Innovations Syst Softw Eng (2017) 13:201–217
DOI 10.1007/s11334-017-0295-0

ORIGINAL PAPER

The application of ROC analysis in threshold identification, data imbalance and metrics selection for software fault prediction

Raed Shatnawi[1]

Received: 7 March 2016 / Accepted: 26 July 2017 / Published online: 2 August 2017

Abstract Software engineers have limited resources and need metrics analysis tools to investigate software quality such as fault-proneness of modules. There are a large number of software metrics available to investigate quality. However, not all metrics are strongly correlated with faults. In addition, software fault data are imbalanced and affect quality assessment tools such as fault prediction or threshold values that are used to identify risky modules. Software quality is investigated for three purposes. First, the receiver operating characteristics (ROC) analysis is used to identify threshold values to identify risky modules. Second, the ROC analysis is investigated for imbalanced data. Third, the ROC analysis is considered for feature selection. This work validated the use of ROC to identify thresholds for four metrics (WMC, CBO, RFC and LCOM). The ROC results after sampling the data are not significantly different from before sampling. The ROC analysis selects the same metrics (WMC, CBO and RFC) in most datasets, while other techniques have a large variation in selecting metrics.

Keywords ROC analysis · Imbalanced data · Feature selection · Fault-proneness models · Software metrics

1 Introduction

Software quality engineers must exploit tools to monitor, audit and verify the software fault-proneness during the life cycle of a project. Software engineers keep records of fault data in a special repository such as Bugzilla. The fault data can be used to find where faults are likely to occur. However, the fault data may not be recorded or available for many reasons. There is a need for indirect measurement of the software, which can be used as a surrogate of software fault-proneness. Software metrics, for example the Chidamber and Kemerer metrics (CK) [15], were validated as indicators of fault-proneness of classes. CK metrics found to have significant relationships with faults using many machine learning and statistical techniques [2,4,18,19,28,37,48,54,63]. Machine learning and statistical techniques are rigorous, and dedicated software tools are needed to analyze such relationships. However, software engineers need more easy tools to investigate fault-proneness in modules. Identifying which classes are more likely to have faults is necessary to guide software testers to improve their performance and reduce the costs of activities such as testing and maintenance [18]. The presence of faults can be used to profile software modules into several risk levels (threshold value or reference value). However, we face some problems in the quality of data. Fault data may not be collected for part of the system because the costs of collection may be extremely expensive [11]. Two major issues are considered in analyzing the fault-proneness in classes: (1) few classes have faults, while the majority do not have faults (imbalance fault distribution) [42,57], and (2) many metrics already exist and can be used to evaluate software fault-proneness. Khoshgoftaar et al. [38] proposed two techniques to confront these issues: a data sampling technique to overcome the class imbalance problem in fault distribution and a feature selection technique for selecting the important metrics. According to Khoshgoftaar et al. [38], the performance of a prediction model depends on both the selected metrics and faults distribution in modules.

✉ Raed Shatnawi
raedamin@just.edu.jo

[1] Software Engineering Department, Jordan University of Science and Technology, Irbid 22110, Jordan

In this study, we propose to use the receiver operating characteristic (ROC) analysis as a quality assurance tool. The ROC analysis is proposed as a quality assurance tool in selecting software metrics as fault-proneness indicators. The ROC identifies a threshold value that separates software modules into two areas: low quality (fault-prone) and high quality (not fault-prone). The results of ROC analysis can be used in quality assurance tools such as JArchitect[1] to identify, for example, the most coupled classes or the lowest cohesive classes in a system. JArchitect uses thresholds to identify bad quality areas and allows the user to change thresholds. The tool reports all classes that exceed thresholds and requires more quality inspection. In a white paper, Gronback [27] reported on using threshold values to identify bad smells in a commercial quality assurance tool (Borland together). The tool uses threshold values to identify potential bad smells such as god classes, god methods and data class. The used threshold values were subjectively reported to be used to identify bad smells in code [44]. This study introduces ROC analysis to investigate the relationship between the CK metrics and the faults in five open-source systems. The study aims to validate the use of ROC in threshold identification. We also validate the consistency of the ROC analysis after two major sampling techniques: oversampling and undersampling. The ROC is used to select metrics to include in learners, and the stability of selection is assessed and compared with other traditional feature selection methods. Finally, we use the metrics that are selected via the ROC analysis to build fault-proneness models using four well-known learners: logistic regression, naïve bayes, the nearest neighbors and C4.5 decision trees [10].

The rest of this paper is organized as follows: In Sect. 2, we discuss the related work on fault prediction and identification of threshold values. Section 3 discusses the experimental design of this research and provides a detailed description of the research methodology. In Sect. 4, we present and discuss the results of this work. Finally, we conclude this work.

2 Related work

Studies on fault-proneness categorized software classes into several groups based on the number of faults. Usually, classes are divided into two groups: faulty classes that have one or more faults in the current release under investigation and non-faulty classes that do not have any faults. Other researchers utilized the threshold values of software metrics in many applications. Metric threshold values can help developers in identifying the most risky classes in software design. Using threshold values, a developer can bookmark such classes during the daily development tasks and make quick decisions whenever needed. However, there were only few empirical studies on threshold values. Erni et al. [21] proposed to use normal distribution parameters (average and standard deviation) to determine thresholds. Erni et al. calculated threshold values as follows, Tmin $= \mu - s$ and Tmax $= \mu + s$, being μ the average of a metric and s the standard deviation. However, these thresholds were not empirically validated. Daly et al. [16] studied the effect of two arbitrary levels of inheritance (three and five) on maintenance time. Other researchers, [8,30,49], replicated the study of Daly et al. with different systems and only considered the effect of three levels of inheritance on maintenance time and found different results. However, these experiments were conducted on few undergraduate students. Benlarbi et al. [6] and El Emam et al. [20] estimated the threshold values using a logistic regression and could not find valid thresholds. Shatnawi [55] used a logistic regression method reported in Bender [5] to find thresholds for the Chidamber and Kemerer suite. Shatnawi [55] used the derivatives of the logistic regression to identify several risk levels in software classes, e.g., CBO could have four different thresholds at four different risk levels, CBO = 6, 9, 16 or 29. In another study, Shatnawi et al. [56] studied the use of ROC curve to identify threshold values of object-oriented metrics. They conducted the study on three releases of a large open-source system—Eclipse. The results could identify thresholds for only three metrics, RFC = 44, CBO = 13 and WMC = 24, whereas the LCOM, DIT and NOC could not have plausible and practical thresholds. Catal et al. [11] proposed a modified ROC analysis of Shatnawi et al. [56] to obtain thresholds for structural metrics. Catal et al. [11] used outlier detection techniques to improve the performance of fault prediction by labeling non-faulty classes as faulty if they exceed threshold criteria and faulty classes otherwise. For example, the threshold values for the SLOC falls in the range between 11 and 33. Ferreira et al. [23] derived thresholds for some OO metrics using statistical properties, such as power law behavior, of the metrics. For example, the authors assigned three rankings based on ranges of metrics: good, regular and bad. Ferreira et al. [23] found different thresholds for different application domains (11 domains reported in the study).

To our knowledge, previous studies on identifying threshold values have not addressed the problems of data imbalance and metric selection. Since the same data are used to build fault prediction models and threshold identification, we need to study these two important issues on threshold identification. Many studies have already addressed data imbalance in fault prediction [3,38,42,57].

The ROC analysis was proposed to identify threshold values in Shatnawi et al. [56]. The objectives of this research are to extend our previous work [56] on identifying threshold values using ROC curves to new contexts and to test it under the effect of data imbalance and feature selection. The previ-

[1] www.jarchitect.com.

ous and the current works are both empirical but have many differences. The previous work included only one project (Eclipse), while we study five projects in this work. However, the objective of this work is different. In this work, we provide more evidence on using the ROC to identify thresholds even for imbalanced data which have a major effect on the validity of the results of fault prediction. In addition, we introduce the ROC analysis as a method of metric selection, which is also important to provide a better performance in quality assurance activities.

3 Experimental design

In this section, we discuss the details of the ROC analysis, the metrics under investigation, the research objectives, data sources and feature selection approaches, and finally, we provide a brief description of classification models.

3.1 Area under the curve (AUC)

The area under the receiver operating characteristic (ROC) analysis is used in classifiers evaluation. The ROC curve is plotted using two variables: one is binary and another is continuous. The binary variable is the event of faults or not in software modules (i.e., 1 and 0). The continuous variable is one of the CK metrics, e.g., WMC metric in Fig. 1. Each metric is analyzed separately by considering all values as potential thresholds that can be used to categorize classes into either faulty (≥threshold) or not faulty (<threshold). For each potential threshold, a classification table (confusion matrix) is produced as shown in Table 1.

