Published in IET Software
Received on 18th January 2013
Revised on 27th July 2013
Accepted on 15th August 2013
doi: 10.1049/iet-sen.2013.0008

ISSN 1751-8806

Empirical study of fault prediction for open-source systems using the Chidamber and Kemerer metrics

Raed Shatnawi

Software Engineering Department, Jordan University of Science and Technology, Irbid 22110, Jordan
E-mail: raedamin@just.edu.jo

Abstract: Software testers are usually provoked with projects that have faults. Predicting a class's fault-proneness is vital for minimising cost and improving the effectiveness of the software testing. Previous research on software metrics has shown strong relationships between software metrics and faults in object-oriented systems using a binary variable. However, these models do not consider the history of faults in classes. In this work, a dependent variable is proposed that uses fault history to rate classes into four categories (none, low risk, medium risk and high risk) and to improve the predictive capability of fault models. The study is conducted on many releases of four open-source systems. The study tests the statistical differences in seven machine learning algorithms to find whether the proposed variable can be used to build better prediction models. The performance of the classifiers using the four categories is significantly better than the binary variable. In addition, the results show improvements on the reliability of the prediction models as the software matures. Therefore the fault history improves the prediction of fault-proneness of classes in open-source systems.

1 Introduction

A common problem in software systems is that they are complex and always have faults. The anticipation of faults in software systems proves to be a difficult, if not an impossible, task. The appearance of faults mean the product was not validated enough. Many validation techniques can be used to find software faults such as software testing, code inspection and walkthroughs. However, software testing is very costly in terms of time and resources (e.g. testing takes more than 50% of the total project effort for some complex systems). Software testers cannot be sure that the program is fault-free because of limited resources available during the testing process. Software testing is a cost-effective process that should be focused on the most complex parts of software that require more testing effort. Measuring software quantitatively and identifying where faults are more likely to occur in a system can make the work of software testers more effective and efficient. Predicting which classes are faulty is necessary to guide software testers to improve the software quality and to reduce the costs of testing and maintenance. Prediction models are effective tools for reducing software faults when the right software measurements are selected [1]. Moreover, knowing why some classes are more fault-prone helps in understanding the nature of faults [2]. The probability of discovering faults can be identified from the history of classes. The fault-proneness can be divided into several categories such that classes that fall within the high category are assigned more resources than the classes within the low category. However, in previous studies on software fault prediction, many research studies have shown that the used data are incomplete and imbalanced, which has a significant effect on the predictive capability of fault-proneness models [3, 4]. The knowledge of fault incidents from previous releases is not always considered although it can give an indication of the recurrence of faults. We expect that the performance of classifiers should improve when more information is added such as the existence of faults in previous releases. In addition, such classes may have inherent complexity and need more attention of the developers and the testers of systems.

Software metrics are used in this study to quantify the risks of having faults in future releases to focus limited resources on significant problems that face the project. We investigate the fault prediction in open-source systems using the Chidamber and Kemerer (CK) metrics [5]. In particular, we propose to use the past incidents of faults to rate classes into several categories: low, medium and high. These ratings are tested to build several fault-prediction models using many data mining tools. The rest of this paper is organised as follows: Section 2 discusses the related work. In Section 3, we describe the research methodology. The results of each project under investigation are reported in Section 4 with a summary of all projects. Finally, we conclude our work.

2 Related work

Previous works on the CK metrics [5] have found an empirical association with fault-proneness [6–8], fault counts [9] and fault categories [10]. Such studies on fault-proneness categorised software classes into several

groups. Usually, classes are divided into two groups: faulty classes that had one or more faults in the current release, and not faulty classes that did not have zero faults [11, 12]. Szabo and Khoshgoftaar classified software classes into low-risk (#faults ≤ 2), medium-risk ($2 <$ #faults ≤ 19) and high-risk (#faults > 19) with respect to the number of faults instead of a binary coded variable (faulty/no faulty) [10]. Some other researchers used the severity of faults to create several categories (low, medium and high) [13, 14]. These researchers used the bug repository to extract the information on the severity. The authors could build a sound and plausible models using the severity levels. Other researchers have used preprocessing and noise reduction to improve the quality of the prediction models [15, 16].

In this research, the history of faults is considered to improve the reliability of fault predictions by rating faulty classes into three ratings (low, medium and high) and by reducing the imbalance in fault distributions. This rating results in a new dependent variable that is compared with the usually considered classification of classes into faulty and not faulty.

