150/2003

Raport Badawczy

Research Report

RB/5/2003

# Genetic granular classifiers in modeling software quality

W. Pedrycz, G. Succi

**Instytut Badań Systemowych**
**Polska Akademia Nauk**

**Systems Research Institute**
**Polish Academy of Sciences**

**POLSKA AKADEMIA NAUK**

**Instytut Badań Systemowych**

ul. Newelska 6

01-447 Warszawa

tel.: (+48) (22) 8373578

fax: (+48) (22) 8372772

Kierownik Pracowni zgłaszający pracę:
Prof. dr hab. inż. Janusz Kacprzyk

Warszawa 2003

# Genetic Granular Classifiers in Modeling Software Quality

Witold Pedrycz* and Giancarlo Succi**

*Department of Electrical and Computer Engineering
Universtiy of Alberta
Edmonton, AB, Canada T6G 2G7

** Department of Computer Science
University of Bozen
Bozen, Italy

**Abstract** Hyperbox classifiers are one of the most appealing and intuitively transparent classification schemes. As the name stipulates, these classifiers are based on a collection of hyperboxes – generic and highly interpretable geometric descriptors of data belonging to a given class. The hyperboxes translate into conditional statements (rules) of the form "if $feature_1$ is in [a,b] and $feature_2$ is in [d,f] and .. and $feature_n$ is in [w,z] then class ω" where the intervals ([a,b],…[w,z]) are the respective edges of the hyperbox. The proposed design process of hyperboxes comprises of two main phases. In the first phase, a collection of "seeds" of the hyperboxes is formed through data clustering (realized by means of the Fuzzy C-Means algorithm, FCM). In the second phase, the hyperboxes are "grown" by applying mechanisms of genetic optimization (and genetic algorithm, in particular). We reveal how the underlying geometry of the hyperboxes supports an immediate interpretation of software data concerning software maintenance and dealing with rules describing a number of changes made to software modules and their linkages with various software measures (such as size of code, McCabe cyclomatic complexity, number of comments, number of characters, etc.)

**Keywords** Software quality, hyperbox geometry of classifiers, software measures, genetic algorithms, fuzzy clustering, fuzzy C-Means (FCM)

## 1. Introductory Comments

Pattern classifiers [4] come with their underlying geometry. As a matter of fact, the geometry and related learning mechanisms are the two essential elements that determine the resulting performance of any pattern classifier. What we witness today is an evident panoply of technologies used to design classifiers: neural networks, fuzzy rule-based systems (and ensuing fuzzy classifiers), k-Nearest Neighbor (k-NN) classifiers, and genetic optimization are just a few representative categories of pattern classifiers. The assumed geometry of the classifier (say, hyperplanes, receptive fields, etc.) implies its learning capabilities and resulting accuracy. Quantitative Software Engineering inherently dwells on empirical data that need to be carefully analyzed. As there are no physical underpinnings characterizing software processes and ensuing software products, the commonly encountered assumptions that govern "standard" regression models and classifiers do not hold and could be difficult to justify. On the other hand, we anticipate that models developed in the realm of Software Engineering should be transparent meaning that their readability and a level of comprehension are high. Being faced by the lack of physical underpinnings on one hand and the need for the user-friendliness on the other, it becomes advisable to set up a logic-based development framework and adopt logic as a paramount feature of the resulting constructs. In this way the interpretability of the models is inherently associated with the logic roots of the environment. Likewise their geometry is also implied by the logic fundamentals we have started with at the beginning. The quest for interpretability of models in Software Engineering comes with a variety of facets and diverse applications including software specification, software maintenance, reliability, portability. Apparently, some categories of models are geared towards interpretational clarity and make it one of the dominant features. This becomes visible in rule-based systems, cf. [3][6] [16] in which this feature comes hand in hand with the requirement of high accuracy and substantial prediction capabilities. Overall, there are several general observations worth making with this respect

- software processes and products are human-centric and do not adhere to any physical underpinnings. It light of their origin, it is legitimate to focus on logic-rich and transparent models

- domain knowledge becomes an integral part of the model especially in case of its availability and limited availability of experimental data as well as substantial variability of the software products and processes

- interpretability of the developed models becomes an important and highly desirable feature of models of software artifacts which helps designer and manager gain a better insight into the specificity of the particular model and derive conclusions. In a nutshell, the geometry of the model (say a predictor or classifier0 needs to be easily comprehended by the user.

