Using Metrics for Risk Prediction in Object-Oriented Software: A Cross-Version Validation

Salim Moudache*, Mourad Badri

Software Engineering Research Laboratory, Department of Mathematics and Computer Science, University of Quebec, Trois-Rivières, Canada

* Corresponding author. Tel.: +1 (438) 725-4259; email: salim.moudache@uqtr.ca

Manuscript submitted January 17, 2021; accepted May 11, 2021.
doi: 10.17706/jsw.17.1.1-20

Abstract: This work aims to investigate the potential, from different perspectives, of a risk model to support Cross-Version Fault and Severity Prediction (CVFSP) in object-oriented software. The risk of a class is addressed from the perspective of two particular factors: the number of faults it can contain and their severity. We used various object-oriented metrics to capture the two risk factors. The risk of a class is modeled using the concept of Euclidean distance.

We used a dataset collected from five successive versions of an open-source Java software system (ANT). We investigated different variants of the considered risk model, based on various combinations of object-oriented metrics pairs. We used different machine learning algorithms for building the prediction models: Naive Bayes (NB), J48, Random Forest (RF), Support Vector Machines (SVM) and Multilayer Perceptron (ANN). We investigated the effectiveness of the prediction models for Cross-Version Fault and Severity Prediction (CVFSP), using data of prior versions of the considered system. We also investigated if the considered risk model can give as output the Empirical Risk (ER) of a class, a continuous value considering both the number of faults and their different levels of severity. We used different techniques for building the prediction models: Linear Regression (LR), Gaussian Process (GP), Random forest (RF) and M5P (two decision trees algorithms), SmoReg and Artificial Neural Network (ANN).

The considered risk model achieves acceptable results for both cross-version binary fault prediction (a g-mean of 0.714, an AUC of 0.725) and cross-version multi-classification of levels of severity (a g-mean of 0.758, an AUC of 0.771). The model also achieves good results in the estimation of the empirical risk of a class by considering both the number of faults and their levels of severity (intra-version analysis with a correlation coefficient of 0.659, cross-version analysis with a correlation coefficient of 0.486).

Key words: Cross-version validation, fault proneness, fault severity, machine learning algorithms, object-oriented metrics, prediction, risk.

1. Introduction

Nowadays, software engineering needs are growing more and more knowing the increasing complexity of developed applications, which increases the risk of faults and their severity. The presence of faults in a software system not only degrades its quality, but also increases its development and maintenance costs [1]. To prevent these faults, particularly the most severe ones, and ensure having high quality software, software systems need to be rigorously tested. However, software testing often has to be done under severe pressure due to limited resources and tight time deadlines constraints. In addition, exhaustive testing is cost prohibitive and would take too much time and resources. So, it is typically not feasible, except perhaps in extremely trivial cases [2]. Hence, software tests prioritization [3]-[5] appears as the best solution.

Many studies were conducted in the area of Software Quality Assurance addressing this problematic. A trivial solution proposed was random testing [6]. This method could be described as naive, and it is not accurate enough to cover all the faults [7]. Another widely active proposal was software fault prediction [8]-[12]. Many studies focused on the investigation of the simple research question: Is a given class fault-prone? So, they proposed different binary classification models. The weakness of the use of binary classification models to support the construction (selection) of test cases is that it provides just simple information about whether a given class is fault-prone or not [12], what we denote as P, the probability of a class to be fault-prone. We believe, in fact, that fault severity is also an important information that must be considered in the construction of fault-proneness prediction models [12], [13]. Recently, some studies have addressed the prediction of fault severity levels. Fault severity measures the impact of a fault on a system and its users [9], [14], what we denote as I. The output of these studies is severity-based fault prediction models [15] that can classify a fault into different levels of severity, for example: low or high. This is much more useful than simple binary classification models. In fact, severity-based prediction models allow much more effective testing by predicting (and focusing testing efforts on) relatively high-risk classes. However, they do not consider the number of faults, which in turn can affect the testing efforts allocation.

In a previous work [16], we addressed the risk of a class from the perspective of two factors: the number of faults it can contain and their severity. We investigated different variants of the metrics-based risk model proposed in [17], which is based on the concept of Euclidean distance. The logic behind this 2D-model for the choice of the metrics is: one axe will be represented by a metric that better describes P (probability of fault-proneness), and another axe that will be represented by a metric that better describes I (severity of faults). We investigated two research questions: (1) Is the considered risk model (variants) suitable for software fault prediction? and (2) Can the risk model (variants) predict different levels of fault severity? We considered two levels of severity: high and normal. Results showed that the risk model achieves good results for both binary fault prediction (with a g-mean of 0.821) and multi-classification of severity levels (with a g-mean of 0.827).

In this study, we explored the effectiveness of the risk model (variants) for Cross-Version Fault and Severity Prediction (CVFSP), using data of prior versions of the considered system (ANT - five successive versions). We investigated the three following research questions:

RQ1: Can a risk model built for a previous version of a given system be reused on its next version to predict fault-prone classes?

Here, we wanted to perform cross-version fault-proneness prediction using the risk model to investigate if its eventual use in a real-life context will be accurate.

RQ2: Can a risk model built for a previous version of a given system be reused on its next version to predict fault severity?

Here, we wanted to perform cross-version severity level prediction using the risk model to investigate if its eventual use in a real-life context will be accurate.

RQ3: Can the risk model be used to predict the empirical risk of a class, a continuous value considering both the number of faults the class can contain and their severity?

As mentioned previously, we investigated in a previous work [16] two research questions: (1) Is the considered risk model suitable for software fault prediction? and (2) Can the risk model predict different levels of fault severity? Here, we aimed to investigate if the considered risk model can give valuable insight about faults severity levels without losing information about the number of faults that a given class can contain. If yes, this will give a serious advantage to the considered risk model compared to the prediction models proposed in the literature which predict either fault-prone classes, their number of faults or their severity. This will be much more useful for tests prioritization by predicting (and focusing testing efforts on) high-risk classes. We defined a software risk prediction methodology. We investigated the risk model in two different situations: (1) intra-version validation, and (2) cross-version validation.

The rest of the paper is organized as follows. Section 2 presents a summary of different studies related to software fault prediction. Section 3 presents the methodology we followed in this study. First, we present some descriptive statistics about the dataset we used to conduct our experiments, and an overview of the software metrics we used to build our different risk models (variants). Second, we explain how each experiment was conducted and how the different models were evaluated. Section 4 presents the different results obtained. Section 5 discusses the possible threats to validity of our study. Finally, Section 6 concludes this paper by summarizing the major contributions of this study and giving some future work directions.

2. Related Work

Shatnawi presented in [18] a study to determine the thresholds values for the CK (Chidamber and Kemerer) [19] metrics beyond which a given class would present a risk to be fault-prone. He led an empirical analysis using three software systems inspired by an epidemiological risk assessment model [20], to calculate the values of acceptable thresholds for each metric. The author performed a logistic regression analysis to restrict the CK suite to the complexity and coupling metrics WMC (Weighted Methods per Class), RFC (Response for a Class) and CBO (Coupling between Objects) as the more correlated with the probability of faults. After determining the thresholds values on version 2.0 of Eclipse, the author made a validation over version 2.1 by building a model (as a regression tree) and crossed the experimental results with the actual results. He showed the good ability of the model to predict faults using the determined thresholds.

In addition, semi-supervised prediction models were used. This kind of models require less fault data than supervised ones. Lu *et al*. in [21], [22], have investigated the use of semi-supervised learning for software fault-proneness prediction with Random Forest and Dimension Reduction techniques. They showed that reducing the dimensionality of the source code metrics significantly improved the semi-supervised learning model. Moreover, Catal explored different semi-supervised classification algorithms for fault prediction [23]. He compared four methods, namely: Support Vector Machines, Class Mass Normalization, Low-Density Separation and Expectation-Maximization. He concluded that Low-Density Normalization gave the best results for large datasets but could also be used for small ones.