3 Study design

The objective of this empirical study is to analyse the use of fault history in building fault-prediction models in the open-source field. We aim to use class history to rate classes into s several areas. To validate the research objectives, we use the evolution of many open-source systems. The quality of these systems is measured using a set of metrics. We use seven machine learning classification techniques that were widely used to build fault-prediction models. Finally, we statistically test the significance of the difference between the proposed models (multi-category classification) and traditional models (binary classification).

3.1 Proposed dependent variable

A binary variable is usually used as a dependent variable in fault-prediction models. Usually, a class is assigned a value false if there are no faults fixed and true otherwise. In this work, we propose a new dependent variable based on fault incidents in the previous release. Instead of using the full history of a software since creation, we use the previous release only to provide indications of evolution of faults in classes. Fault counts represent the number of code fixes in a particular class. Using the fault fixes of successive releases of a system provides an audit trail of the classes that have suffered throughout the lifecycle of the system. The proposed variable is composed of information from two releases: previous and current. The classifiers use this variable to predict the fault proneness of classes. Such classifiers can be used to allocate resources more effectively for future activities such as testing and maintenance. The proposed dependent variable is merely an updated version of the binary variable. The proposed variable has the information of the binary variable in addition to history information. Table 1 presents the new dependent variable and shows four categories for a class in the current release (V_{n+1}). Table 1 is summarised as follows.

- None: The class does not have faults in both the current and the previous release.
- Low: There are faults fixed for a class in the previous release (i.e. fault-prone) and no faults are fixed in the current release (i.e. not fault-prone); such classes are considered as faulty but within the low category.
- Medium: There are no faults for a class in the previous release (i.e. not fault-prone) and there are faults in the current release (i.e. fault-prone).
- High: There are faults in both the previous and the current release (i.e. both classes are fault-prone in both releases). Such classes are given higher priority than other categories and require more attention from the developers and testers of the software. We assume detecting classes in the high category is more important than the low and medium categories.

Table 1 Ratings of software classes

Case#	Current (#faults V_{n+1})	Previous (#faults V_n)	Rating
1	zero faults	zero faults	none
2	zero faults	*N* faults	low
3	*N* faults	zero faults	medium
4	*N* faults	*N* faults	high

There are two situations that are not covered in Table 1. First, removed classes are not included in building the model because they are not part of the software release anymore. Second, new classes are added and considered as having a clear history (i.e. covered in either Case #1 or Case #2).

3.2 Independent variables (software metrics)

CK have proposed and validated a suite of six metrics that can be used as indicators of design quality [5]. These metrics are summarised as follows:

- Coupling between objects (CBO): the CBO metric finds couplings with other classes. Larger values of CBO metrics mean that the class is highly coupled.
- Depth of inheritance hierarchy (DIT): the DIT finds the depth of a class in the inheritance hierarchy. It may require developers and testers to understand all ancestors to comprehend all specialisations of the class.
- Number of child classes (NOC): the NOC metric counts the number of subclasses. The number of children represents the number of specialisations and uses of a class. Therefore understanding all children is important to understand the parent.
- Lack of cohesion of methods (LCOM): the LCOM metric is the number of pairs of methods in the class using no attributes in common (refer to as P), minus the number of pairs of methods that do (refer to as Q). The LCOM is set to zero if the difference is negative. After considering each pair of methods

$$\text{LCOM} = (P > Q)?(P - Q):0$$

The LCOM metric measures the coherence among methods in a class. The class that does one thing is easier to reuse and maintain than the class that does many different things.
- Response for class (RFC): the RFC metric counts the number of methods in the response set for a class, which includes the number of local methods and the number of remote methods invoked by local methods. The class that has a large number of responsibilities tends to be large and has many interactions with other classes.

doi: 10.1049/iet-sen.2013.0008

• Weighted methods complexity (WMC): the WMC metric is the sum of the complexity of all methods for a class. Larger values of WMC metric mean large complexity as well.