- Software measures (metrics) [7][14][15][17] becomes essential indicators of software quality (such as reliability, maintenance effort, development cost, etc). It becomes then essential to develop models that are easily understood by the managing personnel and designers to look at possible scenarios and pursue any "what-if" considerations.

The geometry of the feature space imposed by the classifier is also inherently associated with our ability to interpret classification processes and comprehend the decision boundaries produced by the classifier. In general, these are nonlinear. In an ideal situation they may coincide with those produced by a Bayesian classifier [4]. From the interpretation point of view, the most intuitive ones are those built in the form of boxes (in two-dimensional space) or hyperboxes (in multi-dimensional space), refer to Figure 1.

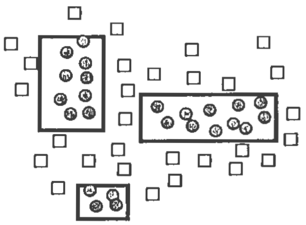


Figure 1. Examples of hyperbox-based classifier formed in a two-dimensional space; note a number of hyperboxes (boxes) formed there and "covering" patterns (data) belonging to class $\omega_1$ (dots). The second class is denoted by small squares and its elements are excluded from the hyperbox

Subsequently, when using boxes (and hyperboxes in general), any classification rule becomes straightforward and emerges as a result of enlisting of the edges of the boxes:

> *assign (classify) pattern* $\boldsymbol{x} \in \boldsymbol{R}^n$ *to class* $\omega_1$ *if* $\boldsymbol{x}$ *belongs to one of the hyperboxes* $\boldsymbol{H}_1, \boldsymbol{H}_2, \ldots, \boldsymbol{H}_c$ *describing (localizing) patterns belonging to the class under interest.*

The hyperbox classifier exhibits two interesting properties. First, the concept of the classifier is profoundly simple. Potentially we can improve the classification rate by moving to the higher level of detail and increasing the number of the boxes while making them smaller and in this way refining the classifier. Second, the classifier directly translates into a set if transparent rules (since each box is a rule itself) whose condition parts assume a straightforward interpretation. Note that a hyperbox is just a Cartesian product of the intervals forming its edges. The rules read as "if **x** is in $\mathbf{H}_i$ then $\omega_1$", i=1, 2,…,c. Alternatively, we can allude to the edges of the hyperbox and spell out a collection of the conditions, that is " if $x_1$ is in $[a_1, b_1]$ and $x_2$ is in $[a_2, b_2]$ and … and $[a_n, b_n]$ then ωi". Owing to their interpretability, hyperbox classifiers have been studied in the literature quite intensively, cf. [6][18][19][16]. The most popular approach to the design of these classifiers is perhaps the one proposed by Simpson [18][19] where he discusses both supervised and unsupervised mode of learning. What somewhat hampers the development of the hyperbox classifiers is a lack of learning algorithms which is not so surprising considering the geometry of the classifiers (that do not come with differentiable boundaries of the hyperboxes and in this way are not suitable for gradient-based optimization techniques).

The objective of this study is to develop a hybrid, two-phase design of hyperbox classifiers and discuss their essential role in analysis and classification of software data. In the first phase of the design, we "seed" the hyperboxes by using fuzzy clustering in which designed are the prototypes (centers) of the clusters. In our case there play a role of seeds around which the we start ":growing' the boxes by expanding the size of the box and pushing its walls further from the center. This process is followed by the second phase in which we "grow " the hyperboxes via genetic optimization. This hybrid approach helps us capture the nature of the software data by identifying the clusters first and then form its geometry with the aid of genetic optimization.

The material is arranged into 4 sections. Following the introductory comments, Section 2 provides with a detailed design process of the hyperbox classifier. This is followed by intensive experimental studies involving MIS software data [15] presented in Section 3. Conclusions are covered in Section 4.

In this study, we adhere to the basic notation being commonly encountered in pattern recognition. To focus attention on the essence of the problem, we consider a two-class problem involving two categories of patterns (denoted here by $\omega_0$ and $\omega_1$, respectively). Patterns are distributed in an n-dimensional space of reals that is $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N \in \mathbf{R}^n$. As already stated, the hyperbox classifier is based on a collection of hyperboxes represented here as

$$\mathcal{H} = \{\mathbf{H}_1, \mathbf{H}_2, \ldots \mathbf{H}_c\}$$

The ith hyperbox, $\mathbf{H}_i$, is fully described by two vectors of its bounds, namely $\mathbf{H}_i = (\mathbf{a}_i, \mathbf{b}_i)$ where $a_{ij} \leq b_{ij}$ denote the lower and upper bounds of the jth feature for the ith hyperbox.

