An Empirical Investigation of Predicting Fault Count, Fix Cost and Effort Using Software Metrics

Raed Shatnawi
Software Engineering Department,
Jordan University of Science and Technology,
Irbid, Jordan

Wei Li
Computer Science Department,
University of Alabama in Huntsville,
Huntsville, AL, USA

***Abstract*—Software fault prediction is important in software engineering field. Fault prediction helps engineers manage their efforts by identifying the most complex parts of the software where errors concentrate. Researchers usually study the fault-proneness in modules because most modules have zero faults, and a minority have the most faults in a system. In this study, we present methods and models for the prediction of fault-count, fault-fix cost, and fault-fix effort and compare the effectiveness of different prediction models. This research proposes using a set of procedural metrics to predict three fault measures: fault count, fix cost and fix effort. Five regression models are used to predict the three fault measures. The study reports on three data sets published by NASA. The models for each fault are evaluated using the Root Mean Square Error. A comparison amongst fault measures is conducted using the Relative Absolute Error. The models show promising results to provide a practical guide to help software engineers in allocating resources during software testing and maintenance. The cost fix models show equal or better performance than fault count and effort models.**

Keywords*—Software metrics; fault prediction; fix cost; fix effort; regression analysis

I. INTRODUCTION

Predicting faults in modules is important to assess software quality and to direct software engineers' effort to spend more time on more trouble-prone modules. Software metrics are surrogates for fault measures such as fault-proneness, fault count, fault-fix cost, and effort. Software metrics measure the complexity of software and can be used to identify the faulty modules using statistical and machine-learning techniques. These techniques can be used to build prediction models such as fault count, fix cost, and fix effort to predict which modules are likely to have these problems. Software systems are becoming larger and larger and contain thousands of modules that are investigated in testing and maintenance phases. However, the cost of testing and maintenance are growing with the size of systems. This growing trend leads to either very costly system or compromised quality. Software engineers can use prediction models to prioritize modules to focus the testing and maintenance activities on the modules that are either have more faults, more costly to fix or demand more efforts to fix. Hence, detecting and ranking faulty modules is an important engineering task for improving system quality and reducing cost. There are usually two measures of module quality: fault count or fault-proneness. In most systems, a small number of modules have faults and the majority of modules have zero faults. Researchers use fault-proneness by using binary coding of modules (zero for no faults and one if there are faults in a module) to build prediction models that are usually easy to interpret [1][2][3][4][5][6][7]. However, the binary coding does not explore all information available about faults. Fault count is an indicator of quality in a module but may not provide enough information about the fix cost or effort. Therefore, regression and machine-learning models are used to identify complex modules by considering fault count, fix cost and effort. In this paper, five regression and machine-learning techniques are used to predict the three fault measures. Twenty procedural metrics used as independent variables in the prediction models. The models are trained and tested on three data sets provided by NASA. Overall, fifteen models were built for each data set using 10-fold cross-validation. The results for the three fault measures have shown similar results, but the cost-fix models are slightly better. These models can help in allocating resources for software testing and maintenance. The results of the models are used to rank the modules based on the fault measures, and the results are promising and commensurate with previous works [8][9].The performance of the three fault measures is compared to find the best ranking. The results show similar results for the three measures with some advantage for fault count and fix cost over fix effort.

The rest of the paper is organized as follows: related work to the three fault measures are discussed in Section 2. In section 3, the study design is discussed which includes a description of the dependent, independent variables and regression models used in this paper. The data analysis is presented in Section 4, which also evaluates the predictions of the fault measures. Validity threats to the study are discussed in Section 5. The study is concluded in Section 6.

II. RELATED WORK

Fault prediction has been discovered in many previous research in two major themes: fault-proneness and fault count. Studies on fault proneness categorized software classes into groups. Usually, classes are divided into two groups: faulty classes that had one or more faults in the current release, and non-faulty classes. Software metrics have shown significant relations with fault-proneness using many machine learning and statistical techniques [1][2][3][5][6][7][10]. Many research studies used the NASA fault data to build fault-proneness models. For example, Pai and Dugan [11] conducted a Bayesian analysis of fault count and fault proneness. The study produced statistical significant results using linear, Poison, and binomial logistic regression. The modeling of the results have

shown 20:60 relationship when classes were ranked using module-fault order. Catal and Diri [12] used the NASA data sets to predict fault-prone modules and proposed an artificial immune system (semi-supervised approach) that uses a recent algorithm called YATSI. Gondra [13] also used the NASA's Metrics Data Program data to build prediction models of fault-proneness of modules using two machine l++-earning techniques: Artificial Neural Networks (ANN) and Support Vector Machines (SVM). Zheng [14] used four datasets from NASA projects to compare the effect of cost-sensitive boosting algorithms on the performance of neural networks for predicting fault-prone parts. In other studies on fault measures , Ohlsson and Alberg [15] noted that in commercial products, the average cost of fixing an operational fault was $7000. Biyani and Santhanam [16] found correlation between the number of faults found in development and the number of faults remaining in operation. Ostrand et al. [17] developed a negative binomial regression model to predict the number of faults in each file for many consecutive releases of a software. Khoshgoftaar and Gao [9] used two statistical models: Poisson regression model and the zero-inflated Poisson to predict fault count in two industrial case studies. The zero inflated model showed better performance than poison regression model. Other researchers focused on other fault measures such as fix cost and effort. For instance, [18] used the KC1 data to build faults fix cost using Neural Networks. Panjer [19] proposed to build machine-learning models to predict fault-fix time. (Khoshgoftaar and Gao [9] proposed to use a program module-order models to explore the relationship between %modules and %faults as a more practical model that is based on the predictions resulting from machine learning models. Khoshgoftaar et al. found that 80% of faults are found in the top 20% of files when ordered by faults predicted by models [9]. In a recent study, Hamill and Goseva-Popstojanova [20] studied the relationship between faults and failure of 21 large-scale software components extracted from a safety-critical NASA mission. However, the study focused more on fault types.

