Combining Software Quality Predictive Models: An Evolutionary Approach

Salah Bouktif, Balázs Kégl, Houari Sahraoui
Dept. of Computer Science and Op. Res., University of Montreal
C.P. 6128 Succ. Centre-Ville, Canada, H3C 3J7
{bouktifs,kegl,sahraouh}@iro.umontreal.ca

Abstract

During the past ten years, a large number of quality models have been proposed in the literature. In general, the goal of these models is to predict a quality factor starting from a set of direct measures. The lack of data behind these models makes it hard to generalize, to cross-validate, and to reuse existing models. As a consequence, for a company, selecting an appropriate quality model is a difficult, non-trivial decision. In this paper, we propose a general approach and a particular solution to this problem. The main idea is to combine and adapt existing models (experts) in such way that the combined model works well on the particular system or in the particular type of organization. In our particular solution, the experts are assumed to be decision tree or rule-based classifiers and the combination is done by a genetic algorithm. The result is a white-box model: for each software component, not only the model gives the prediction of the software quality factor, but it also provides the expert that was used to obtain the prediction. Test results indicate that the proposed model performs significantly better than individual experts in the pool.

1. Introduction

Object oriented (OO) design and programming have reached the maturity stage. OO software products are becoming more and more complex. Quality requirements are increasingly becoming determining factors in selecting from design alternatives during software development. Therefore, it is important that the quality of the software be evaluated during the different stages of the development.

During the past ten years, a large number of quality models have been proposed in the literature. In general, the goal of these models is to predict a quality factor starting from a set of direct measures. There exist two basic approaches of building predictive models of software quality. In the first, the designer relies on historical data and applies various statistical methods to build quality models (e.g., see [2]). In the second approach, the designer uses expert knowledge extracted from domain-specific heuristics (e.g., see [6]).

The role of real software systems is crucial in both approaches: in the first, we need them to build the models, and in the second, we need them for validation. In most of the domains where predictive models are built (such as sociology, medicine, finance, and speech recognition) researchers are free to use large data repositories from which representative samples can be drawn. In the area of software engineering, however, such repositories are rare. Two main reasons can explain this fact. First, there are not many companies that systematically collect information related to software quality (such as development effort, maintenance effort, reusability effort and bug reports). The second reason is that this type of information is considered confidential. Even if the company can make it available in a particular project, only the resulting models are published. Each particular model reflects the programming style, the type of the software system, the application domain, and the profile of the company. The lack of data makes it hard to generalize and to reuse existing models. Since universal models do not exist, for a company, selecting an appropriate quality model is a difficult, non-trivial decision.

In this paper, we propose a general approach and a particular solution to this problem. The main idea is to combine and adapt existing models in such way that the combined model works well on the particular system or in the particular type of organization. The combination is entirely data-driven: we use the existing models as independent experts, and evaluate them on the data that is available to us. An important property of software quality models, beside their reliability, is their interpretability [8]. Practitioners want *white-box* models: they want to know not only the predicted quality of a certain software component but also the *reason* of the assessment, so models without sufficient semantics are unlikely to be used in practice. This makes sophisticated *black-box* type models (such as non-linear statistical models or neural networks) unsuitable for the process of decision making, in spite of the predictive power of these techniques. In our particular solution, the experts are as-

sumed to be decision tree or rule-based classifiers and the combination is done by a genetic algorithm. The result is a white-box model: for each software component, not only the model gives the prediction of the software quality factor, but it also provides the expert that was used to obtain the prediction. As a comparison, we show the results obtained by using a simpler and more general although less powerful method developed for similar tasks in the domain of machine learning.

The paper is organized as follows. Section 2 formulates the problem, introduces the formalism used throughout the paper, and gives a short overview of the techniques used to combine the models. The two techniques are described in detail in Sections 3 and 4. Test results are given in Section 5.

2. Problem formulation

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

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 a *attribute vector* or *observation vector* of d attributes, and $y_i \in C$ 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 *class* in the case of OO software). 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, 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 C is a finite set of these possible values. In software quality prediction the output space C is usually an ordered set $c_1, \ldots, c_k$ of labels. In the experiments described in Section 5, we consider predicting the *stability* of a software component. In this case, y_i is a binary variable, taking its values from the set $C = \{-1(\text{unstable}), 1(\text{stable})\}$. For the sake of simplicity, the machine learning method in Section 3 is described for the binary classification case (it can easily be extended to the k-ary case). The genetic algorithm-based technique in Section 4 considers the general n-ary case.