Prior to moving into the development of the classifier, it is advantageous to elaborate here in more detail on the software data being used in this case study. The Medical Imaging System (MIS) [15] is a commercial software system consisting of approximately 4500 routines written in about 400,000 lines of Pascal, FORTRAN and PL/M assembly code. The MIS development took five years, and the system has been in commercial use at several hundred sites for quite a few years. The MIS data were collected as the number of changes made to each module due to faults discovered during system testing and maintenance over an observation period of three years. Along with the above parameter, eleven software complexity measures (metrics) were provided, refer to Table 1. In this study, MIS is represented by a subset of the whole MIS data with 390 modules written in Pascal and FORTRAN. These modules consist of approximately 40,000 lines of code. Our goal is to develop a prediction model of software quality in which the number of modifications (changes) is projected on a basis of the values of the 11 software metrics that is used to characterize a software module. We cast the problem in the setting of classification and because of this assumed format, the output variable (number of changes) is discretized (categorized) by distinguishing between several limited, but intuitively appealing, categories (say, class_1: software modules with no changes; these could be eventually deemed to be fault-free, class_2: software modules with number of modifications between 1 and 10, and software modules with the number of changes over 10; this category can be sought as potentially highly faulty modules where most of our testing and maintenance effort should be focused).

| *Software Measure* | *Detailed Description* |
|---|---|
| Changes | Number of changes |
| LOC | Number of lines of code including comments, declarations and the main body of the code |

| | |
|---|---|
| CL | Number of lines of code, excluding comments |
| TChar | Number of characters |
| TComm | Number of comments |
| MChar | Number of comment characters |
| DChar | Number of code characters |
| N | Halstead's Program Length $N = N_1+N_2$, $N_1$ is the total number of operators, $N_2$ is the total number of operands |
| N' | Halstead's Estimate of Program Length $N' = n_1 \log_2 n_2 + n_2 \log_2 n_2$, $n_1$ is the number of unique operators, $n_2$ is the number of unique operands |
| NF | Jensen's Estimate of Program Length Metric $\log_2 n_1! + \log_2 n_2!$ |
| V(G) | McCabe's Cyclomatic Complexity Metric, where $V(G)=e-n+2$, and e represents the number of edges in a control graph of n nodes |
| BW | Belady's bandwidth measure $BW=1/n\ \Sigma_i iL_i$, $L_i$ represents the number of nodes at level "i" in a nested control flow graph of n nodes. This measure indicates the average level of nesting or width of the control flow graph representation of the program |

Table 1. Description of the MIS dataset with a detailed characterization of the software measures; for details refer to [18]

## 2. The Design of the Classifier

In this section, we elaborate on the development of the classifier. As already stated in the previous section, the hyperboxes being a crux of the classifier are formed in two phases. First, Fuzzy C-Means (FCM) [2] traverses trough the multidimensional data space, finds clusters and describe them via their prototypes $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_c$. These prototypes form "seeds" of the hyperboxes to be developed in the second design phase. In essence, this second step is about "growing" the hyperboxes around the prototypes. As each hyperbox is completely described by its lower and upper bounds (yielding corresponding hyperplanes) that are positioned relative to the prototype, we are concerned with the determination of the values of such parameters. In general, gradient-based methods (especially in case of highly dimensional feature space and a significant number of the prototypes) may not allow us to explore intensively the resulting search space and arrive at a global minimum. Therefore our intent is to apply techniques of genetic optimization, and genetic algorithms [1][5][8][9][10][12][13] in particular. In what follows, we move to the pertinent algorithmic details and afterwards discuss an overall flow of the algorithmic pursuits along with the corresponding implementation details.