Fault prediction models are reported frequently in previous works as reported in surveys on software fault prediction [21] [22]. This study provides an exploration of the added dimension for the relationships between software metrics and fault measures such as fix cost and fix effort. In addition, the module-order models proposed in Briand et al. and Khoshgoftaar and Gao [8][9] are used to prioritize modules according to models predicting fix cost and fix effort.

III. Study Design

Fault data are becoming more available on many repositories such as PROMISE [23], Eclipse Bug Data [24], and NASA fault data [25]. The NASA data provides more details on the costs and efforts of fixing software faults, which are the focus of this research. Three data sets, KC1, KC3 and KC4 report the cost of fault fixes in terms of person hours and effort measured in Source Line of Code (SLOC) modified to accomplish the fix. Table 1 shows a summary of the three data sets. All these projects were built in similar software development environments and analyzed by the same set of software product metrics. These data sets are available publicly and other researchers can repeat and verify this study's results. The MDP is funded by NASA's Software Independent Verification & Validation (IV&V) facility. These systems met the requirements to support NASA mission [26].

TABLE I. A Summary of Data Sets

Data set	Description	Language	#instances	#faulty instances	%faulty instances
KC1	is a system implementing storage management for receiving and processing ground data	C++	2107	278	13%
KC3	Storage management for ground data	Java	458	25	5.5%
KC4	a ground-based subscription server	Perl	125	60	48%

A. Research Questions

Given the information available on fault count, fix cost and fix effort, this research aims to find answers for the following research questions.

***RQ1**: Can software metrics predict fault count?*

Fault count is defined as the number of faults fixed in a module. This question is already answered in previous research as explained in more details in the related work section. However, this study adds the evaluation of faults prediction using other machine learning techniques. Fault prediction is important to assess the complexity of software modules. Five prediction models are conducted to answer this question. The results of the prediction models are used to rank the modules by sorting according to the predicted fault count. The models can be used to allocate resources efficiently to identify for instance the 20% modules that have the most faults.

RQ2: Can software metrics predict fix cost as measured in man-hours?

Fix cost is defined as the total number of hours the developers spent to fix all faults in a module. For each module, the cost of fault fixes are aggregated. The fix cost in hours is an indicator of the complexity of code. A positive relationship is expected between the studied metrics and fix cost, i.e., more complex modules cost more than less complex modules. To put the cost prediction models in practical use, the results of prediction models are used to sort the modules by the predicted fix cost. The models can be used to allocate resources efficiently to identify for instance the 20% modules that have the most fix cost.

RQ3: Can software metrics predict fix effort as measured in SLOC modified?

Fix effort is defined as the actual number of SLOC added or modified to fix all faults in a module. In this study, the aggregation of all modified SLOC for a particular module is used to investigate the relationship between the fix effort and the complexity of modules. To put the effort prediction in

practical use, the results of the prediction models are used to sort the modules by the predicted fix effort. The models can be used to allocate resources efficiently to identify for instance the 20% modules that need the most fix effort.

The results of the three quality predictions are compared using the relative absolute error to find which models are better.

B. Dependent variables

NASA MDP has many projects, but only three of these projects have details on fault fixes, cost and effort. For each module, the number of faults (fault content), the total fix hours, and the total SLOC changed or added are aggregated. Table 2 provides a summary of the fault measures used in this study. The scale for fix cost and effort are larger than the fault count. The scale has effect on the performance measures used in evaluating the prediction models and the comparison should be based on unbiased performance measures. Relative absolute error is used to evaluate models besides the root mean square error.

C. Independent variables - software metrics

The software metrics under investigation are procedural metrics for three systems collected by NASA MDP. The metrics collection were applied to the lowest level functional unit, procedures. The data were stored in a structured format. For example, a file named KC1_static_defect_data.csv, keeps all information related to faults including severity, priority, fix hours, the actual number of SLOC changed or added. Another file includes all the static metrics for each module and recognized using a unique variable, MODULE_ID. These files are then combined together into one file using the MODULE_ID, which is an identifier of module records in all files.

The NASA MDP data needs preprocessing as reported in [27]. Therefore, we use only those metrics that were reported by [27] which had 21 metrics as reported in Table 3. The LOC_BLANK metric is deleted because it is not meaningful and its interpretation is not clear. These metrics were originally proposed in [28][29]. The McCabe and Halstead measures are module-based where a module is the smallest unit of functionality. McCabe argued that code with complicated pathways are more error prone. Halstead considered the code readability as indicator of fault proneness. Halstead metrics measure software complexity by counting the number of concepts in a module [26].

