Network Reliability Assessment through Empirical Models using a Machine Learning Approach

Claudio M. Rocco S.

Facultad de Ingeniería, Universidad Central Venezuela, Caracas

Marco Muselli

Istituto di Elettronica e di Ingegneria dell'Informazione e delle Telecomunicazioni, Consiglio Nazionale delle Ricerche, Genova, Italy

6.1 Introduction: Machine Learning (ML) Approach to Reliability Assessment

The reliability assessment of a system requires knowledge of how the system can fail, failure consequences and modeling, as well as selection of the evaluation technique [4].

For a reliability evaluation, almost all the systems are modeled using a Reliability Block Diagram (RBD), that is, a set of components that interact in some way to comply with the system purpose. System components are represented by blocks connected together either in series, in parallel, meshed or through a combination of these. For example, if the system failure occurs when all the components are failed, they are represented in a reliability network as a parallel set.

An RBD can be considered as an undirected or a directed connected graph. For example, in a communication system each node represents a communication center and each edge a transmission link between two such centers. It is assumed that each edge (link) functions independently of all other edges and that the edge operation probability is known.

The literature offers two main categories of techniques to evaluate the reliability of a system: analytical and simulation methods. The first one analyzes the topology of the equivalent graph to obtain a symbolic expres-

M.R.S. Claudio and M. Muselli: *Network Reliability Assessment through Empirical Models using a Machine Learning Approach*, Computational Intelligence in Reliability Engineering (SCI) **40**, 145–174 (2007)
www.springerlink.com © Springer-Verlag Berlin Heidelberg 2007

sion for the occurrence of a failure. On the other hand, the simulation approach allows a practical way of evaluating the reliability of the network in specific situations. In particular, when complex operating conditions are considered or the number of events is relatively large, Monte Carlo (MC) techniques offer a valuable way of evaluating reliability [4]. This approach is widely used when real engineering systems are to be analyzed.

In general, any reliability index can be obtained as the *expected value* of a System Function (*SF*) [14] or of an Evaluation Function (*EF*) [25] applied to a system state $\boldsymbol{x}$ (vector representing the state of each element in the network). This function determines whether a specific configuration corresponds to an operating state or a failed one [24], according to a specific criterion. For example, if connectivity between two particular nodes, s (the source) and t (the terminal) must be ensured, the system is operating if there exists at least a working path from the source node s to the terminal node t.

The corresponding reliability measure (s-t reliability) has been widely studied in the literature; in this case a depth-first procedure [23, 29] can be employed as an *EF*.

In other systems, for example in communication networks, the success criterion assumes that a network performs well if and only if it is possible to transmit successfully a specific required capacity. For these systems, the connectivity is not a sufficient condition for success, as it is also required that an adequate flow is guaranteed between s and t, taking into account the capacity of the links involved. In this case, the max-flow min-cut algorithm [23,29] can be adopted to evaluate if a given state is capable or not of transporting a required flow; alternatively, procedures based on the concept of composite paths [1, 28] can be used as the *EF*.

In general, the reliability assessment of a specific system requires the computation of performance metrics using special *EF*. An important characteristic of these metrics and their extensions is that the solution of an NP-hard problem [35] is needed for their evaluation in almost all the contexts of interest. In this situation MC techniques are used to estimate performance metrics. However, an MC simulation requires a large number of *EF* evaluations to establish any reliability indices with high computational effort. For this reason, it is convenient to approximate the *EF* using a Machine Learning (ML) method.

Two different situations can be identified: ML predictive methods reconstruct the desired SF through a black box device, whose functioning is not directly comprehensible. On the contrary, ML descriptive methods provide a set of intelligible rules describing the behavior of the *SF* for the system at hand. Support Vector Machines (SVM) is a widely used predictive method, successfully adopted in reliability assessment [30], whereas

Decision Trees (DT) [31] and Shadow Clustering (SC) [20] are two descriptive methods that are able to discover relevant properties for reliability analysis.

The chapter is organized as follows: In Sec. 2 some definitions are presented. Section 3 introduces the three machine learning methods considered for approximating the reliability of a network, while Sec. 4 compares the results obtained by each method for a specific network. Finally, Sec. 5 contains the conclusions.

Acronyms:

ARE	Approximate Reliability Expression
DT	Decision Tree
EF	Evaluation Function
HC	Hamming Clustering
ML	Machine Learning
RBD	Reliability Block Diagram
RE	Reliability Expression
SC	Shadow Clustering
SF	Structure Function
SVM	Support Vector Machine

6.2 Definitions

Consider a system *S* composed by several units interconnected by *d* links; the functioning of *S* directly depends on the state x_i, $i = 1, \ldots, d$, of each connection, which is viewed as an independent random variable assuming two possible values 1 and 0, associated with the operating and the failed condition, respectively. In particular, we have [3]:

$$x_i = \begin{cases} 1 \text{ (operating state)} & \text{with probability } P_i \\ 0 \text{ (failed state)} & \text{with probability } Q_i = 1 - P_i \end{cases} \quad (1)$$

where P_i is the probability of success of component (link) *i*.

The state of a system *S* containing *d* components is then expressed by a random vector $\boldsymbol{x} = (x_1, x_2, \ldots, x_d)$, which uniquely identifies the functioning of *S*. Again, it can be operating (coded by the value 1) or failed (coded by 0). To establish if $\boldsymbol{x}$ leads to an operating or a failed state for *S*, we adopt a proper Evaluation Function (*EF*):

$$y = EF(\mathrm{x}) = \begin{cases} 1 & \text{if the system is operating in state x} \\ 0 & \text{if the system is failed in state x} \end{cases} \quad (2)$$

If the criterion to be used for establishing reliability is simple connectivity, a depth-first procedure [23,29] can be employed as an *EF*. In the case of capacity requirements, the *EF* could be given by the max-flow min-cut algorithm [23, 29]. For other metrics, special *EF* may be used. Since $\boldsymbol{x}$ and y include only binary values, the functional dependence between y and $\boldsymbol{x}$ is given by a Boolean function, called Structure Function (*SF*) [10], which can be written as a logical sum-of-products involving the component states x_i or their complements $\bar{x}_i$. Consider the system in Fig. 1 that contains four links.

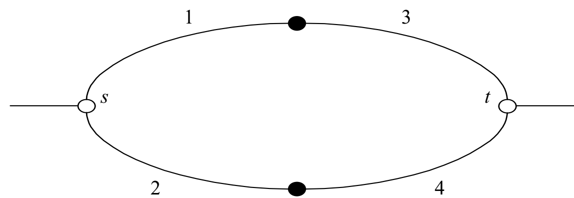

Fig. 1. A 4-components network

If the connectivity between the source node s and the terminal node t must be ensured in an operating state for the network, the following *SF* is readily obtained:

$$y = SF(\boldsymbol{x}) = x_1x_3 + x_2x_4 \tag{3}$$

where the OR operation is denoted by '+' and the AND operation is denoted by '·'. Like for standard product among real numbers, when no confusion arises the AND operator can be omitted.