### 2.1 The Fuzzy C-Means (FCM) Algorithm

The aim of the FCM approach (which is regarded as one among standard and commonly exploited clustering techniques) is aimed at revealing a structure in multidimensional data sets. This is accomplished by minimizing the following objective function

$$Q = \sum_{i=1}^{c} \sum_{k=1}^{N} (u_{ik})^m d_{ik}^2 \tag{1}$$

The minimization of Q is realized with respect to the pivotal descriptors of the structure that is a partition matrix $\mathbf{U} = [u_{ik}]$, $i = 1, 2, \ldots, c$, $k = 1, 2, \ldots, N$ and prototypes (centers) of the clusters $\mathbf{v}_1$, $\mathbf{v}_2 \ldots \mathbf{v}_c$. (1) is a sum of distances from the prototypes where these distances are weighted by the values of the partition matrix (membership grades of patterns to the clusters). $d_{ik}$ denotes the distance between pattern (datum) $\mathbf{x}_k$ and prototype $\mathbf{v}_i$: $d_{ik} = \|\mathbf{x}_k - \mathbf{v}_i\|$. The fuzzification factor m (>1) controls a shape of the fuzzy sets and, in the sequel, a level of overlap occurring between the clusters. Usually we set up 'm' to 2. The minimization of (1) is carried out under a set of constraints expressed with respect to **U**, namely

$$0 < \sum_{k=1}^{N} u_{ik} < N \qquad i=1,2\ldots c \tag{2}$$

$$\sum_{i=1}^{c} u_{ik} = 1 \qquad k=1,2\ldots n \tag{3}$$

By introducing constraint (2) we make sure that the clusters are neither empty nor consist of the entire space. Constraint (3) states that the total membership of each pattern over all clusters is equal to 1.

The constrained optimization is realized through an iterative process by updating prototypes and partition matrix **U** governed by the following expressions

$$\mathbf{v}_i = \frac{\sum_{k=1}^{N} (u_{ik})^m \mathbf{x}_k}{\sum_{k=1}^{N} (u_{ik})^m} \tag{4}$$

$$u_{ik} = \frac{1}{\sum_{j=1}^{c} \left( \frac{d(\mathbf{v}_i, \mathbf{x}_k)}{d(\mathbf{v}_j, \mathbf{x}_k)} \right)^{\frac{2}{m-1}}} \tag{5}$$

The FCM phase is introduced to prevent the GA from exploring empty regions of the space or areas where only patterns belonging to class $\omega_0$ are found. This is a waste of computational power if it can be avoided, as we know that including such areas would not improve the performance of the classifier. We therefore looked for a way to narrow down the search space and make the search more focused. FCM directs the GA to explore only areas that are known to have concentration of class $\omega_1$ vectors. By its nature, FCM is known to locate (position) prototypes in regions where we encounter the highest density of data, which is an essence of clustering. In light of the two-class pattern classification we consider here, we cluster only patterns belonging to class $\omega_1$, leaving out patterns of class $\omega_0$. We can therefore assume that wherever the prototypes are found, should be contained inside the hyperboxes and grow the hyperboxes around these seeds.

Let us stress that the FCM algorithm does come with certain pitfalls. These are primarily related to the number of the clusters (prototypes). This parameter is usually fixed in advance and may not coincide with the "real" structure existing in the data set. If this is the case, we should become aware of the consequences of such mismatch. If there are fewer prototypes than clusters, some or all of the prototypes may be placed by the algorithm in regions between the clusters. There is no information about this region, and we do not know whether this region is empty or filled with class $\omega_0$ vectors, which are omitted from the FCM algorithm. Another problem arises in the opposite case as well. When there are more prototypes than clusters, the arrangement of the prototypes is highly dependent on their original placement, which, as mentioned, is random. For example, given 4 clusters, and 8 prototypes, there is no guarantee that the final arrangement would include 2 prototypes per cluster. Therefore, a cluster that would need more hyperboxes to be defined accurately may only be able to utilize fewer hyperboxes than another cluster, which may not require as many.

## 2.2 Genetic Algorithms as a Vehicle of Evolutionary Optimization

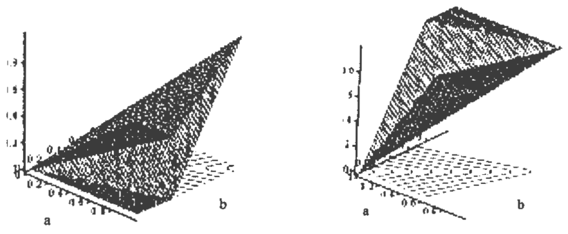
The genetic optimization phase of the design uses a "standard" real coded GA (RCGA) [5][9][10] where chromosomes are represented using real valued vectors. This representation poses some advantages over the traditional binary coded GA (BCGA) where chromosomes are represented using binary strings. RCGA offers higher precision while avoiding coding problems, such as the "Hamming cliff" phenomenon [8], and the representation of the problem to GA space is fairly straightforward. The fundamental architectural and functional considerations involve problem representation, genetic operations, and a fitness function.