On the other hand, unsupervised learning models come across this shortcoming since they do not require any fault data. In [24], Bishnu & Bhattacherjee used a similar approach as Catal et al. in [3], [25], using K-means algorithm and the same threshold values to predict faulty modules. However, they used the Quad-Tree algorithm combined to a genetic algorithm to initialize the clusters used in the K-means algorithm. According to the authors, the classification performance of their model is as good as to the results obtained with supervised models which are built with fault data. Boucher and Badri investigated in [26] three threshold calculation techniques that can be used for fault-proneness prediction: ROC Curves [27], VARL (Value of an Acceptable Risk Level) [20] and Alves rankings [28]. They stand that ROC curves gives the best performance, but Alves Ranking is a good choice too. The advantage of Alves Rankings over ROC Curves technique is that it is completely unsupervised, so it does not require fault data and could be used when it is not available. In [29], the same authors adapted the HySOM model [30], originally working at function-level granularity, to work at class-level granularity for Object-Oriented Software.

Moreover, several studies have shown that severity-based fault proneness prediction models are much more useful than binary classification models that simply determine whether a module is fault-prone or not. E. Hong [31] proposed new severity-based prediction models using two module severity metrics: MS (Module Severity) and MSD (Module Severity Density). The proposed models are different from the previous ones [15], [32], [33] by assigning a severity value to the model rather than treating each severity level alone. Through an empirical study using the JM1 and PC4 projects from the NASA[1] dataset, they concluded that the MS models outperform the previous prediction models, and that the MSD models show good performance with JM1. Besides, multi-layer perceptron neural network showed the best performance among the three classification algorithms used for model building.

Recently, new techniques were proposed such as dynamic tracking fault diagnosis [34]. In [35], the authors showed the gain in baseline predictors combination in comparison to standalone baseline predictors. They conducted an empirical study, comparing ten ensemble predictors with baseline predictors and demonstrated the performance improvement of ensemble predictors. Another avenue, genetic based machine learning algorithms, was also explored [36], [37]. A well-known issue in software fault prediction is the imbalanced data. The authors in [38] proposed a new distance metric based on cost-sensitive learning to reduce the class imbalance. Their experimental results confirm the positive impact of their approach.

3. Research Methodology

3.1. Data Collection

We used the dataset, extracted from five successive versions of the ANT system (from version 1.3 to version 1.7), constructed by Touré [17] for his thesis. He actually used the QA-metrics plugging integrated with the Borland Together tool to calculate the object-oriented metrics, and Bugzilla for collecting faults and their levels of severity. Each class is described with the number of faults it contains and their severity, which can be either: blocker, critical, major, normal, minor or trivial. This dataset was used in multiple studies [39]-[42]. In fact, version 1.7 was widely used, but we decided to include the four previous versions as well in order to investigate fault-proneness and fault-severity prediction in a cross-version analysis context. ANT is a command-line tool developed in Java and mainly used for building Java applications [40]. Besides that, the dataset also includes a couple of object-oriented source code metrics as defined by Chidamber & Kemerer [19] in addition to Ca (Afferent Coupling), Ce (Efferent Coupling) and Fan-in. For this reason, our choice was naturally focused on these metrics to conduct our study. The selected metrics can be subdivided as follows:

3.1.1. Complexity metrics

WMC (Weighted Methods per Class): Is a sum weighted by the cyclomatic complexity of each method of a class [43]. If the value of this metric is high for a given class, the class will be more difficult to test and even to be understood [44].

3.1.2. Size metrics

LOC (Lines Of Code): Gives an idea on the size of a class in terms of the number of lines of code. The larger a class is, the more it is disposed to contain faults [45].

3.1.3. Coupling metrics

CBO (Coupling Between Objects): The coupling between objects describes the number of classes to which a given class is coupled and vice versa [46]. In fact, it is the sum of two other existing coupling metrics that are Fan-in and Fan-out [47]. It is used to evaluate the interdependence between classes of a same system.

Ca (Afferent Coupling): The afferent coupling counts the classes of a given package that depend on outer classes [48].

Ce (Efferent Coupling): The efferent coupling, unlike Ca, represents the number of outer classes that depend on classes in a given package [48].

Fan-in: Gives the number of module calls for a function / method for a given class [49]. This measure is mainly used to identify modules that require restructuring and high testing effort [50].

RFC (Response For a Class): Is a metric of both complexity and coupling at the same time. It is defined by

[1] NASA data-sets available at PROMISE Software Engineering Repository

the number of methods that can be called by an instance of a class [19]. The RFC metric indicates the degree of communication between a class and the rest of the system through the invocations of external methods that it can potentially call [51]. RFC is closely related to the testing effort and to the complexity.

Table 1 describes the different versions of the ANT system we used and highlights the number of classes, faulty classes and the ratio of faulty classes before and after fault duplication (the -D stand for duplication, #C-D gives the number of classes after duplication and #F-D the number of faulty classes after duplication).

Looking back to the literature, we can notice that the number of faults was not often taken into account. However, it represents an important information with valuable insight. Thus, we decided to make use of it. To do so, we were inspired by a simple methodology called 'Fault Duplication', used among others by Shatnawi [52], Zhou and Leung [15], and Boucher and Badri [26], which involves duplicating each class containing more than one fault in the dataset. For example, if a class contains 3 faults, it will be present 3 times in the dataset, each one marked as containing one single fault. This allows taking into consideration the number of faults in the analysis without having to do much preprocessing.

Table 1. ANT Versions Statistics

System	#Classes	#Faulty	Ratio	#C-D	#F-D	Ratio
ANT 1.3	126	60	47.62%	201	135	67.16%
ANT 1.4	178	38	21.35%	190	50	26.21%
ANT 1.5	293	106	36.18%	366	179	48.91%
ANT 1.6	352	45	12.78%	368	61	16.58%
ANT 1.7	745	70	10.37%	776	101	13.01%

3.2. Risk Model

Starting from the data presented above, we built a set of variants of the risk model defined in F. Touré thesis [17]. The risk model is, in fact, built on Bernoulli's risk theory [53], which is widely used in project management. The advantage of this method is that it provides a simple way to assess the risk associated to the classes of an object-oriented software. The risk of a class is considered from two perspectives: the probability of the class to be fault-prone and its impact (in terms of severity) on the rest of the system. Reducing the quantified risk of a class asks to reduce one or both axes that makes it up. We selected different combinations of metrics respecting the following disposition: (1) by taking m_P as a metric that better describes fault probability (many studies e.g., [15], [54]-[56] stand that size and complexity metrics play this role), and (2) m_I as a metric that better describes fault severity (many studies e.g., [50], [57], [58] indicate that coupling metrics better fulfill this aim).

Considering a system of k classes, $K \in \mathbb{N}^2$ and $\forall\, i \in k$, the theoretical risk (TR) of a given class C is calculated as follows:

$$TR_{m_P,m_I}(C) = \sqrt{\left(m_P(C) - max_i\left(m_P(C_i)\right)\right)^2 + \left(m_I(C) - max_i\left(m_I(C_i)\right)\right)^2}$$

Fig. 1 illustrates the two-dimensional risk model. The x-axis represents the metric mP and the y-axis represents the metric m_I. Point M represents the maximum of the two metrics and is calculated as follows:

$$M = \left(max\left(m_P(C_i)\right), max\left(m_I(C_i)\right)\right)$$

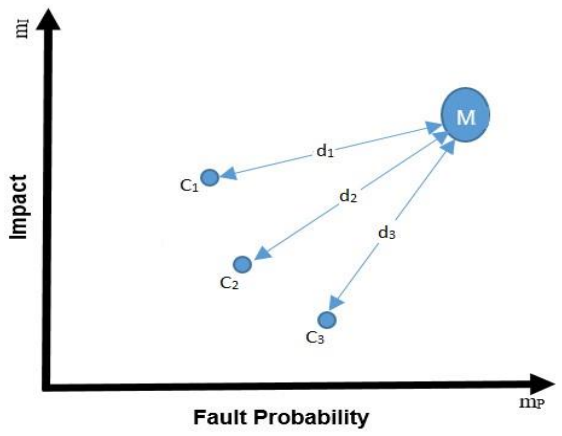


Fig. 1. Risk model.