A *classifier* is a function $f\colon \mathbb{R}^d \mapsto C$ 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 $\mathbb{P}_\mu[f(\mathbf{X}) \neq Y]$. Since the data distribution μ is unknown, both the selection and the evaluation of f must be based on the data D_n. To this end, D_n is cut into two parts, 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 [14] or the C4.5 algorithm for decision trees [13]. To evaluate the chosen function, the error probability $\mathbb{P}_\mu[f(\mathbf{X}) \neq Y]$ is estimated by the *test error* $L(f, D_{n-m})$. In Section 5 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 decision tree classifiers $f_1, \ldots, f_N$ called *experts* into a classifier that works well on the available data set. The first and simplest way to combine N experts is to find the one expert that works the best on the training data, that is,

$$f_{\text{best}} = \arg\min_{f_j} L(f_j, D_m).$$

The advantage of this method is its simplicity and that it keeps the full interpretability of the original experts, while its disadvantage is that it does not use the combined knowledge of several experts. We will use f_{best} as a benchmark for evaluating more sophisticated techniques.

A somewhat more complicated way to combine the experts is to construct a normalized weighted sum of their outputs and then to threshold the output at zero. Formally, let

$$f(\mathbf{x}) = \begin{cases} 1 & \text{if } \sum_{j=1}^{N} w_j f_j(\mathbf{x}) \geq 0, \\ -1 & \text{otherwise.} \end{cases} \tag{2}$$

where $w_j \geq 0, j = 1, \ldots, N$. To learn the weights, we use the AdaBoost [7] algorithm from the family of *ensemble* or *voting* methods. We will refer to the classifier produced by this algorithm as $f_{\text{boost}}(\mathbf{x})$. The method is described in detail in Section 3. The weights w_j of the experts in classifiers of the form (2) have a natural interpretation that they quantify our confidence in the jth expert on our data set D_n. However, the white-box property of the original experts is somewhat reduced in the sense that there is more than one expert responsible for each decision.

Machine learning algorithms that intend to combine a set of experts into a more powerful classifier are generally referred to as *mixture of expert* algorithms [10]. These methods are more general than ensemble methods in two aspects. First, the expert weights w_j are functions $w_j(\mathbf{x})$ of the input rather than constants, and secondly and more importantly, after learning the weights, the experts f_j are *retrained* and the two steps are iterated until convergence. Because of this second property, we cannot use general mixture of expert algorithms since our purpose is to investigate the reusability of *constant* experts trained or manually built on different data. There are algorithms halfway between ensemble methods and the mixture of expert approach that learn localized, data-dependent weights but keep experts constant [12, 11]. 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.

In Section 4 we describe an approach of combining experts using a genetic algorithm. The method is designed specifically to combine and evolve decision tree classifiers into one final expert $f_{\text{gen}}(\mathbf{x})$ in such a way that it retains the full transparency (white box property) of decision trees.

3. The AdaBoost algorithm

The basic idea of the algorithm is to iteratively find the best expert on the weighted training data, then reset the weight of this expert as well as the weights of the data points. Hence, the algorithm maintains two weight vectors, the weights $\mathbf{b} = (b_1, \ldots, b_m)$, $b_i \geq 0, i = 1, \ldots, m$ of the data points and the weights $\mathbf{w} = (w_1, \ldots, w_N)$, $w_j \geq 0, j = 1, \ldots, N$ of the expert classifiers. Intuitively, the weight b_i indicates how "hard" it is to learn the point $\mathbf{x}_i$, while the weight w_j signifies how "good" expert f_j is. The tth iteration starts by finding the expert $f_{j_t^*}$ that minimizes the *weighted training error*

$$L_{\mathbf{b}}(f, D_m) = \frac{1}{m} \sum_{i=1}^{m} b_i I_{\{f(\mathbf{x}_i) \neq y_i\}}.$$

Then both the weight of $f_{j_t^*}$ and the weights b_i of the data points are reset. The weight adjustment of $f_{j_t^*}$ depends on the weighted error $L_{\mathbf{b}}(f_{j_t^*}, D_m)$ of $f_{j_t^*}$: the smaller the error, the more the weight $w_{j_t^*}$ of $f_{j_t^*}$ increases. The weight of the data point $\mathbf{x}_i$ increases if $f_{j_t^*}$ commits an error on $\mathbf{x}_i$, and decreases otherwise. In this way as the algorithm progresses, experts are asked to concentrate more and more on data points with large weights, that is, points that are "hard to learn". The number of iterations T can be either preset to a constant value or decided by using validation techniques. At the end of the routine, the weighted sum of the experts $\sum_{j=1}^{N} w_j f_j(\cdot)$ is returned and used as a classifier as in (2). Figure 1 summarizes the algorithm. The explanation of the details of the algorithm and the theoretical justification of the steps can be found in [7, 15].