1. *Problem Representation*: As noted, the number of hyperboxes is fixed in advance as being in one-to-one correspondence with the prototypes determined by the FCM algorithm. We organize "n" variables (edges) of the "c" hyperboxes as successive entries of the chromosome of length 2*n*c. All the entries in the chromosome are confined to the unit interval; this leads to a chromosome whose content is more homogeneous and therefore easily interpretable. When returning the content of the chromosome and produce a result in a phenotype space, we convert the entry of the chromosome by scaling it. For instance, we transform $a'_{14}$ to $a_{14}$ by multiply it by the range it could have assumed that is D; we thus have $a_{14}=D^* a'_{14}$. If the prototype for box 1 is found to be at (40, 90), the domain for the feature $x_1$ is [60, 100] (determined by finding the range of this feature), and $a'_{14}$ assumes the value equal to 0.6, then the value of $a_{14}$ computes as (100-90)*0.6=6. Likewise the upper bound for box 1 in the feature space (phenotype) $x_1$ is given as 90+6=96.

2. *Selection:* we use an elitist ranking selection [1]. This selection mechanism means that individuals to be selected for the next generation are based on their relative rank in the population, as determined by the fitness function. The best individual from each generation is always carried over to the next generation, so the best solution found so far during the genetic optimization is guaranteed to never disappear.

3. *Mutation:* we use the random mutation operator [9]. Given an individual $\mathbf{a} = [a_1,a_2,\ldots,a_{2n}]$ we generate $\mathbf{a}' = [a_1',a_2', \ldots,a_{2n}']$ where $a_i'$, i=1, 2,...,2n, is a random number confined in the range of [0,1] and subject to the following rule: $a_i$ is mutated that is replaced by $a_i'$ with some probability of mutation ($P_m$) otherwise the entry of the chromosome is left intact, $a_i'=a_i$.

4. *Crossover:* the crossover operation is realized as the BLX-0.5 crossover operator [11], which is carried out as follows. Given two individuals **a** = [$a_1$, $a_2$, ..., $a_{2n}$] and **b** = [$b_1$, $b_2$, ..., $b_{2n}$], their resulting offspring are formed as **a'** = [$a_1$', $a_2$', ..., $a_{2n}$'] and **b'** = [$b_1$', $b_2$', ... , $b_{2n}$'], where $a_i$', $b_i$', i=1,2,...,2n, are random numbers in the range [max(0, $\min_i$-0.5I), min(1, $\max_i$+0.5I)]. In the above calculations, $\min_i$ = min ($a_i$, $b_i$), $\max_i$ = max ($a_i$, $b_i$), and I = $\max_i$ - $\min_i$, see Figure 2 for a detailed illustration of the realization of this operation. This particular crossover operation provides a good balance between using the information of the parents and avoiding premature convergence, cf. 0. Furthermore the crossover operator defined in this manner ensures that all values of the generated offspring are confined to the unit interval [0, 1]. The operator is employed with probability of crossover, $P_c$, otherwise the individuals are left unchanged **a'**=**a** and **b'**=**b**.



(a) (b)

Figure 2. A realization of the mutation operator used in the genetic optimization: lower (a) and upper (b) bounds of the range

5. *Fitness Function:* As each individual is an instance of a hyperbox classifier, the fitness function is the same is the primary evaluation criteria of the classifier, which is its accuracy [4]. The classifier's accuracy is defined as (TP+TN)/N whose definition is self-evident once related to the confusion matrix shown in Table 1, where N denotes the total number of patterns. The confusion matrix summarizes the number of correctly predicted negatives, or True Negative (TN), incorrectly predicted positives, or False Positive (FP), etc. Once we reach a fitness value equal to 1, the GA process is stopped. In our study, the genetic optimization is guided by the accuracy of the developed classifier

Table 1 **Confusion matrix for a two-class problem**

| | Actual $\omega_0$ | Actual $\omega_1$ |
|---|---|---|
| Predicted $\omega_0$ | TN | FN |
| Predicted $\omega_1$ | FP | TP |

## 3. Experiments with Software Data

This section reports on the hyperbox classifiers applied to the MIS data. Prior to these experiments, we present several illustrative two-dimensional examples whose selection helps quantify and visualize the resulting two-dimensional constructs (boxes) of the classifier.