The *reliability* of a system is defined as the expected value of its structure function [15]. When the $SF(\boldsymbol{x})$ has the form of a logical sum-of-products, a closed-form expression for the system reliability, called *Reliability Expression* (*RE*), can be directly obtained by substituting in the logical sum-of-products, according to (1) and to the independence of the random variables x_i, every term x_i with P_i and every $\bar{x}_i$ with Q_i. After this substitution logical sums and products must be changed into standard sums and products among real numbers. For example, the *RE* deriving from the $SF(\boldsymbol{x})$ in (3) is $P_1P_3+ P_2P_4$, which gives the value of the system reliability when substituting the actual values of P_i into this expression.

Since the Boolean expression for the *SF* of a system can be derived only for very simple situations, it is important to develop methods that are able to produce an estimate of the system reliability by examining a reduced number of different system states $\boldsymbol{x}_j$, $j = 1, \ldots, N$, obtained by as many ap-

plications of the *EF*. A possible approach consists in employing machine learning techniques, which are able to reconstruct an estimate of the system function *SF*($\boldsymbol{x}$) starting from the collection of *N* states $\boldsymbol{x}_j$, called in this case *training set*. Some of these techniques generate the estimate of the *SF* as a logical sum-of-products, which can be used to produce (through the simple procedure described above) an *Approximate Reliability Expression* (*ARE*) that is (hopefully) close to the actual *RE*.

In this case, the estimation of the system reliability can be easily performed, by substituting into the *ARE* the actual values of the P_i. On the other hand, when an approximation to the *SF* is available and cannot be written in the form of a logical sum-of-products, a standard Monte Carlo approach can be adopted to estimate the system reliability. By using the approximate *SF* to establish the state y instead of the *EF*, the computational cost is reduced.

6.3 Machine Learning Predictive Methods

In this section three different machine learning techniques are described: Support Vector Machines (SVM), Decision Trees (DT) and Shadow Clustering (SC). All these methods are able to solve two-class classification problems, where a decision function g: $\Re^d \rightarrow \{0,1\}$ have to be reconstructed starting from a collection of examples $(\boldsymbol{x}_1,y_1)$, $(\boldsymbol{x}_2,y_2)$, …, $(\boldsymbol{x}_N,y_N)$. Every y_j is a (possibly noisy) evaluation of $g(\boldsymbol{x}_j)$ for every $j = 1, \ldots, N$. Thus, they can be employed to reconstruct the *SF* of a system *S* when a set of *N* pairs $(\boldsymbol{x}_j,y_j)$, where $\boldsymbol{x}_j$ is a system state and $y_j = EF(\boldsymbol{x}_j)$, is available.

In this case, DT and SC are able to generate a logical sum-of-products that approximates the *SF*. On the other hand, SVM produces a linear combination of real functions (Gaussians, polynomial, or others) that can be used to establish the state y associated to a given vector $\boldsymbol{x}$. Consequently, SVM cannot be directly used to generate an ARE for the system at hand.

6.3.1 Support Vector Machines

In the last ten years Support Vector Machines (SVM) have become one of the most promising approach for solving classification problems [11,36]. Their application in a variety of fields, ranging from particle identification, face identification and text categorization to engine detection, bioinformatics and data base marketing, has produced interesting and reliable results, outperforming other widely used paradigms, like multilayer perceptrons and radial basis function networks.

For symmetry reasons, SVM generates decision functions $g : \Re^d \to$ 1,+1}; like other classification techniques, such as multilayer perceptrons or radial basis function networks, SVM constructs a real function $f : \Re^d \to \Re$ starting from the collection of N samples $(\boldsymbol{x}_j, y_j)$, then writing $g(\boldsymbol{x}) = \text{sign}(f(\boldsymbol{x}))$, being $\text{sign}(z) = +1$ if $z \geq 0$ and $\text{sign}(z) = -1$ otherwise.

The learning algorithm for SVM stems from specific results obtained in statistical learning theory and is based on the following consideration [36]: every classification problem can always be mapped in a high-dimensional input domain $\Re^D$ with $D \gg d$, where a linear decision function performs very well. Consequently, the solving procedure adopted by SVM amounts to selecting a proper mapping $\Phi : \Re^d \to \Re^D$ and a linear function $f(\boldsymbol{x}) = w_0 + \boldsymbol{w} \cdot \Phi(\boldsymbol{x})$, such that the classifier $g(\boldsymbol{x}) = \text{sign}(w_0 + \boldsymbol{w} \cdot \Phi(\boldsymbol{x}))$ gives the correct output y when a new pattern $\boldsymbol{x}$ not included in the training set has to be classified.

Some theoretical results ensure that, once chosen the mapping Φ, the optimal linear function $f(\boldsymbol{x})$ is obtained by solving the following quadratic programming problem:

$$\begin{aligned} \min_{w_0, \mathrm{w}, \xi} \quad & \frac{1}{2} \boldsymbol{w} \cdot \boldsymbol{w} + C \sum_{j=1}^{N} \xi_j \\ \text{subject to} \quad & y_j (w_0 + \boldsymbol{w} \cdot \Phi(\boldsymbol{x}_j)) \geq 1 - \xi_j \\ & \xi_j \geq 0 \quad \text{for every } j = 1, \ldots, N \end{aligned} \tag{4}$$

where the variables ξ_j take account of possible misclassifications of the patterns in the training set. The term C is a regularization constant that controls the trade-off between the training error $\sum_j \xi_j$ and the regularization factor $\boldsymbol{w} \cdot \boldsymbol{w}$.

The parameters $(w_0, \boldsymbol{w})$ for the linear decision function $f(\boldsymbol{x})$ can also be retrieved by solving the Lagrange dual problem:

$$\begin{aligned} \min_{\alpha} \quad & \frac{1}{2} \sum_{j=1}^{N} \sum_{k=1}^{N} \alpha_j \alpha_k y_j y_k \Phi(\boldsymbol{x}_j) \cdot \Phi(\boldsymbol{x}_k) - \sum_{j=1}^{N} \alpha_j \\ \text{subject to} \quad & \sum_{j=1}^{N} \alpha_j y_j = 0 \\ & 0 \leq \alpha_j \leq C \quad \text{for every } j = 1, \ldots, N \end{aligned} \tag{5}$$

which is again a quadratic programming problem where the unknowns α_j are the Lagrange multipliers for the original (primal) problem. The directional vector $\boldsymbol{w}$ can then be obtained by the solution $\boldsymbol{\alpha}$ through the equation

$$\boldsymbol{w} = \sum_{j=1}^{N} \alpha_j y_j \Phi(\boldsymbol{x}_j) \tag{6}$$

It can be easily seen that there is a 1-1 correspondence between the scalars α_j and the examples $(\boldsymbol{x}_j, y_j)$ in the training set.

