Expert Systems With Applications 61 (2016) 106–121

Contents lists available at ScienceDirect

Expert Systems With Applications

journal homepage: www.elsevier.com/locate/eswa

Deriving thresholds of software metrics to predict faults on open source software: Replicated case studies

Ömer Faruk Arar[a,*], Kürşat Ayan[b]

[a] TUBITAK BILGEM, Information Technologies Institute, Kocaeli-Turkey
[b] Department of Computer Engineering, Faculty of Computer and Information Science, Sakarya University, Sakarya-Turkey

ARTICLE INFO

Article history:
Received 10 January 2016
Revised 9 April 2016
Accepted 11 May 2016
Available online 11 May 2016

Keywords:
Machine learning
Software quality metrics
Threshold
Software fault prediction
Logistic regression
Bender method

ABSTRACT

Object-oriented metrics aim to exhibit the quality of source code and give insight to it quantitatively. Each metric assesses the code from a different aspect. There is a relationship between the quality level and the risk level of source code. The objective of this paper is to empirically examine whether or not there are effective threshold values for source code metrics. It is targeted to derive generalized thresholds that can be used in different software systems. The relationship between metric thresholds and fault-proneness was investigated empirically in this study by using ten open-source software systems. Three types of fault-proneness were defined for the software modules: non-fault-prone, more-than-one-fault-prone, and more-than-three-fault-prone. Two independent case studies were carried out to derive two different threshold values. A single set was created by merging ten datasets and was used as training data by the model. The learner model was created using logistic regression and the Bender method. Results revealed that some metrics have threshold effects. Seven metrics gave satisfactory results in the first case study. In the second case study, eleven metrics gave satisfactory results. This study makes contributions primarily for use by software developers and testers. Software developers can see classes or modules that require revising; this, consequently, contributes to an increment in quality for these modules and a decrement in their risk level. Testers can identify modules that need more testing effort and can prioritize modules according to their risk levels.

1. Introduction and motivation

Software systems play a significant role in our daily lives and are becoming more common day-by-day. Most machines and services have some type of built-in software. Developers aim to advance such software as quickly as possible in order to expand its everyday use and to stay competitive. Every phase of developing a software project should be operated effectively. Testing is one of the most important phases and one that should be considered carefully. An ineffectively processed testing phase can lead to problems, which may result in a variety of negative outcomes. A bug in the software missed by the test team may be costly during field runs. It can interrupt an ongoing service or it may result in a crash of another connected system. If it is part of a complex whole, for example banking software or the software of a satellite system, then costs associated with disruption may increase dramatically. For example, a $125 million NASA spacecraft was lost in space because of a small data conversion bug (Michaels, 2008). More recently, a newly designed Airbus A400M cargo plane crashed during a test flight and investigators discovered it was caused by faulty engine control software (Flottau & Osborne, 2015). The U.S. Department of Defense spends $4 billion annually because of software failures (Dick, Meeks, Last, Bunke, & Kandel, 2004).

In the last decade, academicians have put great effort into studying automatic fault prediction and early warning systems in order to find an effective method for the testing process. Such a fault prediction system would facilitate the activities of a test team in a software project, by providing a list of software modules categorized as either fault-prone or non-fault-prone. As a result, testers could focus on fault-prone modules rather than non-fault-prone ones. Internal quality features of a software system provide some clues regarding the fault-proneness of related software modules. This relationship between quality and fault-proneness has been confirmed by numerous studies, which will be referenced in the next section. Software metrics aim to reflect the internal quality of software systems. Different metric suites have been proposed. Some of the more widely studied metrics suites are McCabe (McCabe, 1976), Halstead (Halstead, 1977), and Chidamber and

* Corresponding author. Information Technologies Institute of TUBITAK, BILGEM Gebze, Kocaeli, Turkey. Fax: +90 262 646 31 87.
E-mail addresses: omer.arar@tubitak.gov.tr, omer.arar@ogr.sakarya.edu.tr (Ö.F. Arar), kayan@sakarya.edu.tr (K. Ayan).

http://dx.doi.org/10.1016/j.eswa.2016.05.018
0957-4174/

Kemerer (CK) (Chidamber & Kemerer, 1994). McCabe's and Halstead's metrics represent quality at a method-level, while CK metrics represent quality at class-level. Different metric groups assess software from different aspects. Deriving the threshold values of these metrics would contribute to project developers, testers, and customers. Developers could focus on threshold-exceeding modules before releasing a software version. Customers, on the other hand—who generally do not have enough software knowledge to judge the quality of the source code—could form an opinion about that quality with the aid of such threshold values. Threshold values are used to categorize an instance into two groupings so that each sample is acted upon according to its category. Henderson-Sellers (1996) highlights the practical utility of thresholds, asserting "an alarm would occur whenever the value of a specific internal metric exceeded some predetermined threshold".

In this paper we present a replication of Shatnawi's study, henceforth referred to as the original study. Through this replication we expanded the study by using a broader range of datasets and metrics. Furthermore, the original study carried out experiments for individual projects, i.e. choosing training and test sets from the same project; thresholds derived, therefore, were project-dependent. Other project teams may not stand to benefit from the threshold results of the original study. In this replicated study we experimented to derive generalized threshold results for a particular project group. Thus, other projects having similar characteristics could benefit from the results of this study. We followed the guidelines for replication papers proposed by Carver to organize the structure of the paper (Carver, 2010).

The remainder of this paper is organized as follows: Section 2 describes the original study. Section 3 presents our replication with goals, hypothesis and context. Section 4 presents a summary of the previous studies. Metrics and the datasets used in this paper are described in Section 5. Section 6 explains the threshold derivation model along with algorithms. Section 7 is devoted to the validation, results, and findings of the study. Threats to validity are explained in Section 8. Conclusions and possible future works of the research are presented in Sections 9 and 10.

2. Information about the original study

We briefly describe the original study in this section. We give other information about the original paper in related sections, e.g. metrics and datasets used in the original paper are given in the section on metrics and datasets. Shatnawi (2010) used logistic regression based on the Bender method (Bender, 1999) in his study, using Eclipse 2.0 as the training dataset and Eclipse 2.1 as the testing dataset. Research question investigated in his study was to validate threshold effects of the CK metric suite. He reached effective threshold values for three out of six CK metrics, passing the statistical significance filter and performance limit. However, when he used same model on two other open-source systems (Mozilla and Rhino), only a few positive threshold values were achieved—most yielded negative thresholds. Shatnawi concluded that the proposed model could not be generalized for all open-source software systems.

3. Replicated study

3.1. Research question

Our research question includes the same as in the original study, but the difference is that we investigate effective generalized thresholds that can be utilized in other similar software systems.

3.2. Hypotheses

The aim of this paper is to empirically examine whether there are effective generalized threshold values for source code metrics or not. Thus, the hypotheses to be tested along with the null hypotheses in this paper are as follows for each metric:

- Hypothesis: There is a relationship between the metric threshold and fault-proneness of a module; i.e., a metric value of a software module that is higher than a derived threshold is more fault-prone.
- Null Hypothesis: There is no relationship between the metric threshold and fault-proneness of a module; i.e., a metric value of a software module that is higher than a derived threshold may or may not be more fault-prone.

3.3. Case study setup

We categorized software modules into three fault-proneness groups: zero-fault-prone (0fp), 1+fault-prone (1+fp), and 3+fault-prone (3+fp). On one hand, a 1+fp software module refers to a module that tends to cause at least one fault during the run of that software. A 3+fp module, on the other hand, tends to have at least three different faults during a run. Such a three-level flagging approach would give a developer company insight to be able to prioritize the coding and the testing activities. From a testing standpoint, the company could then allocate resources more effectively. In the case of limited resources, only 3+fp modules would be taken into consideration. With enough test staff and time, 1+fp modules would be taken into consideration too. Moreover, three-level flagging has also been used by other studies to identify modules according to their risk rank. We identified three fault-proneness group after analyzing the distribution of fault numbers. Modules which have zero faults are quite frequent (76%); modules which have between 1 and 3 faults are less frequent (21%); and modules which have more than 3 faults are infrequent (5%).

Open-source software systems from PROMISE[1] repository (Jureczko & Madeyski, 2010; Menzies, Caglayan, Kocaguneli, Krall, & Turhan, 2012) were used as our benchmarking datasets. The performance indicator for the case studies was the geometric mean (g-mean) of true positive rate (TPR) and true negative rate (TNR). A logistic regression based on the Bender method successfully applied a statistical significance test used to derive thresholds for each source code metric.

3.4. Changes to the original study

In this study, our threshold derivation model was based on the study of Shatnawi (2010). Our contribution to and difference from his study is threefold:

1. We merged 10 datasets from different open-source systems in order to reach generalized threshold values, and then did validation checks on 27 datasets. Original study: applied the learning model to different open-source systems individually.
2. We used an extensive number of metrics, including different metric groups, as well as the CK suite. Original study: included only six CK metrics.
3. We introduced three risk levels (0fp, 1+fp, and 3+fp), which is expected to increase practical benefits. Original study: used two risk levels (fault-prone and non-fault-prone).

[1] All dataset along with description are avaliable at http://openscience.us/repo/

4. Related work and background

4.1. Fault prediction

Several machine learning techniques have been used for the purpose of predicting software faults. The following techniques were used for this problem: Random forest (Cukic & Singh, 2004), artificial immune system (Catal & Diri, 2009b), naive bayes (Menzies, Greenwald, & Frank, 2007; Padberg, Ragg, & Schoknecht, 2004; Turhan, Tosun Mısırlı, & Bener, 2013), J48 (Koru & Liu, 2005), tree-based methods (Cukic & Singh, 2004; Khoshgoftaar, Allen, Jones, & Hudepohl, 1999; Selby & Porter, 1988), case-based reasoning (Khoshgoftaar & Seliya, 2003), Bayesian network (Pérez-Miñana & Gras, 2006), and artificial neural network (Arar & Ayan, 2015; Zheng, 2010). According to a study by Menzies et al. (2007), naïve bayes by preprocessed data with log filter gives the best performance among other learners. Some statistical techniques including support vector machine (Elish & Elish, 2008) and logistic regression (Gao, Khoshgoftaar, Wang, & Seliya, 2011; Olague, Etzkorn, Gholston, & Quattlebaum, 2007) were also applied to this problem, and their results are also notable. Univariate logistic regression was used to analyze the impact of each individual metric other than the combination of different metrics. Thus, the relation (strength) of each metric with fault-proneness emerged. Studies of Gyimothy, Ferenc, and Siket (2005) and Zhou et al. (2006) are two examples that use univarite logistic regression and multivariate logistic regression as well. Hall, Beecham, Bowes, Gray, and Counsell (2011) used an extensive literature review to analyze 208 defect prediction studies published from 2000 to 2010. Using criteria they developed and applied, they eliminated most of the studies from further analysis due to the lack of adequate contextual and methodological information. They then synthesized the quantitative and qualitative results of the remaining 36 studies. In their conclusion they noted that "although there are a set of fault prediction studies in which confidence is possible, more studies are needed that use a reliable methodology and which report their context, methodology, and performance comprehensively." This was also another motivation for our study.

