Information and Software Technology 52 (2010) 1298–1311

Contents lists available at ScienceDirect

Information and Software Technology

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

A novel composite model approach to improve software quality prediction

Salah Bouktif [a,b,*], Faheem Ahmed [a], Issa Khalil [a], Giuliano Antoniol [b]

[a] Faculty of Information Technology, United Arab Emirates University, United Arab Emirates
[b] Ecole Polytechnique de Montreal, Montreal, Canada

ARTICLE INFO

Article history:
Received 21 January 2010
Received in revised form 11 July 2010
Accepted 13 July 2010
Available online 17 July 2010

Keywords:
Software quality prediction
Fault-proneness
Decision trees
Genetic algorithm

ABSTRACT

Context: How can quality of software systems be predicted before deployment? In attempting to answer this question, prediction models are advocated in several studies. The performance of such models drops dramatically, with very low accuracy, when they are used in new software development environments or in new circumstances.
Objective: The main objective of this work is to circumvent the model generalizability problem. We propose a new approach that substitutes traditional ways of building prediction models which use historical data and machine learning techniques.
Method: In this paper, existing models are decision trees built to predict module fault-proneness within the NASA Critical Mission Software. A genetic algorithm is developed to combine and adapt expertise extracted from existing models in order to derive a "composite" model that performs accurately in a given context of software development. Experimental evaluation of the approach is carried out in three different software development circumstances.
Results: The results show that derived prediction models work more accurately not only for a particular state of a software organization but also for evolving and modified ones.
Conclusion: Our approach is considered suitable for software data nature and at the same time superior to model selection and data combination approaches. It is then concluded that learning from existing software models (i.e., software expertise) has two immediate advantages; circumventing model generalizability and alleviating the lack of data in software-engineering.

1. Introduction

Over the past two decades, software quality has inspired intensive research effort [1–4]. One reason behind this interest is trying to answer the following question: How can the quality of a software system be predicted before deployment? In attempting to answer this question, several predictive models are advocated. In these models, a number of metrics representing internal attributes of a software artifact are often used in the hope of predicting some quality characteristics such as reliability, fault-proneness, complexity, maintainability, stability, etc. In fact, there are two basic approaches to building predictive models of software quality. In the first approach, the designer relies on historical data and applies various statistically-based methods and machine learning algorithms in order to build the models (e.g., see [1]). In the second approach, the designer uses expert knowledge extracted from domain-specific heuristics (e.g., see [5]). In both approaches, representative data sets are necessary. In the first approach, data are needed to learn the relationship within the models, whereas in the second one, data are needed to validate expert opinions. Unfortunately, in the software-engineering field, the availability of data is often problematic. Compared with other fields such as sociology, medicine, finance, and speech recognition, data on real software systems are not systematically collected partly because many features and indicators (e.g, maintenance effort, number of defects, etc.) need to be collected over a long operational period of the software [4]. In the few cases where data is collected, it is minimal, incomplete, inaccurate or nonrepresentative [6]. In addition, when collected, data is usually not published because companies often provide data for a research group in order to build models under non-disclosure agreements. For this reason, the research group generally agrees to publish the obtained models but not the data which can be explained by the fact that data can be considered confidential and may lead to negative effects for the organization's brand image [7,3,8]. The published models are proposed in the literature in order to be applied to other systems, i.e., to different systems developed within the same or different environments. Nevertheless, little is known about the model generalization and success of these proposed models. On the other hand, the perfor-

* Corresponding author at: Faculty of Information Technology, United Arab Emirates University, United Arab Emirates.
E-mail addresses: salahb@uaeu.ac.ae (S. Bouktif), f.ahmed@uaeu.ac.ae (F. Ahmed), ikhalil@uaeu.ac.ae (I. Khalil), antoniol@ieee.org (G. Antoniol).

0950-5849/$ - see front matter Crown Copyright © 2010 Published by Elsevier B.V. All rights reserved.
doi:10.1016/j.infsof.2010.07.003

mance of such models drops dramatically, with very low accuracy and precision when they are applied to different domains. Despite considerable efforts towards using different statistical methods [8,2,9] and emergent machine learning based techniques [10,3,11], the problem of model generalization remains unresolved up to certain extent.

In this paper, we propose a new approach that aims to circumvent the model-generalization problem in software quality prediction. Our approach is based on the idea of surrogating data sets usually used to train or validate models by already-built prediction models. These existing models are then combined and adapted in order to obtain a "composite" one that performs well in a given context. We consider this approach an alternative to the best model selection, which is common practice in industry for the purpose of addressing the problem of models' generalization. Furthermore, our approach is compared to the usual approach that combines all the available data, called "data combination". This approach is considered as an intuitive solution for the models' generalization problem, and stems from the conventional motto "the more data the better" [12]. In this comparison, we take advantage of the availability of the datasets that served to build the "existing models". The results show that the model obtained by our proposed approach is not only comparable to the hypothetical model that could be built using the combined data, but also significantly outperforms it. We evaluate our approach on module fault-proneness one of the critical factors of software quality prediction. We also illustrate the application of our approach on emerging types of software quality classification models, especially those derived by machine learning based processes, namely decision trees (DT).

The rest of the paper is organized as follows: In Section 2, we attempt to summarize the issues affecting the success of software quality prediction and we discuss the solutions proposed in the literature for such problems. In particular, we discuss the issues of generalization and predictive accuracy. In Section 3, we introduce model combination as a solution for the issues identified in Section 2 and we discuss various ways of model combination in the context of software quality prediction. In Section 4, we introduce our approach of combining and adapting existing prediction models. In Section 5, we describe its implementation. Section 6 details an experimental study to validate the model combination approach. The validation of our approach uses prediction models built from the NASA Software Center. Finally, in Section 7, we present our conclusions and future work.

2. Related work and discussion

2.1. Main problems of building software quality prediction models

A model is generally an abstraction of reality that strips away details and views an entity or concept from a particular perspective [7]. In particular, software quality prediction models permit managers and developers to examine only the software aspects that contribute to the software's final quality *a priori* unknown. These models often express the relationship between internal tangible aspects or attributes and external quality characteristics of the software. These relationships come in many different forms such as equations, mappings, diagrams, rule sets, decision trees and Bayesian classifiers. For example, predicting which modules are likely to contain faults during the operational phase is important for developers and managers to be able to focus their available resources on the most fault-prone modules early in order to prevent poor quality problems and avoid the high cost of repairing faults detected by users. A fault-proneness prediction model often considers measures of structural aspects such as coupling and complexity as indicators of whether a module or a class is fault prone. Building such prediction models is not an easy task due to several problems and constraints that rely on three main factors: (1) the characteristics of datasets used to train models, (2) the technique of model construction and (3) the validity of assumptions made on the nature of the relationship between internal attributes and the external quality characteristic. Due to the lack of systematic data collection in software-engineering, the datasets used to train models are often small, imprecise and incomplete. Missing information is a common problem caused especially by the shortage of data collection funds and effort, a lack of tools and a limited understanding of the measurement field. These problems make data analysis difficult and consequently affect the building of accurate prediction models. In addition to the large number of attributes and the strong collinearity between them, statistical characteristics of data can also include what is known as "heteroscedasticity", which is a phenomenon that occurs when internal attributes may have different variances, i.e., an internal attribute may be a much stronger predictor in a particular portion in its range/value domain. Models that do not consider such issues may be affected in terms of accuracy to a certain extent that they may not provide trustworthy information for the decision making process [6].

A model will be accurate if its conditions of use are the same as those of its construction. Unfortunately, there is no guarantee of accuracy since representative data that simulates all the software environments and conditions is not available. Studies carried out by Fenton et al. [13,3] and Gray and MacDonell [14], on software quality, revealed that the failure of prediction models to achieve their goals is directly related to the limitations of their building techniques. In fact, several problems of prediction are inherited directly from the techniques used to build the models. For example, linear regression is still among the most widely-used techniques; however, it presents several limitations like the non justified data point suppression, uniform data distribution requirement and attributes'non-collinearity assumption [13]. The problem with neural networks is that they tend to behave as black boxes because they do not provide an explanation concerning the way in which the outputs of the model are obtained. They also suffer from a problem referred to as *catastrophic forgetting*: after its original training, if a network is exposed to the learning of new information, then the originally learned information will be disrupted or lost [15]. With fuzzy based techniques, it is difficult to guarantee a high predictive accuracy without losing the ease of model interpretation. This problem appears when the model is built with a very large number of rules [18]. Case based reasoning techniques are known to be intolerant to noise, which is often present in software data [15]. In addition, Aha reported in [14], that the performance of the models built with such techniques strongly depends on the non-trivial task of choosing a similarity function. In order to use a particular model building technique, researchers make assumptions on the form of relationships between internal attributes and external quality characteristics, on the attribute values distributions, or on the correlation between attributes. Since it is difficult for these assumptions to be valid and realistic in most software environments, the produced models are not accurate and weakly accepted. Several techniques proposed to build models are mostly based on machine learning. They contribute to resolving some of the above problems with the hope of building models superior to those produced by classical statistic approaches. Fenton, in particular, in most of his works [13,3] criticized existing techniques of software quality prediction. He described them as naive and proposed Bayesian networks as a probabilistic technique to improve quality prediction. By using a Bayesian network, he combined knowledge and expert opinions on the quality characteristic being predicted in order to improve the understanding and interpretability of models. With more interest in improving the model-generalization performance, Khoshgoftaar introduced dif-

ferent techniques such as regression tree [16] and genetic programming [11]. He also used neural networks [17] and discriminant analysis [24] to predict fault-pronenesses. In order to promote model interpretability, Sahraoui [18] distinguished between two types of models: (1) naive and built using historical data; and (2) causal and built using expert opinions and domain-specific heuristics. Hence, he proposed a fuzzy-logic based approach to derive causal models and improve the capability of models to explain their predictions. In order to predict software quality when only a small number of data samples are available, Xing used support vector machine based techniques to train classification models [4]. More detailed discussion of other techniques of model building involving different machine learning algorithms can be found in [19]. Lyu and Nikora [20] used linear combinations of models in order to improve the accuracy of reliability predictions. Model calibration is a very old practice used mostly for statistical models to tune parameters and make models more appropriate to a new environment. A well-known example of model calibration is COCOMO coefficient adjustment [21].