ADABOOST$(D_m, (f_1, \ldots, f_N), T)$

1 $\mathbf{b} \leftarrow (1/m, \ldots, 1/m)$ ▷ *initial point weights*
2 $\mathbf{w} \leftarrow (0, \ldots, 0)$ ▷ *initial expert weights*
3 **for** $t \leftarrow 1$ **to** T
4 $j_t^* \leftarrow \arg\min_j L_{\mathbf{b}}(f_j, D_m)$ ▷ *best expert*
5 $\varepsilon_t \leftarrow L_{\mathbf{b}}(f_{j_t^*}, D_m)$ ▷ *weighted error*
6 $w_{j_t^*} \leftarrow w_{j_t^*} + \frac{1}{2} \ln\left(\frac{1-\varepsilon_t}{\varepsilon_t}\right)$ ▷ *weight adjustment of* $f_{j_t^*}$
7 **for** $i \leftarrow 1$ **to** m ▷ *reweighting the points*
8 **if** $f_{j_t^*}(\mathbf{x}_i) \neq y_i$ **then**
9 $b_i \leftarrow \frac{b_i}{2\varepsilon_t}$ ▷ *error* $\Rightarrow b_i \uparrow$
10 **else**
11 $b_i \leftarrow \frac{b_i}{2(1-\varepsilon_t)}$ ▷ *no error* $\Rightarrow b_i \downarrow$
12 **return** $\sum_{j=1}^{N} w_j f_j(\cdot)$

Figure 1. The pseudocode of the AdaBoost algorithm.

4. A genetic algorithm-based technique

Combining models is a difficult problem. Exhaustive and local-search methods are inefficient when the problem involves a large set of models that use different metrics. Genetic algorithms (GA) [9] offer an interesting alternative to these approaches. The basic idea of a GA is to start from a set of initial solutions, and to use biologically inspired evolution mechanisms to derive new and possibly better solutions [9]. The derivation starts by an initial solution set P_0 (called the initial *population*), and generates a sequence of populations $P_1, \ldots, P_T$, each obtained by "mutating" the previous one. Elements of the solution sets are called *chromosomes*. The fitness of each chromosome is measured by an objective function called the *fitness function*. Each chromosome (possible solution) consists of a set of *genes*. At each generation, the algorithm selects some pairs of chromosomes using a selection method that gives priority to the fittest chromosomes. On each selected pair, the algorithm applies one of two operators, crossover and mutation, with probability p_c and p_m, respectively, where p_c and p_m are input parameters of the algorithm. The crossover operator mixes genes of the two chromosomes, while the mutation operator randomly changes certain genes. Each selected

pair of chromosomes produces a new pair of chromosomes that constitute the next generation. The fittest chromosomes of each generation are automatically added to the next generation. The algorithm stops if a convergence criterion is satisfied or if a fixed number of generations is reached. The algorithm is summarized in Figure 2.

```
GENETICALGORITHM(p_c, p_m, ...)
 1  Initialize P_0
 2  BESTFIT ← fittest chromosome of P_0
 3  BESTFITEVER ← BESTFIT
 4  for t ← 0 to T
 5     Q ← pairs of the fittest members of P_t
 6     Q' ← offsprings of pairs in Q using
          crossover and mutation
 7     replace the weakest members of P_t by Q'
          to create P_{t+1}
 8     BESTFIT ← fittest chromosome in P_{t+1}
 9     if BESTFIT is fitter than BESTFITEVER then
10        BESTFITEVER ← BESTFIT
11  return BESTFITEVER
```

Figure 2. The summary of a genetic algorithm.

To apply a GA to a specific problem, elements of the generic algorithm of Figure 2 must be instantiated and adapted to the problem. In particular, the solutions must be encoded into chromosomes, and the two operators and the fitness function must be defined. In the rest of this section, we present each of these aspects for our algorithm.