3.3 Data collection and fault distributions

The fault data for the systems under investigation are provided by the Promise Data Repository and faults have been collected from repositories of the projects by [3, 17]. With the help of two tools, BugInfo(https://kenai.com/projects/buginfo) and Ckjm, the number of faults and metrics are collected for all software classes. The ckjm tool has been used to calculate the CK metrics. BugInfo analyses the history of changes of the classes as recorded in the code repositories using regular expressions. BugInfo uses a configuration file that describes how the tool collects data. When a comment in a revision fits to the following regular expression, the revision is interpreted as a bug fix.

$$.*(([bB][uU][gG][fF][iI][xX])|([bB][uU][gG][zZ][iI][lL][lL][aA])).*$$

If a change log matches a bug fix then the fault count of all modified classes was incremented. The faults distributions are shown for each system in Table 2. Most of these systems have imbalanced distributions; that is, the faulty classes are minority while the non-faulty classes are the majority. For example, the percentages of the classes that have faults are between 10.9 and 26.2 in Ant project. The incidents of faults are also counted as proposed in Table 1. The usage of fault incidents helps in balancing the fault distributions, that is, the low category increases the minority class. For example, the percentage of not faulty classes decreased from 77.5% to 69.7 for Ant 1.4 project. This behaviour applies to all systems under the study as shown in Table 2. For all systems, the number of classes ranged between 178 (first release of Ant project) and 965 (latest release for Camel project). The descriptions of these systems are provided online (http://ant.apache.org/, http://camel.apache.org/, http://xerces.apache.org/xerces-j/, www.jedit.org).

3.4 Classification techniques and performance evaluation

In this study, we use seven machine-learning classifiers to test the main hypothesis of this research. All selected classifiers are commonly used in the field of software engineering and these classifiers are suitable for both the binary and multi-category dependent variables. Weka is employed to train and test these classifiers and the default settings of these learners are used [18]. In the following, we provide a brief description of each classifier.

• Bayes networks (BN): A Bayesian network is a graphical model that displays nodes (metrics) in a data set and the conditional independence between them to predict a categorical field (faulty or not faulty) [19].
• Support vector machines (SVM): SVM learners use hyperplanes (works even when the data are not linearly separable) to find the best function that discriminates between classes (faulty or not faulty) by maximising the margin between classes [20, 21].
• Neural networks: The multi-layer perceptron (MLP) is similar to the organisation of the brain neurons. Neurons are arranged in layers (i.e. input layer, number of hidden layers and output layer). Connections between the neurons provide the network with the ability to learn patterns. In MLP, each hidden layer neuron uses a combination of

Table 2 Fault distributions

Data set	#Classes		Binary			Four		
			NFP	FP	None	Low	Medium	High
Ant1.4	178	#	138	40	124	14	34	6
		%	77.5%	22.5%	69.7%	7.9%	19.1%	3.4%
Ant1.5	293	#	261	32	229	32	25	7
		%	89.1%	10.9%	78.2%	10.9%	8.5%	2.4%
Ant1.6	351	#	259	92	244	15	75	17
		%	73.8%	26.2%	69.5%	4.3%	21.4%	4.8%
Ant1.7	745	#	579	166	550	29	104	62
		%	77.7%	22.3%	73.8%	3.9%	14.0%	8.3%
camel1.2	608	#	392	216	390	2	205	11
		%	64.5%	35.5%	64.1%	0.3%	33.7%	1.8%
camel1.4	872	#	727	145	614	113	48	97
		%	83.4%	16.6%	70.4%	13.0%	5.5%	11.1%
camel1.6	965	#	777	188	715	62	106	82
		%	80.5%	19.5%	74.1%	6.4%	11.0%	8.5%
jedit4.0	306	#	231	75	188	43	28	47
		%	75.5%	24.5%	61.4%	14.1%	9.2%	15.4%
jedit4.1	312	#	233	79	209	24	30	49
		%	74.7%	25.3%	67.0%	7.7%	9.6%	15.7%
jedit4.2	367	#	319	48	270	49	19	29
		%	86.9%	13.1%	73.6%	13.4%	5.2%	7.9%
jedit4.3	492	#	481	11	436	45	9	2
		%	97.8%	2.2%	88.6%	9.1%	1.8%	0.4%
xerces1.2	440	#	369	71	334	35	39	32
		%	83.9%	16.1%	75.9%	8.0%	8.9%	7.3%
xerces1.3	453	#	384	69	333	51	52	17
		%	84.8%	15.2%	73.5%	11.3%	11.5%	3.8%
xerces1.4	588	#	151	437	149	2	402	35
		%	25.7%	74.3%	25.3%	0.3%	68.4%	6.0%

weighted outputs of the neurons from the previous layer. In the final hidden layer, neurons are combined to produce an output, which is compared with the correct output and the difference between the two values (the error) is fed back to update the network [21]. The most important parameter in setting the neural network is the number of hidden layers. In Weka, the number chosen hidden layers is 'a' = (attributes + classes)/2.