2.2. Discussion

The purpose of building prediction models is to apply them to new software components, new versions of a system, new systems, or even new environments. Managers want completely reliable predictions. Unfortunately, experience reveals a great variation in the accuracy of the proposed software quality models [7]. Therefore, it is necessary for a model to have prior knowledge about the conditions in which it will be applied. Calibration and combination approaches are attempts to enrich existing models with this type of knowledge. In fact, what is meant by calibration is tuning the model according to some characteristics of the new environment in which the model will be used. However, combination means building a composite model with knowledge derived from various environments. Unfortunately, the proposed calibration approach has two limitations. The first one is due to the limited capability of the calibrating action, which often consists of tuning only parameter values and multiplicative coefficients of a model. The second limitation is the model over fitting resulting from such a naive calibration. As for the proposed combination techniques, they consist mostly of a weighted sum of model outputs (i.e., linear combination). Such a combination method can be qualified as "black box". Indeed after combination, we are not able to determine which model is responsible for the decision taken; therefore, no causal relationship is visible between the decision and the attribute values. It is thus unhelpful for the decision making process because models without sufficient semantics are unlikely to be accepted and used in practice [14].

Our approach is inspired by the fact that existing quality prediction models are results of considerable experiences from which we can learn and draw valid conclusions. The majority of these models embody valuable knowledge and expertise that should be reused. On the other hand, an ideal model must shelter two types of knowledge. The first type is context specific knowledge required to make accurate predictions in the current conditions of an organization willing to apply the model. The second type is domain common knowledge required to guarantee a compromised generalization of the model in the sense that predictions remain accurate for non drastic change in the context of the organization. Consequently, by reusing existing models, the common domain knowledge is reused to reflect a wide range of software environments. This knowledge can be intuitively considered to cover a great part of an organization's new conditions when they change.

In this work we present an approach that uses both adaptation and combination while overcoming their shortcomings. We consider each model as a set of expertise chunks. Each chunk could cover a subset of the potential input set for a decision-making problem. It could also covers a particular region of the input domain of a model. For example in the case of rule-based models, the input space is naturally subdivided into a number of decision regions. In this case, an expertise chunk is simply one rule (IF–THEN statement) in the rule sets composing the model. It is needed to decide the output of the model when it is applied to the region defined by the condition of IF–THEN statement.When we consider all the expertise chunks of all the models together, combination and adaptation can be seen as a search for the best subset of expertise chunks which, when combined, constitute a model with higher predictive accuracy. By decomposing models into expertise chunks and combining them, we ensure the interpretation of the predictions performed by the resulting model. Not only is the prediction of the software quality factor provided, but we are also able to trace the model and the input attribute values responsible for the obtained prediction. Consequently, unlike the combination of model outputs proposed in the literature and described above, our approach behaves as a white-box model combination. The reuse of existing models is not straightforward as some embodied expertise must be adapted to the particular domain and context of a target organization in which the obtained model will be applied. The adaptation of expertise chunks is more intelligent and precise than a simple tuning of parameters as it adjusts the expertise characteristics. The main difference between the present work and our previous work [22] resides firstly in the application of the model combination's approach to a different problem of software quality prediction, namely, fault-proneness. In particular new data on real software systems is used to run experimental validation fault-proneness prediction problem. Secondly, our approach implementation is significantly improved, especially for GA customization.

3. Problem description and background

In this section we introduce the formalism used throughout the paper and give a cursory overview of the techniques used to combine the models. The notations and the concepts originate from machine learning formalism. To make the paper clear and transparent, we shall relate them to the appropriate software-engineering notations and concepts wherever possible.

3.1. Software quality prediction problem

The *data set* or *sample* is a set $D_n = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$ of *n examples* or *data points* where $\mathbf{x}_i \in \mathbb{R}^d$ is an *attribute vector* or *observation vector* of *d* attributes, and $y_i \in \mathscr{C}$ is a *label*. In the particular domain of software quality models, an example $\mathbf{x}_i$ represents a well-defined component of a software system (e.g., a module in the case of classical software system). The attributes of $\mathbf{x}_i$ (denoted by $x_i^{(1)}, \ldots, x_i^{(d)}$) are software *metrics* (such as the number of methods, the depth of inheritance or size of module, etc.) that are considered to be relevant to the particular software quality factor being predicted. The label y_i of the software component $\mathbf{x}_i$ represents the software quality factor being predicted. In this paper we consider the case of *classification* where the software quality factor can take only a finite number of values, so $\mathscr{C}$ is a finite set of these possible values. In software quality prediction the output space $\mathscr{C}$ is usually an ordered set $c_1, \ldots, c_q$ of labels. In the experiments described in Section 6, we consider predicting the fault-proneness of a software module. In this cases, y_i is a binary variable, taking its values in the set $\mathscr{C}$. In the case of fault-proneness $\mathscr{C} = \{$not-fault-prone, fault-prone$\}$. The proposed approach in Section 4 considers the general *q*-ary case where $\mathscr{C} = \{c_1, \ldots, c_q\}$.

A *classifier* is a function $f : \mathbb{R}^d \mapsto \mathscr{C}$ that predicts the label of any observation $\mathbf{x} \in \mathbb{R}^d$. In the framework of supervised learning, it is

assumed that (observation, label) pairs are random variables $(\mathbf{X}, Y)$ drawn from a fixed but unknown probability distribution μ, and the objective is to find a classifier f with a low error probability $\Pr_\mu[f(\mathbf{X}) \neq Y]$. Since the data distribution μ is unknown, both the selection and the evaluation of f must be based on the data D_n [23]. To this end, D_n is split into two subsets: the *training sample* D_m and the *test sample* D_{n-m}. A *learning algorithm* is a method that takes the training sample D_m as input, and outputs a classifier $f(\mathbf{x};D_m) = f_m(\mathbf{x})$. The most often used learning principle is to choose a function f_m from a function class that minimizes the *training error*

$$L(f, D_m) = \frac{1}{m}\sum_{i=1}^{m} I_{\{f(\mathbf{x}_i)\neq y_i\}}, \tag{1}$$

where I_A is the indicator function of event A. Examples of learning algorithms using this principle include the back propagation algorithm for feedforward neural nets [24] or the C4.5 algorithm for decision trees [23]. To evaluate the chosen function, the error probability $\Pr_\mu[f(\mathbf{X}) \neq Y]$ is estimated by the *test error* $L(f, D_{n-m})$. In Section 6 we will use a more sophisticated technique called *cross-validation* that allows us to use the whole data set for training and to evaluate the error probability more accurately.

In this paper, we consider the problem of combining N predefined classifiers $f_1, \ldots, f_N$ into a classifier that works well on the available dataset D_c. We call D_c the *context data* of an organization or a particular circumstance of an organization (e.g., a particular project). The dataset D_c is a sample, as defined above, collected from a random set of software components in order to represent a particular software development context. A context could be a particular software environment, a particular nature project, a particular scale of software systems, etc. The predefined classifiers are already trained of different sets of data. These data are supposed to represent different circumstances of software development. They could be from different organizations and different environments or from the same organization but collected on different software systems representing different circumstances such are different scales of software systems. The datasets that have been used to construct all the available models $f_1, \ldots, f_N$ are combined to build up a dataset called D_T. Normally, in software-engineering field, these kind of datasets are not available. However, in order to better supervise the evaluation of our proposed approach, we will simulate the already-built models by training them on a number of available datasets. These datasets are collected on different software systems representing a spectrum of projects within a software organization.

3.2. Short overview of techniques for model combination

The first and simplest way to combine N models is to find one model that works the best on the context data D_c, that is,

$$f_{\text{best}} = \arg\min_{f_j} L(f_j, D_c), \tag{2}$$

where $L(f_j, D_c)$ is the error of classification made by f_j on the context data D_c. The advantages of this method are its simplicity and the fact that it keeps the full interpretability of the original models, while its disadvantage is that it does not use the existing knowledge of several models. We will use f_{best} as a basis of comparison for evaluating more sophisticated techniques.

A somewhat more complicated way to combine the models is to construct a normalized weighted sum of their outputs. In the case of classifiers, the most commonly used combining method is known as *weighted majority* scheme, from the Basic Ensemble Method (BEM) [25]. It determines the final class label of an observation $\mathbf{x}$, in the following way:

$$f(\mathbf{x}) = \arg\max_{c\in\mathscr{C}} \sum_{j=1}^{N} w_j \|f_j(\mathbf{x}) = c\|, \tag{3}$$

where $w_j \geqslant 0, j = 1, \ldots, N$ is the weight, $\mathscr{C}$ is the set of possible class labels, and $\|a = b\|$ is one if a is equal to b and zero if otherwise. Thus, the "vote" of a classifier for a given class label has a strength proportional to its assigned weight, and the final class label is the one receiving the most "votes" [26]. To learn the weights, we can use algorithms from the family of *voting* methods such as AdaBoost [27]. The weights w_j have a natural interpretation in that they quantify our confidence in the jth model on the data set D_c, the context data. However, the "white-box" property of the original models is somewhat reduced in that there is more than one model responsible for each decision.

Machine learning algorithms that aim to combine a set of models into a more powerful classifier are generally referred to as *mixture of model* algorithms [28]. These methods are more general than ensemble methods in two respects. First, the model weights w_j are functions $w_j(\mathbf{x})$ of the input rather than constants, and more importantly, after learning the weights, the models f_j are *retrained* and the two steps are iterated until convergence. Because of this second aspect, we cannot use a general mixture of model algorithms since our purpose is to investigate the reusability of *constant* models trained or manually built on different data no longer available. There are algorithms halfway between ensemble methods and the mixture of model approach that learn localized, data-dependent weights but keep models constant. Details of this type of methods can be found in [29]. While in principle these methods could be applied to our problem, the increased complexity of the weight functions $w_j(\mathbf{x})$ makes the interpretation of the classifier rather difficult. Such an interpretation is important for the purpose of verification as well as for theory building. In particular, it is essential for the ability of software quality managers to understand the processes being modeled and to see how a model is arriving to its conclusions [14,13].