Example 0. We are concerned with the two-class problems where one class forms an oval shape being surrounded by the second class, Figure 3. When designing the classifier, we require a number of boxes whose union attempts to "cover" data belonging to a certain class while eliminating patterns belonging to second class. As becomes obvious, the resulting union follows the original geometry with the intent of the minimal approximation error (viz. the minimal classification error). A single box provides some approximation and we see that it attempts to do the best by minimizing the involvement of the pattern belonging to the second class. Overall, the classification rate is around 97% (and this figure is almost the same for the training and testing set). With the increase of the boxes to two, the approximation improves (as is well visualized) resulting in the classification rate increased to 98%. Further increase to 4 boxes has resulted in the 99% classification rate.

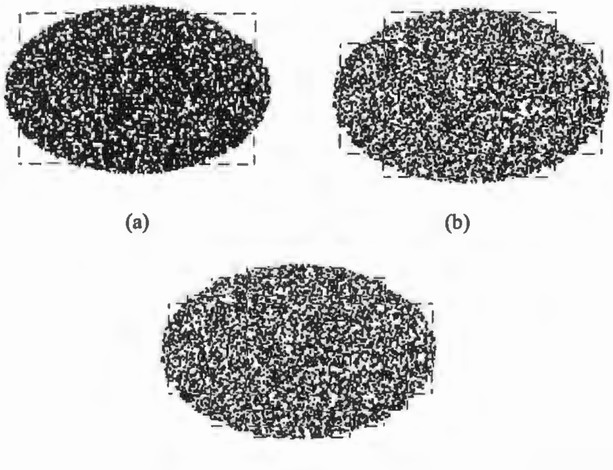


Figure 3. Examples of two-dimensional data belonging to two classes and its box classification involving one (a), two (b), and four (c) boxes

Proceeding with the MIS data, the hyperbox classifier is developed by treating the output (the number of changes) as a binary variable so that we end up with a two-class classification problem. This quantification is very much problem-dependent and varies from the standpoint we assume when analyzing the data. As commonly encountered in the literature and software engineering practice, the number of changes made to some software artifact impacts its quality, projects on the complexity of the undertaking, and reflects upon its future performance.

In the ensuing experiments, we fix the parameters of the environment of the genetic optimization. The size of the population was set to 100 individuals; the GA was run for 500 generations with the probability of crossover and mutation equal to 0.80 and 0.05, respectively.

Experiment 1. Here we describe a category of software modules for which the number of changes varies between 5 and 12. Those are the modules that could be characterized as requiring a *medium* level of maintenance effort. The remaining modules form the alternative class. The genetic optimization was carried out for c = 1, 2,3, and 4 hyperboxes. To assess the performance of the classifier, the dataset was split into the three disjoint subsets of a training, validation, and testing set with each of them playing the role outlined in the previous section. For each case, the experiment was repeated 10 times so that the results become statistically legitimate. The classification rate obtained for the training and testing set are shown in Figure 4 (reported here are the mean values of the classification rate that is the classification accuracy).

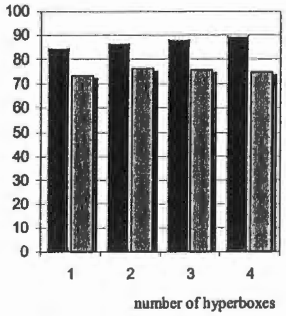


Figure 4. Accuracy of the hyperbox classifier versus number of hyperboxes used in the construct: training set – black bars, testing set – gray bars

From this figure we can conclude that the optimal number of hyperboxes is two; while the training set returns better results for three hyperboxes, we see a slightly better performance on the testing set when using two hyperboxes. As usual, we always prefer to use the most compact structure and thus the choice of the two hyperboxes. Here the best experiment out of the ten runs produced 86.6% and 82.2% for the training and testing set, respectively. The performance of the GA is reported in the form of the fitness function, Figure 5, whose values are visualized for the best and average individual as obtained in the successive generations of the genetic optimization. These are typical plots for the genetic optimization: the best individual (as we used the elitist strategy) is subject to substantial improvements at the early generations and afterwards is left unchanged. The average fitness of the entire population fluctuates from generation to generation and becomes a reflection of the ongoing search completed in the geometry of the hyperboxes.