3.3. Pre-processing

Data pre-processing [59], [60], which allows to obtain a suitable dataset for the mining stage, includes: cleaning, normalization, transformation, feature extraction, selection and so on. Data pre-processing stage could take up to 80% of the time and resources of a data science project. Fortunately, there are some tools that could be used to avoid GIGO (Garbage IN Garbage Out) situations in the form of filters. In terms of data pre-processing, the following operations were performed on each ANT version:

- Binarization of the number of faults of a given class in a new attribute called 'BugsBinary' i.e. 0 means not faulty and 1 means that the class contains one fault or more.
- Grouping [15], [61], [62]: blocker, critical and major faults will form the high severity fault level. On the other hand, normal, minor and trivial faults will form the normal severity fault level.
- For the sake of compatibility when using certain machine learning algorithms, the transformation filter in nominal type has been applied to the class attribute to predict.
- In order to improve the results of the prediction, normalization has been carried out on the models to obtain values in the range [-1, 1].
- We also explored the contribution of stratification techniques such as oversampling by applying the smote filter to 35 – 65% as advised in [63] and the duplication of faulty classes.

3.4. Machine Learning

In this section, we present a brief description of the machine learning algorithms we used from the Weka-API [64]. Machine learning algorithms are commonly used in fault-proneness classification to learn relationships between source code metrics (attributes) and faults (the class to predict or the target). These algorithms are trained using the datasets, including faults. In our study, we used the following machine learning algorithms: Naive Bayes (NB), J48, M5P, Random Forest (RF), Support Vector Machine (SVM), SmoReg, Linear regression (LR), Gaussian Process (GP) and Multilayer Perceptron (ANN). These algorithms have been widely used in literature and showed good performances.

3.4.1. Naive bayes

The Bayes Network algorithm classifies the given instances (classes in our case) by building a directed graph, where risk models will be represented by nodes and their independencies as links, to sort classes as faulty or not [52]. Many studies have investigated the use of this algorithm [41], [52], [65]. It can be used in different variants. The most popular one is the Naive Bayes Network that we chose in our study.

3.4.2. Gaussian process

Gaussian process is a collection of random variables such that every finite collection has a multivariate normal distribution [66]. Various studies explored the use of this process for software fault prediction among other domains as well [67], [68].

3.4.3. Decision trees

The J48, M5P and Random Forest algorithms are used for building decision trees. These algorithms calculate how efficiently each attribute is in splitting the data (in our case as fault-prone or not in a prior experiment than into levels of severity). The resulting decision tree is easy to understand as it is self-explanatory [11]. Weka includes a tool for decision trees visualization. There are two main advantages of visual data exploration over data mining in addition to the direct involvement of the user [69]: it can easily deal with highly nonhomogeneous and noisy data, and visual data exploration is intuitive and requires no understanding of complex mathematical or statistical algorithms or parameters. This kind of algorithms were also used in numerous studies dealing with software fault prediction [11], [55], [65].

3.4.4. Support vector machine

The Support Vector Machine algorithm is founded on the statistical learning theory, which makes it perfect for both regression and classification. This algorithm allows less weight to individuals that are far away from the tendency. It is also suitable for data that do not follow a linear distribution [11]. Malhotra et al. in [41] describe it as the best machine learning algorithm to use. It was also used in other studies addressing fault-proneness prediction [11], [70]. SMOreg implements the support vector machine for regression.

3.4.5. Artificial neural network

Artificial Neural Networks (ANN) are various and widely used in several fields. In our case, a Multilayer Perceptron (feed forward ANN with back-propagation algorithm) is used, as in [70]. This specific ANN topology consists in having several layers of neurons, where each layer can have a different number of neurons. Each neuron of each layer is linked to the previous and next layer's neurons. The network is first trained using training data, and the back-propagation algorithm will update the neurons' weights. Several studies investigated this algorithm in fault-proneness prediction [41], [52], [55], [65].

3.5. Evaluation Method

The machine learning algorithms mentioned above were used to build prediction models for CVFSP using the considered risk model. We performed 10-fold cross-validation. To evaluate the prediction efficiency of our machine learning prediction models, we used the geometric mean (g-mean) and the AUC (Area Under the Curve) metrics, which can be easily calculated using the confusion matrix resulting from the classification. Table 2 gives an example of a confusion matrix.

Table 2. Confusion Matrix

Classified	Actual	
	Faulty	Not Faulty
Faulty	True Positives (TP)	False Positives (FP)
Not Faulty	False Negatives (FN)	True Negatives (TN)

3.5.1. G-mean

The g-mean metric was defined specifically for imbalanced data classification [41] and was also used in other studies on fault-proneness prediction [18], [41]. FPR (False Positive Rate) and FNR (False Negative Rate) are being calculated as follows:

$$FPR = \frac{FP}{FP + TN}$$

$$FNR = \frac{FN}{FN + TP}$$

In regards to g-mean calculation, in case of a binary classification, we need to calculate first the accuracy of positives (TPR) and the accuracy of negatives (TNR) [41] (which are the opposite metrics of FNR and FPR respectively). On the contrary to FPR and FNR, where lower is better, the higher are TPR, TNR and g-mean, better is the classification. The g-mean metric will be satisfying if both TPR and TNR are high, otherwise it will not. These metrics are calculated as follow:

$$TPR = 1 - FNR = \frac{TP}{TP + FN}$$

$$TNR = 1 - FPR = \frac{TN}{TN + FP}$$

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

In case of a multi-class classification, there is another formula for g-mean calculation [71] with the use of the recall for each class R_i, where k is the number of distinct classes. Table 3 describes a confusion matrix of a k-classes prediction, where C_i denotes the i^{th} class:

Table 3. Confusion Matrix Multi-Class

		Predicted	Class		
		C_1	C_2	...	C_k
True	C_1	n_{11}	n_{12}	...	n_{1k}
Class	C_2	n_{21}	n_{22}	...	n_{2k}
	.	.	.	...	.
	.	.	.	...	.
	C_k	n_{k1}	n_{k2}	...	n_{kk}

The recall of the i^{th} class denoted R_i is given by:

$$R_i = \frac{n_{ii}}{\sum_{j=1}^{k} n_{ij}}$$

Afterwards, g-mean can be calculated as:

$$g - mean = \left(\prod_{i=1}^{k} R_i\right)^{1/k}$$

It is common to use a correction Ɛ especially when the number of classes is consequent when $Ri = 0$, in our study we chosen Ɛ = 0.001. We followed the next g-mean interpretation [26] for our results:

- g-mean < 0.5 means no good classification.
- 0.5 ≤ g-mean < 0.6 means poor classification.
- 0.6 ≤ g-mean < 0.7 means fair classification.
- 0.7 ≤ g-mean < 0.8 means acceptable classification.
- 0.8 ≤ g-mean < 0.9 means good classification.
- 0.9 ≤ g-mean means outstanding classification.

3.5.2. AUC

The Receiver Operating Characteristic (ROC) analysis is used in classifiers evaluation [72]. The Area Under the Curve (AUC) 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 [72]. Hosmer and Lemeshow proposed the use of the following rules to evaluate the performance of classifiers [73]:

- $AUC = 0.5$ means no good classification (random classifier).
- $0.5 \le AUC < 0.6$ means poor classification.
- $0.6 \le AUC < 0.7$ means fair classification.
- $0.7 \le AUC < 0.8$ means acceptable classification.
- $0.8 \le AUC < 0.9$ means excellent classification.
- $0.9 \le AUC$ means outstanding classification.