The diversity of model types and quality prediction techniques makes it difficult to define and describe generic solutions for a model combination problem. One effective approach imposes adapted solutions to each type of model. For this reason, we limit our proposal in this paper to decision trees (DT). This choice is motivated by the fact that decision tree is widely-used techniques in decision-making problems because of its comprehensibility and white-box modeling features [30,13,31,11]. Decision tree classifiers are obtained by using respectively C4.5 [23] as machine learning algorithm.

4. New approach for model combination

4.1. Approach overview

In our approach we assume the availability of a set of already-built models predicting a particular quality characteristic as well as a data set representing the context in which the resulting model will be applied. Three challenges motivate our approach of model combination. The first relies on the predictive accuracy of the resulting model. The second focuses on the adaptability of the model resulting from combination to the environment changes. With this concern the proposed approach saves the model accuracy from deterioration when it is used in new circumstances. The third deals with the ability of the approach to improve, or at least maintain the interpretability of the original models. If we consider the combination of decision trees as an example of models of high interpretability, our challenge is to produce a new model that performs accurately in a given context and has a potential of generalization higher than the original models. In addition, the resulting model is as interpretable as a decision tree.

In order to take into account the above challenges and achieve our objectives, we adopt a process of model combination and adaptation based on the following steps:

1. Decompose the existing models into *expertise chunks* to ensure the interpretability of the resulting model.
2. Combine the expertise chunks to improve the generalizability of the resulting model.
3. Adapt/calibrate the expertise chunks to improve the predictive accuracy of the resulting model.

Before introducing the main steps of our approach, we have to distinguish between two families of models. In the first, knowledge is represented by causal relationships that are often mapped into rules (e.g., decision tree, Bayesian classifier, rule based systems, fuzzy-logic based model, etc.). In the second family of models, knowledge representation is driven by statistical and regression-based relationships (e.g., neural networks, regression tree, linear regression, etc.).

Although the models used in this paper belong to the first family, we make the above distinction between predictive model structures to claim that our approach is a general one that could be applicable to the second family of models as well. In the following subsections we will be focusing on the description of steps of our approach for the first family of models (Fig. 1).

4.2. Model decomposition and expertise chunk determination

We define an expertise chunk as a "unit" of knowledge (i.e., a part of the model) needed to compute the output of the model on a particular region of the input space. Based on the above identified families of models, the decomposition can be accomplished in two ways. For the rule-based models, the input space is naturally subdivided into a number of decision regions. In this case, an expertise chunk is simply one rule (IF–THEN statement) in the rule sets composing the model. It is needed to decide the output of the model when it is applied to the region defined by the condition of IF–THEN statement.

For statistical and regression-based models, the input space is seen as a single entity without predefined rules and decision regions. In this case, we may subdivide the space into subspaces in order to make the model modular. Thus, a chunk of expertise is the piece of knowledge the model uses to determine its output on a particular subspace defined therein. In other terms, an expertise chunk is seen as a local expertise of the model on a specific region of the input space. For example, in the case of a linear regression, an expertise chunk is simply the restriction of the model linear-function on a particular subspace delimited by a hyper-rectangle.

The subdivision of the model input space can be performed by cutting each attribute domain into a number of intervals, thus a subspace is a hyper-rectangle (i.e., isothetic box) defined by the cross product of intervals from distinct attributes. The details of input space subdivision are beyond the scope of this paper.

Let us consider a decision tree that predicts whether a software module is fault prone or not. In this example the decision tree uses two structural metrics at module level as inputs; LOC-TOTAL, a measure of the total number of lines of code, and NODE-COUNT, a measure of the number of predicate nodes in a module (see Fig. 2). All the questions in the inner nodes are of the form "Is $x^{(j)} < \alpha$?" (as in the tree depicted by Fig. 2) and an expertise chunk is an IF–THEN statement such as IF $0 <$ *LOC-TOTAL* < 76 THEN the Module is *not fault prone*. Such a statement represents a path from the root of the tree down to one of the leaves. Therefore, a decision region is predefined by the IF-clause of the statement. Thus a decision tree can be seen as a set of paths from the root down to a leaf. Each of these paths represents an expertise chunk. In our example of Fig. 2, the decision tree is decomposed into three expertise chunks represented by three shaded decision regions (A, B and C). The shades attributed to the regions represent different output values (i.e.,*Fault Prone* or *Non Fault Prone* in our case).

4.3. Model preprocessing

This operation consists of identifying all the attributes (i.e., metrics) used by the available models $f_1,\ldots,f_N$ and then defining a uniform space $\mathcal{V} \subset \mathbb{R}^d$ as the cross product $\mathcal{V} = A_1 \times, \ldots, \times A_d$, where $A_j \subset \mathbb{R}$ is the uniform domain of the attribute j and d is the number of the distinct attributes used over all the models. A common domain A_j is computed for each attribute j as the union of the domains of the attribute throughout the existing models. Lower and upper boundaries L_j and U_j are then determined for each attribute. The result is that some models for which the original space will be extended will have missing decisions (i.e., model output missing values) for the new added regions. This is known as *incompleteness problem*. The missing decision can be computed and added to the model in different ways using several possible strategies. One possibility is by extending an expertise chunk among those associated

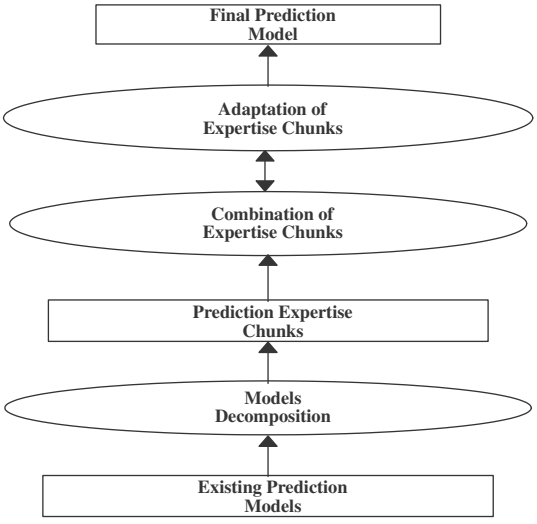

Fig. 1. Summary of the combination–adaptation process.

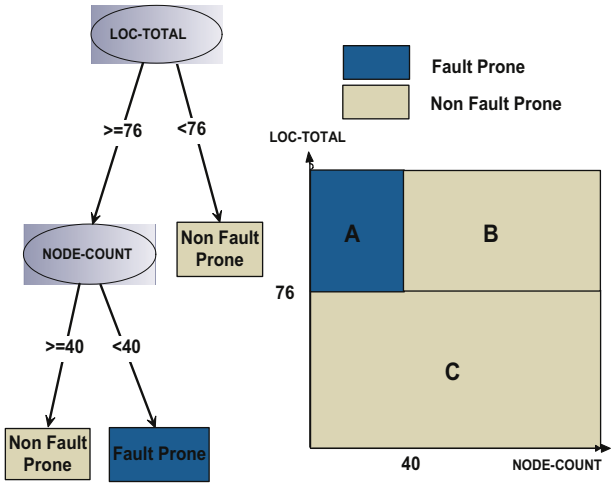

Fig. 2. A two-dimensional example of decision-tree chunks of expertise.

with the neighboring regions. Further discussion and details will be given in Section 5.

Model preprocessing is carried out prior to the combining process in order to ensure a uniform input space shared by all the models in which the combination will make sense by having different expertise chunks for the same input space regions.

4.4. Combination–adaptation process of expertise chunks

The main purpose of this process is to build a new model that performs accurately on the context data D_c using the available expertise chunks. Our proposed process builds progressively, from iteration to more accurate models by combining, adjusting or modifying a subset of available expertise chunks. This can be accomplished either by varying the scope of an expertise chunk (i.e., the dimensions of its covered region), or by modifying some of the information needed to compute the associated output. However, putting together expertise chunks from different models can lead to combinations that no longer represent well-defined models. Indeed, two general problems can occur. If two regions with conflicting decisions overlap, the obtained model is inconsistent and is not even a decision function. The second problem occurs when certain regions in the input space are not covered by any expertise chunk. This is an incompleteness problem owing to the fact that the model does not contain expertise chunks for all the regions. These two problems are illustrated for decision tree classifiers in Section 5.

Given the above described principle of the combination and adaptation operations, our approach can be seen as a learning process that builds a model not from data but from expertise chunks. However, a small data set D_c remains necessary to guide the learning process toward constructing the final model. Therefore, the context data D_c will play the role of "training data" in the sense that it "trains" the successive candidate combinations of expertise chunks. During this training process D_c is indeed used to evaluate the accuracy of the candidate combinations.

5. Approach implementation

One of the main objectives of our approach is to derive a model that makes the minimum number of errors once used in a particular context represented by the dataset D_c. Hence we are dealing with an optimization problem that aims to maximize the prediction accuracy of a searched model on D_c. On the other hand, the combination-adaptation process gradually produces new models by exploring a growing search space of potential candidate solutions which consists of all the possible combinations of expertise chunks, either original or adapted. Original expertise chunks are those used directly from the original models, while adapted ones are those modified by the adaptation step of our approach process. The rapid growth of the search space of expertise makes our combinatorial optimization a hard one.

To better understand the complexity of this optimization problem, let us suppose a very simple instance, in which we want to combine three models that use three metrics as inputs. If we assume that each metric domain is arbitrarily subdivided into three sub-domains, our models will be made up of 81 original expertise chunks ($27 = 3 \times 3 \times 3$ chunks for each model). The number of possible combinations would be greater than 2^{81}; however, this does not account for adaptations, and thus it can be considered a lower limit for the search space dimension. In a more general case of the expertise combination problem, the search space dimension could be greater than 2^N, where N is the number of original expertise chunks collected from the existing models. This problem is known to be NP-hard [32], i.e., a problem almost certainly unsolvable with a polynomial time algorithm.