- Nearest neighbours (kNN): kNN assigns the dominant label of the closest group of k objects in the training set. kNN uses distance (similarity) metric to find the nearest neighbours and assigns the label that has the majority [22]. In this research, five nearest neighbours are used ($k = 5$).
- Decision trees (C4.5): C4.5 is an extension of the basic ID3 algorithm designed by Quinlan. C4.5 is a well-known decision tree classifier in the fault-prediction domain. C4.5 uses information gain to build decision trees [23]. The tree grows by selecting the decision for the attribute with the highest information gain.
- Decision trees (CART): CART builds decision trees using the Gini diversity index. The CART grows recursively by partitioning the training data set into subsets with similar values for the class (faulty or faultless). The CART algorithm grows the tree by conducting an exhaustive search of all attributes (i.e. metrics) and all possible splitting values, selecting the split that reduces impurity in each node [24].
- Decision trees (random forest): Random forests are extended form decision trees. Random forests build many classification trees (hundreds or even thousands) using subsets of the training data. To classify a class, they use the software metrics as input for all trees in the forest to find a classification [25]. The forest chooses the classification having received the most predictions (votes) from all trees [26].

The fault-prediction models are assessed in previous works by considering all faults equally distributed. However, this evaluation might not work properly when the data are imbalanced [27]. The overall accuracy or error rate is usually used to evaluate the prediction performance. Overall accuracy does not consider the distribution of the faulty data and the misclassification costs [28]. The classification of classes is based on a threshold value, which is usually considered 0.5. The probabilities that are larger than the threshold are considered faulty, otherwise not faulty. However, in imbalanced data sets, a particular threshold does not always work. In such cases, a search for another threshold that can discriminate well between classes is needed.

In this research, we compare the results of the classifiers among data sets using the receiver–operator characteristic curve (ROC). The area under the ROC curve (AUC) is used as a single value to assess the classifier performance. The ROC analysis is a diagnostic performance test that was originated from the field of signal detection [29]. The ROC area can be used to assess the quality of the information provided by classifying software classes into two or more categories (e.g. fault-prone and not fault-prone). The ROC curve depicts the benefits of using the model against the costs of using the model for all possible thresholds. Each point in the curve is a pair of sensitivity and 1-specificity values that are calculated at a particular threshold. These values are calculated for all possible threshold values, which are used to draw the ROC curve. The sensitivity and specificity are calculated from the confusion matrix,

Table 3 Confusion matrix based on a threshold value

	Actual	
Predicted	Fault-prone	Not fault-prone
fault probability >= threshold	true positives (TP)	false positives (FP)
fault probability < threshold	false negatives (FN)	true negatives (TN)
totals	P (TP + FN)	N (FP + TN)

Table 3, as follows. *Sensitivity = TP rate = TP/P; Specificity = 1 – FP rate = 1 – FP/N.* The ROC analysis is very effective for data with skewed distributions and unequal classification error costs [29]. The characteristics of the ROC analysis help researchers in generalising results even in case of changing data distributions. In addition, the ROC analysis is preferable both for practical choices and for drawing scientific conclusions [29]. The classifiers are trained and tested for all algorithms for the two dependent variables. The proposed variable is a multi-class and less common than the binary variable. There are two approaches to extend the binary classification to a multi-class classification [30]. In the first approach, a classifier between one class and the rest ($k - 1$ other classes) is trained. In the second approach, a classifier is trained between each pair of classes. Weka uses the first approach to evaluate the seven classifiers. The AUC value is produced for binary variables. For the multi-category variable, an AUC value is calculated for four pairs (none and the rest, low and the rest, medium and the rest, high and the rest). To report a single value, a weighted average of AUC is calculated as follows

$$\text{Weighted AUC} = \frac{\sum_{i=0}^{n} \text{AUC}i * \text{class Counts } i}{\sum_{i=0}^{n} \text{class Counts } i}$$

where n is 3 in this context and the denominator is the number of classes in a software release.

3.5 Hypothesis testing

This research aims to test the following null hypothesis.

H0: the performance of the classification techniques using the proposed variable is not statistically different than the performance of the classification techniques using a binary variable.