4. Results and Discussion

4.1. Summary of Previous Work

We investigated in a previous work [16], as mentioned above, different variants of the risk model following various combinations of object-oriented metrics, namely: TR (LOC, Fan-in), TR (WMC, fan-in), TR (WMC, CBO), TR (LOC, CBO), TR (LOC, Ce), TR (LOC, Ca), TR (WMC, Ce), TR (WMC, Ca), TR (RFC, Ca), TR (LOC, WMC), TR (LOC, RFC) and finally TR (WMC, RFC). As the work presented in this paper is built on our previous work, we give in what follows a brief summary of obtained results.

In a first step (Is the considered risk model (variants) suitable for software fault prediction?), we have investigated the potential of the risk model (variants) to predict whether a given class is fault-prone or not. We explored the contribution of stratification methods. Here we give the statistics after Smote [74] filter application, an oversampling technique. -B aims for balanced, #C-B gives the number of classes after applying the Smote filter and #F-B the number of faulty classes after applying the same transformation. It can be seen that we did not touch either version 1.3 or version 1.5 because the average of faulty classes is already beyond 35%. As explained in the preprocessing subsection, we followed a stratification of 35 – 65%. Many studies [75]-[77] explored the use of this technique by the past.

Table 4. ANT Versions Statistics Oversampling

System	#C-D	#F-D	Ratio	#C-B	#F-B	Ratio
ANT 1.3	201	135	67.16%	-	-	-
ANT 1.4	178	38	21.35%	216	76	35.18%
ANT 1.5	366	179	48.91%	-	-	-
ANT 1.6	368	61	16.58%	473	166	35.09%
ANT 1.7	776	101	13.01%	1039	364	35.05%

The best results have been obtained with the following variants: TR (LOC, Ce), TR (LOC, Fan-in), TR (LOC, RFC), TR (RFC, Ca) and TR (WMC, CBO); and that the variants including the metric LOC are a step above. For space limitations reasons, we give in Table 5 the best results which are obtained with TR (LOC, Fan-in). The table shows the results before, after duplication of faults and after oversampling. The best performances were put in bold to better draw attention. For space limitation reasons, we give only results of version ANT 1.3.

Table 5. TR (LOC, Fan-in) Binary Classification Results

		TPR	TNR	g-m	AUC	TPR-D	TNR-D	g-m-D	AUC-D	TPR-B	TNR-B	g-m-B	AUC-B
ANT 1.3	NB	0.733	0.758	**0.745**	**0.781**	0.741	0.909	**0.821**	**0.867**	-	-	-	-
	J48	0.733	0.773	**0.753**	**0.732**	0.852	0.727	**0.787**	**0.815**	-	-	-	-
	RF	0.633	0.682	0.657	**0.730**	0.837	0.682	**0.792**	**0.885**	-	-	-	-
	RLog	0.667	0.818	**0.739**	**0.836**	0.677	0.788	**0.812**	**0.892**	-	-	-	-
	SVM	0.400	0.955	0.618	0.677	0.993	0.076	0.275	0.530	-	-	-	-
	ANN	0.733	0.818	0.774	**0.830**	0.875	0.732	**0.817**	**0.888**	-	-	-	-

We can see that the best performances were obtained with ANN and NB algorithms. We have a good classification for the ANT version 1.3 (with a g-mean of 0.821 and an AUC of 0.892), no doubt, due to the relatively high number of faults in this early version. For the rest of the versions, we still have acceptable classifications (a g-mean > 0.745). Duplication of faults brings a big performance lift especially for versions 1.4, 1.6 and 1.7 where the ratio of faults is lower (approximately 20% for version 1.4 and only 11% for the two other versions).

The variant TR (LOC, Fan-in) includes a size metric (LOC). It has already been found in the literature that LOC (Lines Of Code) is a good indicator of the probability of occurrence of faults [45]. It is also for this reason that among the five most conclusive models, namely: TR (LOC, Ce), TR (LOC, Fan-in), TR (LOC, RFC), TR (RFC, Ca) and TR (WMC, CBO), models with LOC metric perform better. It seems logical that the larger is the code, the more vulnerable it is to contain faults. In addition, Fan-in (as a coupling metric) is a good indicator of classes that requires restructuring and more testing effort [50]. We can see that the combination of two metrics: size and coupling, more particularly LOC and Fan-in, in our risk model is satisfactory for fault prediction.

In a second step (Can the risk model (variants) predict different levels of fault severity?), we investigated the efficiency of our risk model (different variants) for level of severity prediction. In order to fulfill this aim, we created two categories of faults: Normal and High as explained earlier. Table 6 and Table 7 give some statistics about Normal and High severity faults respectively within the ANT system, respectively before and after fault duplication.

Table 6. ANT Versions Normal Severity Faults Statistics

System	#Classes	#Faulty	Ratio	#C-D	#F-D	Ratio
ANT 1.3	126	55	43.64%	285	214	75.09%
ANT 1.4	178	29	16.29%	186	37	19.89%
ANT 1.5	293	64	21.84%	330	101	30.61%
ANT 1.6	352	32	9.09%	362	42	11.60%
ANT 1.7	675	56	7.51%	765	76	9.93%

For lightness, we restricted the number of our variants to only five; the best performing ones for binary classification which are: TR (LOC, Ce), TR (LOC, Fan-in), TR (LOC, RFC), TR (RFC, Ca) and TR (WMC, CBO). Moreover, it can be seen that there is no duplication for high severity faults on ANT 1.6. Thus, this version is discarded in that particular case.

Table 7. ANT Versions High Severity Faults Statistics

System	#Classes	#Faulty	Ratio	#C-D	#F-D	Ratio
ANT 1.3	126	60	47.62%	279	147	52.69%
ANT 1.4	178	12	6.74%	179	13	7.26%
ANT 1.5	293	55	18.77%	298	60	20.13%
ANT 1.6	352	19	5.40%	-	-	-
ANT 1.7	745	22	2.95%	748	25	3.34%

We applied a multi-classification prediction. We tried to build models that can predict the level of severity of a given class in one multi-classification model we denoted by S. i.e. a single model that can predict whether a class is not faulty or faulty with a High or Normal severity level as follows:

$$S = \begin{cases} NFC & Not\ Faulty \\ NSF & Normal\ Severity\ Faults \\ HSF & High\ Severity\ Faults \end{cases}$$

A class is considered as HSF if it contains at least one high severity fault discarding how many normal severity faults it contains. Afterwards, applying fault duplication will consider the number of faults of the same level of severity. Table 8 gives a brief description of the number of faults in the ANT system following their level of severity S, where: #C is the number of classes, NFC is the number of not faulty classes, NSF is the number of classes containing normal severity faults, HSF is the number of classes containing high severity faults and finally the suffix -D means after duplication of faults.

Table 8. Statistics of ANT Faults' Severity

System	#C	#NFC	#NSF	#HSF	#C-D	#NFC-D	#NSF-D	#HSF-D
ANT 1.3	126	66	0	60	213	66	0	147
ANT 1.4	178	140	26	12	185	140	32	13
ANT 1.5	293	193	51	55	321	193	68	60
ANT 1.6	352	307	26	19	358	307	32	19
ANT 1.7	745	675	48	22	765	675	65	25