Now, the Karush-Kuhn-Tucker (KKT) conditions for optimization problems with inequality constraints [34], assert that in the minimum point of the problem at hand it must be

$$\alpha_j (y_j (w_0 + \boldsymbol{w} \cdot \Phi(\boldsymbol{x}_j)) - 1 + \xi_j) = 0 \tag{7}$$

Thus, for every $j = 1, \ldots, N$ either the Lagrange multiplier α_j is null or the constraint $y_j (w_0 + \boldsymbol{w} \cdot \Phi(\boldsymbol{x}_j)) \geq 1 - \xi_j$ is satisfied with equality. It should be noted that only the points $\boldsymbol{x}_j$ with $\alpha_j > 0$ gives a contribution to the sum in (6); these points are called *support vectors*. If $\boldsymbol{x}_{j^-}$ and $\boldsymbol{x}_{j^+}$ are two support vectors with output +1 and –1, respectively, the bias w_0 for the function $f(\boldsymbol{x})$ is given by

$$w_0 = \frac{\boldsymbol{w} \cdot \Phi(\boldsymbol{x}_{j^+}) + \boldsymbol{w} \cdot \Phi(\boldsymbol{x}_{j^-})}{2} = \frac{\sum_{j=1}^{N} \alpha_j y_j \Phi(\boldsymbol{x}_j) \cdot \Phi(\boldsymbol{x}_{j^+}) + \sum_{j=1}^{N} \alpha_j y_j \Phi(\boldsymbol{x}_j) \cdot \Phi(\boldsymbol{x}_{j^-})}{2}$$

The real function $f(\boldsymbol{x})$ is then completely determined if the mapping $\Phi : \Re^d \to \Re^D$ is properly defined according to the peculiarities of the classification problem at hand. However, if the dimension D of the projected space is very high, solving the quadratic programming problem (4) or (5) requires a prohibitive computational cost.

A possible way of getting around this problem derives form the observation that both in the optimization problem (5) and in the expression for w_0 always appears the inner product between two instances of the function Φ. Therefore, it is sufficient to define a *kernel* function $K: \Re^d \times \Re^d \to \Re^+$ that implements the inner product, i.e. $K(\boldsymbol{u},\boldsymbol{v}) = \Phi(\boldsymbol{u}) \cdot \Phi(\boldsymbol{v})$. This allows control of the computational cost of the solving procedure, since in this way the dimension D of the projected space is not explicitly considered. As the kernel function K gives the result of an inner product, it must be always non negative and symmetric. In addition, specific technical constraints, described by the Mercer's theorem [38], have to be satisfied to guarantee consistency.

Three typical choices for $K(\boldsymbol{u},\boldsymbol{v})$ are [36]:

the linear kernel $K(\boldsymbol{u},\boldsymbol{v}) = \boldsymbol{u} \cdot \boldsymbol{v}$

the Gaussian radial basis kernel (GRBF) $K(\mathrm{u},\mathrm{v}) = \exp\left(-\|\mathrm{u}-\mathrm{v}\|^2 / 2\sigma^2\right)$

the polynomial kernel $K(\boldsymbol{u},\boldsymbol{v}) = (\boldsymbol{u}\cdot\boldsymbol{v} + 1)^p$

where the parameters σ and p are to be chosen properly.

By substituting in (5) the kernel function K we obtain the following quadratic programming problem:

$$\min_{\boldsymbol{\alpha}} \frac{1}{2}\sum_{j=1}^{N}\sum_{k=1}^{N}\alpha_j\alpha_k y_j y_k K(\boldsymbol{x}_j, \boldsymbol{x}_k) - \sum_{j=1}^{N}\alpha_j$$

$$\text{subject to } \sum_{j=1}^{N}\alpha_j y_j = 0$$

$$0 \le \alpha_j \le C \qquad \text{for every } j = 1,\ldots, N$$

which leads to the decision function $g(\boldsymbol{x}) = \mathrm{sign}(f(\boldsymbol{x}))$, being

$$f(\boldsymbol{x}) = w_0 + \sum_{j=1}^{N}\alpha_j y_j K(\boldsymbol{x}_j, \boldsymbol{x}) \tag{8}$$

Again, only support vectors with $\alpha_j > 0$ gives a contribution to the summation above; for this reason classifiers adopting (8) are called *Support Vector Machines* (*SVM*). The average number of support vectors for a given classification problem is strictly related to the generalization ability of the corresponding SVM: the lower is the average number of support vectors, the higher is the accuracy of the classifier $g(\boldsymbol{x})$.

The bias w_0 in (8) is given by:

$$w_0 = \frac{1}{2}\left(\sum_{j=1}^{N}\alpha_j y_j K(\boldsymbol{x}_j, \boldsymbol{x}_{j^+}) + \sum_{j=1}^{N}\alpha_j y_j K(\boldsymbol{x}_j, \boldsymbol{x}_{j^-})\right)$$

where again $\boldsymbol{x}_{j^-}$ and $\boldsymbol{x}_{j^+}$ are two support vectors with output (+1) and (–1), respectively.

The application of SVM to the problem of estimating the system function $SF(\boldsymbol{x})$ starting from a subset of possible system states can be directly performed by employing the above general procedure to obtain a good approximation for the SF. Since SF is a Boolean function we must substitute the output value $y = -1$ in place of $y = 0$ to use the standard training procedure for SVM.

The choice of the kernel is a limitation of the SVM approach. Some work has been done on selecting kernels using prior knowledge [7]. In any case, the SVM with lower complexity should be preferred. Our experience

in the reliability field has confirmed the good quality of the GRBF kernel having parameter $(1/2\sigma^2) = 1/d$, as suggested in [9].

For example, consider the system shown in Fig. 1, whose component and system states are listed in Tab. 1, if a continuity criterion is adopted. As it is usually the case in a practical application, suppose that only a subset of the whole collection of possible states (shown in Tab. 2) is available.

Table 1. Component and system states for the network shown in Fig. 1

x_1	x_2	x_3	x_4	$y = F(\boldsymbol{x})$
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	1
0	1	1	0	0
0	1	1	1	1
1	0	0	0	0
1	0	0	1	0
1	0	1	0	1
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

Table 3 shows the support vectors obtained using the linear kernel and the LIBSVM software [9]. Note that in this case only 6 support vectors are derived. These support vectors are able to completely separate the training set. However, when the model is applied to the test set (states from Tab. 1 not included in Tab. 2), only 6 out of 8 states are correctly classified, as shown in Table 4.

From Tab. 3 it is clear that support vectors can not be easily interpreted, since the expression generated does not correspond to a logical sum-of-products.