4.2. Metric threshold derivation

Compared to the high interest in software fault prediction studies, metric threshold derivation studies are fewer. Generally, metric threshold values are defined by senior software developers relying on their coding experience (Oliveira, Valente, & Lima, 2014). However, different threshold derivation methods based on machine learning techniques have also been proposed. The Bender method and the receiver operating characteristics (ROC) method are among the more popular. Shatnawi (2010) used logistic regression based on the Bender method in his study. Catal, Alan, and Balkan (2011), used a ROC-based threshold evaluation in their study on publicly available NASA datasets written in C++ language. These datasets included mostly method-level metrics. Sánchez-González, García, Ruiz, and Mendling (2012), applied both techniques in order to get threshold values for business process models. Then, they compared performance in terms of recall, precision, and f-measure. More suitable results were gathered on two different experiment families when using the ROC curve–based approach. Shatnawi, Li, Swain, and Newman (2010), also used ROC to find thresholds for binary and ordinal categorization, using different releases of Eclipse as datasets. However, derived thresholds for binary categorization did not produce acceptable results.

Other statistical and data mining techniques were also used for the same purpose. Ferreira, Bigonha, Bigonha, Mendes, and Almeida (2012), categorized software modules as good, regular, and bad according to metric values. The metrics' thresholds were defined according to the statistical distribution of data from 40 open-source software programs, implemented in Java. Data was converted to a best-fit probability distribution (Poisson or Weibull). Categorization was done according to the frequency of each metric value. Ferreira et al. (2012) noted that these thresholds might strongly contribute to decision making, but could not substitute for the judgments of experts. Ferreira et al., checked modules categorized as bad manually to see whether there were some possible design flaws or structural coding problems. This inspection revealed that suggested thresholds can aid in detecting modules that are potentially problematic. Erni and Lewerentz (1996), proposed a methodology to find thresholds depending on mean and standard deviations of metric values. They created two formulas for defining two thresholds: $T_{\min} = \mu - \sigma$ and $T_{\max} = \mu + \sigma$ (μ is mean symbol and σ is standard deviation symbol). Herbold, Grabowski, and Waack (2011), used another machine learning method, called Rectangle Learning, to find thresholds for different programming languages by separate experiments. Rosenberg, Stapko, and Gallo (1999), discussed the threshold values of CK metrics proposed by the Software Assurance Technology Center to be used in NASA projects.

Table 1 shows the derived thresholds from the various studies summarized above. The second column presents threshold values obtained by the original study. Metrics which are not statistically significant on the related study are shown by NA (Not Applicable). Some studies do not include all metrics in the table (shown by a dash), while other studies include more metrics than those that are shown in the table; only metrics which are also used in this study are transferred here.

A threshold definition methodology should meet some requirements: i) it should be built on representative benchmark data, not opinions; ii) data should be analyzed statistically, e.g., distribution and scale conditions; and iii) it should be repeatable, clear, and easily utilized on other systems (Alves, Ypma, & Visser, 2010; Sánchez-González et al., 2012). It is considered that these three conditions were fulfilled in this study.

4.3. Generalized models (cross-project fault prediction)

One of the goals of this study is to obtain generalized threshold values that can be used for other projects similar to those under investigation in our case studies. Fault prediction studies generally use the historical data of preceding versions of software to build learning models and apply those results to ongoing versions. This strategy is referred to as within-project fault prediction. However, such an approach is not applicable for projects that are in development for the first version, or for companies that do not gather historical data. Therefore, there is a need to identify new strategies that require no historical data (Jureczko & Madeyski, 2010). A cross-project fault prediction strategy aims to achieve a successful prediction performance for projects with limited or no historical data by using data from other projects to train learning models. Several studies have discussed the possibility of this scenario (Canfora et al., 2013; Zimmermann, Nagappan, Gall, Giger, & Murphy, 2009). Cross-project fault prediction is of interest for two reasons: i) it can be used for projects with limited data, and ii) it allows generalized fault prediction models. Since projects exhibit heterogeneous characteristics, this problem is challenging and prediction accuracy is not always promising (Canfora et al., 2013) However, according to a study by Koru (2005), building generalized fault prediction models is possible and such models may be quite favorable. Such an opinion promotes attempts on this issue.

Table 1
Derived threshold values from different studies[c].

	Shatnawi (2010)	Ferreira et al. (2012)	Herbold et al. (2011)[a]	Rosenberg et al. (1999)
Dataset	Eclipse	40 open source systems	Eclipse	No dataset
Method	Bender	Data distribution	Rectangle learning	–
Language	Java	Java	Java	General
CBO	9	–	5	5
RFC	40	–	98	100
WMC[b]	20	–	99	100
DIT	NA	2	NA	2–5
NOC	NA	–	NA	–
LCOM	NA	20	–	–
LOC	–	–	–	–
Ca		20	–	–

[a] Thresholds were obtained also for C++ and C, but for the relevance of this study only Java thresholds were taken.
[b] WMC calculation differs according to tools: some assign a complexity of 1 to each method, others calculate McCabe's complexity for each method. Consequently, we will not compare our results in terms of this metric.
[c] All metrics studied in this paper are explained in the Appendix.

5. Metrics and datasets

5.1. Metrics

Software metrics reflect the qualitative characteristics of a software module in a quantitative manner. Metrics describe a module in a more understandable way. A module may be a method/function in structured languages (e.g., C or Ada) or a class in object-oriented languages (e.g., Java or C++). However, methods can also be acted upon as a module in object-oriented languages according to the nature of the study or data collection methodology. Metrics, which represent features of a method, are called method-level metrics; the features of a class, on the other hand, are called class-level metrics. In other words, values of class-level metrics are measured by using the source code of the entire class. The focus of this study is object-oriented languages, so we used class-level metrics. We will mostly use the term "module" throughout this paper, i.e. a module refers to a java class. The term "class" is used for a different meaning in classification problems, so it can cause confusion for readers. In addition, Fenton and Pfleeger (1998) have divided metrics into two categories: internal and external. Internal metrics are measured by considering only the product or resource itself, without dealing with its behavior. External metrics, in contrast, are related to the behavior of the product only. The focus of this study is on internal metrics: the source code of software systems, but not their behavior or functionalities.

CK is a metric set to be used to quantify object-oriented programming paradigms (Chidamber & Kemerer, 1994). This set includes six well-known metrics empirically validated by a number of relevant studies (Catal & Diri, 2009a; Radjenović, Heričko, Torkar, & Živkovič, 2013). In addition to CK metrics, we also used other metrics in the case studies to analyze the validation results of different metrics for the purpose of deriving threshold values. Radjenović et al.,'s systematic literature review of software fault prediction comprehensively surveyed studies using different metrics (Radjenović et al., 2013). The list of all metrics investigated in this study can be seen in Table A1 in the Appendix. All metrics, except Cyclomatic Complexity (CC), are class-level metrics. Therefore, MAX_CC and AVG_CC are derived from CC in order to convert CC to class-level metrics.

5.2. Datasets

In our case studies, 37 datasets from the PROMISE (PRedictOr Models In Software Engineering) data repository (Jureczko & Madeyski, 2010; Menzies et al., 2012) were used. These datasets were created from 10 widely used Java-based open-source systems along with their multiple releases. These datasets included the values of 20 metrics and an output value indicating the number of bugs that a related module had. Metric values of each module were calculated by using the ckjm[2] tool (Spinellis, 2005). The ckjm calculates object-oriented software metrics by processing the bytecode of compiled Java files. Then another tool, BugInfo[3], was used to analyze the commits history by regular expression matching style and to decide whether a commit was a bug fix or not. When a commit comment in a code revision meets the regular expression noted below, this revision is counted as a bug fix. Then, BugInfo increments the defect count for all modules that have been modified in the commit.

```
.* [bB][uU][gG][fF][iI][xX]
| [bB][uU][gG][zZ][iI][lL][lL][aA] .*
```

The first releases of ten open-source systems were merged into one file and used as a training dataset. The remaining datasets (27) were used as test datasets to validate the effectiveness of derived thresholds. These systems, with their descriptions, version information, and sizes in KLOC are listed in Table 2. As can be seen from the table, number of test versions for systems are not the same. This unequal distribution does not affect the validation phase, since we act on each dataset as if it were a separate sample with equal importance.

Training data and the learning algorithm are the two main factors affecting prediction performance (He, Shu, Yang, Li, & Wang, 2011). An arbitrary selection of training data will more likely yield a poor learner, which causes wrong results and decisions (Zimmermann et al., 2009). Nagappan, Ball, and Zeller (2006) studied the generalized ability of fault prediction models and conducted experiments on five Microsoft projects. They found no single metric set applicable for all projects, but prediction models were found to be accurate when they were trained and applied to same or similar projects (Nagappan et al., 2006). Therefore, we should select projects and create training sets carefully. Previous studies indicate that some characteristics can be used to group projects (Jureczko & Madeyski, 2010). We chose ten projects, considering these common characteristics:

- Open-source
- Same language (Java)
- Similar domain (most of them are connected with text-processing)
- Medium-sized development team (consisting of fewer than 25 developers)

[2] ckjm tool is avaliable at http://www.spinellis.gr/sw/ckjm/
[3] BugInfo tool is available at https://kenai.com/projects/buginfo

Table 2
Datasets used in this study.

Name	Description	Version used as training set	KLOC of training set	Versions used as test set	KLOC of test sets
ant	Build tool and library, aiming to build processes according to build files	1.3	37	1.4, 1.5, 1.6, 1.7	54, 87, 113, 208
camel	Integration framework, providing rule-based routing between Java objects	1.0	33	1.2, 1.4, 1.6	66, 98, 113
jedit	Advanced text editor, providing to write software codes	3.2	128	4.0, 4.1, 4.2, 4.3	144, 153, 170, 202
log4j	Logging library, providing an infrastructure to log defined messages during system run.	1.0	21	1.1, 1.2	19, 38
lucene	Advanced text search engine library.	2.0	50	2.2, 2.4	63, 102
poi	Java library, providing to read and write files in Microsoft Office formats.	1.5	55	2.0, 2.5, 3.0	93, 119, 129
synapse	High-performance and lightweight Enterprise Service Bus (ESB), supporting several interchange formats such as plain text, binary, JSON.	1.0	28	1.1, 1.2	42, 53
velocity	Java-based template engine, enabling to transform data from a format to another.	1.4	51	1.5, 1.6	53, 57
xalan	XSLT processor, converting XML documents to different document types.	2.4	225	2.5, 2.6, 2.7	388, 412, 428
xerces	XML parser library, providing utilities for XML manipulation.	1.2	159	1.3, 1.4	167, 141

- Medium class-size (class number between 101 and 1000)
- Development by similar companies (mostly Apache)

There are several previous studies using the same dataset group as in our study (Canfora et al., 2013; Herbold, 2013; Jureczko & Madeyski, 2010; Shatnawi, 2015; Turhan et al., 2013). Other software systems which have the above characteristics can exploit the findings of this study. Eclipse was used as a main dataset by the original study, but does not fulfill the conditions listed above. Therefore, this dataset was neither included in our case studies nor applied to our derived thresholds.