Table 9. TR (LOC, RFC) Results for Level of Severity Prediction

		TPR	TNR	g-m	AUC	TPR-D	TNR-D	g-m-D	AUC-D
ANT 1.3	NB	0.762	0.747	0.739	0.815	0.808	0.813	0.810	0.879
	J48	0.730	0.726	0.729	0.729	0.779	0.726	0.749	0.797
	RF	0.675	0.675	0.675	0.766	0.840	0.736	0.776	0.879
	RLog	0.762	0.756	0.757	0.834	0.826	0.797	0.811	0.892
	SVM	0.754	0.738	0.728	0.746	0.831	0.824	**0.827**	**0.827**
	ANN	0.683	0.667	**0.764**	**0.763**	0.83	0.787	0.812	0.885
ANT 1.4	NB	0.781	0.371	0.325	0.728	0.768	0.434	0.393	0.755
	J48	0.787	0.213	0.316	0.471	0.751	0.273	0.236	0.575
	RF	0.657	0.434	0.398	0.665	0.719	0.604	**0.600**	**0.738**
	RLog	0.781	0.331	0.325	0.737	0.773	0.403	0.382	0.759
	SVM	0.787	0.213	0.316	0.500	0.757	0.243	0.316	0.500
	ANN	0.784	0.259	0.330	0.733	0.773	0.403	0.382	0.757
ANT 1.5	NB	0.672	0.455	0.000	0.736	0.614	0.538	0.442	0.775
	J48	0.655	0.427	0.000	0.635	0.620	0.700	0.534	0.715
	RF	0.584	0.607	0.362	0.677	0.660	0.740	0.680	0.750
	RLog	0.666	0.436	0.000	0.760	0.626	0.560	0.513	0.779
	SVM	0.659	0.341	0.000	0.500	0.611	0.448	0.339	0.528
	ANN	0.674	0.563	0.000	0.756	0.626	0.674	**0.691**	**0.768**
ANT 1.6	NB	0.849	0.222	0.294	0.678	0.832	0.223	0.279	0.713
	J48	0.872	0.128	0.316	0.468	0.858	0.142	0.316	0.486
	RF	0.787	0.256	0.342	0.602	0.802	0.420	0.436	0.687
	RLog	0.866	0.147	0.248	0.703	0.846	0.191	0.280	0.726
	SVM	0.872	0.128	0.316	0.500	0.858	0.142	0.316	0.500
	ANN	0.872	0.128	0.316	0.679	0.858	0.142	0.316	0.722
ANT 1.7	NB	0.887	0.298	0.385	0.800	0.877	0.408	0.425	0.812
	J48	0.906	0.120	0.254	0.591	0.894	0.264	0.341	0.750
	RF	0.839	0.296	0.332	0.684	0.856	0.553	**0.600**	**0.776**
	RLog	0.907	0.223	0.334	0.817	0.890	0.302	0.375	0.834
	SVM	0.906	0.094	0.316	0.500	0.882	0.118	0.316	0.500
	ANN	0.906	0.111	0.316	0.817	0.891	0.287	0.368	0.831

The TR (LOC, RFC) variant gives the best results (before and after fault duplication, see Table 9). The best performing models are NB, SVM and ANN. We obtained a very good classification for the ANT version 1.3 (with a g-mean equal to 0.827 and an AUC of 0.892), but only acceptable results for the other versions, apart version 1.6 where we have a poor classification. It is important to mention that for this particular version, duplication of faults brings the least, since there are only six added classes. The strength of this model is that it consists of a size metric (LOC) and a Complexity / Coupling metric (RFC), which seems giving it a big informative potential.

4.2. Cross-Version Validation Results

We present in this section the results for CVFSP, obtained when training the risk model on a particular version of ANT and testing it on the successive one. The aim of these experiments is to answer to RQ1: Can a risk model built for a previous version of a given system be reused on its next version to predict fault-prone classes? and RQ2: Can a risk model built for a previous version of a given system be reused on its next version to predict fault severity? We used the best performing models presented in the previous section.

Table 10 presents the results obtained for binary classification with TR (LOC, Fan-in). Table 11 shows the results for levels of severity prediction with TR (LOC, RFC).

Table 10. Cross-Version Binary Classification TR (LOC, Fan-in)

	1.3 on 1.4				1.4 on 1.5				1.5 on 1.6				1.6 on 1.7				
	TPR	TNR	g-m	AUC	TPR	TNR	g-m	AUC	TPR	TNR	g-m	AUC	TPR	TNR	g-m	AUC	AUC
NB	0.720	0.686	0.703	0.772	0.095	0.984	0.306	0.761	0.557	0.857	0.691	0.724	0.257	0.982	0.502	0.853	0.720
J48	0.860	0.493	0.651	0.676	0.156	0.963	0.388	0.559	0.820	0.596	0.699	0.676	0.000	1.000	0.000	0.774	0.860
RF	0.720	0.571	0.641	0.694	0.235	0.85	0.447	0.603	0.475	0.632	0.548	0.600	0.248	0.892	0.470	0.674	0.720
RLog	0.780	0.607	0.688	0.780	0.095	0.984	0.306	0.839	0.672	0.759	0.714	0.725	0.218	0.993	0.465	0.854	0.780
SVM	0.960	0.114	0.331	0.537	0.045	0.995	0.212	0.520	0.672	0.759	0.714	0.668	0.000	1.000	0.000	0.500	0.960
ANN	0.880	0.486	0.654	0.780	0.095	0.984	0.306	0.839	0.787	0.632	0.705	0.725	0.000	1.000	0.000	0.854	0.880

Table 11. Cross-Version Levels of Severity Classification TR (LOC, RFC)

	1.3 on 1.4				1.4 on 1.5				1.5 on 1.6				1.6 on 1.7				
	TPR	TNR	g-m	AUC	TPR	TNR	g-m	AUC	TPR	TNR	g-m	AUC	TPR	TNR	g-m	AUC	AUC
NB	0.846	0.679	0.758	0.771	-	-	-	-	0.813	0.436	0.019	0.740	0.886	0.302	0.014	0.500	0.846
J48	0.846	0.571	0.695	0.675	-	-	-	-	0.668	0.673	0.274	0.732	0.882	0.118	0.001	0.500	0.846
RF	0.769	0.636	0.699	0.727	-	-	-	-	0.665	0.625	0.267	0.665	0.833	0.318	0.171	0.500	0.769
RLog	0.846	0.614	0.721	0.773	-	-	-	-	0.799	0.434	0.019	0.740	0.890	0.235	0.012	0.500	0.846
SVM	0.846	0.650	0.742	0.710	-	-	-	-	0.827	0.255	0.011	0.541	0.882	0.118	0.001	0.500	0.846
ANN	0.923	0.493	0.675	0.773	-	-	-	-	0.746	0.611	0.023	0.740	0.882	0.118	0.001	0.500	0.923

Regarding cross-version analysis, we can notice (tables 10 and 11) that, in general, we obtained good results when using ANT 1.3 or ANT 1.5 as a training set to build the classifier model, except when using version ANT 1.5 as training set for High severity level (where the number of high severity faults is very low) prediction.

In the case of binary classification (Table 10), we can see that the considered model achieved an acceptable classification with a g-mean superior to 0.70 (an excellent evaluation according to the AUC value (0.854)). We can notice that the best g-mean obtained is 0.714 (ANT 1.5 on 1.6) using Rlog and SVM. Regarding Levels of Severity prediction (Table 11), only models trained on ANT 1.3 achieved acceptable results. The best performing model was obtained using NB with a g-mean of 0.76 (best AUC is 0.773). These results can be explained by the decremented number of bugs, probably due to the testing efforts, which allow fixing the high severity faults. Besides that, a huge number of new classes were added to this system on new versions. The cross-version validation seems giving good results (which is plausible) only if the version used as a training set contains enough faults.

In a slightly different approach of cross-version validation, we tried building prediction models using the entire data gathered from N-1 previous versions as input data and performing prediction on version N (for example, we would use ANT 1.3 + ANT 1.4 + ANT 1.5 as input data to build our model and then perform the prediction on ANT 1.6). However, results of this approach did not give significant improvement then using the direct precedent version (N-1) to build our model. It would be interesting to investigate this avenue using datasets collected from other systems.

The cross-version validation gives good results for fault prediction and more than acceptable results for levels of severity prediction in early versions. The choice of the version which will be the training set when using this method is crucial for achieving good results. According to the results, we can answer positively to both RQ1 and RQ2.