TABLE II. DESCRIPTIVE STATISTICS FOR THE THREE FAULT MEASURES (FAULT COUNT, FIX HOURS, SLOC MODIFIED)

Fault count	**Min**	**Max**	**Mean**	**stdev**	**Total**
KC1	0	11	0.30	0.991	631
KC3	0	3	0.114	0.50	52
KC4	0	23	2	3.60	248
Fix cost	**Min**	**Max**	**Mean**	**stdev**	**Total**
KC1	0	397	5.99	26.7	12629
KC3	0	190	6.62	29.365	3032
KC4	0	498	28.6	62.43	3548
Fix effort	**Min**	**Max**	**Mean**	**stdev**	**Total**
KC1	0	1016	14.57	57.59	30713
KC3	0	512	7.63	44.00	3496
KC4	0	467	19.24	62.70	7176

D. Regression Models

We propose to use a set of data mining techniques to predict the value of a numerical variable (e.g., fix cost) by building a model based on many software metrics. This research uses the following regression techniques to predict fault count, fix cost and fix effort.

Regression Decision Trees (M5P): Decision tree is used to build regression models in the form of a tree structure using the M5 algorithm [30]. The algorithm constructs a decision tree for regression different from classification by using Standard Deviation Reduction instead of Information Gain. A dataset is continuously partitioned into smaller subsets while the standard deviation is larger than zero.

Multiple Linear regression (MLR): Multiple linear regression (MLR) is a well-known statistical technique used to model the linear relationship between a count variable and many independent variables. MLR is based on calculating ordinary least squares (OLS), the model is fit such that the differences between actual and predicted instances are minimized.

k Nearest Neighbors (kNN): The kNN algorithm is an instance-based method that is not used to build a model from training data; rather, it keeps the training instances with the intention of analyzing future instances. The kNN algorithm searches the training instances to find the closest instances to a new unknown instance to be analyzed. The search starts by finding the distance with all other instances using the Euclidean Distance. The kNN algorithm selects the average of the closest group of k objects in the training set [31].

TABLE III. SOFTWARE METRICS USED IN THE EMPIRICAL WORK

Metrics	description or formula
LOC_CODE_AND_COMMENT:	The number of lines which contain both code and comment in a module
LOC_COMMENTS	The number of lines of comments in a module
LOC_EXECUTABLE	The number of lines of executable code for a module (not blank or comment)
LOC_TOTAL	The total number of lines for a given module
BRANCH_COUNT	Branch count metrics
CYCLOMATIC_COMPLEXITY:	The cyclomatic complexity of a module $v(G) = e - n + 2$
DESIGN_COMPLEXITY:iv(G)	The design complexity of a module
ESSENTIAL_COMPLEXITY:ev(G)	The essential complexity of a module
NUM_OPERATORS:N1	The number of operators contained in a module
NUM_OPERANDS:N2	The number of operands contained in a module
NUM_UNIQUE_OPERATORS:µ1	The number of unique operators contained in a module
NUM_UNIQUE_OPERANDS:µ2	The number of unique operands contained in a module

HALSTEAD_CONTENT:μ	The halstead length content of a module μ = μ1 + μ2
HALSTEAD_LENGTH:N 2	The halstead length metric of a module N = N1 + N
HALSTEAD_LEVEL:L	The halstead level metric of a module L = (2∗μ2)/μ1∗N2
HALSTEAD_DIFFICULTY:D	The halstead difficulty metric of a module D = 1/L
HALSTEAD_VOLUME:V	The halstead volume metric of a module V = N ∗ log2(μ1 + μ2)
HALSTEAD_EFFORT:E	The halstead effort metric of a module E = V/L
HALSTEAD_PROG_TIME: T	The halstead programming time metric of a module T = E/18
HALSTEAD_ERROR_EST: B	The halstead error estimate metric of a module B = $E^{2/3}$/1000

Multi-layer Perceptron - Backpropagation algorithm: The multi-layer perceptron (MLPRegressor) is similar to the organization of the brain neurons. Artificial neurons are arranged in layers (i.e., input layer, hidden layers and output layer). Connections between the neurons provide the network with the ability to learn patterns. In MLP, each neuron in the hidden layer uses a combination of weighted outputs of the neurons from the previous layer. In the final hidden layer, neurons are combined to produce an output, which is compared to the correct output and the difference between the two values (the error) is fed back to update the network [13].

Support Vector Machine (SMOreg): SMOreg implements the support vector machine for regression. SMOreg is more complicated to be taken into consideration than the classification version. However, both aim to minimize error, i.e., individualizing the hyperplane which maximizes the margin while error is tolerated [32].

E. Regression performance evaluation

The models are trained and tested using 10-fold cross-validation, in which data is partitioned into ten equal sample sizes. Nine partitions are used for training while the last partition is used for testing. This process is repeated ten times to use all partitions in testing. The performance of regression models is usually evaluated using the Root Mean Squared Error (RMSE) as defined in Eq. (1). RMSE is frequently used to measure the difference between predicted and actual values. RMSE is calculated as follows.

$$\text{RMSE} = \sqrt{\frac{\sum_{i=1}^{n}(p_i - a_i)^2}{n}} \quad (1)$$

In this research the dependent variables have different units and to be able to compare models on different units, the Relative Absolute Error (RAE) is used as defined in Eq. (2). [32]. RAE is calculated as follows.

$$\text{RAE} = \frac{\sum_{i=1}^{n}|p_i - a_i|}{\sum_{i=1}^{n}|\bar{a} - a_i|} \quad (2)$$

In both measures, a is the actual value, p is the predicted value, and $\bar{a}$ is the mathematical mean.

IV. Data Analysis

In the following, the evaluation of the prediction performance for fault measures are reported using RMSE and then compared using RAE.