5.3. Diving into the training set

Our approach to training dataset creation relied upon studies by (Liu, Khoshgoftaar, & Seliya, 2010; Turhan et al., 2013). The hypothesis worked by Liu et al. (2010) is that using multiple software systems rather than a single system during the training process provides additional information that can advance the prediction performance of the learning model. They used seven datasets from NASA's repository and they employed different strategies for the creation of training and test sets. One of the strategies named as multiple datasets baseline classifier. In this strategy, one system was used as a test set, while the remaining six systems were merged in order to build one single training set (Liu et al., 2010). The study by Turhan et al. (2013) is based on a mixed project data approach. In this approach, they used data from other projects to improve the performance of within-project fault prediction. They constructed models from a mix of within- and cross-project data and concluded that in the case of limited data, mixed-project fault predictions are justifiable. Thus, we also used mixed project data to derive thresholds of software metrics. In general, data from the first version of software is available to mine, with other versions released later. Therefore, to make the model more realistic, our learner model built on data from the first versions.

The fault (bug) percentages of each training set—the first versions of the systems—are given in Table 3. The module number means the number of classes included in the related set. The 1+faults (%) row corresponds to percentages of modules that have at least one fault (1 is included). Similarly, the 3+faults (%) row corresponds to percentages of modules that have at least three faults (3 is included). After merging each individual training set, there were a total of 2819 modules, of which 26.00% were found to have 1+faults and 5.14% 3+faults.

Table 4 represents the statistical characteristics of the training set in terms of twenty software metrics. Statistics in all columns except the last column are derived from the merged single training set. The training dataset is created by merging first versions of the software systems as shown in Fig. 1. The last column (Std. Dev. 2) indicates differences of the mean of metrics based on 10 different training sets individually. The interquartile range (IQR) column represents the difference of the 25th percentile value (1st quartile) and 75th percentile value (3rd quartile). It gives an idea of how the data is distributed. The column (N>0) shows the count of modules which have a value greater than zero for the related metric. As can be seen in the table, among CK metrics, the variability of DIT and NOC is very low. More than 75% of modules have a 0 NOC value

Table 3
Fault (bug) percentages of 10 training sets.

	ant	camel	jedit	log4j	lucene	poi	synapse	velocity	xalan	xerces	total
Modules (#)	125	339	272	135	195	237	157	196	723	440	**2819**
1+faults (%)	16.00	3.83	33.09	25.19	46.67	59.49	10.19	75.00	15.21	16.14	**26.00**
3+faults (%)	3.02	0.00	15.07	4.44	13.85	17.72	0.64	6.63	1.24	0.45	**5.14**

Table 4
Descriptive statistics of the training set; The training set is created by merging first versions of the software systems under investigation. Only the last column is derived from individual (unmerged) training sets.

Metric	N	N>0	Min	Median	Max	Mean	Std. Dev. 1	IQR	Skewness	Std. Dev. 2
WMC	2819	2781	0	6	399	10.49	15.59	9	8.73	2.19
DIT	2819	2813	0	2	8	2.15	1.39	2	1.28	0.42
NOC	2819	330	0	0	100	0.53	3.25	0	15.76	0.14
CBO	2819	2633	0	6	185	10.48	14.96	9	4.68	2.87
RFC	2819	2781	0	18	487	26.82	32.86	26	4.45	6.08
LCOM	2819	2052	0	5	11469	90.20	450.15	33	12.98	47.26
Ca	2819	2166	0	2	184	5.19	13.11	3	6.06	1.38
Ce	2819	2350	0	4	93	5.90	7.43	7	3.11	2.02
LCOM3	2819	2690	0	0.87	2	1.12	0.68	1.36	0.25	0.18
NPM	2819	2687	0	5	169	8.10	10.97	7	4.57	1.94
DAM	2819	1556	0	0.5	1	0.47	0.46	1	0.09	0.19
MOA	2819	960	0	0	38	0.80	1.93	1	7.20	0.25
MFA	2819	1554	0	0.43	1	0.43	0.42	0.9	0.15	0.11
CAM	2819	2770	0	0.42	1	0.47	0.24	0.3	0.66	0.03
IC	2819	1142	0	0	5	0.59	0.86	1	1.55	0.21
CBM	2819	1142	0	0	30	1.68	3.16	2	3.14	0.91
AMC	2819	2463	0	13.60	780	22.67	35.79	22.16	8.92	6.28
MAX_CC	2819	2648	0	2	236	4.08	8.69	3	12.91	1.46
AVG_CC	2819	2648	0	1	22	1.30	1.17	0.75	4.91	0.25
LOC	2819	2795	0	107	23350	281.21	770.55	243	14.00	106.8

Fig. 1. The training dataset is created by merging first versions of the software systems.

(mean is 0, IQR is zero, and N>0 is only 330). The median value of DIT is only 2, and the maximum value is 8. This low variability is found also in other studies (Basili, Briand, & Melo, 1996; Herbold et al., 2011; Shatnawi, 2010; Zhou & Leung, 2006). Conversely, LCOM variability is high, with a standard deviation of 450.15). Further, its high values are scattered in the 4th quartile (50% of modules have only at most a value of 5 and IQR is only 33, while the mean is about 90). This distribution of LCOM is similar to that in the study by Shatnawi (2010). He did not include this metric in his studies because of these statistics, but we used it since the variability of it is not as low as DIT and NOC. All metrics of QMOOD (Quality Model for Object-Oriented Design), except NPM, have low variability too. So we did not include the following metrics based on their statistics: DIT, NOC, LCOM3, DAM, MOA, MFA, CAM, IC, and CBM.

The metric values belonging to the 10 individual training sets show similar statistical characteristics based on the Std. Dev. 2 results as shown in Table 4. Std. Dev. 2 for each metric is calculated by the following formula:

$$Mean2 = \frac{\sum_{i=1}^{N} \mu_i}{N} \quad (1)$$

$$Std.Dev.2 = \sqrt{\frac{\sum_{i=1}^{N} (\mu_i - Mean2)^2}{N-1}} \quad (2)$$

Where N refers to number of individual training set, 10; μ is mean; and μ_i is the mean of the metric for the ith training dataset. The Std. Dev. 2 and mean columns should be read in conjunction. It is probable that a metric which has a high mean also might have a high Std. Dev. 2. As can be seen from the table, datasets are not very different from each other, but they are not the same as well. This can be a welcoming situation in order to reach generalized threshold values. Unlike the others, LCOM and LOC metrics differ more, having 47.26 and 106.8 standard deviations, respectively.

Software metrics data are skewed, which means data generally do not follow normal distribution. There have been many studies investigating the distribution of software metrics. A conclusion drawn by those studies is that software metrics data seems to follow by a power law (Baxter et al., 2006; Ferreira et al., 2012; Shatnawi & Althebyan, 2013). A power law distribution is heavy-tailed; that is, large values for the random variable are very rare, whereas lower values are very common. A power law is a special kind of probability distribution in which the probability that a random variable X takes a value x is proportional to a negative power of x, i.e., $P(X=x) \propto cx^{-k}$ (Ferreira et al., 2012). Fig. 2 visualizes distribution of each metric based on the probability distribution function (pdf), f(x), which displays the probability the random variable gets a value x. Furthermore, numerical methods, such as skewness,

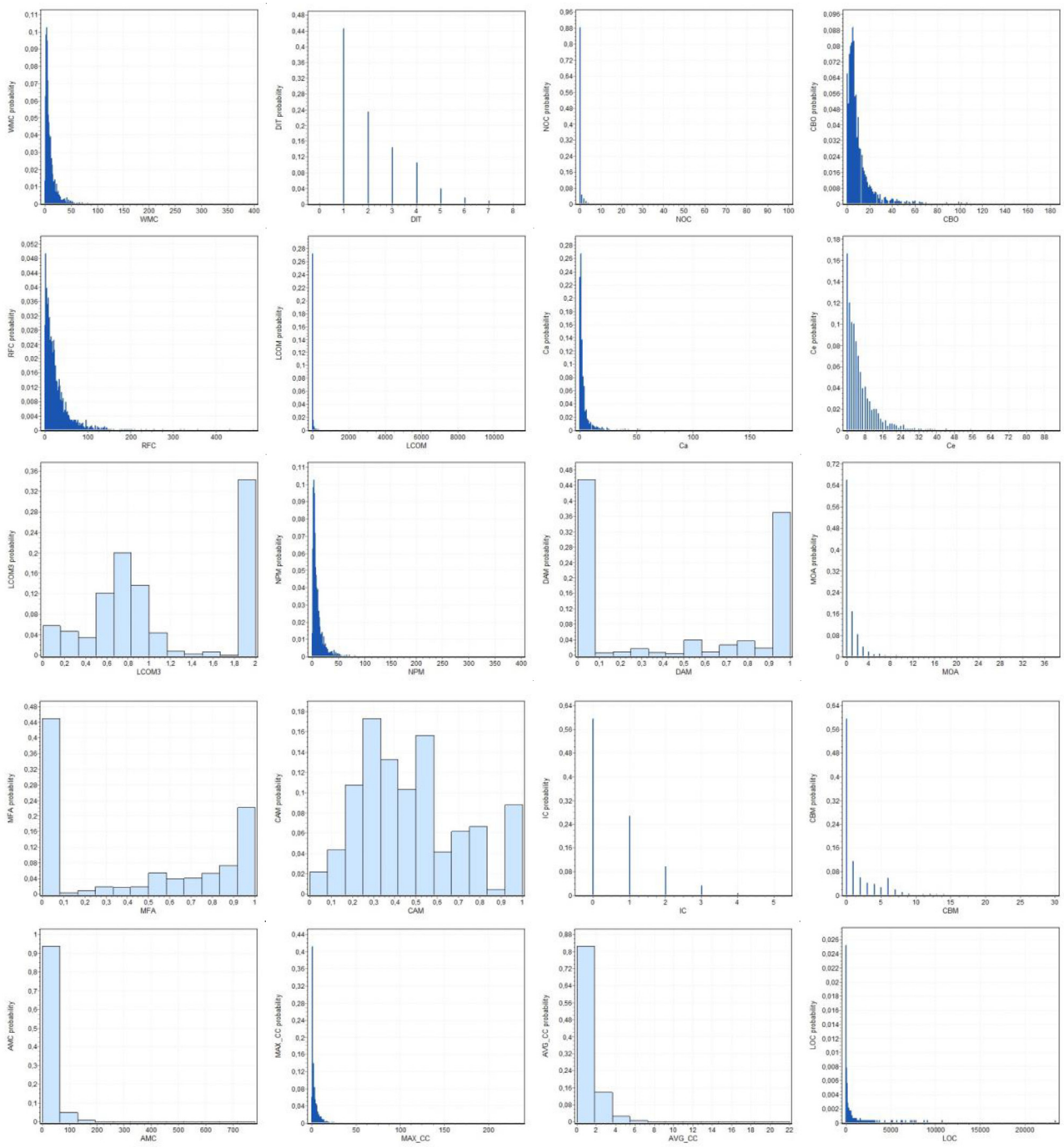

Fig. 2. Distribution of each metric based on probability distribution function.

give an idea about data normality. The last column of Table 4 includes the skewness measure. Skewness measures the degree of asymmetry of a probability distribution. If skewness is greater than zero, the distribution is right-tailed, having more observations on the left. Normally distributed data should have a skewness value near zero (Shatnawi & Althebyan, 2013).