The application of metaheuristic search techniques to this class of problems is a promising solution [32]. Metaheuristics are high-level frameworks that employ heuristics to find solutions for combinatorial problems at a reasonable computational cost. Moreover, they are strategies ready for adaptation to specific problems; in particular they are naturally suitable for our combination-adaptation process of expertise chunks. Among metaheuristic methods, we mention those which are based on population such as Genetic Algorithm and Ant Colony Optimization and those based on local search such as simulated annealing and Taboo Search. In particular, GA is one of the most commonly used techniques and has proved its effectiveness in combinatorial optimization [32]. Besides, GA is easily customizable for our problem, easier to understand and well accepted by the software-engineering community [33]. In this section, we will focus on the description of GA and its adaptation to our approach.

5.1. A Genetic Algorithm-based technique

A GA [34] starts with a set of initial solutions and uses biologically inspired evolution mechanisms to derive new and possibly better solutions. In other words, a GA starts with an initial solution set P_0 called the initial *population*, and generates a sequence of populations $P_1, \ldots, P_T$. Each is obtained by "mutating" the previous one and constitutes a new generation. Individuals of a population are represented by *chromosomes*, each one (a possible solution) consists of a set of *genes*. The ability of a chromosome to evolve is quantified by an objective function called *fitness function* that captures the quality of the individual it represents. In each generation, the algorithm selects some chromosomes using a selection method that gives priority to the fittest solutions. On the selected chromosomes, the algorithm applies two operators, the crossover and the mutation. These operators are applied with probabilities p_c and p_m, respectively, which are input parameters of the algorithm. The crossover operator mixes the genes of the two chromosomes, while the mutation operator randomly changes certain genes of a single chromosome. The new chromosomes produced by these operations constitute the next generation. The fittest chromosomes of each generation are automatically added to the next. The algorithm stops if a convergence criterion is satisfied or if a maximum number of generations is reached. The GA is summarized in Algorithm 1.

Algorithm 1. Summary of a genetic algorithm

```
1  Algorithm: GeneticAlgorithm()
2  Initialize P_0
3  BestFit ← fittest chromosome of P_0
4  BestFitEver ← BestFit
5  for t ← 0 to T do
6     Q ← pairs of the fittest members of P_t
7     Q' ← offsprings of pairs in Q using crossover and
   mutation
8     replace the weakest members of P_t by Q' to create P_{t+1}
9     BestFit ← fittest chromosome in P_{t+1}
10    if BestFit is fitter than BestFitEver then
11       BestFitEver ← BestFit
12    end
13 end
14 return BestFitEver
```

To apply a GA to a specific problem, elements of Algorithm 1 must be instantiated and adapted to the problem. In particular, the solutions must be encoded into chromosomes; the crossover

and mutation must be defined, and the fitness function must be specified. Customizing these elements will mostly depend on the optimization problem characteristics and the representation of the solutions, in our case, on the type of models and their knowledge representation. As mentioned earlier in this paper, we describe a customization of GA for the combination and adaptation of decision tree based models (see Section 5.2).

5.2. GA for decision tree combination

5.2.1. Decision tree encoding

A decision tree that uses d input attributes subdivides the input space of d dimensions into a number of hyper-rectangles. A hyper-rectangle or a box corresponds to a path from the root of the tree down to a leaf. To represent a decision tree as a chromosome, we consider a path as a gene, which can be represented by a (box, label) pair, where the box boundaries can be read on the nodes of the path and the label is read on the leaf node. A box $b = \{\mathbf{x} \in \mathbb{R}^d : \ell_1 < x^{(1)} \leqslant u_1, \ldots, \ell_d < \mathbf{x}^{(d)} \leqslant u_d\}$ can be represented by the vector $((\ell_1, u_1), \ldots, (\ell_d, u_d))$, where ℓ_i and u_i are the extremities of the box with respect to the attribute j. A decision tree is then encoded into a chromosome represented by a vector of (box, label) pairs (i.e., labeled boxes). To close the boxes at the extremities of the input domain, for each input variable $x^{(j)}$, we determine lower and upper boundaries L_j and U_j, respectively as shown in Section 4.3. For example, assuming that in the decision tree of Fig. 2 we have $L_{LOC\text{-}TOTAL} = 0 < \text{LOC-TOTAL} \leqslant 130 = U_{LOC\text{-}TOTAL}$ and $L_{NODE\text{-}COUNT} = 0 < \text{NODE-COUNT} \leqslant 70 = U_{NODE-COUNT}$, the tree is represented by the following structure

$$\begin{pmatrix} ((0,40),(76,130);0), \\ ((40,70),(76,130);1), \\ ((0,70),(0,76);0) \end{pmatrix},$$

Each line of this structure encodes a box defined by a set of d-segments, e.g., ((0,40),(76,130)) followed by its label (1 for *Fault Prone* and 0 for *Not Fault Prone*).

5.2.2. Crossover operator for decision trees

The simplest way to perform the crossover between the chromosomes is to use a single-point crossover [35]. Thus, each of the two parent chromosomes is cut into two subsets of genes (i.e., labeled boxes in our case). Two new chromosomes are created by interleaving the subsets. If we apply such an operation in our problem, it is possible that the resulting chromosomes may no longer represent well-defined decision functions. Two specific problems can occur. If two boxes with different decisions overlap, the model is considered *inconsistent*. In this case, the model represented by the chromosome is not even a function. The second problem occurs when the model is *incomplete*, thus, certain regions in the input space are not covered by any box. Fig. 3 illustrates these two situations.

To preserve consistency and completeness of the offspring, we propose a new crossover operator inspired by the operator defined for grouping problems [35]. In the new operator, one of the two parents is copied as is into the offspring, and then a fraction v of boxes randomly selected from the second parent is added to the offspring. The fraction v of boxes is a parameter of the algorithm. By keeping all the boxes of the first parent, completeness of the offspring is automatically ensured. To guarantee consistency, we make the added boxes predominant; that is, the added boxes are "laid over" the original ones. Fig. 4 illustrates the new crossover operator.

A *level of predominance* is added to each box as an extra element of the genes. Therefore, each gene is now a three-tuple (box, label, level). The boxes of the initial population P_0 have level 1. Each time a box is added to a chromosome, its level is set to 1 plus the maximum level of predominance in the hosting chromosome. To find the label of an input vector **x** (a software component), first we find all the boxes that contain **x**, then assign to **x** the label of the box that has the highest level of predominance. This scheme is similar in spirit to rule systems where rules have priorities that are used to resolve conflicting rules. Note also that this model retains the full

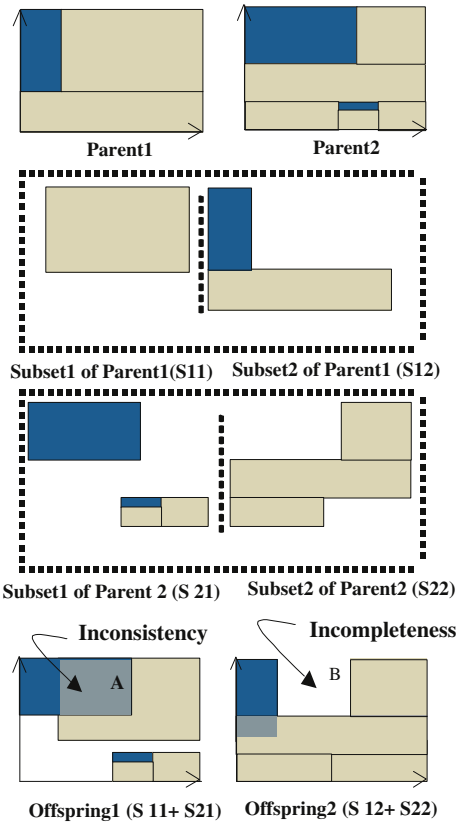

Fig. 3. Problems when using standard crossover.

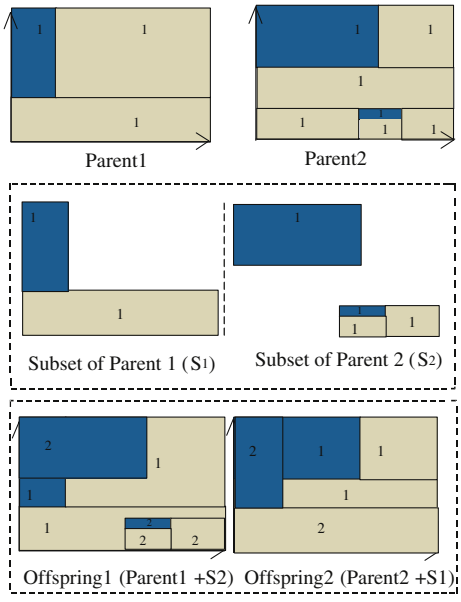

Fig. 4. Crossover that preserves consistency and completeness.

"white-box" property of the original models since each decision can be associated with a unique box from a unique model.

5.2.3. Mutation operator for decision trees

Mutation is as defined in the GA-related literature [32], a random change in the genes that has a low probability. In our problem, the mutation operator also makes a random change in the label of a box. In software quality prediction the output space $\mathscr{C}$ is usually an ordered set $c_1,\ldots,c_q$ of labels. With probability p_m, a label c_i is mutated to c_{i+1} or c_{i-1} if $1 < i < q$, to c_2 if $i = 1$, and to c_{q-1} if $i = q$. Such a strategy of mutation is implemented in this paper because of its simplicity and its closeness to real world situations. Indeed a classifier could likely make a mistake in assigning a wrong label to a particular region of the input space [23]. Thus the chosen mutation strategy would be an attempt to repair such a misclassification error.

However, other choices of mutation strategies are possible and could also be meaningful. For example, we could mutate a decision tree's chromosome by modifying the sizes of boxes, or sending the label of a box to the label of an adjacent box. But, to better focus on our main idea of model combination, we leave the investigation of the GA-operators'alternative choices for future work.

5.3. Prediction accuracy measurement

To measure the prediction accuracy of a classification model, i.e., a decision function f, we could use the *correctness function* which gives the proportion of vectors correctly labeled by f in the data set D_c [36]. The correctness measure is based on a *confusion matrix*, a commonly used notation for presenting the results of a classification accuracy evaluation. Table 1 shows a confusion matrix for q-label classification. The correctness of a decision function f is defined by the following formula:

$$C(f) = \frac{\sum_{i=1}^{k} n_{ii}}{\sum_{i=1}^{k}\sum_{j=1}^{k} n_{ij}},$$

where n_{ij} is the number of vectors in D_c with real label c_i classified as c_j (Table 1). It is clear that $C(f) = 1 - L(f)$ where $L(f)$ is the training error (see Section 3).