Table 2. Available subset of states for the network shown in Fig. 1 (training set)

x_1	x_2	x_3	x_4	$y = EF(\boldsymbol{x})$
0	0	0	1	0
0	0	1	1	0
0	1	1	1	1
1	0	0	1	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	1	1

Table 3. Support vectors for the available subset of states shown in Tab. 2

x_1	x_2	x_3	x_4	$y = EF(\boldsymbol{x})$
0	0	0	1	0
0	0	1	1	0
0	1	1	1	1
1	0	0	1	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	1	1

Table 4. SVM estimation for the test set

x_1	x_2	x_3	x_4	$y = EF(\boldsymbol{x})$	SVM estimate
0	0	0	0	0	0
0	0	1	0	0	0
0	1	0	0	0	0
0	1	0	1	**1**	**0**
0	1	1	0	0	0
1	0	0	0	0	0
1	0	1	0	**1**	**0**
1	1	1	0	1	1

6.3.2 Decision Trees

Decision tree based methods represent a non-parametric approach that turns out to be useful in the analysis of large data sets for which complex data structures may be present [2,5,27]. A DT solves a complex problem by dividing it into simpler sub-problems. The same strategy is recursively applied to each of these sub-problems.

A DT is composed of nodes, branches and terminal nodes (leaves). For our network problem, every node is associated with a component of the network. From each node start two branches, corresponding to the operat-

ing or failed state of that component. Finally, every terminal node represents the network state: operating or failed. Many authors use the convention to draw the false branch on the left side of the node and the true branch on the right.

Consider for example the DT shown in Fig. 2 and suppose a new system state is presented for classification. At the root node the state of the component x_2 is checked: if it is failed, the left branch is chosen and a new test on component x_1 is performed. Again, if x_1 is failed, the left branch is chosen and $y = 0$ is concluded.

Even if it may seem reasonable to search for the smallest tree (in terms of numbers of nodes) that perfectly classifies training data, there are two problems:

1) its generation requires the solution of an NP-hard problem and
2) it is not guaranteed that this tree performs well on a new test sample.

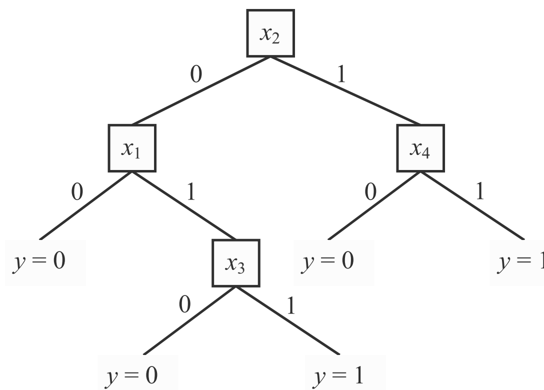

Fig. 2. Example of a decision tree

For this reason DT methods usually exploit heuristics that locally perform a one-step look-ahead search, that is, once a decision is taken it is never reconsidered. However, this heuristic search (hill-climbing without backtracking) may be stuck in a local optimal solution. On the other hand, this strategy allows building decision trees in a computation time that increases linearly with the number of examples [26].

A DT can be used to derive a collection of intelligible rules in the form **if-then**. It is sufficient to follow the different paths that connect the root to the leaves: every node encountered is converted into a condition to be added to the **if** part of the rule. The **then** part corresponds to the final leaf: its output value is selected when all the conditions in the **if** part are satisfied.

Since the tree is a directed acyclic graph, the number of rules that can be extracted from a DT is equal to the number of terminal nodes. As an example, the tree in Fig. 2 leads to three rules for the output $y = 0$ and two rules for $y = 1$.

All the conditions in the **if** part of a rule are connected by a logical AND operation; different rules are considered as forming an **if-then-else** structure. For example, the problem in Fig. 1 is described by the following set of rules:

if $x_1 = 0$ AND $x_2 = 0$ **then** $y = 0$
else if $x_1 = 0$ AND $x_4 = 0$ **then** $y = 0$
else if $x_2 = 0$ AND $x_3 = 0$ **then** $y = 0$
else if $x_3 = 0$ AND $x_4 = 0$ **then** $y = 0$
else $y = 1$

which is equivalent to the *SF* for this network $SF(\boldsymbol{x}) = x_1x_3+x_2x_4$.

Since all the possible output values are considered for rule generation, a complex decision tree can yield a very large set of rules, which is difficult to be understood. To recover this problem, proper optimization procedures have been proposed in the literature, which aim at simplifying the final set of rules. Several different tests have shown that in many situations the resulting set of rules is more accurate than the corresponding decision tree [26].

6.3.2.1 Building the Tree

In general, different algorithms use a *top-down induction* approach for constructing decision trees [22]:

1. If all the examples in the training set *T* belong to one class, then halt.
2. Consider all the possible tests that divide *T* into two or more subsets. Employ a proper measure to score how well each test splits up the examples in *T*.
3. Select the test that achieves the highest score.
4. Divide *T* into subsets according to the selected test. Run this procedure recursively by considering each subset as the training set *T*.

For the problem at hand, a test on a component state with two possible values will produce at most two child nodes, each of which corresponds to a different value. The algorithm considers all the possible tests and chooses the one that optimizes a pre-defined goodness measure.

Since small trees lead to simpler set of rules and to an increase in performance, the procedure above is performed by searching for tests that best separates the training set *T*. To achieve this goal, the most predictive components are considered at Step 3 [5,27].

6.3.2.2 Splitting Rules

Several methods have been described in the literature to measure how effective is a split, that is how good is a component attribute (operating or failed) to discriminate the system state. The most used are:

1. Measures depending on the difference between the training set T and the subsets obtained after the splitting; a function of the class proportion, e.g. the entropy, is typically employed.
2. Measures related to the difference between the subsets generated by the splitting; a distance or an angle that takes into account the class proportions is normally used.
3. Statistical measures of independence (typically a χ^2) between the subsets after the splitting and the class proportions.

In this paper the method used by C4.5 [27] is considered; it adopts the information gain as a measure of the difference between the training set T and the subsets generated by the splitting. Let p be the number of operating states and n the number of failed states included in the training set T. The entropy $E(p,n)$ of T is defined as:

$$E(p,n) = -\frac{p}{p+n}\log(\frac{p}{p+n}) - \frac{n}{p+n}\log(\frac{n}{p+n}) \tag{9}$$

Suppose the component x_j is selected for adding a new node to the DT under construction; if the test on attribute x_j leads to a splitting of T in k disjoint subsets, the average entropy $E_j(p,n)$ after the splitting is given by

$$E_j(p,n) = \sum_{i=1}^{k} \frac{p_i + n_i}{p+n} E(p_i,n_i) \tag{10}$$

where p_i and n_i are the number of instances from each class in the ith subset. Note that in our case $k = 2$ since every component has only two possible states (operating or failed).

The information gain $I_j(p,n)$ is then given by the difference between the values of the entropy before and after the splitting produced by the attribute x_j:

$$I_j(p,n) = E(p,n) - E_j(p,n) \tag{11}$$

At Step 3 of the DT procedure the component x_j that scores the maximum information gain is selected; a test on that component will divide the training set into $k = 2$ subsets.