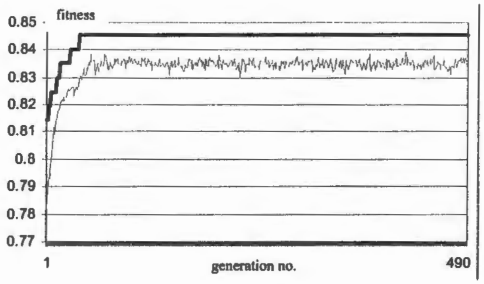
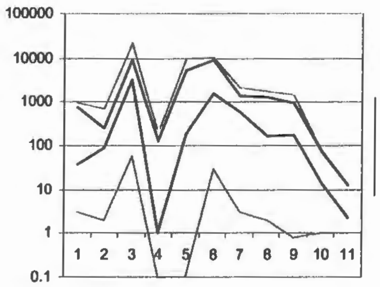
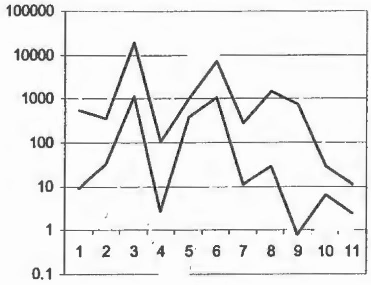


Figure 5. Fitness function in successive generations (average and best individual); the results concerns one of the ten runs of the experiment

The location of the hyperboxes in the input space is visualized in Figure 6; here the endpoints of the corresponding axes (variables) are shown vis-à-vis the ranges of the individual software measures.



(a)



(b)

Figure 6. Location of the hyperboxes in the space of the software measures for the first (a) and second (b) hyperbox

This form of visualization becomes beneficial in providing a certain insight into the specificity of the box with respect to the overall range of the feature. We can make two observations with this respect. First, there are some software measures for which the hyperboxes are quite narrow. This concerns the number of lines of code excluding comments, number of characters, and the number of code characters. These software measures could be then alluded to as the most discriminative features in the classification problem. Secondly, the hyperboxes can overlap (which is not surprising as they are formed in a highly dimensional feature space) with respect to some software measures and become quite distinct as far as some other features are concerned. This occurs for the number of lines of code and the number of comments.

Experiment 2. Here we are interested in the description of software modules requiring low maintenance effort by defining a group of modules with the number of changes less than 7 so we are concerned with the description of *low* maintenance software modules. The results are reported in a similar format as shown in the first experiment. Here we consider 1 and 2 hyperboxes as the two design alternatives of interest. Because of the anticipated character of the class (involving low maintenance modules), we can envision that the hyperbox will be eventually spreading from the low bounds assumed by the software measures towards their higher values. As illustrated in Figure 7, one hyperbox leads to better results than those coming with the use of the two or more hyperboxes.

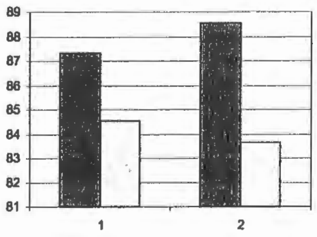


Figure 7. Classification rate for one and two hyperboxes for the training (dark bars) and testing data (gray bars); shown are average values of the rates

The coverage of the hyperbox expressed in terms of the individual software measures helps us assess a discriminative property of the features. To quantify this effect, for each feature we introduce the following ratio

$$\rho = \frac{\text{length of interval (upper - lower bound)}}{\text{lenght of bound of feature}}$$

The lower the value of ρ, the more discriminative the variable. Intuitively, if ρ tends to approach 1, the corresponding software measure is not meaningful (any of its value is relevant to the same class and there is no discrimination abilities present). As visualized in Figure 8, we note that the most discriminative software measures are the number of lines of code, program length, and the number of code characters. The ranking of the features with respect to their discriminative power can be mapped onto the form of the rules where we reflect this in the order of the conditions in the rules. For instance, the rules could read as follows

- if number of lines of code is A and program length is B, and the number of code characters is C…and ….then the software module exhibits *low* level of maintenance

with A, B, and C denoting the intervals of the values of the corresponding features

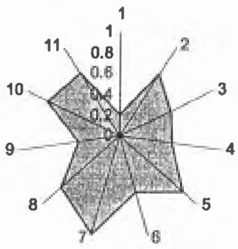


Figure 8. Plot of ratios ρ for software measures in the MIS data

## 4. Conclusions

We have developed a comprehensive design process of hyperbox classifiers. The two-phase development environment appeared to be a viable optimization structure. By initiating the development of the hyperboxes through fuzzy clustering we were able to concentrate a search for the geometry of the data and focus the ensuing genetic optimization on the most promising regions of the feature space. The most visible advantage of the hyperbox classifier lies in its interpretability and this feature is fully exploited in the design of the classifier for the software data. The equivalent representation of the hyperbox classifier comes as a collection of rules where each hyperbox corresponds to a single rule. We experimented with the software MIS dataset and showed how the classifier leads to a collection of rules describing software modules of some assumed quantification of software modules.