6. Threshold derivation model

Our threshold derivation model is based on study of Shatnawi (2010), adapted to flowchart representation, and shown in Fig. 3. This flowchart outlines the model in a straightforward manner. The threshold derivation process comprises three steps: (I) Model Fit and Statistical Analysis, (II) Deriving Thresholds, and (III) Validation of Thresholds. In the first step, learner models are created for each metric according to the training data, and these models are exposed to a significance test to check whether or not the related metric has a statistical meaning. In the second step, thresholds are derived for significant metrics, using the Bender method. In the last step, derived thresholds are examined as to whether they are valid or invalid, by checking the performance result of the classification tree. The merged training dataset is used as input in the first and second steps, and thereafter test datasets are used for the third step to review the performance of the thresholds and the learner model.

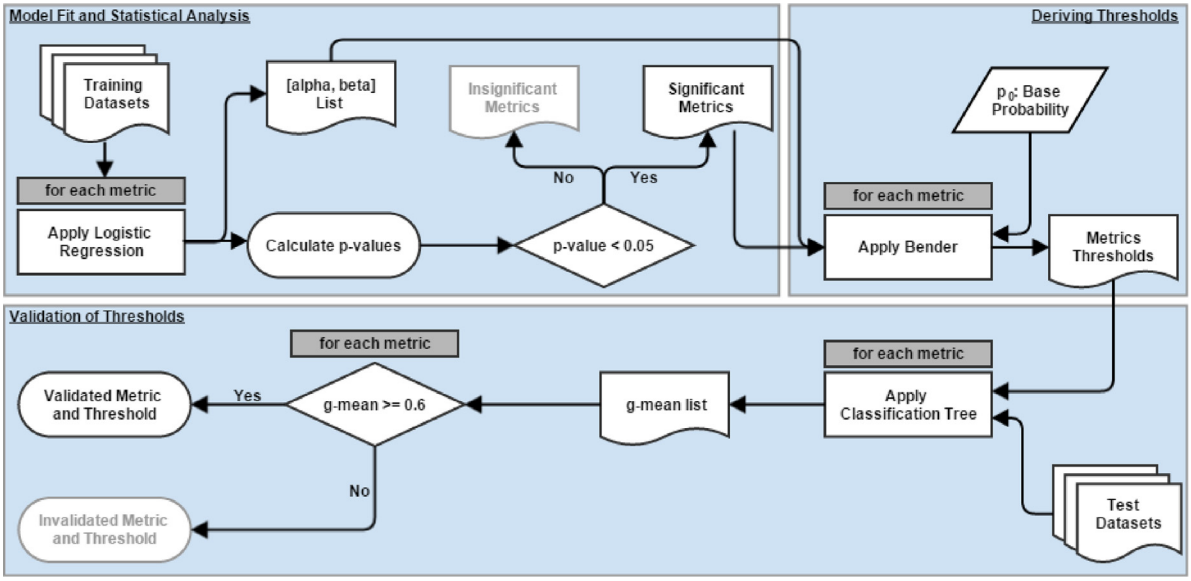

Fig. 3. Flowchart diagram of the threshold derivation model.

Details of all these processes and used algorithms are described in the following sections.

6.1. Model fit and statistical analysis

Univariate logistic regression is used in this study to create the learner model. Logistic regression is a statistical classification technique in which the dependent variable can only be one of two cases (dichotomous output). Since software defect datasets include faulty and non-faulty modules, this method is convenient to use. The model can also take more than one independent variable as input. However, we investigate each metric individually, which makes our model univariate. The output of the logistic regression model is the probability of one of the two cases occurring (i.e., whether the module is faulty/fault-prone). The formula is as follows:

$$P(X) = \frac{e^{\beta_0+\beta_1 X}}{1+e^{\beta_0+\beta_1 X}} \tag{3}$$

where X is the metric value, β_0 is the intercept coefficient, and β_1 is the slope coefficient. Non-linear logistic curves are created with different coefficient values. The independent variable (e.g., the LOC metric value) and dependent variable (probability of being faulty) are represented respectively by X and Y axes of this curve. Intercept and slope coefficients define curvature and the non-linearity level of the curve. The optimum values of these coefficients are scanned using the Artificial Bee Colony (ABC) algorithm. The ABC algorithm simulates the foraging behavior of honey bees. It was proposed by (Karaboga & Basturk, 2007) for the numerical solution of optimization problems. The ABC algorithm is not the focus of this paper; details of it can be accessed from various references (Ayan & Kılıç, 2012; Ayan, Kılıç, & Baraklı, 2015; Karaboga, Akay, & Ozturk, 2007).

A one-tailed p-value with a 95% confidence level ($\alpha = 0.05$) is used to determine whether the corresponding coefficient is statistically significant or not (Bruin, 2006). If the calculated p-value is outside of the confidence interval (below 0.05), then the related metric passes the significance test and is ready for the next step. Otherwise, it is eliminated from further analysis.

6.2. Deriving thresholds

Statistically significant metrics which pass one tailed test are examined to find a threshold value. Thresholds are investigated by using the Bender method (Bender, 1999). Bender proposed a method called Value of an Acceptable Risk Level (VARL), based on a logistic regression equation.

$$VARL = \frac{1}{\beta_1}\left(\ln\left(\frac{p_0}{1-p_0}\right) - \beta_0\right) \tag{4}$$

where p_0 shows the defined base probability. A module with a higher probability value in the logistic regression curve is accepted as fault-prone; a module with a lower probability value in the logistic regression curve is accepted as non-fault-prone. In other words, the fault occurrence probability of a module which has a metric value below VARL is lower than the base probability. For example, if the base probability is chosen as 0.6 (60%) and assuming the metric value 24 corresponds to that base probability from the logistic regression curve, then modules with metric values below 24 have a fault occurrence probability lower than 60%. Conversely, modules whose metric values are above 24 have a fault occurrence probability over 60%. The relation between the base probability and threshold is depicted in Fig. 4.

In the Case studies and results section, it will be explained which values were used as the base probability in our study.

6.3. Validation of thresholds

After finding threshold values for each metric, a validation process is performed by using test datasets as input. Threshold performances on each test dataset are recorded individually and then analyzed individually and wholly together. The classification tree method is used to classify each module as fault-prone or non-fault-prone based on corresponding threshold values. Eq. (3) shows these two case situations where the result of the ith module is set as fault-prone if its jth metric value is higher than the derived threshold for this metric (THR_j). Otherwise, it is set as non-fault-prone.

$$y_i = \begin{cases} \text{faultprone} & \text{if } X_{ij} \geq THR_j \\ \text{non}-\text{faultprone} & \text{if } X_{ij} < THR_j \end{cases} \tag{5}$$

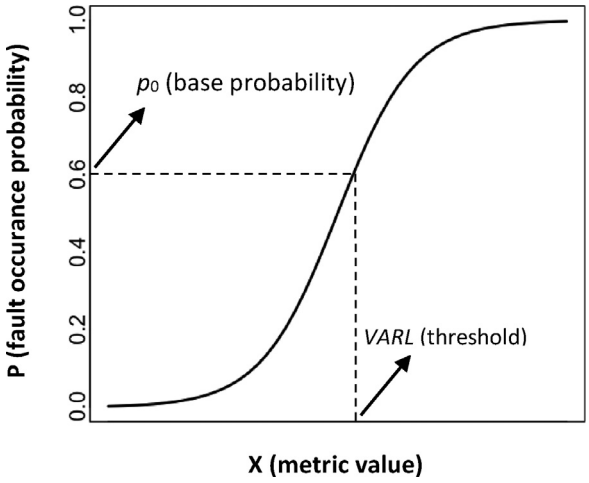

Fig. 4. The relation between the base probability and threshold.

Table 5
Confusion matrix.

		Predicted	
		YES	NO
Actual	YES	TP (True Positive)	FN (False Negative)
	NO	FP (False Positive)	TN (True Negative)

The confusion matrix, which shows each resulting case according to predicted and actual output, is shown in Table 5. Here, YES corresponds to fault-proneness and NO corresponds to non-fault-proneness.

When the dataset includes one class much more than the others, then it is characterized as imbalanced data. A basic simple accuracy indicator is not appropriate for datasets which have an imbalanced class distribution (Nickerson, Japkowicz, & Milios, 2001). For example, the ant dataset contains modules with 3+faults amounting to 5.14% of the total (see Table 3). A predictor could score a 94.86% accuracy performance, even if it predicts all modules as 3+fp without using any machine learning technique. In this study, the geometric mean (g-mean) is used as a performance indicator as it was introduced by Kubat and Matwin (1997). They found that the evaluation of results using the g-mean is more reliable than basic evaluation measures when data is imbalanced. Both the accuracy of majority and the accuracy of minority classes are considered in the g-mean calculation (Malhotra & Bansal, 2015). Shatnawi (2010) used the g-mean measure in his study. Based on his and other studies (Seliya, Khoshgoftaar, & Hulse, 2010; Wang & Yao, 2013) from same domain, we also employ the g-mean. Instead of many measures, using one single measure is also more practical to make the study comparable.

The formula for the g-mean is given by Eq. (4).

$$g - mean = \sqrt{TPR \times TNR} \tag{6}$$

In this equation *TPR* and *TNR* are formulated as follows:

$$TPR = \text{Re}call = \frac{TP}{TP + FN} \tag{7}$$

$$TNR = \frac{TN}{TN + FP} \tag{8}$$

where TPR indicates the rate of fault-prone modules that are classified correctly and TNR indicates the rate of non-fault-prone modules that are classified correctly. The measure *g*-mean is the geometric mean of these both rates. A high *g*-mean is obtained in cases of high TPR and as well as high TNR. When one of these two values is low, it would decrease performance also.

