4 Threats to Validity

Since open source softwares are not developed in controlled environment and unavailability of proper testcases and considering the difficulties of linking them with appropriate files, We decided to do the experiments in proprietary software datasets. Similar experiments in different datasets may not yield similar results.

5 Conclusion

The consistent and repetitive results of three projects and generic Demšar test indicate us that the testing based effort aware models perform significantly better than generic effort aware models. The code review based effort aware model performs slightly better than generic model individually for each project though Demšar test show that there is no significant improvement. Hence, We conclude that effort aware models are sensitive to the type of quality assurance activity one undertakes and the effort required for these activities should be considered while building prediction models.

References

1. Arisholm, E., Briand, L.C.: Predicting fault-prone components in a java legacy system. In: Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering, pp. 8–17. ACM (2006)
2. Arisholm, E., Briand, L.C., Johannessen, E.B.: A systematic and comprehensive investigation of methods to build and evaluate fault prediction models. J. Syst. Softw. **83**(1), 2–17 (2010)

3. Basili, V.R., Briand, L.C., Melo, W.L.: A validation of object-oriented design metrics as quality indicators. IEEE Trans. Softw. Eng. **22**(10), 751–761 (1996)
4. Cartwright, M., Shepperd, M.: An empirical investigation of an object-oriented software system. IEEE Trans. Softw. Eng. **26**(8), 786–796 (2000)
5. Chidamber, S.R., Kemerer, C.F.: A metrics suite for object oriented design. IEEE Trans. Softw. Eng. **20**(6), 476–493 (1994)
6. Demšar, J.: Statistical comparisons of classifiers over multiple data sets. J. Mach. Learn. Res. **7**, 1–30 (2006)
7. Fenton, N.E., Ohlsson, N.: Quantitative analysis of faults and failures in a complex software system. IEEE Trans. Softw. Eng. **26**(8), 797–814 (2000)
8. Friedman, M.: The use of ranks to avoid the assumption of normality implicit in the analysis of variance. J. Am. Stat. Assoc. **32**(200), 675–701 (1937)
9. Khoshgoftaar, T.M., Allen, E.B.: Ordering fault-prone software modules. Softw. Qual. J. **11** (1), 19–37 (2003)
10. Krishnan, S., Strasburg, C., Lutz, R.R., Goševa-Popstojanova, K.: Are change metrics good predictors for an evolving software product line? In: Proceedings of the 7th International Conference on Predictive Models in Software Engineering, ACM, New York, NY, USA, Promise '11, pp. 7:1–7:10. doi:10.1145/2020390.2020397. http://doi.acm.org/10.1145/2020390.2020397 (2011)
11. Lessmann, S., Baesens, B., Mues, C., Pietsch, S.: Benchmarking classification models for software defect prediction: a proposed framework and novel findings. IEEE Trans. Softw. Eng. **34**(4), 485–496 (2008)
12. Mende, T., Koschke, R.: Revisiting the evaluation of defect prediction models. In: Proceedings of the 5th International Conference on Predictor Models in Software Engineering, p. 7. ACM (2009)
13. Mende, T., Koschke, R.: Effort-aware defect prediction models. In: Proceedings of the Software Maintenance and Reengineering (CSMR), 2010 14th European Conference on, pp. 107–116. IEEE (2010)
14. Nagappan, N., Ball, T.: Use of relative code churn measures to predict system defect density. In: Proceedings of the 27th International Conference on Software Engineering, 2005. ICSE 2005, pp. 284–292. IEEE (2005)
15. Nagappan, N., Ball, T., Zeller, A.: Mining metrics to predict component failures. In: Proceedings of the 28th International Conference on Software Engineering, pp. 452–461. ACM (2006)
16. Ohlsson, N., Alberg, H.: Predicting fault-prone software modules in telephone switches. IEEE Trans. Softw. Eng. **22**(12), 886–894 (1996)
17. Ostrand, T.J., Weyuker, E.J., Bell, R.M.: Predicting the location and number of faults in large software systems. IEEE Trans. Softw. Eng. **31**(4), 340–355 (2005)
18. Subramanyam, R., Krishnan, M.S.: Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects. IEEE Trans. Softw. Eng. **29**(4), 297–310 (2003)
19. Tang, M.H., Kao, M.H., Chen, M.H.: An empirical study on object-oriented metrics. In: Proceedings of the Sixth International Software Metrics Symposium, 1999, pp. 242–249. IEEE (1999)