A. Evaluation of fault count prediction

Five prediction models are built for fault count using twenty metrics under investigation. The performance of fault prediction is calculated and summarized in Table 4. The results of the five models do not differ from each other when compared within any data set. However, the LR models look better in two data sets, while KNN models are also better in two data sets as marked in bold. However, the differences in the performance among the models are not enough to provide ranking of the machine learning techniques. The MLP can be considered the worst in performance among all.

TABLE IV. Fault Count Regression Models

Fault Count	LR	kNN	M5P	SMOreg	MLPRegressor
KC1	**0.90**	0.92	0.93	1.00	1.06
KC3	**0.46**	**0.46**	0.48	0.47	0.63
KC4	3.17	**2.60**	2.78	3.16	3.69

To put models in practice, the results of the models are depicted using Alberg diagrams as proposed in [15]. In Figure 1, modules are sorted in decreasing order by the predicted faults. The plot shows the percentage of modules (x-axis) against the percentage of actual faults after sorting the instances. Figure 1 shows the results of fault count prediction in KC1. These results are taken from running the models in the 10-fold cross-validation. The figure can be used as follows, for example at X=20 the value of the curve is 60, which means 20% of modules (369 modules) with highest predicted fault count constitute of 60% of faults. It can be noticed that the top 30% of modules has 70% of actual faults. This behavior is similar in all models.

We also plot the same graph for KC3 and KC4 prediction models in Figure 2 and 3. In Figure 2, we observe similar results for the top 20% modules, i.e., about 60% of faults are found in the top 20% of modules in all prediction models. In Figure 3, we observe similar results for KC4 data in kNN model. Other models show 20:50 relationship, i.e., 50% of faults are found in the top 20 modules. We can conclude that software metrics can be used to predict fault count and models can be used in practice to rank modules based on predicted fault count. Therefore, RQ1 is answered in this research. When planning for quality inspection during the software development process, we can make a trade-off between the resources spent on inspection and the effectiveness of inspections [8]. The prediction models can be used to put the modules in a priority list for more investigation such as testing and maintenance. We can use the graph in Figure 1 to determine the percentage of faults that are expected in the system by inspecting a certain percentage of the system modules. For example, the top 20% modules can be

investigated first if allocated resources are only available for investigating such number of modules.

The graphs in Figures 1-3 have shown similar behavior to works in [33][8][11]. For instance, Briand et al. [8] found that the first 20% of classes have 52% of faults in the system. They also suggested that such curves can be used in practice if they appear to be constant across projects. Software managers can use fault prediction models to allocate more resources on the parts of the code that were predicted to be more fault-prone [5][34].

B. *Evaluation of fix-cost prediction*

We repeated the same experiment to predict fix cost using all metrics and the results are shown in Table 5. We notice no significant differences among the models except MLP, which is again the worst modelling technique. M5P regression trees can be considered the best among all models, while others have almost equal performances.