The t-test was used to assess the significance of findings in many previous researches. However, using parametric tests to compare the performance of machine learners is debated [31]. Thus, this hypothesis is validated statistically using a non-parametric test – Wilcoxon sign ranks test. In this experiment, data sets are randomly generated using the 10-fold cross-validation. The 90% of the data are used for training, and the remaining 10% are used for testing. A 10-fold cross-validation is repeated ten times (e.g. 10×10 data sets are generated) and the results are averaged to reduce the variance. This assures that a similar distribution of outcomes is present in each subset of the data. One of these subsets of the data is reserved for use as a testing data, whereas the $N - 1$ subsets are combined for use as a training data. In this work, we use the AUC to compare the significant differences in performance between the weighted AUC for the multi-category variable and the AUC for the binary variable. We finally present the hypothesis testing using Wilcoxon paired-ranks test.

doi: 10.1049/iet-sen.2013.0008

Table 4 AUC values of all classifiers on the Ant project

	ant1.4		ant1.5		ant1.6		ant1.7	
	Binary	Four	Binary	Four	Binary	Four	Binary	Four
Bayes net	0.50	0.63	0.79	0.71	0.81	0.82	0.79	0.79
neural networks	0.54	0.69	0.81	0.65	0.82	0.83	0.82	0.80
support vector machines	0.46	0.64	0.84	0.68	0.84	0.85	*0.83*	0.79
nearest neighbours	0.61	0.69	0.82	0.68	0.81	0.79	0.78	0.78
C4.5 decision trees	0.49	0.57	0.64	0.59	0.74	0.73	*0.74*	0.70
random forest trees	0.58	0.67	0.73	0.69	0.78	0.79	0.78	0.78
CART decision trees	0.50	0.50	0.59	0.50	0.74	0.75	0.73	0.72

4 Data analysis

Before doing the hypothesis testing, we introduce and describe the performance of the seven classifiers of fault-prone prediction. The study is conducted on many releases of open-source systems to find the effect of fault incidents on the predictive capability of fault-proneness. For each system, the seven classifiers are built using the binary variable (fault/no fault) and are built again using the proposed dependent variable (none, low, medium and high).

4.1 Ant project

Four releases of the Ant project are used to build fault-prediction models using the CK metrics. The summary of the performance (AUC values) of all classifiers is shown in Table 4. The results of the classifiers on Ant1.4 are not strong (poor for the binary and fair for the four categories). The results of the classifiers on Ant1.5 are acceptable for the binary category and fair for the four categories; the models of MLP, SVM and KNN are in the excellent range and higher than other classifiers. The results of the classifiers on Ant1.6 are excellent for both, the binary and the four categories.

4.2 Camel project

Three releases of the Camel project are used to build fault-prediction models using the CK metrics. The performance (AUC values) of all classifiers is shown in Table 5. The results of the classifiers on Camel1.2 are poor. The results of the classifiers are fairly acceptable on Camel1.4. These results also show improvements on the classifiers as the software matures all through the project Lifecycle.

Table 5 AUC values of all classifiers on the Camel project

	Camel 1.2		Camel 1.4		Camel 1.6	
	Binary	Four	Binary	Four	Binary	Four
bayes net	0.50	0.52	0.67	0.72	0.62	0.65
neural networks	0.55	0.56	0.70	0.74	0.66	0.66
support vector machines	0.55	0.58	0.60	0.75	0.61	0.64
nearest neighbours	0.65	0.64	0.68	0.70	0.66	0.65
C4.5 decision trees	0.52	0.51	0.60	0.61	0.54	0.60
random forest trees	0.58	0.59	0.63	0.67	0.66	0.64
CART decision trees	0.50	0.50	0.52	0.55	0.51	0.56

4.3 jEdit project

Four releases of the jEdit project are used to build fault-prediction models using the CK metrics and the results are shown in Table 6. The results of the classifiers on jEdit4.0 are acceptable and for some classifiers are excellent. The results of the classifiers on jEdit4.1 are acceptable or excellent for both dependent variables. These results also show improvements on the classifiers as the software matures all through the project Lifecycle.

4.4 Xerces project

Three releases of the Xerces project are used to build fault-prediction models using the CK metrics and the results are shown in Table 7. The results of the classifiers on Xerces1.2 are not always good (between poor and acceptable). The results of the classifiers are fairly acceptable on Xerces1.3 (two excellent classifiers, *k*NN and random forests). These results also show improvements on the classifiers as the software matures all through the project Lifecycle.

4.5 Results summary and hypothesis testing

In this section, we discuss and summarise the results of the prediction models for all systems. The evolution of these models is very important to understand whether the stability of the systems has a direct effect on the performance of the classifiers. The results for all systems show improvements on the classifiers as the software matures during the project lifecycle. For example, in the Ant project, the initial release has poor performance, whereas the last release has excellent performance (AUC > 0.80). This behaviour meets Hatton's criterion that has suggested using matured releases to build quality models [32].