4.3. Empirical Risk Prediction

In this section, we introduce the concept of empirical risk of a class. The aim of this experiment is to answer to RQ3 (Can the risk model be used to predict the empirical risk of a class, a continuous value considering both the number of faults the class can contain and their severity?). We define the empirical risk (ER) of a given class C using the following formula, which takes into consideration both the number of faults and their different levels of severity:

$$ER(C) = \sum_{i=1}^{6} S_i \times N_i$$

where S_i is the weight given to the level of severity: 6 represents the highest level of severity, which is blocker and 1 corresponds to the lowest one, which represents trivial faults. N_i will be the number of faults observed for the level of severity i.

The objective is to use the risk model (best variant) to build a prediction model giving the ER of a class as output. We used the machine learning algorithms with the regression scheme and tried to predict the ER of a given class always by performing 10-cross validation, because we are trying to predict a continuous class not a categorical one. We used: Linear regression (LR), Gaussian Process (GP), Random forest (RF), M5P a decision tree algorithm, SmoReg and artificial neural network (ANN). Table 12 presents the results. We give the coefficient of correlation (r, Pearson's), Mean Absolute Error (MAE), Root Mean Squared Error (RMSE), Relative Absolute Error (RAE) and Root Relative Squared Error (RSE) of the best variant TR (LOC, RFC) after experimenting all the variants mentioned before.

For intra-version analysis, the best performing variant for ER prediction is TR (LOC, RFC) followed by TR (RFC, Ca). Table 12 gives the results of only (for space limitation reasons) the variant TR (LOC, RFC). The best correlation coefficient completed is 0.659 for the ANT 1.3 version. For the ANT 1.5 version, it is 0.538. For both versions ANT 1.4 and 1.7, it exceeds 0.4 despite the few faults present in these versions. The performance is lower for version 1.6, which contains very few faults.

Table 12. TR (LOC, RFC) Intra-Version Empirical Risk Prediction

		r	MAE	RMSE	RAE	RSE
ANT 1.3	LR	0.651	5.004	7.211	66.902	75.478
	GP	0.629	5.117	7.381	68.424	77.260
	RF	0.439	6.091	9.282	81.437	97.162
	M5P	0.651	5.004	7.211	66.902	75.478
	SmoReg	0.659	4.769	7.358	63.763	77.022
	ANN	0.557	5.659	8.059	75.669	84.354
ANT 1.4	LR	0.391	1.257	2.001	81.595	91.507
	GP	0.355	1.287	2.033	83.597	92.987
	RF	0.203	1.335	2.419	86.661	110.591
	M5P	0.391	1.257	2.001	81.595	91.507
	SmoReg	0.351	0.972	2.378	63.118	108.742
	ANN	0.264	1.371	2.196	88.996	100.391
ANT 1.5	LR	0.468	2.494	3.496	81.300	88.229
	GP	0.504	2.337	3.414	76.189	86.168
	RF	0.347	2.587	4.007	84.311	101.126
	M5P	0.431	2.461	3.594	80.201	90.703
	SmoReg	0.472	2.096	3.607	68.333	91.047
	ANN	0.482	2.644	3.571	86.194	90.104
ANT 1.6	LR	0.232	0.858	1.584	92.253	96.965
	GP	0.243	0.845	1.580	90.890	96.733
	RF	0.070	0.878	1.930	94.385	118.11
	M5P	0.208	0.853	1.596	91.653	97.679
	SmoReg	0.087	0.533	1.710	57.277	104.679
	ANN	0.121	1.004	1.698	107.894	103.903
ANT 1.7	LR	0.452	0.678	1.384	85.464	89.128
	GP	0.459	0.640	1.377	80.728	88.711
	RF	0.280	0.681	1.669	85.763	107.51
	M5P	0.452	0.678	1.384	85.464	89.128
	SmoReg	0.341	0.440	1.501	55.435	96.686
	ANN	0.343	0.870	1.499	109.589	96.517

The results show good correlations for versions 1.3 (0.659) and 1.5 (0.504). Also, for versions 1.4 and 1.7 (between 0.39 and 0.46) although both versions contain fewer faults. On the other hand, the lack of performance persists with version 1.6. As seen previously, due to the absence of faults in this version, the algorithms do not have enough input data for the training phase. We can notice from the error metrics given in Table 12 that predictions are very close to ground truth and that the average error is relatively small. The performance of this model for the prediction of the ER is related to its ability to predict the level of severity of faults as seen in the previous experiments. Indeed, thanks to the "LOC" component, this model is able to predict the probability of presence of faults. Its second component "RFC" allows predicting the level of severity of faults.

For cross-version analysis, we used the variant TR (LOC, RFC), as the best model for intra-version analysis. Table 13 gives the results. We can see that the model created from version N-1 can perform better than the model created from the same version N, except when using ANT 1.4 as a training set. We have already demonstrated the lack of reported faults on this version. We can put in light the performance of the model trained with data from ANT 1.3 and evaluated on ANT 1.4 that went up to a correlation coefficient of 0.417 and ANT 1.6 on ANT 1.7 with a correlation coefficient of 0.466. We can notice that Linear Regression and Gaussian Process are the better performing machine learning algorithms for cross-version risk prediction.

Regarding RQ3 (Can the considered risk model predict fault-prone classes by giving information about both their potential number of faults and levels of severity?), the theoretical risk model estimates the empirical risk with a good correlation coefficient.

Table 13. TR (LOC, RFC) Cross-Version Empirical Risk Prediction

	1.3 on 1.4					1.4 on 1.5				
	r	MAE	RMSE	RAE	RSE	r	MAE	RMSE	RAE	RSE
LR	0.417	9.327	10.914	153.361	173.762	0.486	12.895	13.339	484.575	317.463
GP	0.417	4.775	4.958	78.505	78.941	0.486	2.743	4.713	103.079	112.176
RF	0.389	9.426	11.767	154.986	187.344	0.124	2.391	4.577	89.850	108.934
M5P	0.417	9.327	10.914	153.361	173.762	0.486	12.895	13.339	484.575	317.463
SmoReg	0.417	7.217	8.748	118.657	139.283	0.486	2.405	4.627	90.382	110.122
ANN	0.390	9.511	11.282	156.382	179.621	0.211	7.054	8.024	265.078	190.967
	1.5 on 1.6					1.6 on 1.7				
	r	MAE	r	MAE	r	MAE	r	MAE	r	MAE
LR	0.267	2.363	0.267	2.363	0.267	2.363	0.267	2.363	0.267	2.363
GP	0.267	2.067	0.267	2.067	0.267	2.067	0.267	2.067	0.267	2.067
RF	0.216	2.421	0.216	2.421	0.216	2.421	0.216	2.421	0.216	2.421
M5P	0.307	2.319	0.307	2.319	0.307	2.319	0.307	2.319	0.307	2.319
SmoReg	0.267	1.503	0.267	1.503	0.267	1.503	0.267	1.503	0.267	1.503
ANN	0.287	1.982	0.287	1.982	0.287	1.982	0.287	1.982	0.287	1.982

5. Threats to Validity

The major threats to validity of our study are:

First of all, we relied exclusively on the ANT system to conduct our experiments. The considered risk model needs to be validated on other systems to confirm our results.

Another threat lies in our formulation of the empirical risk. Indeed, the choice of the weights for the levels of severity is still subjective. Even though by giving the highest weight for the highest level of severity, we tried to follow a certain logic. The problem remains in the severity categorization itself which is not formal. The choice of the category of severity of faults is based only on the experience of testers (developers), whom reported them. This assignment could be changed using a more objective methodology which better takes into account the impact of faults.

Besides that, the considered risk model is based on the concept of Euclidean distance. However, this choice is not arbitrary, it is based on the Bernoulli risk theory. It would be interesting to investigate another formula of distance calculation such as the Manhattan distance or a more complex one that better describes the relationship between P and I.