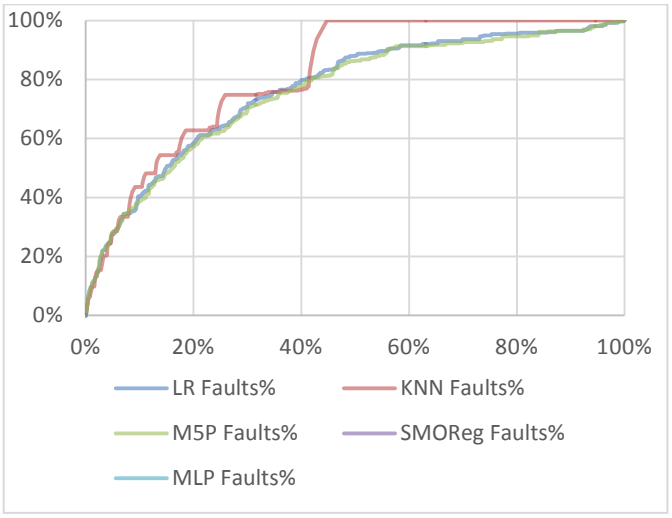

Fig. 1. Alberg diagram for five prediction models of KC1

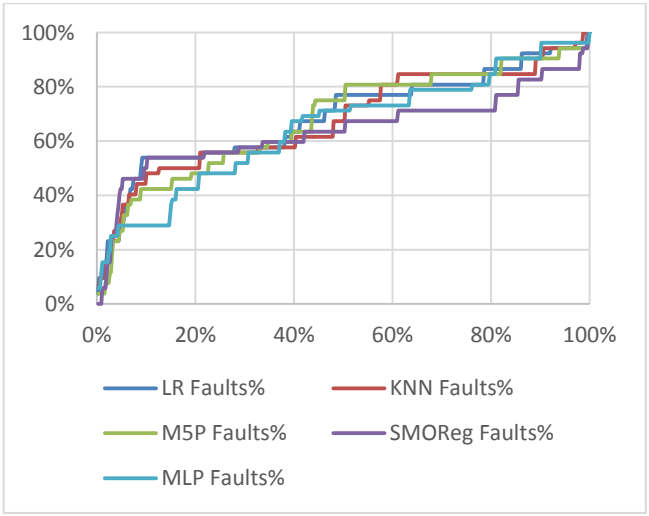

Fig. 2. Alberg diagram for five prediction models of KC3

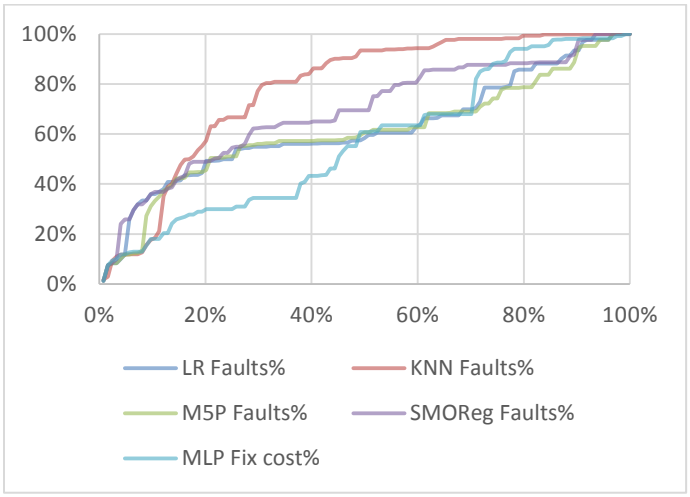

Fig. 3. Alberg diagram for five prediction models of KC4

TABLE V. FIX-COST REGRESSION MODELS

Fix cost	LR	kNN	M5P	SMOreg	MLPRegressor
KC1	24.70	25.05	24.70	25.71	29.70
KC3	25.98	26.3	27.71	24.47	36.38
KC4	59.88	50.83	50.00	55.47	73.21

The fix cost can be used in practice to order modules based on cost prediction. We plot the percentage of modules (x-axis) and the percentage of actual costs after sorting the instances in decreasing order by the predicted fix cost. Figure 4 shows the results of the five prediction models for fix-cost prediction in KC1. The figure can be used, at X=20 the value of the curve is 60% in three models whereas in two models (LR and MLP) is about 50%. This result means 20% of modules (369 modules) with highest predicted fix cost incurred 60% of the spent person hours on fixing cost. It can be noticed that the top 30% of modules ordered by the prediction model has 60-70% of actual fix cost.

We plot the Module-Cost graph for KC3 and KC4 in Figure 5 and 6.

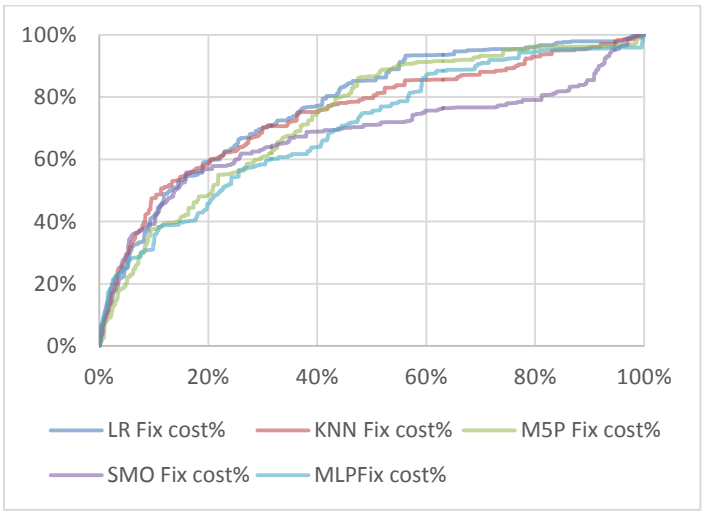

Fig. 4. Alberg diagram for five prediction models of KC1

The graphs show a 10:60 relationship, i.e., 10% of modules incurred 60% of the fix cost in both KC3 and KC4 in four models except the MLP prediction models which shows 20:40 relationship. These results are better than the models in KC1. In addition, the use of fix cost seems more efficient than the use of fault count in two data sets: KC3 and KC4, which show 20:60 relationship. Therefore, RQ2 is answered in this research as well. Fix cost can be predicted using software metrics and models can be used in practice to rank modules based on predicted fix cost. Fix cost can be used to allocate resources in software testing and maintenance activities.

C. Evaluation of fix effort prediction

The SLOC modified in a module is also studied as a fault measure and the results are presented in Table 6. The results are not conclusive in identifying the best model. The MLP models are again the least in performance among all. We plot the percentage of modules (x-axis) and the percentage of actual SLOC modified to fix faults in each module after sorting modules in decreasing order by the predicted effort as shown in