4.1. Model coding

Our algorithm is designed specifically to combine a set of decision tree classifiers into one final classifier. A decision tree is a complete binary tree where each inner node represents a yes-or-no question, each edge is labeled by one of the answers, and terminal nodes contain one of the classification labels from the set C. The decision making process starts at the root of the tree. Given an input vector $\mathbf{x}$, the questions in the internal nodes are answered, and the corresponding edges are followed. The label of $\mathbf{x}$ is determined when a leaf is reached.

If all the questions in the inner nodes are of the form "Is $x^{(j)} > \alpha$?" (as in the tree depicted by Figure 3), the decision regions of the tree can be represented as a set of isothetic boxes (boxes with sides parallel to the axes). Figure 4 shows this representation of the tree of Figure 3. To represent the decision trees as chromosomes in the GA, we enumerate these decision regions in a vector. Formally, each gene is a (box,label) pair where the box $b = \{\mathbf{x} \in \mathbb{R}^d : \ell_1 < x^{(1)} \leq u_1, \ldots, \ell_d < \mathbf{x}^{(d)} \leq u_d\}$ is represented by the vector $\big((\ell_1, u_1), \ldots, (\ell_d, u_d)\big)$, and a vector of these (box,label) pairs constitutes a chromosome representing the decision tree. To close the opened boxes at the extremities of the input domain, for each input variable $x^{(j)}$, we define lower and upper bounds L_j and U_j, respectively. For example, assuming that in the decision tree of Figure 3 we have $0 < \text{NPPM} \leq 100$ and $0 < \text{LCOMB} \leq 50$, the tree is represented by the nested vector

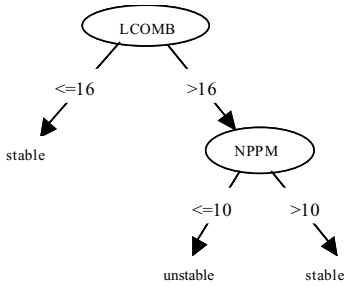

Figure 3. A decision tree for stability prediction.

$$\begin{pmatrix} \big((0,10),(16,50);-1\big), \\ \big((10,100),(16,50);1\big), \\ \big((0,100),(0,16);1\big) \end{pmatrix}.$$

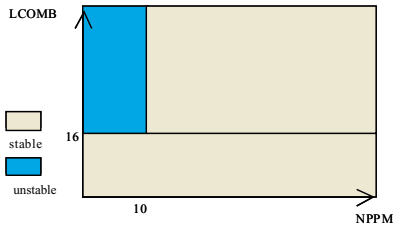

Figure 4. A two-dimensional example of decision tree output regions.

4.2. The crossover operator

A standard way to perform the crossover between the chromosomes is to cut each of the two parent chromosomes into two subsets of genes (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 can no longer represent well-defined decision functions. Two specific problems can occur. If two boxes overlap, we say that the model is *incon-*

sistent. In this case, the model represented by the chromosome is not even a function. The second problem is when the model is *incomplete*, that is, certain regions in the input domain are not covered by any boxes. Figure 5 illustrates these two situations.

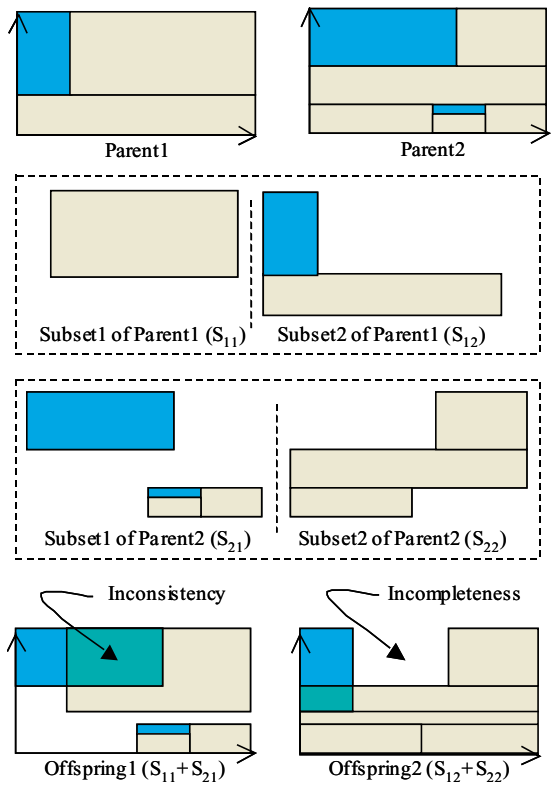

Figure 5. Problems when using standard crossover.