Each table can be used to calculate two important measures of ROC performance: sensitivity and specificity, which are defined as follows.

$$\text{Sensitivity} = \text{TP rate} = \text{TP/P};$$
$$\text{Specificity} = \text{TN rate} = \text{TN/N}.$$

The area under the curve (AUC), as shown in Fig. 1, shows a visual trade-off analysis between the rate of correctly classified classes as fault-prone and the rate of incorrectly classified classes as not fault-prone. The AUC is a single value that evaluates the discrimination power in the curve between the faulty and not faulty classes. The diagonal line in Fig. 1 represents the approach of randomly guessing a class. If a classifier randomly guesses the positive class 50% of the time, then half of the positives and half of the negatives are classified correctly. The area under the diagonal line is calculated as 0.50. Therefore, the curve that discriminates well between the two classes should be larger than 0.5 and should approach the upper left corner. Hosmer and Lemeshow suggested using

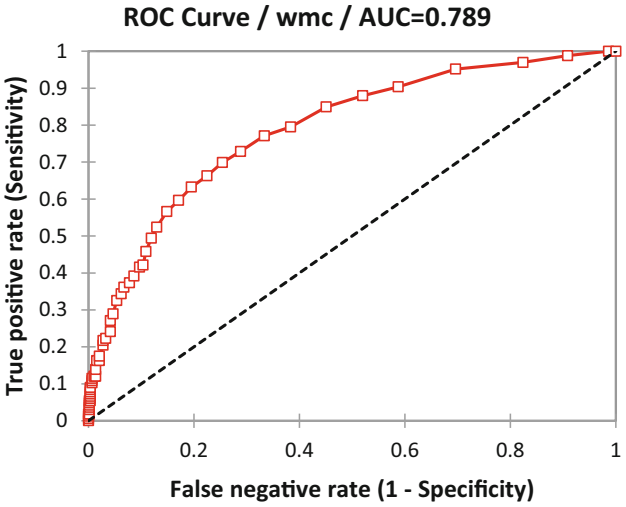

Fig. 1 The ROC curve for the WMC metric in jEdit4.2

Table 1 The confusion matrix based on a threshold value

Actual		
Predicted	Fault-prone	Not fault-prone
Metric ≥ threshold	True positives (TP)	False positives (FP)
Metric < threshold	False negatives (FN)	True negatives (TN)
Totals	P = TP + FN	N = FP + TN

the following rules to evaluate the performance of classifiers using AUC [31]:

- AUC = 0.50: means no good classification and not significantly different from random classifier;
- 0.50 < AUC < 0.60: means poor classification;
- 0.60 ≤ AUC < 0.70: means fair classification;
- 0.70 ≤ AUC < 0.80: means acceptable classification;
- 0.80 ≤ AUC < 0.90: means excellent classification;
- AUC ≥ 0.9: means outstanding classification.

The ROC analysis is very effective for data with skewed distributions and unequal classification error costs [22]. The characteristics of the ROC analysis help researchers in generalizing results even in case of changing data distributions [39,40]. In addition, the ROC analysis is preferred both for practical choices and for drawing scientific conclusions [64]. Koru et al. showed that smaller modules are more fault-prone than larger modules [39]. Kubat and Matwin found that in imbalanced dataset, the effect of the negative cases (no bugs in classes) prevails [40].

3.2 The Chidamber and Kemerer metrics

In this research, the CK metrics are used to build predictive models. Chidamber and Kemerer validated six metrics

both theoretically and empirically and proposed to use these metrics to predict design quality factors. The CK suite is composed of six metrics that measure the complexity of the software design by measuring the properties of classes. In the definition of these metrics, we refer to software modules as classes. These metrics are summarized as follows:

- Coupling between Objects (CBO): the CBO metric counts the number of other classes to which a class is coupled. Larger values of CBO metrics mean that the class is highly coupled. The developers and testers perceive that the maintainability and testability of highly coupled classes are difficult, which makes the process of maintaining and uncovering errors pre-release and post-release difficult as well.
- Depth of Inheritance Hierarchy (DIT): the DIT measures the length of the inheritance chain from the root of the inheritance tree to the measured class. The DIT metric is an indicator of the number of ancestors of a class. It may require developers and testers to understand all ancestors to comprehend all specializations of the class, which is necessary to maintain or uncover pre- and post-release faults.
- Number of Child Classes (NOC): the NOC metric counts the number of descendants of a class. The number of children represents the number of specializations and uses of a class. Therefore, understanding all children is important to understand the parent. Large number of children increases the burden on developers and testers in comprehending, maintaining and uncovering both pre- and post-release faults.
- 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 this difference is negative. After considering each pair of methods:

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

  The LCOM metric measures the coherence among local methods in a class. The class that does one thing (i.e., cohesive class) is easier to reuse and maintain than the class that does many things (i.e., the class provides many services).
- Response for Class (RFC): The size of the response set for the class includes methods in the class inheritance hierarchy and methods that can be invoked on other objects. 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 many responsibilities tends to be large and has many interactions with other classes. Therefore, such classes are complex and incur more time and effort to maintain and test than small classes.
- Weighted Methods Complexity (WMC): the WMC metric is the sum of the complexity of all methods for a class. Normally, many software metrics tools calculate the WMC metric as simply the number of methods in a class. This is equivalent to considering all functions have equal complexity. Therefore, larger values of WMC metric mean large complexity as well.

3.3 Research objectives

We aim to analyze the CK metrics using ROC analysis. The sensitivity and specificity are calculated for all potential thresholds and are used to draw the ROC curve as shown in Fig. 1. One pair, on the ROC curve, is considered better than another, if it approaches the upper left corner (i.e., TP rate is higher, FP rate is lower, or both) [22].

This study aims to use ROC analysis to achieve three objectives:

Objective 1 Study software fault-proneness in open-source systems using ROC curves. We study whether software metrics can discriminate between the fault-prone and not fault-prone classes using ROC curves. The aim of this objective is to find practical threshold values using ROC analysis. The search for threshold values has many steps. The first step in the search for a practical threshold value is to test the hypothesis H_{01} for software metrics. This test is essential step in ROC analysis and vital to assess whether a particular ROC curve is usable or not in the first place [31]. The null hypothesis (H_{01}) is proposed to find whether the curve is significantly different from a random classifier (AUC = 0.5).

$\mathbf{H}_{01}$: The AUC for a particular metric is equal to 0.5. The AUC should be significantly different from the diagonal line to consider the curve as a candidate to identify a threshold.

If the AUC is significantly different from random guessing (i.e., reject H_{01}), then the ROC is usable and we continue the search for practical threshold values; otherwise, the threshold identification for the metric stops at this point. The second step is to identify the metrics that rejected the null hypothesis (H_{01}). The curves are considered as acceptable classification when the ROC value is larger than 0.70 [31]. For the selected curves, many points can be potential threshold values. We suggest two criteria in identifying the best threshold value:

1. Threshold values should be closer to the ideal point (0, 1). We use the Euclidean distance measure to find the distance between the ideal point (0, 1) and every possible threshold. The lowest distance is considered as the closest to the ideal point.
2. The ROC curve is built using two pairs of values (sensitivity, 1-specificity) that are used to evaluate the

performance of classification. We need to use one value to evaluate the performance of a particular threshold. The slope of the tangent line at a threshold on the curve tells us the ratio of the probability of identifying true positives over true negatives, i.e., likelihood ratio (LR) for the test value.

$$\text{LR} = \text{sensitivity}/\ (1\text{-specificity})$$

The likelihood ratio shows the ratio of change in the two values. If the likelihood ratio is equal to one, the selected threshold does not add additional information to identify the true positives and it is equal to the diagonal line shown in Fig. 1. If the likelihood ratio is greater than one, then the selected threshold helps in identifying the true positive result [60]. If the ratio is less than one, then the selected threshold does not help in identifying fault-prone modules. Finally, a threshold value is selected if it has $\text{LR} > 1$.

Objective 2 Validate the effect of different sampling techniques on using ROC curves and derived thresholds. Several sampling techniques were used to improve the performance of fault-proneness models for imbalanced fault distribution. In this research, the ROC analysis is repeated for all systems after applying the following sampling techniques: synthetic minority oversampling (SMOTE) and undersampling (at a rate of 2:1). SMOTE is an oversampling approach that increases the number of instances of the minority class (faulty modules in our study) by creating synthetic examples rather than by oversampling with replacement [14]. In the implementation of SMOTE, five nearest neighbors are used to synthesize the new samples. In undersampling, a sample of the data is created such that the ratio of not faulty to faulty is kept at 2:1. These sampling techniques were widely used to improve the performance of fault-proneness models when the data distribution is imbalanced [9,42,53,57]. The results of the ROC analysis after sampling are compared with the ROC results without sampling. Therefore, we test the null hypothesis H_{02} in two scenarios: AUC values in sampling and SMOTE and AUC values in sampling versus undersampling.