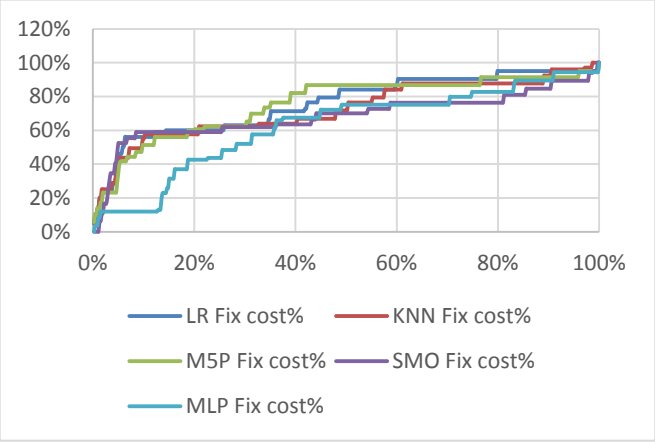

Fig. 5. Alberg diagram for five prediction models of KC3

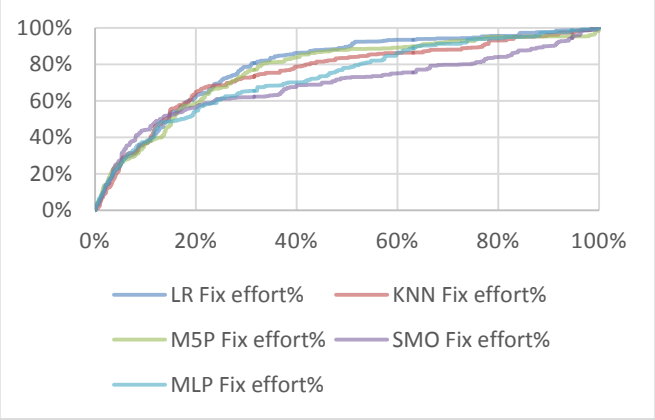

Fig. 6. Alberg diagram for five prediction models of KC4

Figure 7,8 , and 9. The results of the five prediction models do not show consistent results in all data sets. Almost all models show 20:60 relationship in KC1, but are different in KC3 and KC4 for different models. However, the results of the models on KC4 are similar to the models in KC1. While the models obtained from KC3 do not show promising results. These results show that RQ3 is answered. Fix effort as measured using SLOC can be used in practice to order the modules based on fix effort. However, the fault count and fix cost in person hours can be more beneficial to software managers.

TABLE VI. FIX EFFORT REGRESSION MODELS

Fixed SLOC	LR	kNN	M5P	SMOreg	MLPRegressor
KC1	53.05	56.32	53.58	56.66	61.66
KC3	36.91	38.00	35.49	34.58	49.85
KC4	109.63	92.66	90.41	103.79	129.00

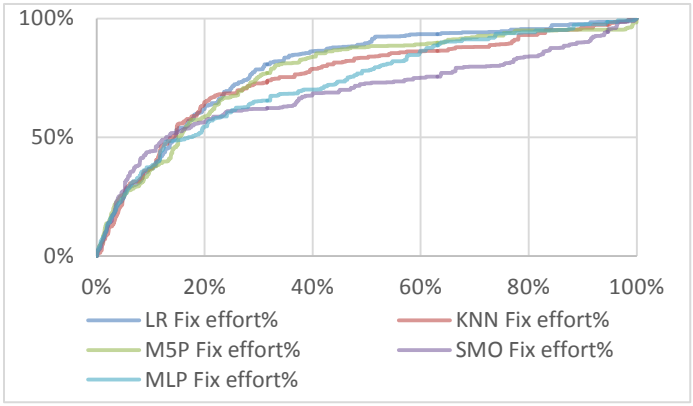

Fig. 7. Alberg diagram for five prediction models of KC1 data

D. Comparison of models performance

The RMSE results cannot be used to compare the results across the three fault measures because of the differences in measurement units. Therefore, we use another measure, the Relative Absolute Error (RAE), to analyze the results among the fault measures. The results of the models performance in RAE are reported in Table 7, where we find the following observations. In KC1, the Fault count models are the best in most models except one. In KC3, the fix cost models are the best except for two models. In KC4, the Fault count models are again the best. Therefore, for the systems under investigation, we can observe that prediction models based on fault count are slightly better in performance than other studied models. However, we do not observe large differences among the three fault measures under investigation. These results help the software engineers to consider other quality factors related to fault discovery and fix processes. The regression models for the fix cost and fix effort can be used similarly to fault count models.