For example, consider the system shown in Fig. 1 and the training set shown in Tab. 2. There are $p = 4$ operating states and $n = 4$ failed states; thus, the entropy $E(4,4)$ assumes the value:

$$E(4,4) = -4/(4+4)\log[4/(4+4)] - 4/(4+4)\log[4/(4+4)] = 1$$

Looking at the class proportion after the splitting we note that for $x_1 = 0$ there are one system operating state and two failed states, whereas for $x_1 = 1$ there are three system operating states and two failed states. Thus:

$E(1,2) = 0.918296$, $E(3,2) = 0.970951$

$E_1(4,4) = 3/8 \cdot E(1,2) + 5/8 \cdot E(3,2) = 0.951205$

$I_1(4,4) = E(4,4) - E_1(4,4) = 1 - 0.951205 = 0.048795$

Now, for $x_2 = 0$ and $x_3 = 0$ there are one system operating state and three failed states, whereas for $x_2 = 1$ and $x_3 = 1$ there are three system operating states and one failed state. Consequently, we have:

$E(1,3) = E(3,1) = 0.811278$

$E_2(4,4) = E_3(4,4) = 4/8 \cdot E(1,3) + 4/8 \cdot E(3,1) = 0.811278$

$I_2(4,4) = I_3(4,4) = 1 - 0.811278 = 0.188722$

Finally, the fourth component x_4 presents only one system failed state for $x_4 = 0$, whereas for $x_4 = 1$ we detect four system operating states and three failed states. Consequently, we obtain:

$E(0,1) = 0$, $E(4,3) = 0.985228$

$E_4(4,4) = 1/8 \cdot E(0,1) + 7/8 \cdot E(4,3) = 0.\ 862075$

$I_4(4,4) = 1 - 0.\ 862075 = 0.137925$

Since both x_2 and x_3 score the maximum information gain, one of them must be considered for the first node of the DT. Suppose that the second component x_2 is selected as the root node. It is important to note that, in general, the component chosen for the first node holds a primary importance [2].

After this choice the training set in Tab. 2 is split into the following two subsets:

x_1	x_2	x_3	x_4	$y = EF(\boldsymbol{x})$
0	0	0	1	0
0	0	1	1	0
1	0	0	1	0
1	0	1	1	1

x_1	x_2	x_3	x_4	$y = EF(\boldsymbol{x})$
0	1	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	1	1

The former contains the examples with $x_2 = 0$ and the latter those with $x_2 = 1$. If we repeat the procedure for the addition of a new node by considering the first subset as T, we obtain:

$I_1(1,3) = I_3(3,1) = 0.311278$, $I_4(3,1) = 0$

Consequently, the highest information gain is achieved with the choice of x_1 or x_3; the same procedure allows to select the component x_4 for the second subset, which yields, after a further splitting, the DT in Fig. 2. A direct inspection of the DT allows generating the following set of rules:

if $x_2 = 1$ AND $x_4 = 0$ **then** $y = 0$
else if $x_1 = 0$ AND $x_2 = 0$ **then** $y = 0$
else if $x_1 = 1$ AND $x_2 = 0$ AND $x_3 = 0$ **then** $y = 0$
else $y = 1$

Although this set of rules correctly classifies all the system configurations in Tab. 2, it is not equivalent to the desired system function $SF(\boldsymbol{x}) = x_1x_3+x_2x_4$. This means that the DT procedure is not able to recover the lack of information deriving from the absence of eight feasible system configurations, reported in Tab. 1 and not included in the training set.

6.3.2.3 Shrinking the Tree

The splitting strategy previously presented relies on a measure of the information gain based on the examples included in the available training set. However, the size of the subset analyzed to add a new node to the DT decreases with the depth of the tree. Unfortunately, estimates based on small samples will not produce good results for unseen cases, thus leading to models with poor predictive accuracy, which is usually known as overfitting problem [13]. As a consequence, small decision trees consistent with the training set tend to perform better than large trees, according to the Occam's Razor principle [13].

The standard approach followed to take into account these considerations amounts to *pruning* branches off the DT. Two general groups of pruning techniques have been introduced in the literature: 1) pre-pruning methods that stop building the tree when some criteria is satisfied, and 2) post-pruning methods that first build a complete tree and then prune it back. All these techniques decide if a branch is to be pruned by analyzing the size of the tree and an estimate of the generalization error; for implementation details, the reader can refer to [26,27].

It is interesting to note that in the example presented in section 3.2.2 neither nodes nor branches can be removed from the final decision tree in Fig. 2 without degrading significantly the accuracy on the examples of the training set. In this case the pruning phase has no effect.

6.3.3 Shadow Clustering (SC)

As one can note, every system state $\boldsymbol{x}$ can be associated with a binary string of length d: it is sufficient to write the component states in the same order as they appear within the vector $\boldsymbol{x}$.

For example, the system state $\boldsymbol{x} = (0, 1, 1, 0, 1)$ for $d = 5$ will correspond to the binary string 01101. Since also the variable y, denoting if the con-

sidered system is operating or failed, is Boolean, at least in principle, any technique for the synthesis of digital circuits can be adopted to reconstruct the desired SF from a sufficiently large training set $\{(\boldsymbol{x}_j, y_j), j = 1, \ldots, N\}$. Unfortunately, the target of classical techniques for Boolean function reconstruction, such as MINI [17], ESPRESSO [6], or the Quine-McCluskey method [12], is to obtain the simplest logical sum-of-products that correctly classifies all the examples provided. As a consequence, they do not generalize well, i.e. the output assigned to a binary string not included in the given training set can be often incorrect.

To recover this drawback a new logical synthesis technique, called Hamming Clustering (HC) [18,19] has been introduced. In several application problems HC is able to achieve accuracy values comparable to those of best classification methods, in terms of both efficiency and efficacy. In addition, when we are facing with a classification problem the Boolean function generated by HC can be directly converted into a set of intelligible rules underlying the problem at hand.

Nevertheless, the top-down approach adopted by HC to generate logical products can require an excessive computational cost when the dimension d of the input vector is very high, as it can be the case in the analysis of system reliability. Furthermore, HC is not well suited for classification problems that cannot be easily coded in a binary form. To overcome this shortcoming, an alternative method, named Shadow Clustering (SC) [20], has been proposed. It is essentially a technique for the synthesis of monotone Boolean functions, writable as a logical sum-of-products not containing the complement (NOT) operator.

The application of a proper binary coding allows the treatment of general classification problems; the approach followed by SC, which resembles the procedure adopted by HC, leads to the generation of a set of intelligible rules underlying the given classification problem. Preliminary tests [21] show that the accuracy obtained by SC is significantly better than that achieved by HC in real world situations.