Software quality prediction data is often *unbalanced*, that is, software components tend to have one label with a much higher probability than other labels. For example, in our experiments we had fewer fault prone than non fault-prone modules. On an unbalanced data set, a low training error can be achieved by the constant classifier function f_{const} that assigns the majority label to every input vector. By using the training error to measure the prediction accuracy, we found that classes of the minority tended to be "neglected". To give more weight to data points with minority labels, we used Youden's *J-index* [37] to measure the prediction accuracy of our classification models. This measure is recommended by El Emam et al. in comprehensive reviews of prediction accuracy measures [36] because it is independent of the proportions of minority labels in a particular data set and therefore, is useful for producing generalizable conclusions. Youden's *J-index* is defined as:

Table 1
The confusion matrix of a decision function f. n_{ij} is the number of training vectors with real label c_i classified as c_j.

		Predicted label			
		c_1	c_2	...	c_k
Real label	c_1	n_{11}	n_{12}	...	n_{1k}
	c_2	n_{21}	n_{22}	...	n_{2k}
	⋮	⋮	⋮	⋱	⋮
	c_k	n_{k1}	n_{k2}	...	n_{kk}

$$J(f) = \frac{1}{k}\sum_{i=1}^{k} \frac{n_{ii}}{\sum_{j=1}^{k} n_{ij}}.$$

Intuitively, $J(f)$ is the average correctness per label. If we have the same number of points for each label, then $J(f) = C(f)$. However, if the data set is unbalanced, where there are significant gaps between the proportions of labels, $J(f)$ gives more relative weight to data points with rare labels. In statistical terms, $J(f)$ measures the correctness assuming that the *a priori* probability of each label is the same. Both a constant classifier f_{const} and a guessing classifier f_{guess}, which assigns random, uniformly distributed labels to input vectors, would have a J-index close to 0.5, while a perfect classifier would have $J(f) = 1$. On the other hand, for an unbalanced training set, the correctness of a guessing classifier (i.e., by chance) $C(f_{\text{guess}}) \simeq 0.5$ but that of a constant classifier (i.e., naive classifier) $C(f_{\text{const}})$ can be close to 1.

6. Evaluation methodology

6.1. Evaluation overview

Given the objectives of our approach, experiments were carried out with the problem of fault-proneness prediction in order to evaluate the performance of the outcoming models. In this case, we applied our approach to decision tree classifiers that predict Fault-Proneness (FP) of software modules.

Our approach assumes that models already exist and can be combined in order to better predict software quality in a context represented by a dataset D_c. We recall that D_c is a sample, as defined in Section 3, collected from a random set of software components in order to represent a particular software development context.

In order to better supervise the evaluation of our proposed approach, we simulate the already-built models by training them on a number of available datasets. These datasets are collected on different software systems representing a spectrum of projects within a software organization. In our experiments the models are trained on a dataset D_T using standard learning algorithms. In particular, the dataset D_T is collected from independent software systems representing different natures and scales of projects developed within NASA.

The next subsection presents our research questions and states the corresponding hypotheses.

6.2. Research questions and hypotheses

The main goal of the evaluation is to compare our combination approach to other methods of improving model generalizability. Three methods were investigated: (1) selection of the best existing model, (2) combination of model data and (3) combination of all available data. These methods are intuitive and have the advantage of not worsening the model interpretability. The models derived by these methods are, respectively, named f_{Best}, $f_{DExpert}$ and f_{DAll}. They are constructed in the following ways:

- f_{Best}: the best existing model is determined after measuring the accuracy of the existing models across a spectrum of available data sets, i.e., context data in our case. Then f_{Best} is the existing model that has the highest accuracy on the context data.
- $f_{DExpert}$: the model derived from the data that has been used to build all the "existing" models. To construct this model, the different datasets that have been used to train existing models are combined into one dataset called *DExpert*. Then *DExpert* is used as a training set to build a new model referred to as $f_{DExpert}$.

- f_{DAll}: the model derived from all available data. To construct this model, *DExpert* and the context data D_c are combined into one data set called *DAll*, and therefrom a new model f_{DAll} is trained on *DAll*.

We developed three research questions to be answered through our evaluation process. The first research question QR1 was developed in order to study the performance of our approach.

- QR1: What is the performance of the model obtained through a combination of models when compared with models obtained by other approaches? The remaining two questions were added to study additional properties of our approach without making our research focus too broad-based.
- QR2: Can the resulting model remain valid and accurate despite the context evolution? Context changes occur when an organization evolves in order to adapt to continuous technological pressure and perpetual market needs. This can be represented by variations in the context data.
- QR3: How is the accuracy of the obtained model affected by the number of existing models used in the combination process?

To give insight into our research questions, we formulated the following null hypotheses H01, H02, H03, H04 and H05 which are tested against the alternative hypotheses HA1, HA2, HA3, HA4 and HA5, respectively.

- H01: There is no difference between the prediction accuracy of f_{Mix}, the combination of existing models, and that of f_{Best}, the best existing model.
- HA1: The combination of existing models, f_{Mix}, provides more accurate predictions than the best existing model, f_{Best}.
- H02: The accuracy of the obtained model f_{Mix} is lower than that of $f_{DExpert}$, the model trained on the combined data sets used to train all the individual existing models.
- HA2: The accuracy of the obtained model f_{Mix} is not lower than that of $f_{DExpert}$.
- H03: The obtained model, f_{Mix}, provides less accurate predictions than f_{DAll}, the model trained on all the available data including the context data.
- HA3: The obtained model, f_{Mix}, is not less accurate than f_{DAll}.
- H04: The context evolution impacts the prediction accuracy of the obtained model f_{Mix} and the accuracy of the best existing model f_{Best} in the same way.
- HA4: The accuracy of the obtained model f_{Mix} is less affected by the context evolution than the accuracy of the best existing model f_S.
- H05: There is no positive correlation between the prediction accuracy of the obtained model and the number of existing models involved in the combination process.
- HA5: There is a positive correlation between the prediction accuracy of the obtained model and the number of existing models involved in the combination process.

6.3. Experiment steps

6.3.1. Model construction

To evaluate our approach with Fault-Proneness models, we consider a module $\mathbf{x}_i$ as fault prone ($y_i = 1$) if it contains faults; if not, it is not fault prone ($y_i = 0$). To build FP models we used the data set of NASA available at [38]. This data is organized into 13 data sets collected from 13 software systems within the NASA critical mission project as a part of the Software Metrics Data Program (MDP).[1] The systems used are summarized in Table 2. For each of the 13 systems, metrics at module level are available. These metrics capture well-known structural attributes of software modules such as size, complexity, coupling, and the McCabe and Halstead metrics. A short description of the metrics used is given in Table 3. In addition to these metrics, the number of faults detected during the development or the deployment is available for each module. This was used to determine for each module a class label indicating whether the module is faulty or not.

In order to experiment with our approach in different circumstances, three different data sets were randomly chosen from NASA data. From these data sets, respectively collected on CM1, PC3 and PC5 systems, three different context data sets D_{C1}, D_{C2} and D_{C3} were constituted by selecting randomly from each system data set 400 data points. The size of datasets, i.e., 400, is arbitrarily chosen, but we believe that in lack-of-data circumstances, a context data set should be relatively small. Besides, the size of context data will be a subject of a research question in our future work. The remaining 10 data sets were used as training data D_T to build fault-proneness models. To imitate the heterogeneity of real-life experts, each model was trained on a different subset of metrics and on a different software system. Thus, 100 models were trained on 100 data sets obtained from the 10 original ones in the following way: From each data set, 10 data subsets of metrics were extracted by selecting a random but different subset of metrics

Table 2
Software systems used to build FP models and constitute the context data.

System(language)	Number of modules	Number of faulty modules
KC4(Perl)	126	62
MC2(C++)	161	52
MW1(C)	403	28
KC3(JAVA)	457	43
CM1(C)	504	48
PC1(C)	1107	76
PC4(C)	1457	178
PC3(C)	1563	160
KC1(C++)	2108	265
PC2(C)	5589	24
MC1(C++)	9466	68
JM1(C)	10,877	2102
PC5(C++)	16,883	462

Table 3
Metrics at module level used as FP-model inputs in the experiment.

Name	Description
Halstead metrics	
NUM OPERANDS	# of operands in a module
NUM OPERATORS	# of operators in a module
HALSTEAD DIFFICULTY	Halstead difficulty of a module
HALSTEAD PROG TIME	Halstead programming time of a module
HALSTEAD EFFORT	Halstead effort metric of a module
MAINT SEVERITY	Maintenance severity
McCabe size and complexity metrics	
CYCLO COMPLEXITY	Cyclomatic complexity of a module
DESIGN COMPLEXITY	Design complexity of a module
DATA COMPLEXITY	Global data complexity
LOC-TOTAL	Total number of lines of a module
LOC EXECUTABLE	# lines of executable code of a module
Coupling metrics	
PARAMETER COUNT	# of parameters of module
CALL PAIRS	# calls to other functions in a module
Control flow metrics	
BRANCH COUNT	Branch count metrics
CONDITION COUNT	Number of conditionals in a module
EDGE COUNT	Number of edges in a module
NODE-COUNT	Number of nodes in a module

[1] NASA IV&V Facility Metrics Data Program

each time. Then we trained a decision tree on each data set using the machine learning algorithm C4.5 [23]. We retained 46 decision trees by eliminating constant classifiers and classifiers with training error greater than 10%. An example of one explicit step of this process could be the following: from the dataset collected on the software system *KC*1, we extract 10 datasets by selecting, each time, a random set of metrics. One of the 10 obtained datasets is using three metrics,*LOC-TOTAL, NUM OPERATORS* and *HALSTEAD EFFORT*, to describe 2108 software modules. The decision tree trained on this dataset is labeled *Model*1 and shown in Fig. 5. Note that *Model*2 and *Model*3, shown in Fig. 5, are derived by two other different software systems using the same process. Therefore, it is visible that the three models are different in terms of number of rules, input metrics and metrics value ranges. Subsequently, they enclose different chunks of expertise. The retained decision trees enclose from 2 to 17 rules.