Table 6
TPR and TNR samples resulting in 0.6 *g*-mean.

TPR	TNR	g-mean
0.6	0.6	0.6
0.51	0.70	0.6
0.70	0.51	0.6

We used a limit value of 0.6 for the *g*-mean. Validation results above this limit indicate acceptable range and it is suggested that the derived threshold value for the corresponding metric is valid. Indeed, there is no consensus among researchers for the limit value, whether the results of fault prediction is acceptable or not. Reasons for using this limit value in our study are summarized below:

(1) According to our own research, coding, and testing experience, this limit value makes sense and it is useful in practice.
(2) Studies of Menzies et al. (2008, 2010), state that some upper bound of performance is reached in the fault prediction problem, since the information presented by static code features is limited. Since we also used static code features, it was hard to attain very high performance results in our case studies. Some studies evaluate their results by alternative measures, such as recall and precision. Zimmermann et al. (2009) characterized predictors as good when they have recall, precision, and accuracy above 0.75. Most studies have used 0.7 for recall and 0.5 for precision as base values (He et al., 2011; Herbold, 2013). Unlike these studies, we use a single g-mean measure. Recall and TPR are the same measure. However, precision and TNR are different. Even this difference, arithmetic mean of recall and precision base values of aforementioned studies (0.7 and 0.5) were considered in our g-mean limit value suggestion.
(3) In the original study, Shatnawi arrived at 0.59, 0.64, and 0.61 *g*-mean results for three metrics; he interpreted these results as effective. Relying upon this interpretation, the recommended limit value is feasible.

Furthermore, Table 6 is beneficial in order to show the practical aspect of the recommended limit value. The three examples in the table present TPR and TNR values resulting in 0.6 *g*-mean. The third example means 70% of faulty modules are captured by the model. If a software consisted of ten faulty modules, seven of them would be captured. Releasing the software after reviewing and editing these modules would decrease the work load of testers. Instead of focusing on ten modules, testers would focus on three modules. Our several years of experience in software development and testing can attest to the practical utility of such a limit value.

6.4. Model changes from the original study

The threshold derivation model depicted in Fig. 3 is mostly the same as in the original study. There are only the following slight changes from the original study:

(1) Base probability selection for the VARL formulation is based on the ratio of faulty modules (not shown in the figure). In the original study, Shatnawi set up his case studies with different base probabilities, those being 0.050, 0.060, 0.065, 0.075, and 0.100. In our case studies, using these values did not yield good results.

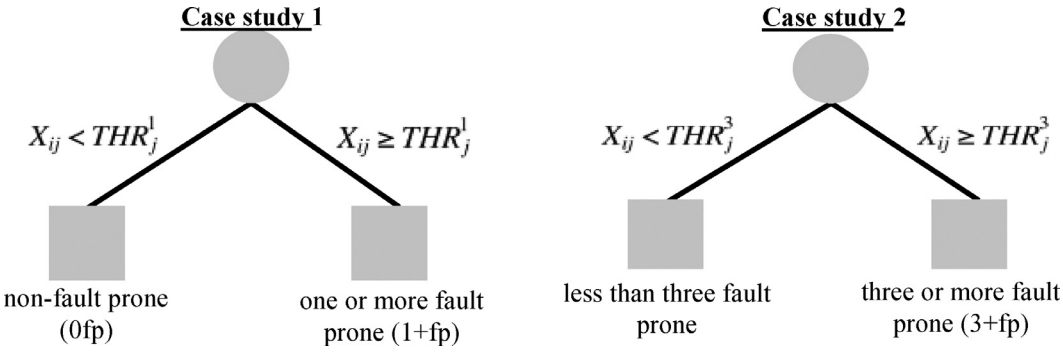

Fig. 5. Functionalities of Threshold-1 and Threshold-3 (shown as classification trees).

(2) Scanning optimum coefficients for the logistic regression formulation is attempted by the ABC algorithm (not shown in the figure). In the original study, there is no mention of an optimization algorithm being used. The ABC algorithm has many advantages, some of which are that: the implementation of this algorithm to all engineering problems is easy; it has broad applicability, even in complex functions or with continuous, discrete, or mixed variables; it does not require that the objective function be differentiable, continuous, or mathematically representable; it has an ability to explore local solutions; it is a global optimizer, with an effective search process even under high complexity and with risk of premature convergence; and it is robust against initialization, regardless of feasibility and the distribution of the initial solutions population. A study of a comparison of ABC with other well-known optimization algorithms revealed its better performance in classification problems (Karaboga & Ozturk, 2009).

(3) Interpretation of results for the metrics' thresholds is done according to a 0.6 *g*-mean limit value (shown in the figure). In the original study, Shatnawi arrived 0.59, 0.64, and 0.61 *g*-mean results for three metrics; although he interpreted these results as effective, there was no remark about g-mean limit value.

7. Case studies and results

We carried out two different case studies to find two different threshold sets. The first was about finding thresholds to categorize modules as 0fp or 1+fp, and the second was to find thresholds to categorize modules as 3+fp or not. These two thresholds are called Threshold-1 (THR^1) and Threshold-3 (THR^3), respectively.

1 Threshold-1 (THR^1): Software modules which have metric values above this threshold tend to have one or more faults (1+fp). Conversely, metric values below this threshold indicate zero-fault-proneness (0fp).
2 Threshold-3 (THR^3): Software modules which have metric values above this threshold tend to have three or more faults (3+fp). Conversely, metric values below this threshold indicate that the related module has less than three faults (including no fault).

Note that, for simplicity and understandability, explanations were done according to only 0fp and 1+fp in previous sections, including no remarks about 3+fp. Representations of both threshold sets are shown in Fig. 5.

7.1. Case study 1: Deriving threshold-1

First, the merged training data is prepared for this case study by labeling modules that have at least one bug as one. Then, a logistic regression with ABC optimization is applied by using training data. Logistic regression curves for each metric are yielded by candidate intercept and slope coefficients. Horizontal decision lines in these curves are used to discriminate between 0fp and 1+fp modules. Modules with a probability of fault occurrence over this decision line are regarded as 1+fp, while modules with a probability of fault occurrence below this decision line are regarded as 0fp. The decision line in terms of probability (*p(dl)*), output of each module (y_i), and cost function to minimize (Cost) are as follows:

$$p(dl) = \frac{N(1 + faults)}{N} \tag{9}$$

$$y_i = \begin{cases} 0\text{f} & \text{if } P(X_{ij}) < p(dl) \\ 1 + \text{fp} & \text{if } P(X_{ij}) \geq p(dl) \end{cases} \tag{10}$$

$$\text{Cost} = 1 - gmean \tag{11}$$

where $N(1+faults)$ represents the number of faulty modules among all N modules. Revisiting Table 3, 74% of all modules include zero faults. Therefore, the probability value of the decision line in this case study is set as 0.74. In each iteration of the ABC algorithm, a logistic regression curve is created by estimated coefficients. The output of the ith module, y_i, is predicted as 1+fp if the probability of fault occurrence corresponding to the metric value under investigation (j), is greater than *p(dl)*; otherwise, it is predicted as 0fp. The probability of fault occurrence, $P(X_{ij})$, is calculated by using the logistic regression equation (Eq. (1)). After applying this comparison to all modules, TP, TN, FP, FN values are gathered, as are *g*-means. The goal of the ABC is to minimize the Cost, i.e., maximize the g-mean. According to the Cost value gathered in each step, coefficients of the logistic regression are updated. When predefined conditions are met, optimal intercept and slope coefficients are recorded to be used within the Bender method. As well as logistic regression coefficients, the Bender method requires base probability p_0. We used the same probability value of the decision line, *p(dl)*, and consequently the corresponding X value is accepted as the metric threshold. The optimal coefficients, *p*-value, and threshold values derived for each metric in this case study are shown in Table 7.

As can be seen from the *p*-value column, all metrics are statistically significant (less than 0.05). As expected, all slope coefficients are positive, indicating the same change of direction in metric value and probability of fault-proneness (i.e., the higher the metric value, the more the fault-proneness). Eventually derived threshold values are shown in the last column.

Assessments of these thresholds were done by using 27 test datasets, altogether totaling 12,090 modules (classes). Results are summarized in Table 8.

Min g-mean and Max g-mean columns in the table show test sets which have the best and worst performances among all others.

Table 7
Results of logistic regression and the Bender method (Case study 1).

Metric	Intercept	Slope	*p*-value	Threshold
WMC	−0.616	0.233	0.035	**7.13**
CBO	−4.000	0.666	0.031	**7.58**
RFC	−0.973	0.100	0.009	**20.19**
LCOM	−0.603	0.222	0.000	**7.42**
Ca	−2.289	2.741	0.000	**1.22**
Ce	−0.572	0.407	0.000	**3.98**
NPM	−0.183	0.251	0.000	**4.90**
AMC	−1.145	0.217	0.011	**11.37**
MAX_CC	−4.000	2.936	0.000	**1.72**
AVG_CC	−1.469	2.449	0.000	**1.03**
LOC	−3.779	0.042	0.000	**114.11**

The Std. Dev. column represents how performances of each test set differ from each other. N(*g*-mean)>0.6 is the number of test datasets which have a *g*-mean performance above our predefined limit (0.6). The last column is simply an average of all *g*-means. Our case study reveals that seven metrics have satisfactory results (above 0.6), validating their derived threshold values. These metrics are WMC, CBO, RFC, LCOM, Ce and LOC. Four metrics, on the other hand, have unsatisfactory results (below 0.6), not validating their derived threshold values. These metrics are Ca, AMC, MAX_CC and AVG_CC. However, all metrics except Ca show at least a 0.57 *g*-mean, indicating that the related threshold values also make sense of fault-proneness of modules. The RFC metric gives better results than others, with 17 out of 27 test sets giving a *g*-mean above 0.6. An interesting finding is that the Ce metric has an acceptable threshold, whereas the Ca has not. This means that a module which has more dependence on other modules is more fault-prone. However, a more responsible module might not be more fault-prone.

A more detailed statistical visualization by box plots is shown in Fig. 6. Box plots are a convenient way to display the distribution of results (Williamson, Parker, & Kendrick, 1989).

The horizontal dotted line represents our predefined limit value. Tiny symbols inside each box represent mean values, and straight lines show medians. The LOC metric returns the best result in terms of stability; nearly bottom section of the box (1st quartile) is above the limit. In other words, the threshold of the LOC metric tends to be more generalized, so it can be applied to other similar software systems less tentatively. Next in line are RFC, CBO, and WMC, respectively. However, none of the thresholds give satisfactory results in more than 20 test sets. When looking from the test sets perspective, some of them give very poor results against the thresholds, although some give better results.

Table 9
Results of logistic regression and the Bender method (Case study 2).