$\mathbf{H}_{02}$: There are no significant differences in AUC values before and after sampling (undersampling and SMOTE) for a metric.

Objective 3 Validate the use of the ROC analysis in selecting metrics that are more significantly associated with fault-proneness in modules. Feature selection is another vital technique in improving the performance of fault prediction models [26,38]. The metrics selected using ROC are used in building fault-proneness models. The performance of four machine learning techniques: logistic regression, decision trees (C4.5), the nearest neighbors (kNN) and naïve bayes, is used in the validation of ROC analysis in selecting metrics. The fault-proneness models that use metrics resulting from the ROC analysis are compared against other models: models including all the CK metrics and models resulting from three different feature selection approaches. Therefore, we aim to test the following hypothesis

$\mathbf{H}_{03}$: There are no significant differences in the performance of the models resulted from metrics subset selection (forward selection, Chi-squared and information gain) and the models resulted from metrics selected via ROC analysis.

3.4 Data sources

Faults are discovered during the software life cycle and especially in testing phases or throughout system evolution. We study five open-source systems that are available publicly. The systems are from different domains and sizes.

- Apache Ant is a Java library and command-line tool whose mission is to drive processes described in build files as targets and extension points dependent upon each other. The main known usage of Ant is to build Java applications. Ant supplies a number of built-in tasks allowing to compile, assemble, test and run Java applications. Ant project is publicly available at: (http://ant.apache.org/)
- Apache Camel is a versatile open-source integration framework based on known enterprise integration patterns. Apache Camel uses URIs to work directly with any kind of transport or messaging models such as HTTP, ActiveMQ, JMS, JBI, SCA, MINA or CXF, as well as pluggable components and data format options. Apache Camel is a small library with minimal dependencies for easy embedding in any Java application. Camel project is publicly available at: (http://camel.apache.org/).
- jEdit is a cross platform programmer's text editor written in Java. It uses the Swing toolkit for the GUI and can be configured as a rather powerful IDE through the use of its plug-in architecture. jEdit project is publicly available at: (www.jedit.org).
- Apache Lucene is a high-performance, full-featured text search engine library written entirely in Java. Lucene provides Java-based indexing and search technology, as well as spellchecking, hit highlighting and advanced analysis/tokenization capabilities. Apache Lucene is an open-source project available for free download (http://lucene.apache.org/core/).
- Apache Synapse is a lightweight and high-performance enterprise service bus (ESB). Powered by a fast and asynchronous mediation engine, Apache Synapse provides exceptional support for XML, Web Services and REST. In addition to XML and SOAP, Apache Synapse supports several other content interchange formats, such as plain

Table 2 The fault distribution for all systems

DataSet	#classes	#Not faulty	#Faulty	%Not faulty	%Faulty
Ant1.4	178	138	40	78	22
Ant1.5	293	261	32	89	11
Ant1.6	351	259	92	74	26
Ant1.7	745	579	166	78	22
camel1.2	608	392	216	64	36
camel1.4	872	727	145	83	17
camel1.6	965	777	188	81	19
jedit4.0	306	231	75	75	25
jedit4.1	312	233	79	75	25
jedit4.2	367	319	48	87	13
jedit4.3	492	481	11	98	2
Lucene2.0	195	104	91	53	47
Lucene2.2	247	103	144	42	58
Lucene2.4	340	137	203	40	60
Synapse1.0	157	141	16	90	10
Synapse1.1	222	162	60	73	27
Synapse1.2	256	170	86	66	34

text, binary, Hessian and JSON. Apache Synapse is a free and open-source software (http://synapse.apache.org/).

The fault data for the systems under investigation were collected from repositories of the projects and reported by the promise data repository [35,36]. The authors used two separate tools: BugInfo[2] and Ckjm.[3] BugInfo was used to collect the fault data, whereas Ckjm was used to collect the CK metrics. BugInfo analyzes the history of the classes by studying the code repositories (Subversion or CVS). If a log contains a fault fix, then the affected classes are determined from the full description and marked as faulty. BugInfo uses regular expressions to extract fault information. If a log description matches a pattern of a regular expression, fault counts are incremented. The faults distributions are shown for each system in Table 2. Mostly, these systems have imbalanced distributions; i.e., 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 11 and 26 in Ant project. The use of sampling techniques helps in balancing the fault distributions.

We provide, in Table 3, the descriptive statistics for one release of each software. The inheritance metrics (DIT and NOC) have low variances, whereas the cohesion metric (LCOM) has the highest variance. The mean values range between 8 and 12 for the WMC metric, 11–14 for CBO metric, 21–40 for RFC metric and 2–3 for DIT metric. The mean of the NOC metric approaches one due to low variance. The LCOM metric shows totally different means in different projects.

3.5 Feature selection approaches

There are two main feature selection approaches: *filter* and *wrapper*. In filter approaches, no data mining techniques are used in the feature selection method. On the other hand, a wrapper approach depends on the results of the data mining algorithm to determine the effectiveness of the resulted feature subset. Many software metrics emerged to assess the fault-proneness in software. These metrics measure different dimensions and may not be significantly associated with fault-proneness of modules. Metrics that are not good indicators of fault-proneness may affect classifiers' performance [2,7]. For example, the nearest neighbor algorithm is prone to the inclusion of irrelevant features, because of their effect on the calculation of similarity measures. In this research, we consider the results of ROC analysis in Sect. 4.1 to select features to include in fault classification models. The proposed feature selection technique is compared against three other feature selection techniques: the forward stepwise greedy algorithm, Chi-squared feature evaluation and information gain feature evaluation. The forward greedy feature selection evaluates subsets by considering the individual predictive ability of each feature along with the degree of redundancy between them. Feature subsets that are highly correlated with the class while having low intercorrelation are preferred. The Chi-squared selection evaluates features by computing the value of the chi-squared statistic with respect to the class [17].

[2] https://kenai.com/nonav/projects/buginfo.

[3] http://www.spinellis.gr/sw/ckjm/.

Table 3 The descriptive statistics for later releases of the five systems

Ant1.7					Lucene 2.4				
Metric	Min	Max	Mean	SD	Metric	Min	Max	Mean	SD
WMC	0	120	11	12	WMC	1	166	10	13
DIT	1	7	3	1	DIT	1	5	2	1
NOC	0	102	1	5	NOC	0	17	1	2
CBO	0	499	11	26	CBO	0	128	11	12
RFC	0	288	34	36	RFC	1	392	25	32
LCOM	0	6692	89	350	LCOM	0	6747	69	443
Camel 1.6					Synapse 1.2				
Metric	Min	Max	Mean	SD	Metric	Min	Max	Mean	SD
WMC	0	166	9	11	WMC	0	67	8	9
DIT	0	6	2	1	DIT	1	5	2	1
NOC	0	39	1	3	NOC	0	19	0	2
CBO	0	448	11	23	CBO	0	83	13	12
RFC	0	322	21	25	RFC	0	172	30	26
LCOM	0	13617	79	524	LCOM	0	1931	41	176

jEdit4.3				
Metric	Min	Max	Mean	SD
WMC	0	351	12	25
DIT	1	8	2	2
NOC	0	38	<1	2
CBO	0	346	14	25
RFC	0	540	40	56
LCOM	0	41713	260	2185

Information gain selection evaluates features by measuring the information gain with respect to the class. Information gain measures how the entropy of the class decreases when the value of a given feature is already known [17]. The selected features are used to build fault prediction classifiers. The resulting classifiers are then compared against the models that include the metrics selected via the ROC analysis. All feature subsets are applied on four machine learning techniques: logistic regression, decision trees (C4.5), the nearest neighbors (kNN) and naïve bayes. The stability of feature selection techniques is an important issue to have consistent results among different techniques. If a technique produces a different subset for different datasets, then that technique becomes unreliable for feature selection [13]. Feature subset stability requires a similarity measure for feature subset. There are three types of feature stability measures [41]. In the first type, a weight or score is assigned to each feature indicating its importance. The second type ranks are assigned to features. The third type consists of sets of selected features in which no weighting or ranking is considered. In this work, stability is measured by considering the overlap between two subsets of features using a straightforward adaptation of the Tanimoto distance measure as follows.

$$S_s\left(s,\ s'\right) = 1 - \frac{|s| + \left|s'\right| - 2\left|s \cap s'\right|}{|s| + |s'| - |s \cap s'|}$$

The Tanimoto distance metric measures the amount of overlap between two sets (s and s'). S_s takes values in [0, 1] with 0 meaning that there is no overlap between the two sets, and 1 that the two sets are identical.