6. Conclusion and Future Work

In this work, we conducted a validation study to investigate the effectiveness of the considered risk model for Cross Version Fault and Severity Prediction (CVFSP). In a previous study [16], we showed that the considered model can be used to predict fault-prone classes and severity of faults as well. Our main contribution in the present work lies in presenting a more practical usage of the considered risk model by showing its effectiveness for Cross-Version Fault and Severity Prediction (CVFSP), using data of prior versions of the considered system. In addition, we showed that the risk model gives good results in the estimation of the empirical risk of a class by considering both the number of faults and their levels of severity in the two situations: (1) intra-version analysis, and (2) cross-version analysis.

About cross-version validation (RQ1 (Can a risk model built for a previous version of a given system be reused on its next version to predict fault-prone classes?) and RQ2 (Can a risk model built for a previous version of a given system be reused on its next version to predict fault severity?)), we wanted to investigate if a model built on a previous version can be reused for fault-proneness and fault-severity prediction for a new version. The objective was to verify if this approach can be used in a real-life context. We managed to achieve acceptable results (for both cross-version binary fault prediction (a g-mean of 0.714, an AUC of 0.725) and cross-version multi-classification of levels of severity (a g-mean of 0.758, an AUC of 0.771)), only when using an early version as training set. We also noticed that the choice of the training set (reference version dataset) is crucial.

Regarding the empirical risk prediction (RQ3 (Can the risk model be used to predict the empirical risk of a class, a continuous value considering both the number of faults the class can contain and their severity?)), the theoretical risk model estimates the empirical risk with a good correlation coefficient when the data are more or less balanced. The risk model presented in this study seems, according to obtained results, having a good potential not only for software fault prediction but also for level of severity prediction.

As future work, we plan to: (1) extend our study by considering many other object-oriented software systems in order to be able to give general conclusions, (2) investigate the risk model in a cross-system situation, using data from a system to build prediction models for other systems, and (3) conduct an empirical comparison of the risk model to other approaches.

Conflict of Interest

Mourad Badri: Project definition, Conceptualization, Methodology, Data Collection, Writing - Review & Editing, Supervision, Project administration

Salim Moudache: Methodology, Software, Data Collection, Formal data analysis, Investigation, Writing - Original Draft, Writing - Review & Editing, Visualization

Authors Contributions

The authors declare no conflict of interest.

7. Acknowledgment

This work was partially supported by NSERC (Natural Sciences and Engineering Research Council of Canada) grant. We would also like to give special thanks to Professor Fadel Touré for his help with the risk models.

References

[1] Tim, M., Zach, M., Burak, T., Bojan, C., Yue, J., & Ayşe, B. (2010). Defect prediction from static code features: Current results, limitations, new approaches. *Automated Software Engineering*, *17(4)*, 375–407.

[2] Antonia, B. (2007). Software testing research: Achievements, challenges, dreams. *Proceedings of the 2007 Future of Software Engineering* (pp. 85–103).

[3] Cagatay, C., Ugur, S., & Banu, D. (2010). Metrics-driven software quality prediction without prior fault data. *Electronic Engineering and Computing Technology*, 189–199, Springer.

[4] Alireza, H., Mika, M., Markku, O., & Pasi, K. (2018). Test prioritization in continuous integration environments. *Journal of Systems and Software*, *146*, 80–98.

[5] Lu, Y. F., Lou, Y., Cheng, S. Y., Zhang, L. M., Hao, D., Zhou, Y. F., & Zhang, L. (2016). How does regression test prioritization perform in real-world software evolution? *Proceedings of the 2016 IEEE/ACM 38th International Conference on Software Engineering* (ICSE).

[6] Chen, T. Y., Kuo, F. C., Liu, H., & Wong, W. E. (2013). Code coverage of adaptive random testing. *IEEE Transactions on Reliability*, *62(1)*, 226–237.

[7] Wu, H. Y., Petke, J., Yue, J., Harman, M., *et al*. (2018). An empirical comparison of combinatorial testing, random testing and adaptive random testing. *IEEE Transactions on Software Engineering*.

[8] Busjaeger, B., & Xie., T. (2016). Learning for test prioritization: an industrial case study. *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (pp. 975–980).

[9] Erturk, E., & Sezer, E. A. (2015). A comparison of some soft computing methods for software fault prediction. *Expert Systems with Applications*, *42(4)*, 1872–1879.

[10] Malhotra, R. (2015). A systematic review of machine learning techniques for software fault prediction. *Applied Soft Computing*, *27*, 504–518.

[11] Moeyersoms, J., Fortuny, E. J., Dejaeger, K., Baesens, B., & Martens, D. (2015). Comprehensible software fault and effort prediction: A data mining approach. *Journal of Systems and Software*, *100*, 80–90.

[12] Rathore, S. S., & Kumar, S. (2017). An empirical study of some software fault prediction techniques for the number of faults prediction. *Soft Computing*, *21(24)*, 7417–7434.

[13] Madeyski, L., & Jureczko, M. (2015). Which process metrics can significantly improve defect prediction models? An empirical study. *Software Quality Journal*, *23(3)*, 393–422.

[14] Harter, D. E., Kemerer, C. F., & Slaughter, S. A. (2012). Does software process improvement reduce the severity of defects? A longitudinal field study. *IEEE Transactions on Software Engineering*, *38(4)*, 810–827.

[15] Zhou, Y. M., & Leung, H. (2006). Empirical analysis of object-oriented design metrics for predicting high and low severity faults. *IEEE Transactions on Software Engineering*.

[16] Moudache, S., & Badri, M., (2019). Software fault prediction based on fault probability and impact. *Proceedings of the 2019 18th IEEE International Conference on Machine Learning And Applications* (ICMLA).

[17] Toure, F. (2016). Effort orientation of unit tests in objectoriented systems, an approach based on software metrics. Ph.D. Dissertation. Université Laval.

[18] Raed, S. (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.

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

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

[21] Lu, H. H., Cukic, B., & Culp, M. (2012). Software defect prediction using semi-supervised learning with dimension reduction. *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*.

[22] Lu, H. H., Cukic, B., & Culp, M. (2014). A semi-supervised approach to software defect prediction. *Proceedings of the 2014 IEEE 38th Annual Computer Software and Applications Conference* (pp. 416–425).

[23] Catal, C. (2014). A comparison of semi-supervised classification approaches for software defect prediction. Journal of Intelligent Systems, *23(1)*, 75–82.

[24] Bishnu, P. S., & Bhattacherjee, V. (2012). Software fault prediction using quad tree-based k-means clustering algorithm. *IEEE Transactions on Knowledge and Data Engineering*, *24(6)*, 1146–1150.

[25] Catal, C., Sevim, U., & Diri, B. (2009). Clustering and metrics thresholds based software fault prediction of unlabeled program modules. *Proceedings of the Sixth International Conference on Information Technology: New Generations* (pp. 199–204).

[26] Boucher, A., & Badri, M. (2018). Software metrics thresholds calculation techniques to predict fault-proneness: An empirical comparison. *Information and Software Technology*, *96*, 38–67.

[27] 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.

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

[29] Boucher, A., & Badri, M. (2017). Predicting fault-prone classes in object-oriented software: An adaptation of an unsupervised hybrid SOM algorithm. *Proceedings of the 2017 IEEE International Conference on Software Quality, Reliability and Security*.

[30] Abaei, G., Selamat, A., & Fujita, H. (2015). An empirical study based on semi-supervised hybrid self-organizing map for software fault prediction. *Knowledge-Based Systems*, *74*, 28–39.

[31] Hong, E. (2017). Software fault-proneness prediction using module severity metrics. *International Journal of Applied Engineering Research*, *12(9)*, 2038–2043.

[32] Shatnawi, R., & Li, W. (2008). The effectiveness of software metrics in identifying error-prone classes in post-release software evolution process. *Journal of systems and Software*, *81(11)*, 1868–1882.

[33] Singh, Y., Kaur, A., & Malhotra, R. (2010). Empirical validation of object-oriented metrics for predicting fault proneness models. *Software Quality Journal*, *18(1)*, 3.