Metric	Intercept	Slope	*p*-value	Threshold
WMC	−3.513	0.384	0.042	**11.87**
CBO	−4.000	0.624	0.011	**8.09**
RFC	−1.905	0.099	0.001	**29.71**
LCOM	0.266	0.037	0.000	**20.83**
Ca	−0.265	0.599	0.000	**2.19**
Ce	−0.751	0.268	0.017	**6.71**
NPM	−0.072	0.159	0.000	**7.05**
AMC	−2.922	0.230	0.021	**17.22**
MAX_CC	−1.751	0.755	0.000	**3.70**
AVG_CC	−0.732	1.733	0.000	**1.03**
LOC	−3.093	0.024	0.000	**170.91**

7.2. Case study 2: *Deriving threshold-3*

First, the merged training data is prepared for this case study by labeling modules that have at least three bugs as 1. Then, logistic regression with ABC optimization is applied. The decision line in this case study is used to discriminate between 3+fp and less-than-3-fault-prone (including 0fp) modules. The output of each module is as follows:

$$y_i = \begin{cases} \text{less than 3 fault prone} & \text{if } P(X_{ij}) < p(dl) \\ 3 + \text{fp} & \text{if } P(X_{ij}) \geq p(dl) \end{cases} \quad (12)$$

Revisiting Table 3, 94.86% of all modules include fewer than three faults. Therefore, the probability value of the decision line in this case study is set as 0.9486. If the result of the logistic regression equation for a module is above this value, then this module is counted as 3+fp; otherwise, it is counted as less-than-3-fault-prone. Optimal coefficients found by the ABC, *p*-value, and threshold values derived for each metric in this case study are shown in Table 9.

As can be seen from the *p*-value column, all metrics are statistically significant. All slope coefficients are positive. Eventually derived threshold values are shown in the last column.

Assessments of these thresholds are done by using 24 test datasets, altogether totaling 10,991 modules (classes). Note that ant-1.5, jedit-4.3, and poi-2.0 test sets include no more than three faulty modules, so we excluded these three datasets. Results are summarized in Table 10.

As can be seen from the last column, all metrics have satisfactory results. Results obtained in this case study are quite good when compared with the first case study. We can claim that metric values belonging to 3+faulty modules are more discriminative in light of our findings. Table 11 is informative with the aim of showing metrics' value differences by including the mean values of metrics for 1+faulty modules and 3+faulty modules. Note that this table was created by the training dataset. The biggest difference

Table 8
Results of assessments (Case study 1).

Metric	Min *g*-mean	Max *g*-mean	Std. Dev.	N(*g*-mean)>0.6	Avg *g*-mean
WMC	0.47	0.79	0.08	15	**0.62**
CBO	0.40	0.69	0.15	16	**0.60**
RFC	0.52	0.77	0.07	17	**0.63**
LCOM	0.47	0.73	0.08	14	**0.60**
Ca	0.39	0.67	0.06	4	**0.54**
Ce	0.51	0.80	0.08	14	**0.62**
NPM	0.43	0.70	0.07	14	**0.60**
AMC	0.44	0.75	0.08	9	**0.57**
MAX_CC	0.32	0.76	0.08	11	**0.57**
AVG_CC	0.39	0.69	0.07	11	**0.57**
LOC	0.48	0.74	0.07	19	**0.62**

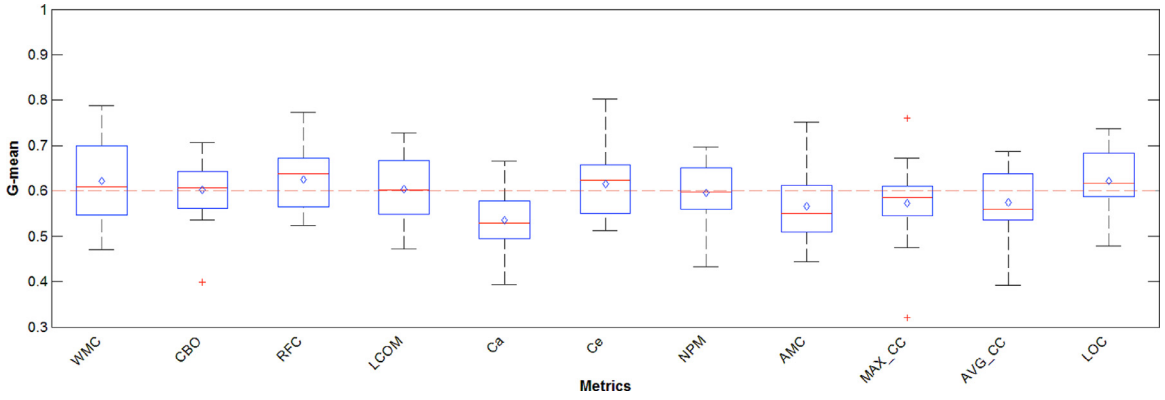

Fig. 6. Box plot visualization of results (Case study 1).

Table 10
Results of assessments (Case study 2).

Metric	Min *g*-mean	Max *g*-mean	Std. Dev.	N(*g*-mean)>0.6	Avg *g*-mean
WMC	0.41	0.85	0.11	20	**0.70**
CBO	0.54	0.82	0.07	22	**0.69**
RFC	0.60	0.82	0.06	24	**0.72**
LCOM	0.47	0.84	0.11	14	**0.65**
Ca	0.35	0.72	0.08	14	**0.61**
Ce	0.51	0.83	0.08	20	**0.68**
NPM	0.46	0.83	0.11	13	**0.66**
AMC	0.43	0.80	0.08	14	**0.62**
MAX_CC	0.53	0.76	0.08	14	**0.64**
AVG_CC	0.50	0.79	0.07	17	**0.64**
LOC	0.61	0.77	0.05	24	**0.71**

Table 11
Means of metric values of 1+faulty modules and 3+faulty modules.

Mean/Metric	WMC	CBO	RFC	LCOM	Ca	Ce	NPM	AMC	MAX_CC	AVG_CC	LOC
1+faulty	14.84	13.52	40.10	161.77	7.16	7.44	11.22	26.68	5.78	1.51	431.61
3+faulty	21.81	20.23	66.29	374.95	11.07	11.23	15.40	32.56	8.66	1.86	741.88

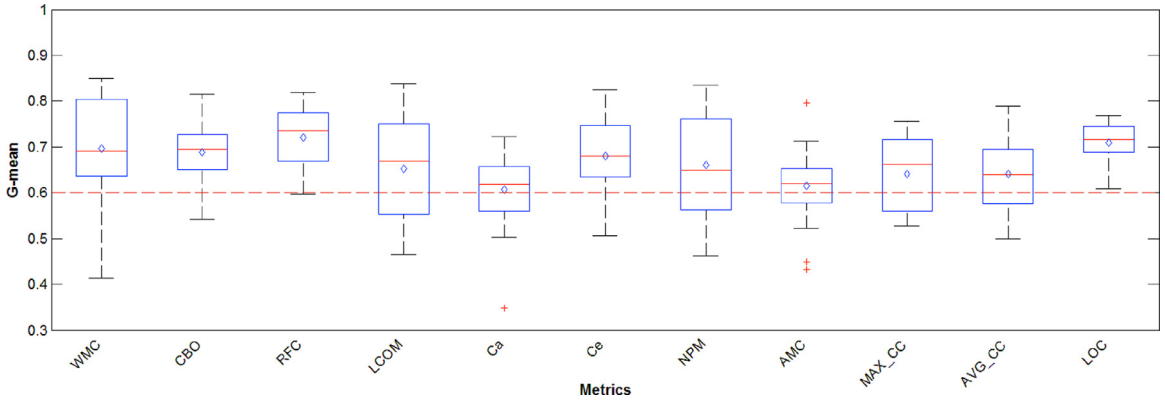

Fig. 7. Box plot visualization of results (Case study 2).

is in the LOC metric, including an average of 431 lines and 741 lines of code, respectively, for both module categories. Returning to discussions pertaining to Table 9, the average g-mean is 0.72 for the RFC metric, while the average is 0.71 for the LOC. The poorest performance again belongs to the Ca metric, but, nevertheless, it is also above the predefined limit with a 0.61 average g-mean.

Box plots visualization of results is shown in Fig. 7.

All 24 test sets give g-mean results above 0.6 for RFC and LOC metric thresholds. In other words, the thresholds of RFC and LOC metrics tend to be more generalized, so can be applied to other similar software systems less tentatively. CBO, WMC and Ce metrics, respectively, come after. This stability order is almost the same as with the first case study. When looking from a test sets perspective, some give poor results against thresholds, but some give excellent results. For example, log4j-1.2 has an average of a 0.57 *g*-mean performance, whereas log4j-1.1 has an average of 0.76. This finding is noticeable, since different versions of the same software system react in a very different way against the same thresholds. After analyzing other results further, we observed that different versions of the same software systems yielded similar results. This implies that log4j-1.2 is an exception, and possible reasons for this situation are addressed in the next section.

Table 12
Derived thresholds from Case study 1 and Case study 2.

Metric/Threshold	WMC	CBO	RFC	LCOM	Ca	Ce	NPM	AMC	MAX_CC	AVG_CC	LOC
THR^1	7	8	20	7	–	4	5	–	–	–	114
THR^3	12	8	30	21	2	7	7	17	4	1	171

Table 13
Findings of the original and replicated studies.

	Dataset	Task	Thresholds			*g*-mean		
			RFC	CBO	WMC	RFC	CBO	WMC
Original study	Eclipse	Within-project	40	9	20	0.635	0.588	0.605
Replicated study	See Table 2	Generalized	20	8	7	0.635	0.600	0.630

7.3. Overview of derived thresholds

Validated threshold sets (Threshold-1 and Threshold-3) from case study 1 and case study 2 are summarized in Table 12. In practice, suggested thresholds, except AVG_CC, should be integer values, but not real values. Therefore, derived thresholds are rounded to the nearest integer. Differences between THR^1 and THR^3 seem reasonable with only the exception of the CBO metric. Both thresholds for CBO are the same, at 8. However, THR^3 outperforms THR^1 in terms of the g-mean, at 0.69 and 0.60, respectively (see Table 10 and Table 8). Therefore, a module having CBO greater than 8 is more likely to return at least three faults (3+fp). On the other hand, a module having more than 114 LOC is likely to have at least one fault (1+fp), whereas if it consists of more than 171 LOC, it is likely to have at least three faults (3+fp). Similar remarks can be made for the other metrics.

7.4. Comparison of results with the original study

General features and findings of the original study and this study are presented in Table 13. In the original study, only three metrics were included in the threshold case study. Therefore, only the results of these metrics from our first case study are transferred to the table. Eclipse was used as a dataset in the original study. It is a large-scale Integrated Development Environment (IDE) software with a 4454 class-size developed by the IBM Corporation. Its features are different than the grouping characteristics of our datasets. Further, the task of the original study was to investigate within-project thresholds, whereas the task in this study is to reach generalized thresholds for one group of open-source software systems. For these reasons, derived thresholds of both studies cannot be compared. An interesting finding is that our g-mean results are better than the original study's despite the performance superiority of within-project predictors in several previous studies (Briand, Melo, & Wust, 2002; Zimmermann et al., 2009).