6.3.2. Experimental design and methodology

6.3.2.1. Algorithmic setting. To verify the above hypotheses, first, we need to set up the implemented genetic algorithm used to perform model combination and generate the composite model f_{Mix}. In fact, a GA has many parameters that have to be set in order to improve the performance of particular implementations. Many studies that have provided useful guidelines for choosing GA parameters [35] were consulted to inform our GA setting. Indeed, after several tuning runs, our GA parameters were set as follows. First, the elitist strategy [35] was adopted. This means that in each iteration, the entire population is replaced, except for a small number N_e of the fittest chromosomes. The number of generations T was set to 100. The maximum number of chromosomes in a generation was 170. The values of N_e was set to 10, p_c (crossover probability) to 0.65, p_m (mutation probability) to 0.05, and v (proportion of the random subset of boxes used in the crossover operation) to 0.3. This GA setting is performed based theoretically on the above worthwhile studies and empirically on several preliminary runs of our GA.

6.3.2.2. Hypotheses verification. To precisely estimate the performance of the obtained models, we used the well-known *10-fold cross validation* for accuracy evaluations. In this technique, an evaluation data set (the context data set D_c in our case) is randomly split into 10 subsets of equal size. Then the model combination is guided by (or trained on) the union of 9 subsets of D_c, and the obtained model is tested on the remaining subset. This training-test process is repeated for all the 10 possible combinations of 9 subsets.

Therefore, to verify hypotheses H01, H02 and H03, each of the context data D_{C1}, D_{C2} and D_{C3} was randomly split into 10 subsets of equal size (i.e., folds). Within each context, the combination of models was trained on the union of 9 subsets, and the obtained model f_{Mix} was tested on the remaining subset of the context. The process was repeated for all the 10 possible combinations of 9 subsets. For each context, mean and standard deviation values were computed for the prediction accuracy of f_{Mix} for both the training and the test sample.

Indeed, by also using 10-fold cross-validation to select the best model f_{Best} for each context, all the existing models were evaluated on both the union of 9 folds (i.e., the training sample) and the remaining fold (i.e., test sample). This was repeated for all the 10 possible combinations of 9 subsets. For each existing model, mean and standard deviation values for the prediction accuracy were computed for both the training and the test sample. Thereafter, the best existing model for each context, f_{Best}, is the model achieving the highest mean of the prediction accuracy on the training sample.

To verify hypothesis H02, the performance of $f_{DExperts}$, i.e., the decision tree trained on *DExperts*, was estimated by evaluating its accuracy on each fold of the context data. For each context, we computed the mean and standard deviation of the 10 accuracies achieved by $f_{DExperts}$ on the 10-folds of the context data.

To verify hypothesis H03, a decision tree f_{DAll} was trained using C4.5 on *DAll*, the union of *DExperts* and 9 folds of the context data. The model f_{DAll} was tested on the remaining fold of the context data. Ten f_{DAll} models were obtained by repeating the training process for all the 10 possible combinations of 9 folds. Then mean and standard deviation values were computed for the prediction accuracy of f_{DAll} on the test sample. The hypothesis H03 is also verified within the three different contexts D_{C1}, D_{C2} and D_{C3}.

In order to verify hypothesis H04, each of the context data sets D_{C1}, D_{C2} and D_{C3} was randomly split into two data sets. The first was called D_{c0} and consists of 300 data points and the second was called D_a and contains 100. We used D_{c0} as context data to derive a new model f_{Mix} in the following way: a model f_{Mix} was built using two-thirds of D_{c0} and tested on the remaining one-third of D_{c0}. At the same time, f_{Best}, the most accurate existing model on the two-thirds of D_{c0} was selected. As described, the data set D_a was not used to train or evaluate f_{Mix}, however, it was used as an *application set* on which f_{Mix} was applied. By such an application of f_{Mix} on D_a we imitate a real-word situation where an organization would use a context data set to derive a prediction model f_{Mix} and apply it to new data points from the same context. More important for the verification of HA4 was to simulate the context evolution. Hence, we created for each context five application datasets by adding to D_a, fractions of 10%, 20%, 30%, 40% and 50% of new data points. These added fractions were randomly chosen from a different context. The accuracies of f_{Mix} and f_{Best} were recorded on the third of D_{c0}, D_a, (D_a + 10%), (D_a + 20%), (D_a + 30%), (D_a + 40%) and (D_a + 50%). The whole process was repeated for 3 two-thirds/one-third splits of the context data set D_{c0}. This means that three models f_{Mix} were created and three f_{Best} were selected for the three possible two-thirds/one-third splits of D_{c0}. This is a widely used method for model evaluation called *Monte Carlo cross-validation* [26]. A comparison between the accuracy variance of f_{Mix} and that of f_{Best} is performed.

To investigate the relationship between the number of used models and final model accuracy (hypothesis H05), we ran our GA algorithm with the reuse of varying number of existing models to be combined. Each time we increased the number of models by involving new ones. The combination of increasing number of models was performed in the three contexts D_{C1}, D_{C2} and D_{C3}. A model f_{Mix} was built using two-thirds of each context data set (i.e., D_{C1}, D_{C2} then D_{C3}) and tested on the remaining one-third. This was repeated three times for the 3 possible two-thirds/one-third splits of the context data set by using Monte Carlo cross-validation.

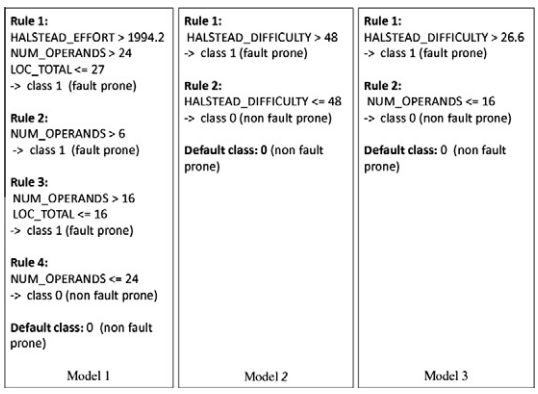

Fig. 5. Example of an already-built fault-proneness models obtained using C4.5.

The variation of the prediction accuracy of f_{Mix} was recorded in the three contexts.

6.3.3. Interpretation of result and hypotheses testing

Figs. 6–8 show, respectively in three contexts, D_{C1}, D_{C2} and D_{C3}, the prediction accuracies of f_{Mix}, f_{Best}, $f_{DExperts}$ and f_{DAll}. The results clearly show that f_{Mix} outperforms the best existing model in all three different contexts. Moreover, it yielded not only a comparable but a higher accuracy than $f_{DExperts}$, the hypothetical model trained on the data used to build the existing models. In addition, when f_{Mix} was compared to f_{DAll}, it clearly showed that model combination yielded higher performance than data combination. Table 4 shows a summary of the results verifying the hypotheses H01, H02, and H03 in three contexts D_{C1}, D_{C2}, and D_{C3}.

The higher performance of the model combination approach is graphically visible in the three different contexts. A statistical testing using the *t*-test and then the Bonferroni procedure showed a significant difference between f_{Mix} and f_{Best} (Two-tailed *t*-test, *p*-value < 0.01 in the three contexts). Using the same statistical testing procedure, the result showed that f_{Mix} is also significantly better than $f_{DExperts}$ (*t*-test, *p*-value < 0.000001) and f_{DAll} (*t*-test, *p*-value < 0.0001). The null hypotheses H01, H02, and H03 are rejected with a statistical significance of less than 1%. Beside the higher performance of the model combination approach, it succeeded to preserve another aspect of interpretability by deriving a decision tree with reasnobale number of rules ranging from 4 to 33.

With respect to the context evolution, the results shown in Figs. 9–11 confirmed the statement of hypothesis HA4 in the Contexts C1, C2, and C3. In particular, two interesting results were clearly noticeable in the three context circumstances. First, the accuracy of the resulting model f_{Mix} did not decrease significantly when changing the context. Second, f_{Mix} remained clearly better than the best model even when the context change rate reached 50%. Moreover, the comparison between the accuracy variance of f_{Mix} and that of f_{Best}, using the F-test, shows in the three contexts that the first variance is significantly smaller than the second one and the null hypothesis H04 is rejected with a statistical significance of less than 5%. Results on variance comparison is summarised in Table 5 for the three contexts. Hence, we can conclude that the behaviour of f_{Mix} is quite stable and predictible in the same context with reasonable evolutions.

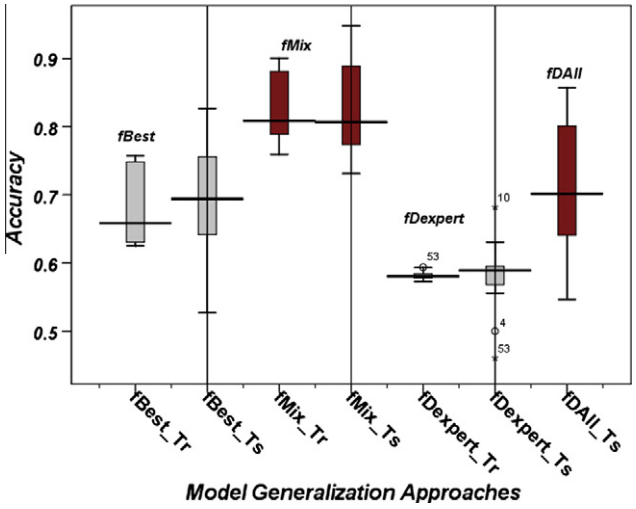

Fig. 6. Prediction accuracies in the context C1: model combination vs. best model, model-data combination, and all available-data combination.