To preserve the consistency and the completeness of the offsprings, we propose a new crossover operator inspired by the operator defined for grouping problems [5]. To obtain an offspring, we select a random subset of boxes from one parent and add it to the set of boxes of the second parent. By keeping all the boxes of one of the parents, completeness of the offspring is automatically ensured. To guarantee consistency, we make the added boxes predominant (the added boxes are "laid over" the original boxes). The size of the random subset is v times the number of boxes of the parent, where v is a parameter of the algorithm. Figure 6 illustrates the new crossover operator.

After the crossover, it is possible that some of the decision regions are not isothetic boxes. To keep the offspring consistent with the model coding described in Section 4.1, we transform all the residual regions into boxes, as indicated by Figure 7.

The residual transformation is extremely time and space

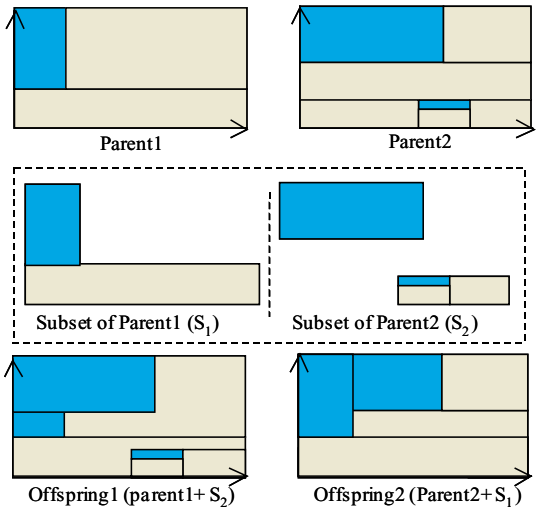

Figure 6. Crossover that preserves consistency and completeness.

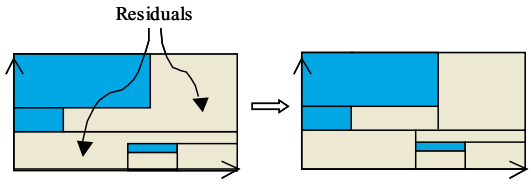

Figure 7. The residual transformation.

consuming. On the one hand, residuals must be computed for each possible pairs of boxes after each crossover, for several crossovers per generation and for several generations. On the other hand, in the worse case, a residual must be decomposed into $2d$ boxes (where d is the dimension of the input space), so after several generations, the space can become very fragmented.

To circumvent these problems, we modify our coding scheme by keeping all the boxes (even those that have hidden parts) and by adding the *level of predominance* as an extra element to 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 predominant box is added to a chromosome, its level is set to 1 plus the maximum level in the hosting chromosome. To find the label of a input vector x (a software component), first we find all the boxes that contain x, and assign to x the label of the box that have 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 white-box property of the original experts since each decision can be associated with a unique

box, and each box comes from a unique expert.

4.3. The mutation operator

Mutation is a random change in the genes that happens with a small probability. In our problem, the mutation operator randomly changes the label of a box. In software quality prediction the output space C is usually an ordered set $c_1, \ldots, c_k$ of labels. With probability p_m, a label c_i is changed to c_{i+1} or c_{i-1} if $1 < i < k$, to c_2 if $i = 1$, and to c_{k-1} if $i = k$.

In general, other types of mutations are possible. For example, we could change the size of a box, or we could set the label of a box to the label of an adjacent box. The main reason of our mutation operator is its simplicity.

4.4. The fitness function

To measure the fitness of a decision function f represented by a chromosome, one could use the *correctness function*

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

where n_{ij} is the number of training vectors 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 defined in (1).

		predicted label			
		c_1	c_2	$\ldots$	c_k
real label	c_1	n_{11}	n_{12}	$\ldots$	n_{1k}
	c_2	n_{21}	n_{22}	$\ldots$	n_{2k}
	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
	c_k	n_{k1}	n_{k2}	$\ldots$	n_{kk}

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.

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 much more stable than unstable classes. On an unbalanced data set, 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 for measuring the fitness, we found that the GA tended to "neglect" unstable classes. To give more weight to data points with minority labels, we decided to use Youden's *J-index* [16] defined as

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

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, $J(f)$ 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} (that 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 $C(f_{\text{guess}}) \simeq 0.5$ but $C(f_{\text{const}})$ can be close to 1.