3.6 Classification models

There are many classification techniques that can be used. However, we limit our work to four techniques only: Naïve Bayes (NB), logistic regression (LR), the nearest neighbors (kNN), and C4.5 decision trees. All selected classifiers are commonly used in the field of Software Engineering. Weka is used to train and test these classifiers, and the default settings of these learners are used [29]. The LR model is a regression model that is suitable for a binary class (faulty or not faulty). LR is a statistical technique that was widely used to build fault-proneness models in many studies including [33,43]. Naïve Bayes (NB) algorithm is a commonly used classifier in the software defect prediction and is used as a classifier for defect prediction in many studies including

[12,46]. NB is intuitive and simple to build and can be viewed as a simple Bayesian network that has two assumptions: the attributes (metrics) are independent given the class (faulty or faultless) and no hidden or underlying attributes affect the prediction [34]. The nearest neighbor (kNN) algorithm uses distance (similarity) metrics to assign the dominant label of the closest group of k objects in the training set [1]. The K is usually selected to be an odd value (1, 3, 5, 7, etc.), and in this research, an arbitrary $k = 5$ is selected. The selection of the best k is not an objective of this research. The nearest neighbors were used as a classifier in fault-proneness models in many previous papers [25,58,63]. 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-based criteria (information gain) to build decision trees [50]. The tree grows by selecting the decision for the attribute with the highest information gain. C4.5 is used as a classifier in fault-proneness models in many previous papers [47,51]. The four classification models are trained and tested using tenfold cross-validation.

4 Results and analysis

4.1 ROC Analysis and Identification of Threshold Values

In this section, we discuss the results of the ROC analysis and the significance of the curves in classifying software classes into faulty or not. In addition, for each metric, we identify the possible threshold values. For each metric, we conduct the two-sided t-test to find the significance of the difference between a curve and the random curve (AUC = 0.50) at the 95% significance level. For the curves that are significantly different from random guessing, we identify threshold values based on both sensitivity (i.e., represents benefits) and specificity (i.e., represents costs). A visual assessment of the relationship between both measurements is shown in Fig. 2. In the following, we present the threshold values that have the lowest distance from the optimal point (0, 1). We provide a comprehensive analysis of ROC; however, due to limited space we do not draw all charts as shown in Fig. 2 for every metric; rather, we provide the candidate thresholds in a tabular format. At the end of this section, we use the selected thresholds to classify modules into two groups: faulty and not faulty.

4.1.1 WMC metric

The results of the ROC analysis for the WMC metric in all releases are shown in Table 4 for the curves that are significantly different from the random classifier. The AUC is calculated, and the p value shows whether there is a significant difference from a random classifier (AUC = 0.50). If the

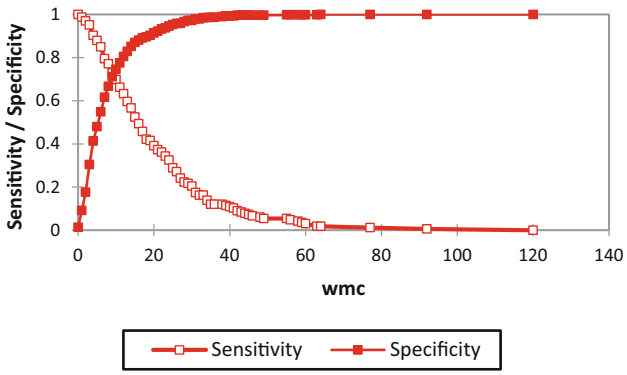

Fig. 2 The sensitivity versus specificity for WMC metric in Ant1.7

p value is lower than the significance level ($\alpha = 0.05$), we reject the null hypothesis H_{01}.

To identify a practical threshold value, the performance of the AUC values should be larger than 0.70. We use the Euclidean distance from the optimal point on ROC curve (i.e., the distance from 0, 1) to find optimal threshold values. The selected thresholds also should have a likelihood ratio larger than one which is satisfied for all systems. Therefore, we get threshold values that fall in the range 6–11.

Thresholds can be used in profiling software into two levels: the low-risk modules and the high-risk modules. The modules in high-risk group require more quality investigations such as looking for potential refactorings, or code improvements. McCabe suggested a threshold for WMC = 10 [45]. When the complexity of a given module exceeds 10, the likelihood of the code being unreliable is much higher. Shatnawi et al. [56] found a threshold WMC = 9 for a large open-source system using ROC analysis. This particular threshold falls in the range that was reported in this work. However, fewer constraints were applied to find a threshold in Shatnawi et al. [56] work. Previously, Rosenberg preferred another possible threshold value at WMC = 20 or WMC = 40 that can be acceptable as well [52], although they found most classes have values less than 20. The works of McCabe and Rosenberg are either anecdotal or based on histogram analysis, which assumes normality of data. The assumption of normality for OO metrics is not valid as shown in the descriptive analysis of all releases under investigation. In addition, thresholds are derived in this work based on the relationship with faults in the system.

4.1.2 CBO metric

The results of the ROC for the CBO metric are shown in Table 5. The p value are lower than the significance level except only for one system (jEdit 4.3). Therefore, we should reject the null hypothesis H_{01}. The AUC values should be larger than 0.70 to consider it practical to identify a threshold

Table 4 The ROC analysis for WMC ordered by AUC

System	Threshold	AUC	p value (two-tailed)	LB (95%)	UB (95%)	Sen	Spec	D	LR
Ant1.5	**8**	**0.82**	<**0.0001**	0.76	0.88	0.88	0.66	0.37	2.54
Ant1.7	**8**	**0.80**	<**0.0001**	0.76	0.85	0.75	0.75	0.36	3.00
jEdit4.2	**10**	**0.79**	<**0.0001**	0.76	0.82	0.70	0.75	0.39	2.75
jEdit4.1	**10**	**0.79**	<**0.0001**	0.74	0.83	0.70	0.81	0.36	3.60
Ant1.6	**8**	**0.78**	<**0.0001**	0.70	0.85	0.81	0.69	0.36	2.62
jEdit4.0	**11**	**0.75**	<**0.0001**	0.70	0.81	0.63	0.86	0.40	4.39
Lucene2.0	**8**	**0.72**	<**0.0001**	0.67	0.78	0.64	0.75	0.44	2.55
Synapse1.0	**6**	**0.72**	<**0.0001**	0.61	0.82	0.69	0.65	0.47	1.98
camel1.4	**7**	**0.71**	<**0.0001**	0.70	0.72	0.63	0.69	0.48	2.01
Synapse1.2	5	0.69	<**0.0001**	0.68	0.71	0.65	0.71	0.45	2.26
Lucene2.4	5	0.68	<**0.0001**	0.68	0.69	0.75	0.54	0.52	1.63
Lucene2.2	6	0.66	<**0.0001**	0.62	0.69	0.63	0.61	0.54	1.61
Synapse1.1	7	0.63	<**0.0001**	0.59	0.67	0.52	0.77	0.53	2.26
camel1.6	8	0.62	<**0.0001**	0.58	0.65	0.48	0.72	0.59	1.69
camel1.2	5	0.58	**0.00**	0.53	0.62	0.54	0.54	0.65	1.18
Ant1.4	*	0.53	0.55	0.44	0.62	*	*	*	*
jEdit4.3	*	0.46	0.68	0.28	0.64	*	*	*	*

The p values in bold are significantly different from a random classifier
* The results are not reported when the p value >0.05

Table 5 The ROC analysis for CBO ordered by AUC

System	Threshold	AUC	p value (two-tailed)	LB (95%)	UB (95%)	Sen	Spec	D	LR
jEdit4.2	**11**	**0.79**	<**0.0001**	0.74	0.85	0.71	0.70	**0.42**	2.35
Synapse1.1	**11**	**0.75**	<**0.0001**	0.69	0.80	0.67	0.74	**0.42**	2.57
Ant1.7	**7**	**0.73**	<**0.0001**	0.72	0.74	0.72	0.66	0.44	2.12
Ant1.6	**7**	**0.73**	<**0.0001**	0.68	0.78	0.75	0.65	0.43	2.16
Ant1.5	**7**	**0.72**	<**0.0001**	0.63	0.80	0.81	0.61	0.43	2.10
jEdit4.0	**8**	**0.71**	<**0.0001**	0.66	0.77	0.68	0.64	0.48	1.89
jEdit4.1	**10**	**0.71**	<**0.0001**	0.65	0.76	0.59	0.72	0.49	2.10
Lucene2.4	**6**	**0.70**	<**0.0001**	0.68	0.73	0.68	0.62	0.49	1.80
Synapse1.2	10	0.69	<**0.0001**	0.63	0.74	0.65	0.59	0.54	1.60
Synapse1.0	12	0.69	**0.01**	0.55	0.82	0.63	0.64	0.52	1.73
Lucene2.0	5	0.68	<**0.0001**	0.63	0.73	0.76	0.54	0.52	1.64
camel1.4	7	0.65	<**0.0001**	0.65	0.66	0.65	0.59	0.54	1.57
camel1.6	7	0.65	<**0.0001**	0.63	0.67	0.62	0.57	0.57	1.45
Lucene2.2	6	0.64	<**0.0001**	0.59	0.68	0.59	0.63	0.55	1.60
jEdit4.3	*	0.63	0.14	0.46	0.81	*	*	*	*
Ant1.4	6	0.61	**0.00**	0.54	0.68	0.75	0.51	0.55	1.54
camel1.2	6	0.57	<**0.0001**	0.54	0.60	0.56	0.57	0.62	1.28

The p values in bold are significantly different from a random classifier
* The results are not reported when the p value >0.05

value. Therefore, we get threshold values that fall in the range 6–11.