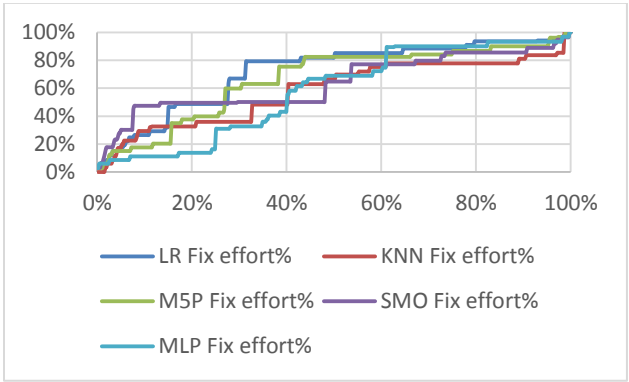

Fig. 8. Alberg diagram for five prediction models of KC3 data

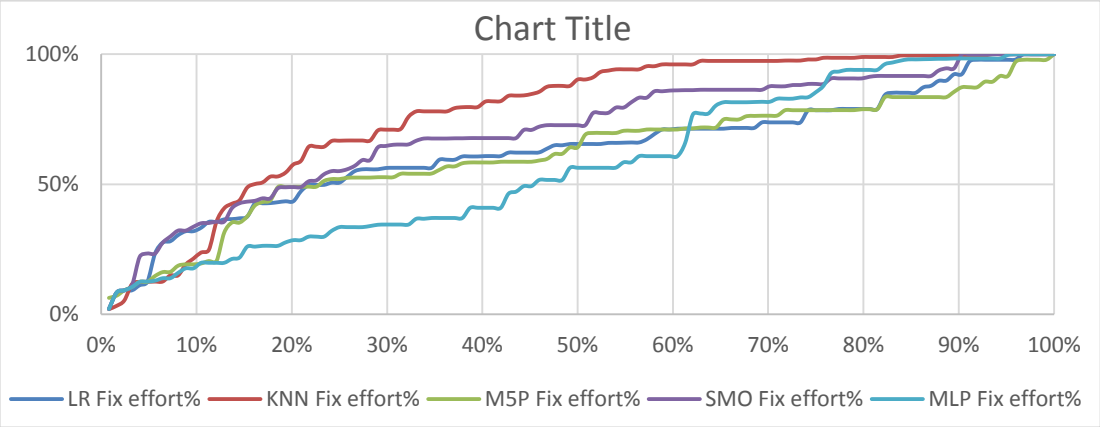

Fig. 9. Alberg diagram for five prediction models of KC4 data

TABLE VII. THE RAE RESULTS FOR FIVE MODELS

	KC1			KC3			KC4		
	Fault count	Fix cost	Fix effort	Fault count	Fix cost	Fix effort	Fault count	Fixcost	Fix effort
LR	**84.2**	91.3	89.7	90.6	**89.8**	101.2	**92.0**	102.2	97.9
KNN	**72.2**	76.4	78.4	71.4	**69.3**	73.5	**67.1**	77.1	73.4
M5P	**85.8**	88.2	87.5	**97.7**	96.1	99.2	**79.9**	84.6	82.7
SMO	58.2	**57.7**	58.4	51.6	51.6	**50.1**	74.9	70.4	**69.3**
MLP	**106**	114.1	109.7	131	**125.4**	137.7	**103**	122.3	114.1

The three quality factors can be used in practice to allocate resources, but it is important to know which models are consistent and always useful. The results of the practical implementation of the models for the three factors when 20% of modules are selected for further investigation are summarized in Table 8. The results show that both fault count and fix cost are more consistent than fix effort. In some cases, the cost models show better results. Furthermore, the use of fix cost in allocating resources provide more insights about the person hours spent to fix faults and can be considered a stronger indicator of where difficulties in code may appear.

V. VALIDITY THREATS

In the following, we address two kinds of possible threats that may affect the conducted research.

Construct Validity Threats: Construct validity refers to the degree to which the dependent and independent variables in this research measure the intended targets. Fix cost as measured in person hours are estimated by the developers and there is no detailed information about how developers estimate the fix cost. However, the data comes from a well-reputed organization, NASA, and their work is focused on quality of data and quality of work. The metrics in this study are well-studied metrics and recommended by many researchers to measure modules at procedural level.

Internal validity threats: internal validity is the degree to which conclusions can be drawn from the proposed data sets. This study depends on data from other organization and there is not enough information available about the development process followed in developing the three applications under study. However, the studied systems were considered in many other research papers and recommended to use by NASA.

External validity threats: External validity is concerned with the degree to which the results can be generalized to other research settings. The results of this study is based on only three data sets published by NASA MDP. We need more data sets to be able to generalize the results of this study into other systems. In addition, the systems are measured at procedural levels and conclusions may not be applicable for other paradigms like object-oriented paradigm.

VI. CONCLUSIONS AND FUTURE WORK

The fault prediction models are surrogates for the software quality. The assessment of faults in modules can be used to direct the efforts of software engineers in assuring software quality. Five well-known regression models were used to predict fault count, fix cost, and fix effort. The results of regression models for three data sets were reported. The results were not conclusive to find the best models in each data set and all regression models had similar performance. The prediction of fault count had a better performance in most models in KC1 and KC4 data sets. We found the prediction of fix cost is the best in KC3 only. Engineers may not have enough time to explore the quality of all modules in large software systems. It is vital to show the value of using these models in doing cost-effective quality assurance, e.g., prioritizing modules for further investigation. We have modeled the results of the prediction models by plotting the relationship between %modules and %faults after sorting the modules by faults predicted.

TABLE VIII. THE RELATIONSHIPS RESULTING FROM IMPLEMENTATION OF THE PREDICTION MODELS

	KC1			KC3			KC4		
	Fault count	Fix cost	Fix effort	Fault Count	Fix cost	Fix effort	Fault count	Fix cost	Fix effort
LR	20:58	20:59	20:62	20:54	20:60	20:49	20:49	20:60	20:43
KNN	20:58	20:58	20:63	20:50	20:58	20:33	20:57	20:58	20:58
M5P	20:59	20:48	20:58	20:48	20:60	20:38	20:46	20:60	20:49
SMO	20:58	20:57	20:56	20:54	20:59	20:50	20:49	20:59	20:49
MLP	20:58	20:44	20:54	20:42	20:43	20:14	20:30	20:43	20:29

We have also plotted the relationship between (%modules, %fix cost) and (%modules, %fix effort). The plots have shown that the 20:60 rule can be applied for the three measures.

These results are important to conclude that we can use the same metrics to predict different fault measures, i.e., answering the three research questions. The software engineers can have alternative methods to select software modules for further verification and validation from different perspectives. In future, we plan to expand this study to more diverse data sets.

REFERENCE

[1] V. R. Basili, L. C. Briand, and W. L. Melo, "A validation of object-oriented design metrics as quality indicators," IEEE Trans. Softw. Eng., vol. 22, pp. 751–761, 1996.

[2] K. El Emam, W. Melo, and J. Machado, "The prediction of faulty classes using object-oriented design metrics," J. Syst. Softw., vol. 56, no. 1, pp. 63–75, Feb. 2001.

[3] T. M. Khoshgoftaar and N. Seliya, "Comparative assessment of software quality classification techniques: An empirical case study," Empir. Softw. Eng., vol. 9, no. 3, pp. 229–257, 2004.

[4] T. Gyimothy, R. Ferenc, and I. Siket, "Empirical validation of object-oriented metrics on open source software for fault prediction," IEEE Trans. Softw. Eng., vol. 31, no. 10, pp. 897–910, 2005.

[5] Y. Zhou and H. Leung, "Empirical analysis of object-oriented design metrics for predicting high and low severity faults," IEEE Trans. Softw. Eng., vol. 32, no. 10, pp. 771–789, 2006.