Since the system function may not be a monotone Boolean function, an initial coding β is needed to transform the training set, so as a logical sum-of-products not including the complement operator can be adopted for realizing the $SF(\boldsymbol{x})$. A possible choice consists in using the coding $\beta(\boldsymbol{x})$ that produces a binary string $\boldsymbol{z}$ with length $2d$, where every component x_i in $\boldsymbol{x}$ gives rise to two bits z_{2i-1} and z_{2i} according to the following rule:

$$z_{2i-1} = 1, z_{2i} = 0 \quad \text{if } x_i = 0, \text{ whereas } z_{2i-1} = 0, z_{2i} = 1 \quad \text{if } x_i = 1$$

It can be easily seen that this coding maps any binary training set into a portion of the truth table for a monotone Boolean function, which can then be reconstructed through SC. A basic concept in the procedure followed by

SC is the notion of *cluster*, sharing the same definition of *implicant* in classic theory of logical synthesis.

A cluster is the collection of all the binary strings having the value 1 in a same fixed subset of components. As an example, the eight binary strings 01001, 01011, 01101, 11001, 01111, 11011, 11101, 11111 form a cluster since all of them only have the value 1 in the second and in the fifth component. This cluster is usually written as 01001, since in the synthesis of monotone Boolean functions the value 0 serves as a don't care symbol and is put in the positions that are not fixed. Usually the cluster 01001 is said *to be covered* by the eight binary strings mentioned above.

Every cluster can be associated with a logical product among the components of $\boldsymbol{x}$, which gives output 1 for all and only the binary strings which cover that cluster. For example, the cluster 01001 corresponds to the logical product x_2x_5, obtained by considering only the components having the value 1 in the given cluster. The desired monotone Boolean function can then be constructed by generating a valid collection of clusters for the binary strings in the training set with output 1. This collection is consistent, if none of its elements is covered by binary strings of the training set having output 0.

After the application of the binary coding β on the examples $(\boldsymbol{x}_j, y_j)$ of the training set, we have obtained a new collection of input-output pairs $(\boldsymbol{z}_j, y_j)$ with $\boldsymbol{z}_j = \beta(\boldsymbol{x}_j)$, which can be viewed as a portion of the truth table of a monotone Boolean function. If T and F contain the binary strings $\boldsymbol{z}_j$ with corresponding output $y_j = 1$ and $y_j = 0$, respectively, the procedure employed by SC to reconstruct the sum-of-products expression for the desired monotone Boolean function $f(\boldsymbol{z})$ consists of the following four steps:

1. Set $S = T$ and $C = \emptyset$.
2. Starting from the implicant $000\cdots0$, turn some 0 in 1 to obtain a cluster $\boldsymbol{c}$ that is covered by the greatest number of binary strings in S and by no element of F.
3. Add the cluster $\boldsymbol{c}$ to the set C. Remove from S all the binary strings that cover $\boldsymbol{c}$. If S is not empty go to Step 2.
4. Simplify the collection C of clusters and build the corresponding monotone Boolean function.

As one can note, SC generates the sum-of-products expression for the desired monotone Boolean function f (Step 4) by examining a collection C of clusters incrementally built through the iteration of Steps 2–3. To this aim, SC employs an auxiliary set S to maintain the binary strings of T that do not cover any cluster in the current collection C.

The following subsections describe the solutions adopted by SC to construct the clusters to be added in C (Step 2) and to simplify the final collection C (Step 4), thus improving the generalization ability of the resulting monotone Boolean function.

6.3.3.1 Building Clusters

Starting from the largest cluster 0···00, containing only 0 values, an implicant $\boldsymbol{c}$ has to be generated for its inclusion in the collection C. The only prescription to be satisfied in constructing this cluster is that it cannot be covered by any binary string in F.

As suggested by the Occam's Razor principle, smaller sum-of-products expressions for the monotone Boolean function to be retrieved perform better; this leads to prefer clusters that are covered by as many as possible training examples in S and contain more don't care values 0 inside them.

However, searching for the optimal cluster in this sense leads to an NP-hard problem; consequently, greedy alternatives must be employed to avoid an excessive computing time. In these approaches an iterative procedure changes one at a time the components with value 0 in the cluster under construction, until no elements of F cover the resulting implicant $\boldsymbol{c}$.

Every time a bit c_i in $\boldsymbol{c}$ is changed from 0 to 1 a (possibly empty) subset R_i of binary strings in S do not cover anymore the new implicant. The same happens for a (possibly empty) subset $G_i \subset F$.

It can be easily seen that the subset R_i contains all the elements in S that cover $\boldsymbol{c}$ and has a value 0 in the ith component; likewise, the subset G_i includes all and only the binary strings in F covering $\boldsymbol{c}$ and having a 0 as the ith bit.

It follows that a greedy procedure for SC must minimize the cardinality $|R_i|$ of the subset R_i, while maximizing the number of elements in G_i, when the ith bit of $\boldsymbol{c}$ is set. In general, it is impossible to satisfy both these prescription; thus, it is necessary to privilege one of them over the other.

Trials on artificial and real-world classification problems suggest that the most promising choice consists in privileging the minimization of $|R_i|$, which leads to the Maximum-covering version of SC (MSC) [21]. Here, at every iteration the cardinality of the sets R_i and G_i is computed for every component i of $\boldsymbol{c}$ having value $c_i = 0$. Then the index i^* that minimizes $|R_i|$ is selected; ties are broken by taking the maximum of $|G_i|$ under the same value of $|R_i|$.

As an example, consider the network in Fig. 1, having system function $SF(\boldsymbol{x}) = x_1x_3+x_2x_4$, and the training set shown in Tab. 2. It can be noted that in this case the SF is a monotone Boolean function and therefore can be reconstructed by SC without recurring to the preliminary coding β. How-

ever, to illustrate the general procedure followed by SC, we apply anyway the mapping β, thus obtaining the binary training set (z_j, y_j) in Tab. 5.

Table 5. Binary training set for the system in Fig. 1 obtained by applying the coding β.

z_1	z_2	z_3	z_4	z_5	z_6	z_7	z_8	$y = SF(\boldsymbol{x})$
1	0	1	0	1	0	0	1	0
1	0	1	0	0	1	0	1	0
1	0	0	1	0	1	0	1	1
0	1	1	0	1	0	0	1	0
0	1	1	0	0	1	0	1	1
0	1	0	1	1	0	1	0	0
0	1	0	1	1	0	0	1	1
0	1	0	1	0	1	0	1	1

It can be directly obtained that the set T contains the strings 10010101, 01100101, 01011001, and 01010101, whereas F includes 10101001, 10100101, 01101001, and 01011010. Starting at Step 2 of SC with the generic implicant 00000000, we compute the cardinalities $|R_i|$ and $|G_i|$ for i = 1, …, 8, thus obtaining:

$$\begin{aligned} &|R_1| = 3, |R_2| = 1, |R_3| = 3, |R_4| = 1, |R_5| = 3, |R_6| = 1, |R_7| = 4, |R_8| = 0 \\ &|G_1| = 2, |G_2| = 2, |G_3| = 1, |G_4| = 3, |G_5| = 1, |G_6| = 3, |G_7| = 3, |G_8| = 1 \end{aligned} \quad (12)$$

Then, $|R_i|$ is maximized for i = 8; by changing the eight bit from 0 to 1, we obtain the cluster 00000001. At this time the subsets R_i for i = 1, …, 7 remain unchanged, whereas the cardinalities of the subsets G_i are

$$|G_1| = 1, |G_2| = 2, |G_3| = 0, |G_4| = 3, |G_5| = 1, |G_6| = 2, |G_7| = 3$$

Now, the minimum value of $|R_i|$ = 1 is obtained for i = 2, 4, 6, but the maximization of $|G_i|$ suggests to change from 0 to 1 the fourth bit, thus obtaining the cluster $\boldsymbol{c}$ = 00010001. Since this implicant is not covered by any element of F, it can be inserted into C (Step 3). Then the set S is reduced by removing from it the binary strings that cover $\boldsymbol{c}$, namely 10010101, 01011001, and 01010101; it follows that S = {01100101}.