Again, the selected thresholds for the CBO can be used in profiling software modules into two levels, the modules with low coupling and the high coupling. McCabe suggested a threshold for CBO = 6 [45], which is more conservative than ours. When a module is coupled to more than six modules, the module can be identified as more fault-prone. Shatnawi et al. [56] found a threshold at CBO = 13 for a large open-source system using ROC analysis, which falls out of the range. Previously, Rosenberg preferred another possible threshold value for CBO = 5 [52].

Table 6 The ROC analysis for NOC

System	AUC	p value (two-tailed)	System	AUC	p value (two-tailed)
Ant1.4	0.54	0.930	jEdit4.2	0.51	0.974
Ant1.5	0.54	0.911	jEdit4.3	0.53	0.940
Ant1.6	0.53	0.936	Lucene2.0	0.48	0.940
Ant1.7	0.54	0.899	Lucene2.2	0.52	0.950
camel1.2	0.53	0.940	Lucene2.4	0.53	0.930
camel1.4	0.54	0.915	Synapse1.0	0.47	0.950
camel1.6	0.56	0.877	Synapse1.1	0.47	0.950
jEdit4.0	0.49	0.980	Synapse1.2	0.51	0.990
jEdit4.1	0.50	0.990			

Table 7 The ROC analysis for DIT

System	AUC	p value (two-tailed)	System	AUC	p value (two-tailed)
Ant1.4	0.60	0.46	jEdit4.2	0.50	0.97
Ant1.5	0.62	0.25	jEdit4.3	0.59	0.40
Ant1.6	0.48	0.89	Lucene2.0	0.48	0.90
Ant1.7	0.53	0.79	Lucene2.2	0.47	0.88
camel1.2	0.49	0.98	Lucene2.4	0.56	0.72
camel1.4	0.56	0.70	Synapse1.0	0.48	0.91
camel1.6	0.53	0.85	Synapse1.1	0.37	0.57
jEdit4.0	0.50	0.96	Synapse1.2	0.53	0.86
jEdit4.1	0.48	0.89			

From this discussion, we do not reach a consensus on a particular threshold value for the CBO metric. In our work, we found thresholds in the range 6–11. Only the work of McCabe suggested a threshold that falls in the range reported in this work.

4.1.3 NOC metric

The results of the ROC analysis for the NOC metric are shown in Table 6. As the p value are larger than the significance level, we should accept the null hypothesis H_{01} and reject the alternative hypothesis H_{a1}. Therefore, all ROC curves that were produced for NOC cannot be used to classify software classes into faulty and not faulty and threshold values cannot be identified for the NOC metric. The work of McCabe and Rosenberg did not suggest any thresholds for the NOC [45,52]. In addition, Shatnawi et al. [56] could not report a threshold for NOC using ROC analysis. These results are consistent, and therefore, we can conclude that there are no thresholds that can be identified for the NOC metric.

4.1.4 DIT metric

The results of the ROC analysis for the DIT metric are shown in Table 7. As the p value are larger than the significance level, we should accept the null hypothesis H_{01} and reject the alternative hypothesis H_{a1}. Therefore, all ROC curves that were produced from DIT metric cannot be used to classify software classes into faulty and not faulty and threshold values cannot be identified for the DIT metric. Therefore, ROC analysis is not suitable to identify such thresholds. ROC analysis can be used to define many thresholds using ordinal variable (non-binary coding of faults), such as using the 3-level severity of faults. However, the results are expected to have monotonic behavior, which is not expected for the DIT. For example, Rosenberg et al. [52] defined a threshold as follows $2 < \text{DIT} < 5$. The modules of $\text{DIT} < 2$ may represent poor exploitation of inheritance, whereas modules of $\text{DIT} > 5$ have larger complexity. McCabe [45] defined thresholds as $2 < \text{DIT} < 6$, i.e., $\text{DIT} > 6$ increases the testing effort and $\text{DIT} < 2$ indicates a poor exploitation of inheritance.

4.1.5 RFC metric

The results of the ROC analysis for the RFC metric are shown in Table 8. The p value is lower than the significance level $\alpha = 0.05$, and we should reject the null hypothesis H_{01}. The AUC values are significantly different from the random classifier for most releases except in two releases (Ant 1.4 and jEdit 4.3). The thresholds that are identified for the RFC metric show a wide range (15–40).

Again, the selected threshold for the RFC can be used in profiling software modules into two levels: the modules with low responsibilities and high responsibilities. McCabe

Table 8 The ROC analysis for RFC ordered by AUC

System	Threshold	AUC	p value (two-tailed)	LB (95%)	UB (95%)	Sen	Spec	D	LR
jEdit4.2	**37**	**0.84**	<**0.0001**	0.79	0.90	0.88	0.75	**0.280**	3.49
Ant1.6	**31**	**0.84**	<**0.0001**	0.80	0.89	0.79	0.78	0.305	3.54
Ant1.7	**32**	**0.83**	<**0.0001**	0.79	0.86	0.79	0.75	0.326	3.17
jEdit4.1	**31**	**0.83**	<**0.0001**	0.78	0.88	0.73	0.75	0.364	2.95
Ant1.5	**40**	**0.83**	<**0.0001**	0.75	0.91	0.69	0.84	0.353	4.17
Synapse1.0	**35**	**0.81**	<**0.0001**	0.69	0.93	0.81	0.78	0.289	3.70
jEdit4.0	**32**	**0.78**	<**0.0001**	0.72	0.84	0.65	0.77	0.418	2.79
Synapse1.2	**29**	**0.76**	<**0.0001**	0.70	0.82	0.66	0.79	0.398	3.13
Lucene2.0	**15**	**0.76**	<**0.0001**	0.69	0.82	0.76	0.63	0.438	2.08
Lucene2.4	**15**	**0.71**	<**0.0001**	0.66	0.76	0.68	0.64	0.477	1.91
camel1.4	**20**	**0.70**	<**0.0001**	0.66	0.74	0.61	0.69	0.497	1.97
Synapse1.1	31	0.69	<**0.0001**	0.61	0.77	0.58	0.78	0.472	2.63
Lucene2.2	17	0.65	<**0.0001**	0.59	0.72	0.60	0.62	0.553	1.58
camel1.6	17	0.63	<**0.0001**	0.59	0.67	0.56	0.62	0.581	1.48
Ant1.4	*	0.58	0.099	0.48	0.68	*	*	*	*
camel1.2	13	0.57	**0.001**	0.53	0.61	0.58	0.53	0.631	1.23
jEdit4.3	*	0.51	0.940	0.33	0.68	*	*	*	*

The p values in bold are significantly different from a random classifier
* The results are not reported when the p value >0.05

Table 9 The ROC analysis for LCOM ordered by AUC

System	Threshold	AUC	p value (two-tailed)	LB (95%)	UB (95%)	Sen	Spec	D	LR
jEdit4.2	**10**	**0.81**	<**0.0001**	0.75	0.88	0.88	0.63	0.39	2.37
Ant1.7	**14**	**0.77**	<**0.0001**	0.75	0.80	0.75	0.69	0.39	2.45
Ant1.6	**16**	**0.77**	<**0.0001**	0.74	0.80	0.68	0.78	0.39	3.06
jEdit4.1	**21**	**0.77**	<**0.0001**	0.71	0.82	0.66	0.81	0.39	3.41
Ant1.5	**10**	**0.74**	<**0.0001**	0.66	0.82	0.78	0.68	**0.39**	2.46
jEdit4.0	**20**	**0.71**	<**0.0001**	0.66	0.76	0.64	0.80	0.41	3.21
Synapse1.0	**26**	**0.71**	**0.00**	0.57	0.84	0.56	0.83	0.47	3.30
camel1.4	9	0.67	<**0.0001**	0.63	0.71	0.63	0.67	0.50	1.90
Synapse1.1	10	0.65	<**0.0001**	0.59	0.70	0.52	0.80	0.52	2.62
Synapse1.2	6	0.62	<**0.0001**	0.62	0.63	0.52	0.78	0.53	2.34
Lucene2.2	*	0.60	0.26	0.43	0.77	*	*	*	*
camel1.2	4	0.58	**0.03**	0.51	0.66	0.55	0.58	0.61	1.32
Lucene2.4	*	0.58	0.24	0.45	0.70	*	*	*	*
camel1.6	*	0.56	0.10	0.49	0.64	*	*	*	*
Ant1.4	*	0.55	0.08	0.49	0.61	*	*	*	*
Lucene2.0	*	0.53	0.70	0.38	0.69	*	*	*	*
jEdit4.3	*	0.48	0.81	0.34	0.63	*	*	*	*

The p values in bold are significantly different from a random classifier
* The results are not reported when the p value >0.05

suggested a threshold for RFC = 40 [45]. Shatnawi et al. [56] found a threshold RFC = 44 for Eclipse using ROC analysis. Previously, Rosenberg preferred another possible threshold value for CBO = 50 [52]. These thresholds fall out of the range reported in this work.