[6] R. Shatnawi, W. Li, J. Swain, and T. Newman, "Finding software metrics threshold values using ROC curves," J. Softw. Maint. Evol. Res. Pract., vol. 22, no. 1, pp. 1–16, 2010.

[7] E. Arisholm, L. C. Briand, and E. B. Johannessen, "A systematic and comprehensive investigation of methods to build and evaluate fault prediction models," J. Syst. Softw., vol. 83, no. 1, pp. 2–17, 2010.

[8] L. C. Briand, J. Wust, J. W. Daly, and D. Victor Porter, "Exploring the relationships between design measures and software quality in object-oriented systems," J. Syst. Softw., vol. 51, pp. 245–273, 2000.

[9] T. M. Khoshgoftaar and K. Gao, "Count models for software quality estimation," IEEE Trans. Reliab., vol. 56, no. 2, pp. 212–222, 2007.

[10] R. Shatnawi, "Empirical study of fault prediction for open-source systems using the Chidamber and Kemerer metrics," IET Softw., vol. 8, no. 3, pp. 113–119, 2013.

[11] G. J. Pai and J. B. Dugan, "Empirical Analysis of Software Fault Content and Fault Proneness Using Bayesian Methods," IEEE Trans. Softw. Eng., vol. 33, no. 10, pp. 675–686, 2007.

[12] C. Catal and B. Diri, "A fault prediction model with limited fault data to improve test process," in Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 2008, vol. 5089 LNCS, pp. 244–257.

[13] I. Gondra, "Applying machine learning to software fault-proneness prediction," J. Syst. Softw., vol. 81, no. 2, pp. 186–195, 2008.

[14] J. Zheng, "Cost-sensitive boosting neural networks for software defect prediction," Expert Syst. Appl., vol. 37, pp. 4537–4543, 2010.

[15] N. Ohlsson and H. Alberg, "Predicting Fault-Prone Software Modules in Telephone Switches," IEEE Trans. Softw. Eng., vol. 22, no. 12, pp. 886–894, 1996.

[16] S. Biyani and P. Santhanam, "Exploring defect data from development and customer usage on software modules over multiple releases," in Proceedings Ninth International Symposium on Software Reliability Engineering (Cat. No.98TB100257), 1998, pp. 316–320.

[17] T. J. Ostrand, E. J. Weyuker, and R. M. Bell, "Predicting the location and number of faults in large software systems," IEEE Trans. Softw. Eng., vol. 31, no. 4, pp. 340–355, 2005.

[18] H. Zeng and D. Rine, "Estimation of software defects fix effort using neural networks," in Proceedings - International Computer Software and Applications Conference, 2004, vol. 2, pp. 20–21.

[19] L. D. Panjer, "Predicting eclipse bug lifetimes," in Proceedings - ICSE 2007 Workshops: Fourth International Workshop on Mining Software Repositories, MSR 2007, 2007, p. 29.

[20] M. Hamill and K. Goseva-Popstojanova, "Exploring fault types, detection activities, and failure severity in an evolving safety-critical software system," Softw. Qual. J., pp. 1–37, 2014.

[21] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell, "A systematic literature review on fault prediction performance in software engineering," IEEE Trans. Softw. Eng., vol. 38, no. 6, pp. 1276–1304, 2012.

[22] D. Radjenović, M. Heričko, R. Torkar, and A. Živkovič, "Software Fault Prediction Metrics:A Systematic Literature Review," Inf. Softw. Technol., vol. 55, no. 8, pp. 1397–1418, 2013.

[23] G. Boetticher, T. Ostrand, and T. Menzies, "Promise repository of empirical software engineering data." 2007.

[24] T. Zimmermann, R. Premraj, and A. Zeller, "Predicting defects for eclipse," in Proceedings - ICSE 2007 Workshops: Third International Workshop on Predictor Models in Software Engineering, PROMISE'07, 2007, p. 9.

[25] NASA M.D.P., "NASA Independent Verification and Validation facility," 2014. [Online]. Available: http://mdp.ivv.nasa.gov.

[26] T. Menzies and J. S. Di Stefano, "How good is your blind spot sampling policy," in Eighth IEEE International Symposium on High Assurance Systems Engineering, 2004. Proceedings., 2004, pp. 129–138.

[27] M. Shepperd, Q. Song, Z. Sun, and C. Mair, "Data quality: Some comments on the NASA software defect datasets," IEEE Trans. Softw. Eng., vol. 39, no. 9, pp. 1208–1215, 2013.

[28] T. J. McCabe, "A Complexity Measure," IEEE Trans. Softw. Eng., vol. 4, no. 2, pp. 308–320, 1976.

[29] M. Halstead, Elements of Software Science. 1977.

[30] J. R. Quinlan, "Learning with continuous classes," in Machine Learning, 1992, vol. 92, pp. 343–348.

[31] D. W. Aha, D. Kibler, and M. K. Albert, "Instance-based learning algorithms," Mach. Learn., vol. 6, no. 1, pp. 37–66, 1991.

[32] S. Sayed, An Introduction to Data Mining. 2014.

[33] L. Briand, J. Wust, S. Ikonomovski, and H. Lounis, "A Comprehensive Investigation of Quality Factors in Object-Oriented Designs: An Industrial Case Study." 1998.

[34] Y. Zhou, B. Xu, L. Chen, and L. Hareton, "An in-depth study of the potentially confounding effect of class size in fault prediction," Trans. Softw. Eng. Methodol., vol. 23, no. 1, p. 51, 2014.