The procedure is then repeated at Step 2, by considering again the generic implicant 00000000 and by computing the cardinalities $|R_i|$ for i = 1, …, 8. We obtain:

$$|R_1| = 1, |R_2| = 0, |R_3| = 0, |R_4| = 1, |R_5| = 1, |R_6| = 0, |R_7| = 1, |R_8| = 0$$

Note that the subsets G_i in (12) are not changed since F was not altered. It is immediately seen that the best choice corresponds to i = 6; changing the corresponding bit from 0 to 1 yields the cluster 00000100. The cardinalities $|G_i|$ now become

$$|G_1| = 0, |G_2| = 1, |G_3| = 0, |G_4| = 1, |G_5| = 1, |G_7| = 1, |G_8| = 0$$

Consequently, the index $i = 2$ is selected, thus leading to the implicant $\boldsymbol{c}$ = 01000100 to be inserted into C.

Removing at Step 3 the last element from S, which covers $\boldsymbol{c}$, we obtain that S becomes empty and the execution of SC follows at Step 4 with the simplification of the resulting collection $C = \{00010001, 01000100\}$.

6.3.3.2 Simplifying the Collection of Clusters

Usually, the repeated execution of Steps 2-3 leads to a redundant set of clusters, whose simplification can improve the prediction accuracy of the corresponding monotone Boolean function. In analogy with methods for decision trees, the techniques employed to reduce the complexity of the resulting sum-of-products expressions are frequently called *pruning algorithms*.

The easiest effective way of simplifying the set of clusters produced by SC is to apply the *minimal pruning* [19,21]: According to this greedy technique the clusters that is covered by the maximum number of elements in T are extracted one at a time. At each extraction, only the binary strings not included in the clusters already selected are considered. Breaks are tied by examining the whole covering.

The application of minimal pruning to the example analyzed in the previous subsection begins with the computation of the covering associated with each of the two clusters generated in the training phase. It can be readily observed that 00010001 covers three examples of Tab. 3 (precisely the binary strings 10010101, 01011001 and 01010101), whereas the covering of 01000100 is equal to 2. Consequently, the cluster 00010001 is firstly selected.

After this choice only the binary string 01100101 does not cover any implicant, which leads to the selection of the second cluster 01000100. No simplification is then possible in the collection C, which leads to the monotone Boolean function $z_4 z_8 + z_2 z_6$. By applying in the opposite way the coding β we obtain the desired expression for the system function $SF(\boldsymbol{x}) = x_2x_4+x_1x_3$, i.e. the correct SF for the system in Fig. 1.

6.4 Example

To evaluate the performance of the methods presented in the previous sections, the network shown in Fig. 3 has been considered [39]. It is assumed that each link has reliability P_i and capacity of 100 units. A system failure occurs when the flow at the terminal node t falls below 200 units. Conse-

quently, a max-flow min-cut algorithm is used to establish the value of the *EF* [23,29].

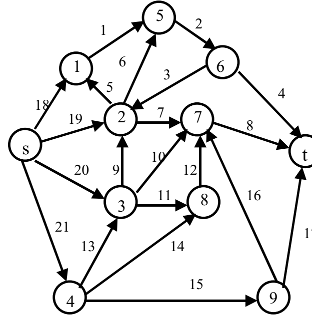

Fig. 3. Network to be evaluated [39]

In order to apply a classification method it is first necessary to collect a set of examples $(\boldsymbol{x},y)$, where $y = EF(\boldsymbol{x})$, to be used in the training phase and in the subsequent performance evaluation of the resulting set of rules. To this aim, 50000 system states have been randomly selected without replacement and for each of them the corresponding value of the *EF* has been retrieved.

To analyze how the size of the training set influences the quality of the solution provided by each method, 13 different cases were analyzed with 1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000, 10000, 15000, 20000 and 25000 examples in the training set. These examples were randomly extracted with uniform probability from the whole collection of 50000 system states; the remaining examples were then used to test the accuracy of the model produced by the machine learning technique. An average over 30 different choices of the training set for each size value was then performed to obtain statistically relevant results.

The performance of each model is evaluated using standard measures of *sensitivity, specificity* and *accuracy* [37]:

sensitivity = TP/(TP+FN); specificity = TN/(TN+FP);

accuracy = (TP+TN)/(TP+TN+FP+FN)

where

- TP (resp. TN) is the number of examples belonging to the class $y = 1$ (resp. $y = 0$) for which the classifier gives the correct output,
- FP (resp. FN) is the number of examples belonging to the class $y = 1$ (resp. $y = 0$) for which the classifier gives the wrong output.

For reliability evaluation, *sensitivity* gives the percentage of correctly classified operating states and *specificity* provides the percentage of correctly classified failed states.

6.4.1 Performance Results

Different kernels were tried when generating the SVM model and it was found that the best performance is achieved with a Gaussian radial basis function (GRBF) kernel with parameter $(1/2\sigma^2) = 0.05$. All SVM models obtained are able to completely separate the corresponding training set. The optimization required in the training phase was performed using the LIBSVM software [9]. Table 6 shows the average performance indices during the testing phase.