4.1.6 LCOM metric

The results of the ROC analysis for the LCOM metric are shown in Table 9. The LCOM has different trends than all other metrics. We notice that the p value is not consistent

Table 10 The application of thresholds on Ant1.5

Percentage of god classes	Number of god classes	Faulty god classes	% Faulty god classes
17.5%	51	21	41%

Table 11 The application of thresholds on jEdit4.2

Percentage of god classes	Number of god classes	Faulty god classes	% Faulty god classes
22.5%	82	37	45%

Table 12 Wilcoxon signed rank tests for H_{02}

	DataSets	WMC p value	CBO P value	RFC P values	Results
Wilcoxon signed rank test on AUC values	Original-SMOTE	0.969	0.735	0.821	Cannot reject H_{02}
	Original undersampling	0.806	0.851	0.939	Cannot reject H02
Wilcoxon signed rank test on threshold values	Original-SMOTE	0.231	0.796	0.499	Cannot reject H02
	Original undersampling	0.112	0.722	0.686	Cannot reject H02

for all systems. Out of seventeen releases, we found eleven releases that have p value lower than the significance level, whereas six releases have p value larger than alpha. Therefore, we should reject the null hypothesis H_{01} for the 11 releases only. Again, a threshold is practical when the AUC values are larger than 0.70. The thresholds that are identified for the LCOM metric shows a wide range (10–26).

Shatnawi et al. [56] could not find a threshold for LCOM using ROC analysis. Previously, Rosenberg did not report any preferred thresholds for the LCOM [52]. McCabe suggested a threshold for LCOM = 75% [45] but for a different definition of the LCOM metric. From this work and the previously reported results on LCOM thresholds, we can conclude that the LCOM metric was not significantly associated with fault-proneness [4,48].

4.1.7 ROC application in god classes identification

We study the application of the identified threshold values in profiling software modules into low- and high-risk groups. Software verification and validation is a lengthy process, and it should be cost-effective. We expect to identify a small proportion of modules that have a large percentage of faults. A threshold value separates modules into two groups: low and high risk. The first group includes the modules with values less than the threshold; otherwise, modules are placed in the second group.

Threshold values are used to identify god classes in code. A god class or large class is one of the important code bad smells reported in Fowler [24]. A god class has large number of responsibilities, complexities and interconnections. The classes that exceed the threshold values are identified as god classes. In this section, we report only the results of the god class analysis on two systems: Ant 1.5 and JEdit 4.2, as shown in Tables 10 and 11. The number of god classes identified is 51 and 82 in Ant 1.5 and JEdit 4.2, respectively. The engineers can choose a proportion of these classes for manual inspection or refactoring. Among the god classes, a large percentage of classes have faults (41 and 45%). The god classes are more fault-prone than other classes, and a large portion already have faults. These results confirm the relationship between faults and god classes.

4.2 ROC analysis after sampling

In this section, we conduct an experiment to find the effect of two sampling techniques: SMOTE and undersampling, on ROC analysis. The systems under investigation have imbalanced fault distributions as shown in Table 2.

We conducted the ROC analysis for all systems under investigation in three scenarios: original data without sampling, data after SMOTE sampling and data after undersampling. Both the AUC and threshold values after data sampling are calculated but are not shown for brevity. To test the null hypothesis (H_{02}), we conducted a pairwise Wilcoxon signed rank test to find the significance of the differences in AUC values for two pairs: without sampling against SMOTE and without sampling against undersampling. The results of the statistical tests (p values) are shown in Table 12 for only three metrics. The results of the statistical analysis for WMC, CBO and RFC do not show significant differences between the two groups. Therefore, we can conclude that sampling does not have significant effect on the ROC analysis, i.e., the null hypothesis (H_{02}) is accepted. The AUC values are not statistically different. We also tested the significance of the differences in threshold values using Wilcoxon signed rank test as shown in the second part of Table 12. The test results show no statistically significant differences between threshold values before and after sampling. These results show that the ROC is robust under imbalanced data; on the other hand,

many studies reported the effect of imbalanced data on fault prediction models. Hall et al. [32] found that fault prediction models using C4.5 underperform for imbalanced data and recommended not to use imbalanced data. Some other researchers, Wang et al. [59] and Yu et al. [62], found similar results to the Hall et al. study and concluded that C4.5 outcomes are unstable on imbalanced datasets. Yan et al. [61] performed fuzzy logic and rules to overcome the imbalance effects on support vector machines. Agrawal and Menzies [3] introduced a tuned SMOTE technique and found improvements on classifiers such as decision trees, logistic regression, K-means, naïve bayes and support vector machines. The authors have recommended that any prior study which did not study the effects of data pre-processing needs to be analyzed again. However, this work presents evidence of robustness in using ROC in identifying threshold values under imbalanced data conditions. The threshold values are not significantly different before and after sampling.

4.3 Feature selection using ROC

In this section, we train and test fault-proneness models using four learners: logistic regression (LR), naïve Bayes (NB), the nearest neighbors (5NN) and C4.5 decision trees. These learners are used to validate the impact of using the ROC analysis as an alternative attribute (metrics) selection. The metrics that have thresholds values resulted from ROC analysis in Sect. 4.1 are only selected to build models. A metric is included in a fault-proneness model if the AUC value is significantly different from the random guessing and AUC $\geq$ 0.70. These models are then compared with well-known feature selection techniques that performs a stepwise search throughout the space of attribute subsets.

First, the results of including all metrics in building classifiers are summarized in Table 13. The AUC values resulted from the application of the four classifiers in the Weka data mining tool. The default settings of the classifiers in Weka are conducted with tenfold cross-validation, and the results are repeated ten times. The results do not show acceptable results (AUC $\geq$ 0.70) for all releases. In summary, the four classifiers showed acceptable results for most releases. The LR and 5NN models showed the best classification performance, while the models of C4.5 showed the least performance among all.

Second, the metrics selected via the ROC analysis and the feature selection are shown in the first column of Table 14. The metrics that are listed in column two are selected based on the significance of ROC from random guessing (AUC = 0.5). For example, the CBO metric is the only metric that is selected in Ant1.4. Therefore, the classifiers are trained and tested using the CBO metric only. The metrics that are selected using other selection techniques are shown in Table 14 as well. For example, the CBO metric is the only metric included in the first model. Several combinations of metric subsets are selected in different datasets. The inheritance metrics (DIT and/or NOC) are not selected in ROC selection. In contrast, the stepwise selection has eleven distinct combinations. Chi-squared and information gain selection resulted in the same results for all datasets, and each produced eight different subsets.

Table 13 The area under curve (AUC) values when all metrics are included

All metrics	LR	NB	5NN	C4.5
Ant1.4	0.55	0.61	0.59	0.49
Ant1.5	**0.84**	**0.77**	**0.80**	0.64
Ant1.6	**0.84**	**0.81**	**0.80**	**0.74**
Ant1.7	**0.83**	**0.79**	**0.76**	**0.74**
Camel 1.2	0.57	0.56	0.64	0.52
Camel 1.4	**0.70**	0.67	0.67	0.60
Camel 1.6	0.65	0.59	0.66	0.54
jEdit4.0	**0.77**	**0.70**	**0.81**	**0.72**
jEdit4.1	**0.82**	**0.75**	**0.80**	0.69
jEdit4.2	**0.84**	**0.75**	**0.77**	0.64
Lucene2.0	**0.77**	**0.75**	**0.70**	0.67
Lucene2.2	0.62	0.61	**0.70**	0.58
Lucene2.4	**0.75**	0.69	**0.73**	0.68
Synapse1.0	**0.81**	**0.71**	**0.75**	0.53
Synapse1.1	**0.72**	**0.75**	**0.77**	0.66
Synapse1.2	**0.75**	**0.71**	**0.73**	**0.71**

The p values in bold are significantly different from a random classifier

To test the significance of the differences in the AUC values among the models, we conducted a pairwise Wilcoxon signed rank test at the 95% confidence level and the results are presented in Table 15. There are no significant differences between the two pairs of models. However, the ROC analysis is more consistent in selecting the same metrics in various datasets than the feature selection techniques. The p values do not show significant differences between models for the four classifiers except the logistic regression, which shows significant differences from the forward stepwise models. Therefore, we could not find differences in the model's performance and we reject the null hypothesis (H_{03}).