We conducted the Wilcoxon signed-ranks test to validate the main hypothesis of this work. Table 8 shows the results of the statistical tests. The two-tailed *P*-value is less than the significance level (0.05); therefore, we should reject the null hypothesis. We can conclude that the performance of the classifiers in the two dependent variables is statistically different.

Fig. 1 shows a summary of the results of the Wilcoxon signed ranks test conducted to find significant differences between the classifiers for the two dependent variables. Wins are the number of times the classifiers for the four categories variable outperforms the binary variable, while losses otherwise. The number of ties between the two variables is also presented in Fig. 1. Therefore we can conclude that the number of ties is prevailing in some classifiers between the two dependent variables. However,

Table 6 AUC values of all classifiers on the jEdit project

	jEdit4.0		jEdit4.1		jEdit4.2		jEdit4.3	
	Binary	Four	Binary	Four	Binary	Four	Binary	Four
bayes net	0.74	0.81	0.75	0.84	0.83	0.90	0.50	0.74
neural networks	0.83	0.82	0.84	0.87	0.83	0.90	0.38	0.79
support vector machines	0.77	0.79	0.82	0.85	0.84	0.90	0.41	0.80
nearest neighbours	0.81	0.82	0.80	0.84	0.77	0.80	0.53	0.70
C4.5 decision trees	0.72	0.70	0.69	0.71	0.64	0.70	0.50	0.66
random forest trees	0.79	0.81	0.78	0.84	0.75	0.85	0.48	0.71
CART decision trees	0.70	0.75	0.72	0.75	0.53	0.71	0.50	0.51

Table 7 AUC values of all classifiers on the Xerces project

	Xerces1.2		Xerces1.3		Xerces1.4	
	Binary	Four	Binary	Four	Binary	Four
bayes net	0.70	0.58	0.71	0.65	0.50	0.50
neural networks	0.62	0.53	0.72	0.60	0.51	0.51
support vector machines	0.55	0.50	0.76	0.70	0.52	0.53
nearest neighbours	0.74	0.65	0.80	0.75	0.45	0.44
C4.5 decision trees	0.61	0.73	0.70	0.63	0.50	0.50
random forest trees	0.75	0.69	0.81	0.71	0.47	0.46
CART decision trees	0.51	0.50	0.65	0.60	0.50	0.50

Table 8 Wilcoxon signed-ranks test statistics

Statistical tests		Descriptive statistics		
	Four–binary (2-tailed)		Binary	Four
Z (observed value)	−2.151	mean	0.661	0.683
Z (critical value)	1.960	maximum	0.840	0.900
P-value	0.031	variance	0.016	0.013

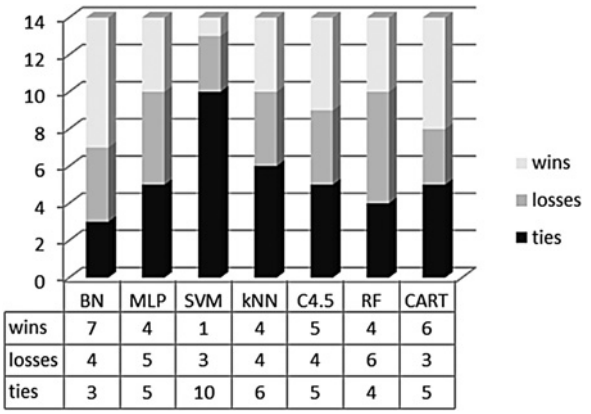

Fig. 1 *Comparing the classifiers, the number of significant wins for the proposed variable, losses and ties*

*The experiment is repeated ten times for each classifier for 10-folds in each variable. For each classifier, we built 14 classifiers. In total, 98 classifiers were trained and tested and each is repeated ten times * 10-folds

the four category variable provides more information about the nature of the faults fixed in systems. In addition, the use of the four-category variable provides more information about the history of faults. The assigned category can be used to give priorities to testing tasks. The previous incident of faults in a class that is already faulty is an indication of the design problems that need serious awareness. For example, predicting which classes are in the high category direct the focus of testing efforts to a small part of the software components (e.g. 5% of the classes). These classes have faults in more than one consecutive release, which is an indicator of a serious complexity in the design or the code of these classes.