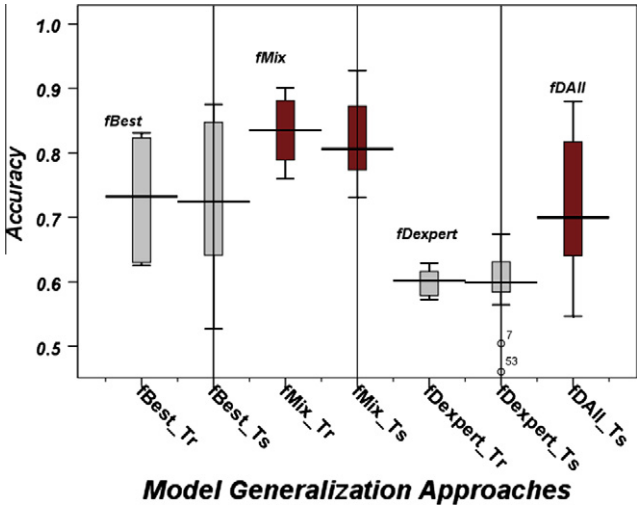

Fig. 7. Prediction accuracies in the context C2: model combination vs. best model, model-data combination, and all available-data combination.

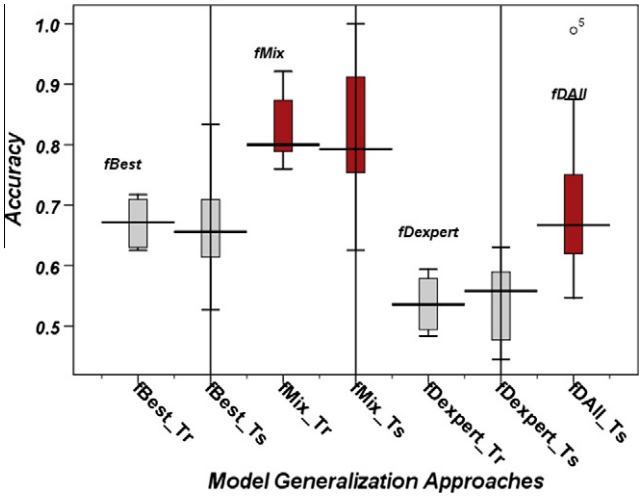

Fig. 8. Prediction accuracies in the context C3: model combination vs. best model, model-data combination, and all available-data combination.

Table 4
Experimental results. Accuracy percentage values of the compared models in contexts C1, C2, and C3 (note that f^* is the model compared to f_{mix}).

	Models			
	f_{Mix}	f_{Best}	$f_{DExperts}$	f_{DAll}
C1				
Mean	88.29	75.76	58.46	79.73
STDEV.	4.60	3.74	4.65	4.80
p-value f_{mix} vs. f^*	–	3.87E−06 (Two-tail)	1.24E−11	0.00036
C2				
Mean	88.22	82.98	61.71	80.86
STDEV.	4.37	3.70	4.71	5.24
p-value f_{mix} vs. f^*	–	0.009 (Two-tail)	6.0E−11	0.0016
C3				
Mean	88.76	69.84	49.06	75.93
STDEV.	12.4	8.42	3.81	11.83
p-value f_{mix} vs. f^*	–	0.001 (Two-tail)	5.2E−07	1.4E−05

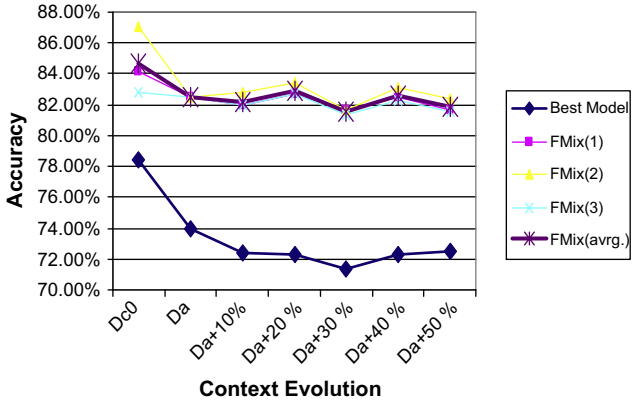

Fig. 9. Variation of f_{Mix} accuracy with context evolution: case C1.

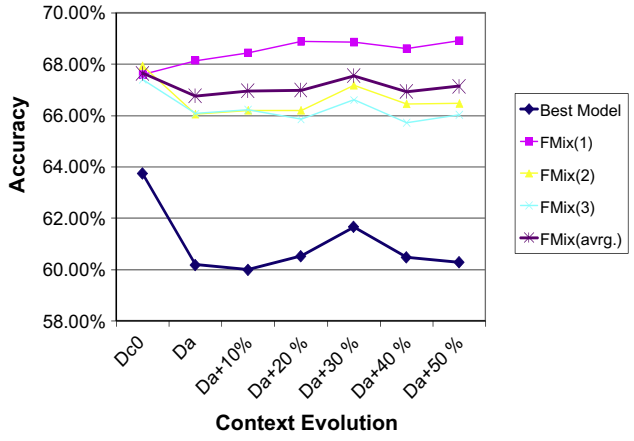

Fig. 10. Variation of f_{Mix} accuracy with context evolution: case C2.

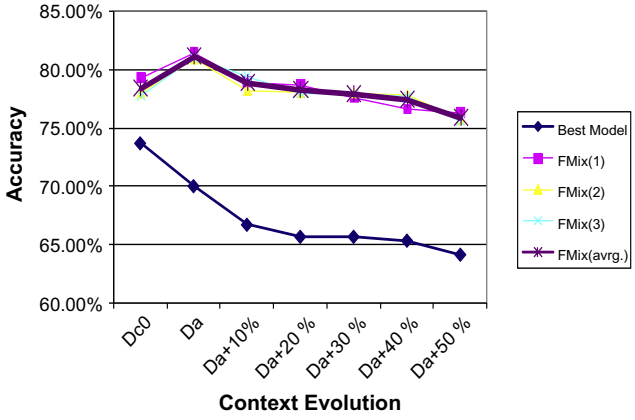

Fig. 11. Variation of f_{Mix} accuracy with context evolution: case C3.

As shown in Fig. 12, the prediction accuracy of the resulting model f_{Mix} increases as the number of models used increases. Moreover, Spearman's correlation coefficient was computed to see whether and how strongly the prediction accuracy and the number of models used are related in the three contexts. As we expected, the results presented in Table 6 show a positive correlation between the prediction accuracy and the number of models used. In all three contexts, the observed correlation coefficients are significant beyond the 1% level (p-values less than 0.01).

Table 5
Comparison between accuracy variances of f_{Best} and f_{Mix}, in the contexts C1, C2 and C3.

Contexts	C1 (%)	C2 (%)	C3 (%)
f_{Mix} accuracy (variance)	82.61 (0.01)	67.13 (0.00113)	78.23 (0.025)
f_{Best} accuracy (variance)	73.33 (0.057)	60.98 (0.018)	67.28 (0.11)
F-test p-value	0.029	0.0020	0.0446

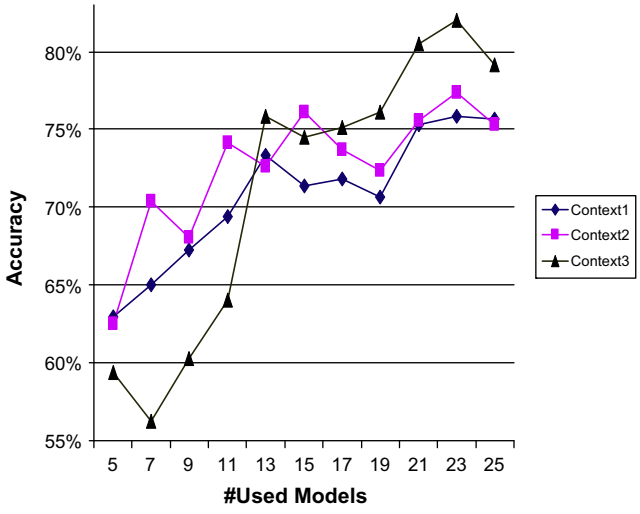

Fig. 12. Variation of f_{Mix} accuracy with the number of models used.

Table 6
Correlation between predictive accuracy and number of models used.

Contexts	C1	C2	C3
Correlation	0.92	0.79	0.92
p-value (sample of size 11)	0.00003	0.0019	0.00003

6.4. Threats to validity

Like any controlled empirical study, our experiment is subject to threats to validity. In this section we use Cook and Campbell's definition [39] in order to discuss the internal, external, construct and conclusion threats to the validity of our experiment.

6.4.1. Internal validity

The primary issue that affects validity is instrumentation. In our case, several programs and tools were required to conduct the experiment, including machine learning tools, data collection programs, metric calculators, the GA tool and model accuracy measurement. These can add variability and negatively affect the experiment. To control this problem, we chose well-known and widely used tools. The programs specifically developed for these experiments were designed and tested rigorously on several software systems and contexts with sizes ranging from 300 to 14,000 modules. A second issue affecting internal validity is the ambiguity of the direction of causal influence. This concerns, for example, the statement of hypothesis HA5 for which we have to make sure that the increase of the accuracy of the resulting model is really caused by the increase of the number of models used and not by other phenomena like the non-deterministic characteristics of GA. This ambiguity and similar issues were mitigated by repeating the experiment several times and carefully observing similar results in different contexts. A third issue concerns the model accuracy evaluation and whether it yields what it claims to measure. As

discussed in Section 5.3, Youden's *J-index* is well suited for classification problems such as quality prediction where data is likely to be unbalanced with respect to the quality factor classification.

6.4.2. External validity

Threats to external validity limit the ability to generalize the results of our experiment to industrial practice. We pretend in our approach to use individual models coming from different environments or from different circumstances within the same environment. In our experiment we use models coming from different software systems within the same organization. In order to avoid the threats to validity in this case, we applied our approach on fault-proneness of software modules for which existing models are constructed from a spectrum of real projects within NASA, a big organization used to develop software systems of different categories and scales. Nevertheless, to completely cope with the threats to external validity, it is necessary to replicate the application of our approach for the second case, where existing models are coming from many distinct industrial environments.

6.4.3. Conclusion validity

To avoid problems that affect our ability to draw correct conclusions, we used tests with high statistical power and rigorous techniques to estimate results, in particular we precisely estimated model accuracy using 10-fold cross-validation. Null hypotheses were rejected with strong significance levels, i.e., an error rate lower than 1% with the *t*-test and lower than 5% when using Bonferroni adjustment.