The stability of feature selection is measured to find the overlap in selected subsets. If a technique produces a different subset for different datasets, then that technique becomes unreliable for feature selection. Table 16 shows the results of the Tanimoto distance metric. The stable technique has value close to 1. The ROC analysis provides more stable and consistent selection subsets than the three feature selection techniques. The reduction in the number of metrics reduces the efforts to collect more metrics when less number is sufficient. The ROC analysis selects at most four metrics, while other techniques select up to six metrics in some models.

Table 14 Metrics selection via ROC and stepwise selection procedures

	ROC selection	Forward stepwise selection	Chi-squared selection	Info. gain selection
Ant1.4	CBO	CBO	CBO	CBO
Ant1.5	WMC, CBO, RFC, LCOM	RFC	RFC	RFC
Ant1.6	WMC, CBO, RFC, LCOM	WMC, CBO, RFC, LCOM	WMC, CBO, RFC, LCOM	WMC, CBO, RFC, LCOM
Ant1.7	WMC, CBO, RFC, LCOM	CBO, RFC, LCOM	RFC	RFC
Camel 1.2	WMC, CBO, RFC, LCOM	DIT, NOC	DIT, NOC	DIT, NOC
Camel 1.4	WMC, CBO, RFC, LCOM	WMC, DIT, CBO, RFC, LCOM	WMC, DIT, CBO, RFC, LCOM	WMC, DIT, CBO, RFC, LCOM
Camel 1.6	WMC, CBO, RFC	DIT, NOC, CBO, RFC, LCOM	WMC, DIT, NOC, CBO, RFC, LCOM	WMC, DIT, NOC, CBO, RFC, LCOM
jEdit4.0	WMC, CBO, RFC, LCOM	WMC, RFC, LCOM	WMC, RFC, LCOM	WMC, RFC, LCOM
jEdit4.1	WMC, CBO, RFC, LCOM	WMC, RFC, LCOM	WMC, RFC, LCOM	WMC, RFC, LCOM
jEdit4.2	WMC, CBO, RFC, LCOM	WMC, CBO, RFC, LCOM	WMC, CBO, RFC, LCOM	WMC, CBO, RFC, LCOM
Lucene2.0	WMC, CBO, RFC	WMC, DIT, CBO, RFC, LCOM	LCOM	LCOM
Lucene2.2	WMC, CBO, RFC	WMC, DIT, CBO, RFC, LCOM	WMC, NOC CBO, RFC, LCOM	WMC, NOC CBO, RFC, LCOM
Lucene2.4	WMC, CBO, RFC	WMC, NOC, CBO, RFC, LCOM	WMC, DIT, CBO, RFC, LCOM	WMC, DIT, CBO, RFC, LCOM
Synapse1.0	WMC, CBO, RFC, LCOM	RFC	WMC, NOC, CBO, RFC, LCOM	WMC, NOC, CBO, RFC, LCOM
Synapse1.1	WMC, CBO, RFC, LCOM	DIT, CBO, RFC, LCOM	RFC	RFC
Synapse1.2	WMC, CBO, RFC, LCOM	WMC, CBO, RFC	WMC, DIT, CBO, RFC, LCOM	WMC, DIT, CBO, RFC, LCOM

Table 15 The Wilcoxon signed rank tests for ROC versus three selection techniques

Model	ROC versus all metrics	ROC versus forward selection	ROC versus Chi-squared	ROC versus IG
Logistic regression	0.33	**0.001**	0.731	0.731
Naïve Bayes	0.387	0.798	**0.03**	**0.03**
5 Nearest neighbors	0.028	0.504	0.195	0.195
C4.5 trees	0.48	0.231	0.221	0.221

The p values in bold are significantly different from a random classifier

Table 16 Subset overlap measurement

	ROC	Forward	Chi-squared	IG
Stability	0.82	0.46	0.41	0.40

5 Limitations and threats to validity

With regard to internal threats, from the viewpoint of the application of the results, different interpretations of the software metrics represent a threat to the validity of the study. The definition of a threshold value might not be valid for every metric, and more investigation is needed. For example, the DIT metric may need a more detailed definition. As previous works show, the DIT metric might need two or more break points, e.g., a bad module might have a DIT < 3 or a DIT > 5, whereas the rest of the modules are regular. We do not claim the collection of the metrics and faults. It is possible that there are mistakes in the fault identification. Faults were identified from the comments in the source code version control system, which are not always well written [36]. The problem with the use of confusion matrix and the ROC in evaluating the fault-proneness of software modules is that they are designed to apply to all classification problems and they do not clearly and directly relate to the cost effectiveness of using fault-proneness models [2].

With regard to external threats, this study considers open-source systems developed in Java and might not be generalized to systems written in other languages. The study is conducted on five systems only, and further investigation is still needed for more software systems. In addition, these systems are representative of open-source systems and might

not be representative of commercial products. However, the systems under investigation are well known and widely used. Four systems are licensed as Apache products and are used as third party in commercial products. This study does not provide a comprehensive investigation of different sampling and feature selection techniques. In addition, only four learners were reported in the study, although there are more to investigate. However, the purpose is not to be comprehensive, but as a basis for comparison only.

6 Conclusion

Using ROC analysis, we identified threshold values for the CK metrics to aid software engineers to identify risky classes. CK metrics are widely validated as predictors of software fault-proneness. The ROC analysis is used to diagnose the relationship between software metrics and faults in five open-source systems. The results of the ROC analysis on five systems showed significant relationships between four metrics (WMC, CBO, RFC and LCOM) and faults on most releases. For each metric, we identified threshold values via a statistical test on the significance of the curve. The identified thresholds are not consistent for all releases under investigation. To validate the consistency of the ROC analysis, we tested the effect of two sampling techniques (oversampling using SMOTE and undersampling) on the area under the ROC curve. The results of oversampling and undersampling the data are not significantly different from the ROC analysis when conducted on the data without sampling. These results confirm our findings in Shatnawi et al. [56] and provide more evidence of the robustness of the ROC in case of data imbalance.

The ROC analysis is used to select metrics for inclusion in fault-proneness models. The metrics that have an area under the curve that can be considered acceptable, larger than 0.70, are selected to assess quality further. The selected metrics are then used to build prediction models using four machine learning techniques: logistic regression, naïve Bayes, the nearest neighbors and C4.5 decision trees. The performance of classifiers using the metrics that were selected via the ROC analysis does not show significant differences from other models that are built from all metrics and via stepwise feature selection techniques. However, the ROC selection is more reliable and consistent in selecting the same metrics than three feature selection techniques. Therefore, we conclude that the ROC analysis can be used to identify which metrics are strongly related to fault-proneness and to identify plausible threshold values. Furthermore, the ROC analysis as proposed in this work can be used to derive thresholds of other software metrics.

For the future work, we plan to investigate other techniques for threshold identification and to use thresholds in applications of SQA. We plan to study a larger number of metrics using many sampling and feature selection techniques to find which technique produces best results. In addition, we plan to study the effect of data transformation on the effectiveness of software metrics in building fault-proneness models.

References

1. Aha D, Kibler D (1991) Instance-based learning algorithms. Mach Learn 6(1):37–66
2. Arisholm E, Briand L, Johannessen E (2010) A systematic and comprehensive investigation of methods to build and evaluate fault prediction models. J Syst Softw 83(1):2–17
3. Agrawal A, Menzies T (2017) "Better Data" is better than "Better Data Miners", arXiv:1705.03697 [cs.SE]
4. Basili V, Briand L, Melo W (1996) A validation of object-oriented design metrics as quality indicators. IEEE Trans Softw Eng 22(10):751–761
5. Bender R (1999) Quantitative risk assessment in epidemiological studies investigating threshold effects. Biom J 41(3):305–319
6. Benlarbi S, El Emam K, Goel N, Rai S (2000) Thresholds for object-oriented measures. In: 11th International symposium on software reliability engineering (ISSRE 2000). IEEE Computer Society, Los Alamitos, CA, pp 24–38
7. Briand LC, Wü st J, Daly JW, Victor Porter D (2000) Exploring the relationships between design measures and software quality in object-oriented systems. J Syst Softw 51(3):245–273
8. Cartwright M (1998) An empirical view of inheritance. Inf Softw Technol 40:795–799
9. Catal C, Diri B (2008) A Fault prediction model with limited fault data to improve test process. In: PROFES 2008, LNCS 5089, pp 244–257
10. Catal C (2011) Software fault prediction: a literature review and current trends. Expert Syst Appl 38:4626–4636
11. Catal C, Alan O, Balkan K (2011) Class noise detection based on software metrics and ROC curves. Inf Sci 181(21):4867–4877
12. Challagulla VU, Bastani FB, Yen I, Paul RA (2005) Empirical assessment of machine learning based software defect prediction techniques. In: Tenth IEEE international workshop on object-oriented real-time dependable systems, pp 263–270
13. Chandrashekar G, Sahin F (2014) A survey on feature selection methods. Comput Electr Eng 40(1):16–28
14. Chawla N, Bowyer K, Hall L, Kegelmeyer W (2002) SMOTE, synthetic minority over-sampling technique. J Artif Intell Res 16:321–357
15. Chidamber S, Kemerer C (1994) A metrics suite for object oriented design. IEEE Trans Softw Eng 20(6):476–493
16. Daly J, Brooks A, Miller J, Roper M, Wood M (1996) Evaluating inheritance depth on the maintainability of object-oriented software. Empir Softw Eng 1(2):109–132
17. Dessi N, Pes B (2015) Similarity of feature selection methods: an empirical study across data intensive classification tasks. Expert Syst Appl 42(10):4632–4642
18. El Emam KE, Benlarbi S, Goel N, Rai SN (2001a) The confounding effect of class size on the validity of object-oriented metrics. IEEE Trans Softw Eng 27(7):630–648
19. El Emam KE, Melo W, Machado J (2001b) The prediction of faulty classes using object-oriented design metrics. J Syst Softw 56:63–75
20. El Emam K, Benlarbi S, Goel N, Melo W, Lounis H, Rai S (2002) The optimal class size for object-oriented software. IEEE Trans Softw Eng 28(5):494–509