5. Evaluation

To test the algorithms, we constructed a "semi-real" environment in which the "in-house" data set is a real software system, but the experts are "simulated": they are decision tree classifiers trained on independent software system data. To imitate the heterogeneity of real-life experts, each expert was trained on a different subset of metrics and on a different software system. Although we are aware of the limitations of this model, we found that it simulated reasonably well a realistic situation and yielded some interesting results.

5.1. Experimental settings

The chosen task in our experiments is to predict the stability of Java classes. The main reason of our choice is that it is relatively easy to objectively measure stability by looking at consecutive major versions of the same software. In our case, we say that a class x_i is stable ($y_i = 1$) if its public interface of the jth version is included in the public interface of the $(j+1)$th version, and unstable ($y_i = -1$) otherwise.

Since the structure of an OO software component is very important in determining its stability, we chose 22 structural software metrics for the attributes constituting the observation vectors $\mathbf{x}_i$ (Table 2). The metrics belong to one of the four categories of coupling, cohesion, inheritance, and complexity, and constitute a union of metrics used in different theoretical models [4, 1, 17, 3].

5.2. Data collection

The selected software metrics were extracted from 11 software systems (Table 3) using the ACCESS tool of the

Name	Description
Cohesion metrics	
LCOM	lack of cohesion methods
COH	cohesion
COM	cohesion metric
COMI	cohesion metric inverse
Coupling metrics	
OCMAIC	other class method attribute import coupling
OCMAEC	other class method attribute export coupling
CUB	number of classes used by a class
CUBF	number of classes used by a memb. funct.
Inheritance metrics	
NOC	number of children
NOP	number of parents
NON	number of nested classes
NOCONT	number of containing classes
DIT	depth of inheritance
MDS	message domain size
CHM	class hierarchy metric
Size complexity metrics	
NOM	number of methods
WMC	weighted methods per class
WMCLOC	LOC weighted methods per class
MCC	McCabe's complexity weighted meth. per cl.
DEPCC	operation access metric
NPPM	number of public and protected meth. in a cl.
NPA	number of public attributes

Table 2. The 22 software metrics used as attributes in the experiments.

Discover© environment[1]. The Jedit and Jetty systems were selected to serve as the "in-house" software systems. We created a data set D_n of 690 data vectors using the classes in these two systems. The remaining 9 systems were used to "create" 23 experts in the following way. First we formed 15 subsets of the 22 software metrics by combining two, three, or four of the metrics categories in all the possible ways, and created $15 \times 9 = 135$ data sets. Then we trained a decision tree classifier on each data set using the C4.5 algorithm [13]. We retained 23 decision trees by eliminating constant classifiers and classifiers with training error more than 10%.

5.3. Algorithmic settings

The only free parameter of the AdaBoost algorithm, the number of iterations T, was set to 100. The GA has several parameters that were set based on experiments. To form successive generations, the elitist strategy was used: 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 160 to have a reasonable execution time. The values of N_e, p_c (crossover probability), p_m (mutation probability), and v (proportion of the random subset of boxes used in the crossover operation) change with the number of generations t. Table 4 indicates the actual values.

[1] Available at http://www.mks.com/products/discover/developer.shtml.

System	Number of (major) versions	Number of classes
Bean browser	6(4)	388–392
Ejbvoyager	8(3)	71–78
Free	9(6)	46–93
Javamapper	2(2)	18–19
Jchempaint	2(2)	84
Jedit	2(2)	464–468
Jetty	6(3)	229–285
Jigsaw	4(3)	846–958
Jlex	4(2)	20–23
Lmjs	2(2)	106
Voji	4(4)	16–39

Table 3. The software systems used to train and to combine the experts.

t	0–10	11–30	31–99
N_e	3	5	10
p_c	0.65	0.65	0.60
p_m	0.02	0.03	0.05
v	0.3	0.1	0.05

Table 4. GA parameters.

5.4. Results

To accurately estimate the correctness and the J-index of the trained classifiers, we used 10-fold cross validation. In this technique, the data set is randomly split into 10 subsets of equal size (69 points in our case). A decision function is trained on the union of 9 subsets, and tested on the remaining subset. The process is repeated for all the 10 possible combinations, and mean and standard deviation values are computed for the correctness and J-index for both the training and the test sample. Table 5 shows our results.