[34] Chen, X. L., Jiang, J. H., Zhang, W., & Xia, X. Z. (2020). Fault diagnosis for open source software based on dynamic tracking. *Proceedings of the 2020 7th International Conference on Dependable Systems and Their Applications (DSA)*.

[35] Yucalar, F., Ozcift, A., Borandag, E., & Kilinc, D. (2020). Multiple-classifiers in software quality engineering: Combining predictors to improve software fault prediction ability. *Engineering Science and Technology, an International Journal*, *23(4)*, 938–950.

[36] Ali, A., & Gravino, C. (2020). Bio-inspired algorithms in software fault prediction: A systematic literature review. *Proceedings of the 2020 14th International Conference on Open Source Systems and Technologies (ICOSST)*.

[37] Kumar, A., & Bansal, A. (2019). Software fault proneness prediction using genetic based machine learning techniques. *Proceedings of the 2019 4th International Conference on Internet of Things: Smart Innovation and Usages*.

[38] Jin, C. (2021). Software defect prediction model based on distance metric learning. *Soft Computing*, *25(1)*, 447–461.

[39] Jureczko, M. (2011). Significance of different software metrics in defect prediction. *Software Engineering: An International Journal*, *1(1)*, 86–95.

[40] Jureczko, M., & Madeyski, L. (2010). Towards identifying software project clusters with regard to defect prediction. *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*.

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

[42] Yu, L. G. (2012). Using negative binomial regression analysis to predict software faults: A study of apache ant.

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

[44] 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.

[45] Jureczko, M., & Spinellis, D. (2010). Using object-oriented design metrics to predict software defects. *Models and Methods of System Dependability. Oficyna Wydawnicza Politechniki Wrocławskiej*, 69–81.

[46] Yourdon, E., & Constantine., L. L. (1979). *Structured Design: Fundamentals of A Discipline of Computer Program and Systems Design (1st ed.)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

[47] Briand, L. C., Wust, J., & Lounis, H. (1999). Using coupling measurement for impact analysis in object-oriented systems. *Proceedings of the IEEE International Conference on Software Maintenance* (pp. 475–482).

[48] Martin, R. (1994). OO design quality metrics. *An Analysis of Dependencies*, *12*, 151–170.

[49] Henry, S., & Kafura, D. (1981). Software structure metrics based on information flow. *IEEE transactions on Software Engineering*, *5*, 510–518.

[50] Rosenberg, L. H., Stapko, R., & Gallo, A. (1999). Risk-based object oriented testing. 24th SWE. NASA, Greenbelt, MD, USA.

[51] Booch, G. (1991). Object-oriented design with applications, Redwood City, CA, enj armnCummings.

[52] Shatnawi, R. (2012). Improving software fault-prediction for imbalanced data. *Proceedings of the 2012 International Conference on In Innovations in Information Technology*.

[53] Risk, O., & Bernoulli. D. (1954). Exposition of a new theory on the measurement. *Econometrica*, *22(1)*, 23–36.

[54] Briand, L. C., Wüst, J., Daly, J. W., & Porter, D. V. (2000). Exploring the relationships between design measures and software quality in object-oriented systems. *Journal of Systems and Software*, *51(3)*, 245–273.

[55] 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.

[56] Iliev, M., Karasneh, B., Chaudron, M. R., & Essenius. E. (2012). Automated prediction of defect severity based on codifying design knowledge using ontologies. *Proceedings of the 2012 First International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering*.

[57] Bearden, W. O., & Netemeyer, R. G. (1999). Handbook of marketing scales: Multi-item measures for marketing and consumer behavior research.

[58] Catal, C., & Diri, B. (2007). Software fault prediction with object-oriented metrics based artificial immune recognition system. *Proceedings of the International Conference on Product Focused Software Process Improvement.*

[59] Olson, D. L., & Delen, D. (2008). Advanced data mining techniques. Springer Science & Business Media.

[60] Refaat, M. (2010). Data preparation for data mining using SAS.

[61] Singh, S., & Kahlon, K. S. (2014). Object oriented software metrics threshold values at quantitative acceptable risk level. *CSI Transactions on ICT*, *2(3)*, 191–205.

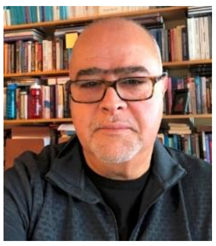
[62] Wendland, M. F., Kranz, M., & Schieferdecker, I. (2012). A systematic approach to risk-based testing using risk-annotated requirements models. *Proceeding of the Seventh International Conference on Software Engineering Advances.*

[63] Khoshgoftaar, T. M., Seiffert, C., Hulse, J. V., Napolitano, A., & Folleco, A. (2007). Learning with limited minority class data. *Proceeding of the Sixth International Conference on Machine Learning and Applications*.

[64] Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., & Witten, I. H. (2009). The WEKA data mining software: An update. *ACM SIGKDD Explorations Newsletter*, *11(1)*, 10–18.

[65] Kaur. A., & Kaur, K. (2014). Performance analysis of ensemble learning for predicting defects in open source software. *Proceeding of the 2014 International Conference on Advances in Computing, Communications and Informatics*.

[66] Christopher, K. I. W. (2006). *Gaussian Processes Formachine Learning*, Taylor & Francis Group.
[67] Goyal, R., Chandra, P., & Singh, Y. (2014). Suitability of KNN regression in the development of interaction based software fault prediction models. *IERI Procedia*, 15–21.
[68] Torrado, N., Wiper, M. P., & Rosa, E. L. (2012). Software reliability modeling with software metrics data via Gaussian processes. *IEEE Transactions on Software Engineering*, *39(8)*, 1179–1186.
[69] Keim, D. A. (2002). Information visualization and visual data mining. *IEEE transactions on Visualization and Computer Graphics*, *8(1)*, 1–8.
[70] Malhotra, R., & Jain, A. (2012). Fault prediction using statistical and machine learning methods for improving software quality. *Journal of Information Processing Systems*, *8(2)*, 241–262.
[71] Sun, Y. M., Kamel, M. S., & Wang, Y. (2006). Boosting for learning multiple classes with imbalanced class distribution. *Proceeding of the Sixth International Conference on Data Mining*.
[72] Shatnawi, R. (2017). The application of ROC analysis in threshold identification, data imbalance and metrics selection for software fault prediction. *Innovations in Systems and Software Engineering*, *13(2-3)*, 201–217.
[73] Hosmer, D. W., & Lemeshow. S. (2000). Special topics. *Applied Logistic Regression*, 260–351.
[74] Chawla, N. V., Bowyer, K. W., Hall., L. O., & Kegelmeyer, W. P. (2002). SMOTE: Synthetic minority over-sampling technique. *Journal of Artificial Intelligence Research*, *16*, 321–357.
[75] Catal, C., & Diri, B. (2008). A fault prediction model with limited fault data to improve test process. *Proceeding of the International Conference on Product Focused Software Process Improvement*.
[76] Seiffert, C., Khoshgoftaar, T. M., Hulse, J. V., & Napolitano, A. (2008). Building useful models from imbalanced data with sampling and boosting. *Proceeding of the Florida Artificial Intelligence Research Society Conference*.
[77] J. V., Hulse, Khoshgoftaar, T. M., & Napolitano, A. (2007). Experimental perspectives on learning from imbalanced data. *Proceedings of the 24th International Conference on Machine Learning*.



Mourad Badri is a full professor of computer science at the Department of Mathematics and Computer Science of the University of Quebec at Trois-Rivières (Canada). He holds a PhD in computer science (software engineering) from the National Institute of Applied Sciences in Lyon, France. His main areas of interest include object and aspect-oriented software engineering, software quality attributes, software testing, refactoring, software evolution and application of machine learning in software engineering.

Salim Moudache is a graduate student (master - applied mathematics and computer science) from the Department of Mathematics and Computer Science of the University of Québec at Trois-Rivières (Canada). His main research fields are software engineering, data science, application of machine learning in software engineering and artificial intelligence.