8. Threats to validity

There are some possible threats that may affect our study and the results of the case studies. These threats are as follows:

1. Metrics related to software modules can be collected by different quality tools by considering formulas and definitions of metrics. However, as a dependent variable, the numbers regarding bug data are reported and recorded by test teams or end users, which makes this process subjective. Testers or end users may not catch all of the bugs during use or they may miss recording one even when it arises. Some software systems, by nature, are exposed to intense usage through all of their features, while others not. This situation is a serious threat to seeing the association between dependent and independent variables. It can cause a potentially available pattern of a dataset to be missed.
2. Another threat is about not reporting all relevant bugs. In the case of finding a tiny bug, the tester may contact the developer directly without recording it, especially in small project teams. Therefore, a reliable prediction model may not be created.
3. Empirical results have indicated that a threshold obtained from one system may not be applied to others because of differences in systems (Briand et al., 2002). A good threshold might change depending on an organization's structure, programming language, tools used, and developers' practice and experience (Herbold et al., 2011). It is found that project-specific context is an issue which should be considered during model development (Dejaeger, Verbraken, & Baesens, 2013). In our case studies, we merged datasets from different open-source software systems to create a single training dataset in order to overcome this threat, and consequently to reach thresholds which can yield generalized satisfactory results for a particular software group. However, our case studies do not also include software systems from a large set of environments and other programming languages (we included only Java). Thus, thresholds derived in this study might not show the same performance when applied to a different environment having dissimilar characteristics to our pre listed ones.
4. Recommended value for *g*-mean limit is 0.6 in this study, based on references listed in previous sections. This limit value assumption affects our drawing conclusions about thresholds. Other researchers may use different limit values in their studies, as there is no consensus among researchers.

9. Conclusion

Software quality metrics reflect the quality of the software quantitatively. They support the management of a project, the development of software, and, especially, the testing of software. Software managers can view the drawbacks of a software before delivering it to the customer. Software developers can see classes or modules that require revising, and consequently this contributes incrementally to the quality of these modules and the decrement in their risk level. Testers can identify modules that need more testing effort and can prioritize modules according to their quality. In this study, we investigated whether or not there are effective threshold values for each metric that supports a defining risk categorization of each module. First, we defined three different risk categories: non-fault-prone (0fp), fault-prone (1+fp), and three-or-more-fault-prone (3+fp). We executed two independent case studies in order to derive two thresholds that support defined

categories of modules. One of the primary objectives of this study has been to achieve generalized thresholds that can be applied to different software systems successfully. Therefore, a total of 37 versions from 10 open-source software systems were used as datasets. A single training set was created by merging 10 of them and derived thresholds were tested on the remaining 27 datasets. We suggested two thresholds for a number of software quality metrics studied in this paper, aiming to support project managers, software developers, testers, and customers.

10. Future works

Future works may include the following issues:

- Derived thresholds can be tested and analyzed in a wider range of datasets, including software systems implemented by different languages (C++, C#, etc.). Furthermore, the effect of these thresholds can be assessed on large scale closed industrial projects other than open-source software systems.
- The same threshold derivation model can be applied to method-level metrics to also suggest thresholds for structured languages.
- The threshold derivation model can be extended to explore correlations between metrics, rather than considering them individually (e.g., deriving different thresholds for CBO metrics corresponding to various LOC ranges).

Appendix

Table A1.

Table A1
Software quality metrics used in this study (Jureczko & Spinellis, 2010).

Short Name	Metric Group	Explanation
WMC	CK (Chidamber & Kemerer, 1994)	*Weighted Methods per Class:* Sum of complexities of all methods within a class. When complexity of each method is assumed to be 1, then this metric value is measured by counting number of method in that class.
DIT	CK	*Depth of Inheritance Tree:* Maximum path length from the current class to its top parent class.
NOC	CK	*Number of Children:* Number of children classes inherited from the current class.
CBO	CK	*Coupling Between Object Classes:* Number of classes coupled to the current class (Afferent couplings + Efferent couplings). A coupling can occur via a method call, a variable access, or a return type.
RFC	CK	*Response For a Class:* Number of methods executed by the current class in response to a potentially received message by the object of that class. For simplicity, this metrics value is measured by summing number of local methods of that class and number of methods called by local methods.
LCOM	CK	*Lack of Cohesion in Methods:* Cohesion of a class indicates how strongly methods of that class are related with each other. Some pairs of methods within a class share at least one common variable, on the hand, some pairs does not share any common variable. This metric is then calculated by subtracting number of latter pairs from number of former pairs. High values of this metric show that there is some separable functionality within that class.
Ca	Martin (1994)	*Afferent couplings:* Number of classes that depend on the current class. (also called as inward couplings)
Ce	Martin (1994)	*Efferent couplings:* Number of classes that current class depends on. (also called as outward couplings)
LCOM3	Henderson- Sellers (1996)	*Lack of Cohesion in Methods:* It indicates cohesion strength of the class according to the following formula: *LCOM3 = (m - sum(mA)/a)/(m-1)* *m: number of methods in the class* *a: number of variables in the class* *mA: number of methods that access a variable* *sum(mA): sum of mA over all variables* Its value varies between 0 and 2. The value of 0 indicates good cohesion and 1 indicates bad cohesion. A value between 1 and 2 happens when some variables are never used or used by other classes (Zimmermann et al., 2009).
NPM	QMOOD (Bansiya & Davis, 2002)	*Number of Public Methods:* Number of public methods within the current class. It is measured simply by counting methods that are declared with "public" keyword.
DAM	QMOOD	*Data Access Metric:* It is the ratio of the number of private and protected attributes to the total number of attributes within the class. It measures encapsulation rate of a class.
MOA	QMOOD	*Measure of Aggregation:* It is the ratio of number of user-defined class types to the number of standard attribute types (int, double, char, etc.)
MFA	QMOOD	*Measure of Functional Abstraction:* It is the ratio of number of methods inherited from parent class to the total number of methods of that class (including methods from inherited parent classes)
CAM	QMOOD	*Cohesion Among Methods of Class:* It measures relatedness among methods of a class based upon the parameter list of the methods.
IC	Tang (Tang, Kao, & Chen, 1999)	*Inheritance Coupling:* It is measured by counting the parent classes to which the current class is coupled. The class is coupled to its parent class if one the following case occurs: - Inherited method invokes a redefined method - Inherited method uses at least one attribute, belonging to another method which is new or redefined - Inherited method is invoked by another method which is new or redefined
CBM	Tang	*Coupling Between Methods:* It is measured by counting number of new/redefined methods to which inherited methods are coupled. Coupling occurs when one of the three conditions, explained in IC, is held.

(*continued on next page*)

Table A1 (*continued*)

Short Name	Metric Group	Explanation
AMC	Tang	*Average Method Complexity:* It is measured by averaging size of methods within the class. Size of a method equals java binary codes in the method.
MAX_CC	McCabe (1976)	*Maximum Cyclomatic Complexity:* Cyclomatic Complexity (CC) is measured by counting the number of different paths within a method (plus one, method itself). It is a method-level metric. MAX_CC, AVG_CC are derived from CC to create class-level metrics from it. MAX_CC is CC of the method which has maximum value among methods of the current class.
AVG_CC	McCabe	*Average Cyclomatic Complexity:* AVG_CC is measured by taking arithmetic mean of all CC values of the current class.
LOC		*Lines of Code:* It is simply lines of java binary source code of the class.

References

Alves, T. L., Ypma, C., & Visser, J. (2010). Deriving metric thresholds from benchmark data. In *2010 IEEE International Conference on Software Maintenance* (pp. 1–10). IEEE.

Arar, Ö. F., & Ayan, K. (2015). Software defect prediction using cost-sensitive neural network. *Applied Soft Computing, 33*, 263–277.

Ayan, K., & Kılıç, U. (2012). Artificial bee colony algorithm solution for optimal reactive power flow. *Applied Soft Computing, 12*(5), 1477–1482.

Ayan, K., Kılıç, U., & Baraklı, B. (2015). Chaotic artificial bee colony algorithm based solution of security and transient stability constrained optimal power flow. *International Journal of Electrical Power & Energy Systems, 64*, 136–147.

Bansiya, J., & Davis, C. G. (2002). A hierarchical model for object-oriented design quality assessment. *IEEE Transactions on Software Engineering, 28*(1), 4–17.

Basili, V. R., Briand, L. C., & Melo, W. L. (1996). A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering, 22*(10), 751–761.

Baxter, G., Frean, M., Noble, J., Rickerby, M., Smith, H., & Al. , E. (2006). Understanding the shape of Java software. *ACM SIGPLAN Notices, 41*, 397–412.

Bender, R. (1999). Quantitative risk assessment in epidemiological studies investigating threshold effects. *Biometrical Journal, 41*(3), 305–319.

Briand, L. C., Melo, W. L., & Wust, J. (2002). Assessing the applicability of fault-proneness models across object-oriented software projects. *IEEE Transactions on Software Engineering, 28*(7), 706–720.

UCLA: Statistical Consulting Group. Retrieved November 24, 2007, from Bruin, J.. *Introduction to SAS*. 2006 http://www.ats.ucla.edu/stat/sas/notes2/.

Canfora, G., De Lucia, A., Di Penta, M., Oliveto, R., Panichella, A., & Panichella, S. (2013). Multi-objective cross-project defect prediction. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation* (pp. 252–261). IEEE.

Carver, J. (2010). Towards reporting guidelines for experimental replications: A proposal. In *Proceedings of the 1st international workshop on replication in empirical software engineering research* (pp. 2–5).

Catal, C., Alan, O., & Balkan, K. (2011). Class noise detection based on software metrics and ROC curves. *Information Sciences, 181*(21), 4867–4877.

Catal, C., & Diri, B. (2009). A systematic review of software fault prediction studies. *Expert Systems with Applications, 36*(4), 7346–7354.

Catal, C., & Diri, B. (2009). Investigating the effect of dataset size, metrics sets, and feature selection techniques on software fault prediction problem. *Information Sciences, 179*(8), 1040–1058.

Chidamber, S. R., & Kemerer, C. F. (1994). A metrics suite for object oriented design. *IEEE Transactions on Software Engineering, 20*(6), 476–493.

Cukic, B., & Singh, H. (2004). Robust prediction of fault-proneness by random forests. In *15th International Symposium on Software Reliability Engineering* (pp. 417–428). IEEE.

Dejaeger, K., Verbraken, T., & Baesens, B. (2013). Toward comprehensible software fault prediction models using bayesian network classifiers. *IEEE Transactions on Software Engineering, 39*(2), 237–257.