7. Conclusion and future work

Software engineers look for trustworthy predictions to organize their effort according to their quality needs. Experiments reported in the literature reveal a great variation in the accuracy of the proposed software quality prediction models. Indeed, when these models are applied to new systems or new environments their performance drops drastically. This is simply because there is no model that can take into account all circumstances of software development. In the present work, we proposed an approach that aims at circumventing the well-known model-generalization problem. Our approach is based on the idea of reusing existing prediction models to derive new ones rich with general knowledge as well as specific organization knowledge. The derived model works more accurately not only for a particular state of an organization context but also for evolving and modified ones. The reuse of existing prediction models is achieved through combination and adaptation of expertise embedded within them. Our approach is a three-step process:

(1) Decomposing the existing models into expertise chunks to ensure the interpretability of the resulting model.
(2) Combination of the expertise chunks to improve the generalizability of the resulting model.
(3) Adaptation/calibration of the expertise chunks to improve the predictive accuracy of the resulting model.

The combination and adaptation steps are data-driven processes and performed using a genetic algorithm. To apply our approach to a software organization, we need two inputs. The first is a set of prediction models and the second is a dataset called context data used to guide combination and adaptation processes. Our approach was developed in a way to circumvent the problem of lack of reliable data for building and validating accurate models. It can be seen as a high-level learning process that trains models not from a large set of data but from already constructed models. Therefore, just a small context dataset was needed in the learning process. In this article, we illustrated the application of our approach on decision trees. We studied the error proneness problem using NASA data. The experiments showed that our approach performs significantly better than model selection (a more obvious approach). Moreover, the combination of existing models was unexpectedly more accurate than models obtained when the data is available and despite the evolution of the organization context, the model derived by our approach succeeded to preserve its high accuracy and robustness. Besides, the genetic combination of individual models succeed to preserve interpretability by deriving a composite model built-up with reasonable number of rules compared to original models.

We plan to study other aspects of our approach such as the impact of the context characteristics on the performance of our approach. Context characteristics could include the size of the data set representing the context where the combination takes place, the nature and size of applications, the programming language, the proportion of fault-prone modules and other attributes of the system(s) from which the context data is extracted. Up to now, the approach has worked well with classification models, i.e., a finite number of model output values. We are currently exploring strategies for combining regression models with an infinite number of output values.

References

[1] L. Briand, J. Wüst, Empirical studies of quality models in object-oriented systems, in: M. Zelkowitz (Ed.), Advances in Computers, Academic Press, 2002.
[2] K. Srinivasan, D. Fisher, Machine learning approaches to estimating software development effort, IEEE Transactions on Software Engineering 21 (2) (1995) 126–137.
[3] N. Fenton, N. Ohlsson, Quantitative analysis of faults and failures in a complex software system, IEEE Transactions on Software Engineering 26 (8) (2000) 797–814.
[4] F. Xing, P. Guo, M.R. Lyu, A novel method for early software quality prediction based on support vector machine, in: ISSRE, IEEE Computer Society, 2005, pp. 213–222.
[5] N. Fenton, M. Neil, Software metrics: a roadmap, in: A. Finkelstein (Ed.), Proceedings of the Future of Software Engineering, 22nd International Conference on Software Engineering, ACM Press, 2000, pp. 357–370.
[6] L. Briand, V. Basili, W. Thomas, A pattern recognition approach for software engineering data analysis, IEEE Transactions on Software Engineering 18 (11) (1992) 931–942.
[7] N. Fenton, S. Pfleeger, Software Metrics: A Rigorous and Practical Approach, 2nd ed., International Thomson Computer Press, Boston, 1997.
[8] T. Khoshgoftaar, J. Munson, B. Bhattacharya, G. Richardson, Predictive modeling techniques of software quality from software measures, IEEE Transactions on Software Engineering 18 (11) (1992) 979–987.
[9] L. Briand, V. Basili, C. Hetmanski, Developing interpretable models with optimized set reduction for identifying high-risk software components, IEEE Transactions on Software Engineering 19 (11) (1993) 1028–1044.
[10] R. Takahashi, Software quality classification model based on mccabe's complexity measure, Journal of Systems and Software 38 (1) (1997) 61–69.
[11] T.M. Khoshgoftaar, Y. Liu, N. Seliya, Genetic programming-based decision trees for software quality classification, in: Proceedings of the Fifteenth International Conference on Tools with Artificial Intelligence (ICTAI 03), IEEE Computer Society, 2003, pp. 374–383.
[12] M. Kai, T. Boon, Model combination in the multiple-data-batches scenario, in: Proceedings of the European Conference on Machine Learning, 1997, pp. 250–265.
[13] N. Fenton, M. Neil, A critique of software defect prediction models, IEEE Transactions on Software Engineering 25 (5) (1999) 675–689.
[14] A. Gray, S. MacDonell, A comparison of techniques for developing predictive models of software metrics, Information and Software Technology 39 (1997) 425–437.
[15] A. Robins, Catastrophic forgetting, Rehearsal and Pseudorehearsal ConnectionScience 7 (1995) 123–146.
[16] Y. Liu, T. Khoshgoftaar, Reducing overfitting in genetic programming models for software quality classification, in: Proceedings of the Eighth IEEE International Symposium on High Assurance Systems Engineering (HASE 2004), Tampa, Florida, USA, 2004, pp. 25–26.
[17] T. Khoshgoftaar, E. Allen, J. Hudepohl, S. Aud, Applications of neural networks to software quality modeling of a very large telecommunications system, IEEE Transactions on Neural Networks 8 (4) (1997) 902–909.
[18] H.A. Sahraoui, M. Serhani, M. Boukadoum, Extending software quality predictive models domain knowledge, in: Proceedings of the 5th

International ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering, 2001.
[19] D. Zhang, J. Tsai, Machine learning and software engineering, Software Quality Journal (1986) 87–119.
[20] M. Lyu, A. Nikora, Applying reliability models more effectively, IEEE Software 7 (9) (1992) 43–52.
[21] P. Goodman, Practical Implementation of Software Metrics, McGraw Hill Company, 1993.
[22] S. Bouktif, B. Kégl, S. Sahraoui, Combining software quality predictive models: An evolutionary approach, in: Proceeding of the International Conference on Software Maintenance, 2002, pp. 385–392.
[23] J. Quinlan, C4.5: Programs for Machine Learning, Morgan Kaufman, 1993.
[24] D.E. Rumelhart, G.E. Hinton, R.J. Williams, Learning representations by back-propagating errors, Nature 323 (1986) 533–536.
[25] M. Perrone, L. Cooper, Artificial Neural Networks for Speech and Vision, Chapman and Hall, London, 1993 (Chapter: When networks disagree: ensemble methods for hybrid neural networks).
[26] C. Merz, Classification and regression by combining models, Ph.D. thesis, University of California, Irvine, 1998.
[27] Y. Freund, R. Schapire, A decision-theoretic generalization of on-line learning and an application to boosting, Journal of Computer and System Sciences 55 (1) (1997) 119–139.
[28] R.A. Jacobs, M.I. Jordan, S.J. Nowlan, G.E. Hinton, Adaptive mixtures of local experts, Neural Computation 3 (1) (1991) 79–87.
[29] P. Moerland, E. Mayoraz, DynaBoost: combining boosted hypotheses in a dynamic way, Tech. rep., IDIAP, Switzerland, 1999.
[30] N.E. Fenton, M. Neil, W. Marsh, P. Hearty, D. Marquez, P. Krause, R. Mishra, Predicting software defects in varying development lifecycles using bayesian nets, Information & Software Technology 49 (1) (2007) 32–43.
[31] T.M. Khoshgoftaar, N. Seliya, A. Herzberg, Resource-oriented software quality classification models, The Journal of Systems and Software 76 (2) (2005) 111–126.
[32] R. Garey, Computers and Intractability: A Guide to the Theory of NP-Completeness, Freeman and Co., 1979.
[33] M. Harman, R. Hierons, M. Proctor, A new representation and crossover operator for search based optimization of software modularization, in: AAAI Genetic and Evolutionary Computation COnference (GECCO), New York, USA, 2002, pp. 82–87.
[34] J. Holland, Adaptation in Natural Artificial Systems, University of Michigan Press, 1975.
[35] E. Falkenauer, Genetic algorithms and grouping problems, John Wiley and Sons, 1998.
[36] K. El Emam, A methodology for validating software product metrics, Tech. rep., Conseil national de recherches Canada Institut de Technologie de l'information, 2000.
[37] W.J. Youden, How to evaluate accuracy, Materials Research and Standards, ASTM, 1961.
[38] NASA, Nasa iv&v facility metrics data program repository, June 2008, <http://mdp.ivv.nasa.gov/>.
[39] T. Cook, D. Campbell, Quasi-experimentation-design and analysis issues for field settings, Tech. rep., Houghton Mifflin Company, 1979.

Update

Information and Software Technology
Volume 53, Issue 3, March 2011, Page 291

DOI: https://doi.org/10.1016/j.infsof.2010.12.004

Information and Software Technology 53 (2011) 291

Contents lists available at ScienceDirect

Information and Software Technology

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

Corrigendum

Corrigendum to "A novel composite model approach to improve software quality prediction" [Information and Software Technology 52 (12) (2010) 1298–1311]

Salah Bouktif [a,b,*], Faheem Ahmed [a], Issa Khalil [a], Giuliano Antoniol [b]

[a] Faculty of Information Technology, United Arab Emirates University, United Arab Emirates
[b] Ecole Polytechnique de Montreal, Montreal, Canada

The authors regret that Prof. Houari Sahraoui (Department of Computer Science & Operations Research, University of Montreal, Canada) was not acknowledged in the original publication.

The correct author listing should read: Salah Bouktif, Faheem Ahmed, Issa Khalil, Giuliano Antoniol, Houari Sahraoui.

DOI of original article: 10.1016/j.infsof.2010.07.003
* Corresponding author at: Faculty of Information Technology, United Arab Emirates University, United Arab Emirates.
E-mail address: salahb@uaeu.ac.ae (S. Bouktif).

0950-5849/$ - see front matter
doi:10.1016/j.infsof.2010.12.004