5 Threats to validity

In this section, we discuss the validity threats of this study as follows:

Construct validity threats: The first threat to the validity of our results is to use all CK metrics in building prediction models. Previous researches have criticised the cohesion metrics, Basili *et al.* [6, 8] noted some problems in the definition of the LCOM metric (i.e. the LCOM metric gives the same value for classes with different cohesion values). Etzkorn *et al.* also have reported similar conclusions [33].
Internal validity threats: This threat comes from the fact that data collection was completed by other authors [3]. Jureczko and Madeyski [17] declared that no guarantees of the quality of the collected data and mistakes in fault identification and assignment might exist. In addition, we do not have detailed information about the software process activities and in particular the effort and cost of the testing and maintenance of the systems under study.
External validity threats: This threat comes from rating the classes into different categories based on history. Since there is no previous history for initial releases of software systems, we cannot generalise this rating to all systems. In addition, we cannot apply the rating of classes into multi-categories if fault data were not collected from previous releases.

6 Conclusions

Successful software projects require implementing quality assurance plans that lead to successful software. However, poor-quality software implies a higher software failure, especially in system operation. Therefore detecting and fixing software faults before release leads to a great reduction in the odds of software failures. Software fault detection is a major factor that leads to a successful project. Many factors have great effect on the performance of a

doi: 10.1049/iet-sen.2013.0008

fault predictor such as the learner used and the quality of the software measurement data set. In this work, a new dependent variable is proposed to improve the quality of the data sets. The proposed dependent variable has four different categories of faults instead of the commonly used categories (faulty and not faulty). The fault incidents in the previous and current release are used to assign multi-categories (low, medium and high). We have investigated the proposed categorisation of faults on the evolution of four different open-source systems. We conducted prediction models using seven different classifiers. The results of classifiers have improved for the later releases as the software matures. The performance of the classifiers has shown acceptable values for two systems (Ant and jEdit); whereas the performance of prediction models on Camel and Xerces systems were less than good (i.e. AUC < 70). The statistical test shows that the fault prediction using the multi-category learners outperforms the binary variable. In addition, the proposed variable provides more information about the history of the fault and gives more insight into how testers can allocate their efforts.

Future works will involve conducting additional experiments to tackle common problems that affect the effectiveness of software prediction models such as data imbalance and noise reduction.

7 Acknowledgment

We would like to thank the anonymous reviewers for their invaluable reviews and suggestions.

8 References

1 Gao, K., Khoshgoftaar, T.M., Seliya, N.: 'Predicting high-risk program classes by selecting the right software measurements', *Softw. Qual. J.*, 2012, **20**, pp. 3–42

2 Zimmermann, T., Nagappan, N., Zeller, A.: 'Predicting bugs from history'. In Software Evolution (Software Evolution), 2008, pp. 69–88

3 Jureczko, M., Spinellis, D.: 'Using object-oriented design metrics to predict software defects'. Proc. Fifth Int. Conf. Dependability of Computer Systems, 2010, pp. 69–81

4 Catal, C.: 'Software fault prediction: a literature review and current trends', *Expert Syst. Appl.*, 2011, **38**, (4), pp. 4626–4636

5 Chidamber, S., Kemerer, C.: 'A metrics suite for object oriented design', *IEEE Trans. Softw. Eng.*, 1994, **20**, (6), pp. 476–493

6 Basili, V., Briand, L., Melo, W.: 'A validation of object-oriented design metrics as quality indicators', *IEEE Trans. Softw. Eng.*, 1996, **22**, (10), pp. 751–761

7 Emam, K.E., Benlarbi, S., Goel, N., Rai, S.N.: 'The confounding effect of class size on the validity of object-oriented metrics', *IEEE Trans. Softw. Eng.*, 2001, **27**, (7), pp. 630–648

8 Briand, L., Wust, J., Daly, J., Porter, D.: 'Exploring the relationship between design measures and software quality in object oriented systems', *J. Syst. Softw.*, 2000, **51**, (3), pp. 245–273

9 Subramanyam, R., Krishnan, M.: 'Empirical analysis of CK metrics for object-oriented design complexity: implications for software defects', *IEEE Trans. Softw. Eng.*, 2003, **29**, (4), pp. 297–310

10 Szabo, R., Khoshgoftaar, T.: 'An assessment of software quality in a C++ environment'. Proc. Sixth Int. Symp. Software Reliability Engineering, 1995, pp. 240–249

11 Olague, H., Etzkorn, L., Gholston, S., Quattlebaum, S.: 'Empirical validation of three software metrics suites to predict fault-proneness of object-oriented classes developed using highly iterative or agile software development processes', *IEEE Trans. Softw. Eng.*, 2007, **33**, (8), pp. 402–419