Dick, S., Meeks, A., Last, M., Bunke, H., & Kandel, A. (2004). Data mining in software metrics databases. *Fuzzy Sets and Systems, 145*(1), 81–110.

Elish, K. O., & Elish, M. O. (2008). Predicting defect-prone software modules using support vector machines. *Journal of Systems and Software, 81*(5), 649–660.

Erni, K., & Lewerentz, C. (1996). Applying design-metrics to object-oriented frameworks. In *Proceedings of the 3rd International Software Metrics Symposium* (pp. 64–74). IEEE Comput. Soc. Press.

Fenton, N. E., & Pfleeger, S. L. (1998). *Software Metrics: A Rigorous and Practical Approach*. London: PWS Publishing Co.

Ferreira, K. A. M., Bigonha, M. A. S., Bigonha, R. S., Mendes, L. F. O., & Almeida, H. C. (2012). Identifying thresholds for object-oriented software metrics. *Journal of Systems and Software, 85*(2), 244–257.

Flottau, J., & Osborne, T. (2015, May 15). *Aerospace Daily & Defense Report*. Retrieved November 27, 2015, from Flottau, J., & Osborne, T.. *Software cut off fuel supply in stricken A400M*. 2015, May 15 http://aviationweek.com/defense/software-cut-fuel-supply-stricken-a400m.

Gao, K., Khoshgoftaar, T. M., Wang, H., & Seliya, N. (2011). Choosing software metrics for defect prediction: An investigation on feature selection techniques. *Software: Practice and Experience, 41*(5), 579–606.

Gyimothy, T., Ferenc, R., & Siket, I. (2005). Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Transactions on Software Engineering, 31*(10), 897–910.

Hall, T., Beecham, S., Bowes, D., Gray, D., & Counsell, S. (2011). A Systematic review of fault prediction performance in software engineering. *IEEE Transactions on Software Engineering, 38*(6), 1276–1304.

Halstead, M. H. (1977). *Elements of Software Science*. New York: Elsevier Science Inc.

He, Z., Shu, F., Yang, Y., Li, M., & Wang, Q. (2011). An investigation on the feasibility of cross-project defect prediction. *Automated Software Engineering, 19*(2), 167–199.

Henderson-Sellers, B. (1996). *Object-Oriented Metrics: Measures of Complexity*. Prentice Hall.

Herbold, S. (2013). Training data selection for cross-project defect prediction. In *Proceedings of the 9th International Conference on Predictive Models in Software Engineering - PROMISE '13* (pp. 1–10). New York: ACM Press.

Herbold, S., Grabowski, J., & Waack, S. (2011). Calculation and optimization of thresholds for sets of software metrics. *Empirical Software Engineering, 16*(6), 812–841.

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 - PROMISE '10* (p. 1). ACM Press.

Jureczko, M., & Spinellis, D. D. (2010). Using object-oriented design metrics to predict software defects. In *In Models and Methods of System Dependability* (pp. 69–81).

Karaboga, D., Akay, B., & Ozturk, C. (2007). In V. Torra, Y. Narukawa, & Y. Yoshida (Eds.). *Artificial Bee Colony (ABC) Optimization Algorithm for Training Feed-Forward Neural Networks*: Vol. 4617. Berlin, Heidelberg: Springer Berlin Heidelberg.

Karaboga, D., & Basturk, B. (2007). A powerful and efficient algorithm for numerical function optimization: Artificial bee colony (ABC) algorithm. *Journal of Global Optimization, 39*(3), 459–471.

Karaboga, D., & Ozturk, C. (2009). Neural networks training by artificial bee colony algorithm on pattern classification. *Neural Network World, 19*(3), 279–292.

Khoshgoftaar, T. M., Allen, E. B., Jones, W. D., & Hudepohl, J. I. (1999). Classification tree models of software quality over multiple releases. In *Proceedings 10th International Symposium on Software Reliability Engineering (Cat. No.PR00443)* (pp. 116–125). IEEE Comput. Soc.

Khoshgoftaar, T. M., & Seliya, N. (2003). Analogy-based practical classification rules for software quality estimation. *Empirical Software Engineering, 8*(4), 325–350.

Koru, A. G. (2005). Building defect prediction models in practice. *IEEE Software, 22*(6), 23–29.

Koru, A. G., & Liu, H. (2005). An investigation of the effect of module size on defect prediction using static measures. *ACM SIGSOFT Software Engineering Notes, 30*(4), 1–5.

Kubat, M., & Matwin, S. (1997). Addressing the curse of imbalanced training sets: one-sided selection. In *In Proceedings of the Fourteenth International Conference on Machine Learning* (pp. 179–186).

Liu, Y., Khoshgoftaar, T. M., & Seliya, N. (2010). Evolutionary optimization of software quality modeling with multiple repositories. *IEEE Transactions on Software Engineering, 36*(6), 852–864.

Malhotra, R., & Bansal, A. J. (2015). Fault prediction considering threshold effects of object-oriented metrics. *Expert Systems, 32*(2), 203–219.

Martin, R. (1994). OO design quality metrics – an analysis of dependencies. In *Proceedings of Workshop Pragmatic and Theoretical Directions in Object-Oriented Software Metrics*.

McCabe, T. (1976). A complexity measure. *IEEE Transactions on Software Engineering, 2*(4), 308–320.

Menzies, T., Caglayan, B., Kocaguneli, E., Krall, J., & Turhan, B. (2012). The promise repository of empirical software engineering data. *West Virginia University: Department of Computer Engineering*. http://promisedata.googlecode.com.

Menzies, T., Greenwald, J., & Frank, A. (2007). Data mining static code attributes to learn defect predictors. *IEEE Transactions on Software Engineering, 33*(1), 2–13.

Menzies, T., Milton, Z., Turhan, B., Cukic, B., Jiang, Y., & Bener, A. (2010). Defect prediction from static code features: Current results, limitations, new approaches. *Automated Software Engineering, 17*(4), 375–407.

Menzies, T., Turhan, B., Bener, A., Gay, G., Cukic, B., & Jiang, Y. (2008). Implications of ceiling effects in defect predictors. In *Proceedings of the 4th international workshop on Predictor models in software engineering – PROMISE '08* (p. 47). ACM Press.

Michaels, P. (2008). *Faulty software can lead to astronomic costs. Retrieved November 27, 2015, from*. 2008 http://www.computerweekly.com/opinion/Faulty-software-can-lead-to-astronomic-costs.

Nagappan, N., Ball, T., & Zeller, A. (2006). Mining metrics to predict component failures. In *Proceeding of the 28th international conference on Software engineering - ICSE '06* (p. 452). ACM Press.

Nickerson, A. S., Japkowicz, N., & Milios, E. (2001). Using unsupervised learning to guide resampling in imbalanced data sets. In *Proceedings of the Eighth International Workshop on AI and Statitsics* (pp. 261–265).

Olague, H. M., Etzkorn, L. H., 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 Transactions on Software Engineering, 33*(6), 402–419.

Oliveira, P., Valente, M. T., & Lima, F. P. (2014). Extracting relative thresholds for source code metrics. In *2014 Software Evolution Week – IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)* (pp. 254–263). IEEE.

Padberg, F., Ragg, T., & Schoknecht, R. (2004). Using machine learning for estimating the defect content after an inspection. *IEEE Transactions on Software Engineering, 30*(1), 17–28.

Pérez-Miñana, E., & Gras, J.-J. (2006). Improving fault prediction using Bayesian networks for the development of embedded software applications. *Software Testing, Verification and Reliability, 16*(3), 157–174.

Radjenović, D., Heričko, M., Torkar, R., & Živkovič, A. (2013). Software fault prediction metrics: a systematic literature review. *Information and Software Technology, 55*(8), 1397–1418.

Rosenberg, L., Stapko, R., & Gallo, A. (1999). Object-oriented metrics for reliability. *Presentation at IEEE International Symposium on Software Metrics*.

Sánchez-González, L., García, F., Ruiz, F., & Mendling, J. (2012). A study of the effectiveness of two threshold definition techniques. In *16th International Conference on Evaluation & Assessment in Software Engineering (EASE 2012)* (pp. 197–205). IET.

Selby, R. W., & Porter, A. A. (1988). Learning from examples: Generation and evaluation of decision trees for software resource analysis. *IEEE Transactions on Software Engineering, 14*(12), 1743–1757.

Seliya, N., Khoshgoftaar, T. M., & Hulse, J. Van. (2010). Predicting faults in high assurance software. In *2010 IEEE 12th International Symposium on High Assurance Systems Engineering* (pp. 26–34). IEEE.

Shatnawi, R. (2010). A quantitative investigation of the acceptable risk levels of object-oriented metrics in open-source systems. *IEEE Transactions on Software Engineering, 36*(2), 216–225.

Shatnawi, R. (2015). Deriving metrics thresholds using log transformation. *Journal of Software: Evolution and Process, 27*(2), 95–113.

Shatnawi, R., & Althebyan, Q. (2013). An empirical study of the effect of power law distribution on the interpretation of oo Metrics. *ISRN Software Engineering, 2013*, 1–18.

Shatnawi, R., Li, W., Swain, J., & Newman, T. (2010). Finding software metrics threshold values using ROC curves. *Journal of Software Maintenance and Evolution: Research and Practice, 22*(1), 1–16.

Spinellis, D. (2005). Tool writing: a forgotten art? *IEEE Software, 22*(4), 9–11.

Tang, M.-H., Kao, M.-H., & Chen, M.-H. (1999). An empirical study on object-oriented metrics. In *Proceedings Sixth International Software Metrics Symposium (Cat. No.PR00403)* (pp. 242–249). IEEE Comput. Soc.

Turhan, B., Tosun Mısırlı, A., & Bener, A. (2013). Empirical evaluation of the effects of mixed project data on learning defect predictors. *Information and Software Technology, 55*(6), 1101–1118.

Wang, S., & Yao, X. (2013). Using class imbalance learning for software defect prediction. *IEEE Transactions on Reliability, 62*(2), 434–443.

Williamson, D. F., Parker, R. A., & Kendrick, J. S. (1989). The box plot: a simple visual method to interpret data. *Annals of Internal Medicine, 110*(11), 916–921.

Zheng, J. (2010). Cost-sensitive boosting neural networks for software defect prediction. *Expert Systems with Applications, 37*(6), 4537–4543.

Zhou, Y., & Leung, H. (2006). Empirical analysis of object-oriented design metrics for predicting high and low severity faults. *IEEE Transactions on Software Engineering, 32*(10), 771–789.

Zimmermann, T., Nagappan, N., Gall, H., Giger, E., & Murphy, B. (2009). Cross-project defect prediction. In *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering on European software engineering conference and foundations of software engineering symposium – E* (p. 91). ACM Press.