While in this study we have confined ourselves to a two-class problem (that is usually treated as a generic classification environment), the approach readily extends to a multiclass problem. Interestingly, the use of the genetic algorithm itself could be helpful in an overall optimization of the resulting ensemble of the classifiers.

**Acknowledgments**

Support from the Canada Research Chair Program (W. Pedrycz), Natural Sciences and Engineering Research Council of Canada (NSERC) and Alberta Software Engineering Research Consortium (ASERC) is gratefully acknowledged.

## 5. References

[1] J. E. Baker, Adaptive selection methods for genetic algorithms, *Proc. of the First International Conference on Genetic Algorithms*, pp. 101-111, 1985.

[2] J. C. Bezdek, *Pattern Recognition with Fuzzy Objective Functions,* Plenum, N.Y. 1981

[3] I. De Falco, A. Della Cioppa, and E. Tarantino, Discovering interesting classification rules with genetic programming, *Applied Soft Computing 1,* pp. 257-269, 2002

[4] R. Duda and P. Hart, *Pattern Classification and Scene Analysis*, Wiley, New York, 1973.

[5] L. J. Eshelman and J. D. Schaffer, *Real-coded Genetic Algorithms and Interval Schemata*, in Foundations of Genetic Algorithms 2, Morgan Kaufman Publishers, San Mateo, CA, pp. 187—202, 1993

[6] B. Gabrys, A. Bargiela, General fuzzy Min-Max neural network for clustering and classification, *IEEE Trans. Neural Networks*, Vol. 11, issue 3, pp 769-783, 2001

[7] D. Garmus, D. Herron, *Measuring The Software Process*, Prentice Hall, Upper Saddle River, NJ, 1996.

[8] D. E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley,1989

[9] D.E. Goldberg, Real-coded genetic algorithms, virtual alphabets, and blocking, *Complex Systems*, (5), pp 139-167, 1991.

[10] R. L. Haupt, S. E. Haupt, *Practical Genetic Algorithms*, J. Wiley & Sons, N. York, 1998.

[11] F. Herrera, M. Lozano, and J.L. Verdegay, Tackling real-coded genetic algorithms: Operators and tools for behavioral analysis, *Artificial Intelligence Review*, vol. 12, pp. 265-319, 1998

[12] J. H. Holland, *Adaptation of Natural and Artificial Systems*, The University of Michigan Press, Ann Arbor, 1975

[13] Z. Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs*. Springer-Verlag, Heidelberg, 3rd edition, 1996

[14] K.H. Muller, D.J. Paulish, *Software Metrics*, IEEE Press/Chapman & Hall, London, 1993.

[15] J. C. Munson, T. M. Khoshgoftaar, Software metrics for reliability assessment, in: *Handbook of Software Reliability and System Reliability*, McGraw-Hill, Hightstown, NJ, 1996.

[16] W. Pedrycz, G. Succi, M.G. Chun, Association analysis of software measures, *Int. J of Software Engineering and Knowledge Engineering*, 12, no.3, 2002, 291-316.

[17] W. Pedrycz, G. Succi, P. Musilek, X. Bai, Using self-organizing maps to analyze object-oriented software measures, *J. of Systems and Software*, 59, 2001, 65-82.

[18] P. K. Simpson, Fuzzy Min-Max Neural Networks – Part 1: Classification, *IEEE Trans. Neural Networks*, vol. 3, pp 776-786, 1992

[19] P. K. Simpson, Fuzzy Min-Max Neural Networks-Part 2: Clustering, *IEEE Trans. Fuzzy Systems*, vol. 1, no. 1, pp. 32-45, 1993

[20] G. Succi, W. Pedrycz, M. Stefanovic, B. Russo, An investigation on the occurrence of service requests in commercial software applications, *Empirical Software Engineering*, 8, 2003, 197-215.