The relatively low correctness rates indicate that the chosen problem of predicting software quality factor itself is difficult problem. Nevertheless, test results show that our

		Correctness $C(f)$	J-index $J(f)$
Training	f_{best}	68.55(0.70)	57.43(0.50)
	f_{boost}	69.55(0.60)	59.41(0.42)
	f_{gen}	73.01(1.10)	69.24(1.81)
Test	f_{best}	68.55(6.30)	57.49(3.52)
	f_{boost}	69.13(6.42)	58.92(3.84)
	f_{gen}	72.12(4.18)	68.89(5.17)

Table 5. Experimental results. The mean(standard deviation) percentage values of the correctness and the J-index.

approach of combining expert knowledge can yield significantly better results than using individual models. We strongly believe that if we use more numerous and real experts on cleaner, less ambiguous data, the improvement will be even more significant. In particular, the results show that AdaBoost performed slightly better than the best expert, and GA performed slightly better than AdaBoost, although the only statistically significant difference is in the J-index in favor of the GA. This is not surprising since the GA was trained with the J-index as the fitness function. The small difference between the training and test results indicate that there is no visible overfitting.

6. Conclusion

In this paper we propose an evolutionary approach for combining and adapting existing software quality predictive models (experts) to a particular context. The resulting model can be interpreted as a "meta-expert" that selects the best expert for each given case. This notion corresponds well to the "real world" in which individual predictive models, coming from heterogeneous sources, are not universal. Indeed, an ideal predictive model can be seen as the mixture of two types of knowledge: (1) domain common knowledge and (2) context specific knowledge. In the existing models, one of the two types is often missing. On one hand, theoretical models are designed to cover the common domain knowledge and their application requires some adaptation/calibration to the particular context of a company. On the other hand, historical-data-based models contain the knowledge that was abstracted from a context-specific data set. Our approach takes the best of both worlds. By combining several existing models, it reuses the common domain knowledge and by guiding this combination by the company specific data, it integrates its specific context. Our preliminary results show that our combination method of models can perform significantly better then individual models. Issues of future research include the evaluation of the approach on real experts proposed in the literature and the comparison of our approach to other white-box techniques. To show the universality of our technique, we also intend to evaluate our method on data coming from other domains where representative benchmarks exist.

References

[1] L. Briand, P. Devanbu, and W. Melo. An investigation into coupling measures for C++. In *Proceedings of the 19th International Conference on Software Engineering*, 1997.

[2] L. Briand and J. Wüst. Empirical studies of quality models in object-oriented systems. In M. Zelkowitz, editor, *Advances in Computers*. Academic Press, 2002.

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

[4] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions of Software Engineering*, 20(6):476–493, 1994.

[5] E. Falkenauer. *Genetic Algorithms and Grouping Problems*. John Wiley & Sons, 1998.

[6] N. E. Fenton and N. M. Software metrics: roadmap. In A. Finkelstein, editor, *The Future of Software Engineering, 22nd International Conference on Software Engineering*, pages 357–370. ACM Press, 2000.

[7] Y. Freund and R. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of Computer and System Sciences*, 55(1):119–139, 1997.

[8] A. Gray and S. MacDonell. A comparison of techniques for developing predictive models of software metrics. *Information and Software Technology*, 39:425–437, 1997.

[9] J. H. Holland. *Adaptation in Natural Artificial Systems*. University of Michigan Press, 1975.

[10] R. A. Jacobs, M. I. Jordan, S. J. Nowlan, and G. E. Hinton. Adaptive mixtures of local experts. *Neural Computation*, 3(1):79–87, 1991.

[11] R. Meir, R. El-Yaniv, and S. Ben-David. Localized boosting. In *Proceedings of the 13th Annual Conference on Computational Learning Theory*, pages 190–199, 2000.

[12] P. Moerland and E. Mayoraz. DynaBoost: Combining boosted hypotheses in a dynamic way. IDIAP-RR 9, IDIAP, Switzerland, 1999.

[13] J. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.

[14] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *Nature*, 323:533–536, 1986.

[15] R. E. Schapire, Y. Freund, P. Bartlett, and W. S. Lee. Boosting the margin: a new explanation for the effectiveness of voting methods. *Annals of Statistics*, 26(5):1651–1686, 1998.

[16] W. J. Youden. How to evaluate accuracy. *Materials Research and Standards, ASTM*, 1961.

[17] H. Zuse. *A Framework of Software Measurement*. Walter de Gruyter, 1998.