21. Erni K, Lewerentz C (1996) Applying design-metrics to object-oriented frameworks. In: Proceedings of the third international software metrics symposium. Society Press, pp 25–26
22. Fawcett T (2004) ROC graphs, notes and practical considerations for researchers. Technical report, HP Laboratories, Page Mill Road, Palo Alto, CA
23. Ferreira KAM, Bigonha M, Bigonha R, Mendes L, Almeida H (2012) Identifying thresholds for object-oriented software metrics. J Syst Softw 85:244–257
24. Fowler M, Beck K, Brant J, Opdyke W, Roberts D (1999) Refactoring: improving the design of existing code
25. Gao K, Khoshgoftaar K, Wang H, Seliya N (2011) Choosing software metrics for defect prediction: an investigation on feature selection techniques. Softw Pract Exp 41(5):579–606
26. Gondra I (2008) Applying machine learning to software fault-proneness prediction. J Syst Softw 81(2):186–195
27. Gronback RC (2003) Software remodeling: improving design and implementation quality, using audits, metrics and refactoring in Borland Together ControlCenter, A Borland White Paper
28. Gyimothy T, Ferenc R, Siket I (2005) Empirical validation of object-oriented metrics on open source software for fault prediction. IEEE Trans Softw Eng 31(10):897–910
29. Hall M, Frank E, Holmes G, Pfahringer B, Reutemann P, Witten I (2009) The WEKA data mining software, an update. Spec Interest Group Knowl Discov Data Min Explor Newsl 11(1):10–18
30. Harrison R, Counsell S, Nithi R (2000) Experimental assessment of the effect of inheritance on the maintainability of object-oriented systems. J Syst Softw 52(2):173–179
31. Hosmer D, Lemeshow S (2000) Applied logistic regression, 2nd edn. Wiley, New York
32. Hall T, Beecham S, Bowes D, Gray D, Counsell S (2012) A systematic literature review on fault prediction performance in software engineering. IEEE Trans Softw Eng 38(6):1276–1304
33. Jiang Y, Cukic B, Ma Y (2008) Techniques for evaluating fault prediction models. Empir Softw Eng 13:561–595
34. John G, Langley P (1995) Estimating continuous distributions in Bayesian classifiers. In: Proceedings of the eleventh conference on uncertainty in artificial intelligence. Morgan Kaufmann Publishers, San Mateo, pp 338–345
35. Jureczko M, Madeyski L (2010) Towards identifying software project clusters with regard to defect prediction. In: Proceedings of the 6th international conference on predictive models in software engineering, pp 1–10
36. Jureczko M, Spinellis D (2010) Using object-oriented design metrics to predict software defects. In: Proceedings of the 5th international conference on dependability of computer systems, pp 69–81
37. Khoshgoftaar T, Seliya N (2004) Comparative assessment of software quality classification techniques, an empirical case study. Empir Softw Eng 9(3):229–257
38. Khoshgoftaar TM, Kehan G, Seliya N (2010) Attribute Selection and imbalanced data: problems in software defect prediction. In: Proceedings of the 22nd IEEE international conference on tools with artificial intelligence (ICTAI), pp 137–144
39. Koru AG, El Emam K, Zhang D, Liu H, Mathew D (2008) Theory of relative defect proneness. Empir Softw Eng 13:473–498
40. Kubat M, Matwin S (1997) Addressing the curse of imbalanced training sets: one-sided selection. In: Proceedings of the fourteenth international conference on machine learning, pp 179–186
41. Kalousis A, Prados J, Hilario M (2007) Stability of feature selection algorithms: a study on high-dimensional spaces. Knowl Inf Syst 12(1):95–116
42. Ma Y, Cukic B (2007) Adequate evaluation of quality models in software engineering studies. In: International workshop on predictor models in software engineering, p 1
43. Marcus A, Poshyvanyk D, Ferenc R (2008) Using the conceptual cohesion of classes for fault prediction in object-oriented systems. IEEE Trans Softw Eng 34(2):287–300
44. Marinescu R (2002) Measurement and quality in object-oriented design. Ph.D. thesis, Politehnica University of Timisoara
45. McCabe Software (2012) Using code quality metrics in management of outsourced development and maintenance, white paper. http://www.mccabe.com/pdf/McCabeCodeQualityMetrics-OutsourcedDev.pdf. Accessed Nov 2012
46. Menzies T, DiStefano J, Orrego A, Chapman R (2004) Assessing predictors of software defects. In: Predictive software models workshop
47. Mertik M, Lenic M, Stiglic G, Kokol P (2006) Estimating software quality with advanced data mining techniques. In: International conference on software engineering advances, p 19
48. Olague H, Etzkorn L, Gholston S, Quattlebaum S (2007) 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 33(8):402–419
49. Prechelt L, Unger B, Philippsen M, Tichy W (2003) A controlled experiment on inheritance depth as a cost factor for code maintenance. J Syst Softw 65:115–126
50. Quinlan JR (1993) C4.5, Programs for machine learning. Morgan Kaufmann, San Mateo
51. Riquelme JC, Ruiz R, Rodrí guez D, Moreno J (2008) Finding defective modules from highly unbalanced datasets. Actas del 8° taller sobre el apoyo a la decisió n en ingenierí a del software, pp 67–74
52. Rosenberg LH, Stapko R, Gallo A (1999) Risk-based object oriented testing. In: 24th Annual software engineering workshop, Goddard Space Flight Center
53. Seiffert C, Khoshgoftaar TM, Van Hulse J, Napolitano A (2008) Building useful models from imbalanced data with sampling and boosting. In: Proceedings of the twenty-first international FLAIRS conference, pp 206–311
54. Shatnawi R, Li W (2008) The effectiveness of software metrics in identifying error-prone classes in post-release software evolution process. J Syst Softw 81(11):1868–1882
55. Shatnawi RA (2010) Quantitative investigation of the acceptable risk levels of object-oriented metrics in open-source systems. IEEE Trans Softw Eng 36(2):216–225
56. Shatnawi R, Li W, Swain J, Newman T (2010) Finding software metrics threshold values using ROC curves. J Softw Maint Evol Res Pract 22(1):1–16
57. Van Hulse J, Khoshgoftaar TM, Napolitano A (2007) Experimental perspectives on learning from imbalanced data. In: Proceedings of the 24th international conference on machine learning, Corvallis, OR, pp 935–942
58. Wang H, Khoshgoftaar TM, Seliya N (2011) How many software metrics should be selected for defect prediction? In: Murray RC, McCarthy, PM (eds) FLAIRS conference. AAAI Press
59. Wang S, Yao X (2013) Using class imbalance learning for software defect prediction. IEEE Trans Reliab 62(2):434–443
60. XLStat, Creating an ROC curve and identify the optimal threshold value for a detection method. http://www.xlstat.com/en/learning-center/tutorials/creating-an-roc-curve-and-identify-the-optimal-threshold-value-for-a-detection-method.html. Accessed 8/2/2014
61. Yan Z, Chen X, Guo P (2010) Software defect prediction using fuzzy support vector regression. In: International symposium on neural networks. Springer, Berlin, pp 17–24
62. Yu Q, Jiang S, Zang Y (2017) The performance stability of defect prediction models with class imbalance: an empirical study. IEICE Trans Inf Syst E100(2):265–272

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

64. Zweig M, Campbell G (1993) Receiver-operating characteristic (ROC) plots, a fundamental evaluation tool in clinical medicine. Clinl Chem 39(4):561–577