12 Aggarwal, K.K., Singh, Y., Kaur, A., Malhotra, R.: 'Empirical analysis for investigating the effect of object-oriented metrics on fault proneness: a replicated case study', Dalcher, D. (Ed.): 'Software Process Improvement and Practice', (Wiley, 2008)

13 Zhou, Y., Leung, H.: 'Empirical analysis of object-oriented design metrics for predicting high and low severity faults', *IEEE Trans. Softw. Eng.*, 2006, **32**, (10), pp. 771–789

14 Shatnawi, R., Li, W.: 'The effectiveness of software metrics in identifying error-prone classes in post-release software evolution process', *J. Syst. Softw.*, 2008, **81**, (11), pp. 1868–1882

15 Gray, D., Bowes, D., Davey, N., Sun, Y., Christianson, B.: 'The misuse of the NASA metrics data program data sets for automated software defect prediction'. Proc. Evaluation and Assessment in Software Engineering (EASE), 2011

16 Al Dallal, J.: 'The impact of accounting for special methods in the measurement of object-oriented class cohesion on refactoring and fault prediction activities', *J. Syst. Softw.*, 2012, **85**, (5), pp. 1042–1057

17 Jureczko, M., Madeyski, L.: 'Towards identifying software project clusters with regard to defect prediction'. Proc. Sixth Int. Conf. Predictive Models in Software Engineering, 2010, pp. 1–10

18 Hall, M., Frate, E., Holmes, G., Pfahringer, B., Reutemann, R., Witten, I.: 'The WEKA data mining software: an update', *SIGKDD Explorat.*, 2009, **11**, (1), pp. 10–18

19 Heckerman, D.: 'A tutorial on learning with Bayesian networks'. In Proc. NATO Advanced Study Institute on Learning in Graphical Models, 1998, pp. 301–354

20 Burges, C.: 'A tutorial on support vector machines for pattern recognition', *Data Min. Knowl. Discov.*, 1998, **2**, (2), pp. 121–167

21 Gondra, I.: 'Applying machine learning to software fault-proneness prediction', *J. Syst. Softw.*, 2008, **81**, (2), pp. 186–195

22 Aha, D., Kibler, D.: 'Instance-based learning algorithms', *Mach. Learn.*, 1991, **6**, (1), pp. 37–66

23 Quinlan, JR.: 'C4.5: Programs for machine learning' (Morgan Kaufmann Publishers, San Mateo, 1993)

24 Ebert, C.: 'Classification techniques for metric-based software development', *Softw. Qual. J.*, 1996, **5**, (4), pp. 255–272

25 Guo, L., Ma, Y., Cukic, B., Singh, H.: 'Robust prediction of fault-proneness by random forests'. 15th Int. Symp. Software Reliability Engineering, 2004, pp. 417–428

26 Breiman, L.: 'Random forests', *Mach. Learn.*, 2001, **45**, (1), pp. 5–32

27 Chawla, N.: 'C4.5 and imbalanced data sets: investigating the effect of sampling method, probabilistic estimate, and decision tree structure'. Workshop on Learning from Imbalanced Data sets II, ICML, Washington DC, 2003

28 Jiang, Y., Cukic, B., Ma, Y.: 'Techniques for evaluating fault prediction models', *Empir. Softw. Eng.*, 2008, **13**, (5), pp. 561–595

29 Fawcett, T.: 'ROC graphs: notes and practical considerations for researchers', Technical report, HP Laboratories, Page Mill Road, Palo Alto, CA 2004, pp. 38

30 Sun, Y., Wong, A., Kamel, M.: 'Classification of imbalanced data: a review', *Int. J. Pattern Recognit. Artif. Intell.*, 2009, **23**, (4), pp. 687–719

31 Demsar, J.: 'Statistical comparisons of classifiers over multiple data sets', *J. Mach. Learn. Res.*, 2006, **7**, pp. 1–30

32 Hatton, L.: 'The Role of empiricism in improving the reliability of future software'. Keynote Talk at TAIC PART, 2008, http://www.leshatton.org/wp-content/uploads/2012/01/TAIC2008-29-08-2008.pdf, accessed, April 2012

33 Etzkorn, L., Davis, C., Li, W.: 'A practical look at the lack of cohesion in methods metric', *J. Object-Oriented Program.*, 1998, **11**, (5), pp. 27–34