Table 6. Average performance indices for SVM (test)

Training Set Size	Accuracy %	Sensitivity %	Specificity %
1000	95.12	93.93	95.61
2000	96.90	96.96	96.87
3000	97.63	97.93	97.51
4000	98.09	98.50	97.92
5000	98.42	98.84	98.25
6000	98.66	99.09	98.48
7000	98.83	99.27	98.64
8000	98.97	99.41	98.79
9000	99.07	99.50	98.90
10000	99.17	99.59	98.99
15000	99.48	99.81	99.35
20000	99.64	99.88	99.53
25000	99.73	99.93	99.64

Table 7. Average performance indices for DT

Training Set Size	Accuracy (%)		Sensitivity (%)		Specificity (%)	
	Training	Test	Training	Test	Training	Test
1000	98.71	95.65	98.26	92.67	98.90	96.91
2000	99.25	97.30	98.89	95.48	99.40	98.08
3000	99.42	98.06	99.15	96.52	99.54	98.71
4000	99.54	98.63	99.44	97.72	99.59	99.01
5000	99.63	98.94	99.51	98.17	99.68	99.27
6000	99.69	99.10	99.58	98.45	99.73	99.38
7000	99.74	99.23	99.67	98.70	99.77	99.45
8000	99.80	99.37	99.71	98.89	99.83	99.57
9000	99.81	99.42	99.75	98.94	99.84	99.63
10000	99.83	99.47	99.75	99.02	99.86	99.66
15000	99.90	99.69	99.85	99.42	99.92	99.80
20000	99.93	99.80	99.88	99.57	99.96	99.89
25000	99.95	99.85	99.92	99.71	99.97	99.92

As for DT, the resulting models do not classify correctly all the examples in the training set. Table 7 presents the average performance indices during training and testing. Finally, Table 8 shows the average performance indices obtained by SC only during testing, since also SC does not commit errors on the system states of the training set.

Table 8. Average performance indices for SC (test)

Training Set Size	Accuracy %	Sensitivity %	Specificity %
1000	96.30	94.52	97.05
2000	98.07	97.34	98.38
3000	98.67	98.11	98.91
4000	99.01	98.64	99.16
5000	99.26	99.00	99.37
6000	99.45	99.32	99.50
7000	99.54	99.43	99.58
8000	99.60	99.49	99.64
9000	99.65	99.54	99.69
10000	99.69	99.59	99.73
15000	99.84	99.80	99.85
20000	99.89	99.89	99.90
25000	99.92	99.92	99.92

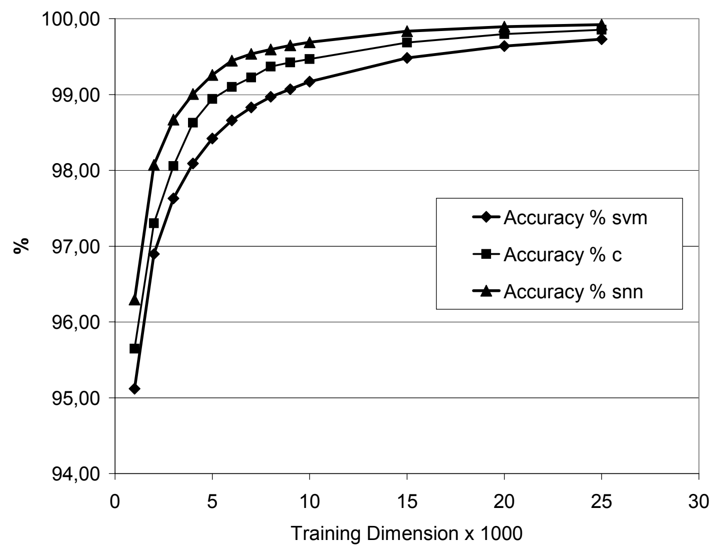

Fig. 4. Average accuracy obtained by each ML method in the test phase

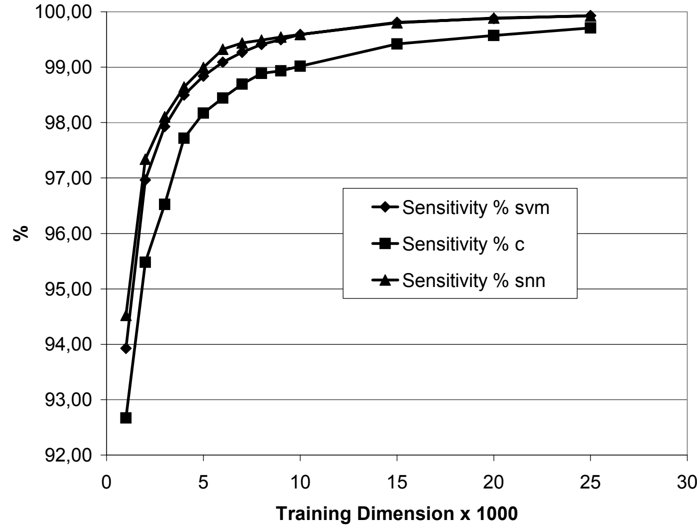

Fig. 5. Average sensitivity obtained by each ML method in the test phase

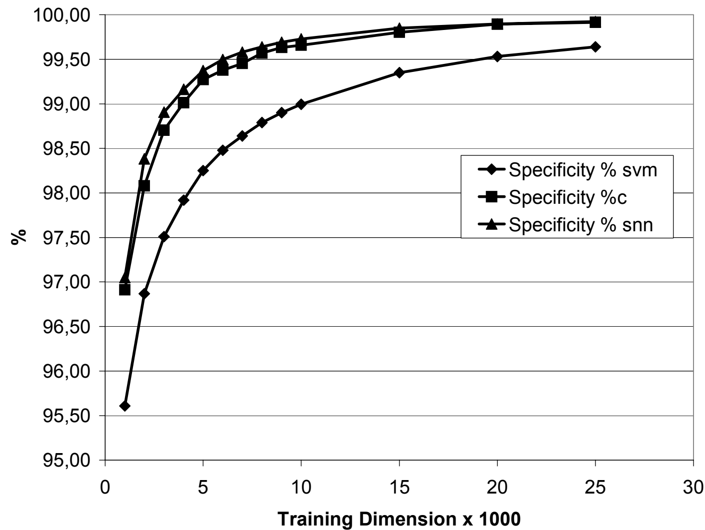

Fig. 6. Average specificity obtained by each ML method in the test phase

Figures 4–6 show the result comparison regarding accuracy, sensitivity and specificity for the ML techniques considered. It is interesting to note that the index under study for each model increases with the size of the training set. SC has the best behavior for all the indices. However, for the

sensitivity index, the performances of SC and SVM are almost equal. For the specificity index, the performance of SC and DT are almost equal.

This means that SC seems to be more stable when considering the three indices simultaneously. In [33] different networks are evaluated using different *EF*: the behavior obtained is similar to the one observed in the network analyzed in this chapter.

6.4.2 Rule Extraction Evaluation

As previously mentioned, DT and SC are able to extract rules that explain the behavior of the systems in the form of a logical sum-of-products approximating the *SF*. DT rules are in disjoint form, so the *ARE* can be easily determined. Rules generated by SC are not disjoint; thus, an additional procedure, such as the algorithm KDH88 [16], has to be used to perform this task.

Table 9 shows the average number of paths and cuts generated by both procedures. As can be seen in Fig. 7, both techniques are able to extract more and more path and cut sets as long as the training set is increased (the system under study has 43 minimal paths and 110 minimal cuts). However, for a given training set size, SC can produce more path and cut sets than DT.

Table 9. Average number of paths and cuts extracted by DT and SC

Training Set Size	PATHS		CUTS	
	DT	SC	DT	SC
1000	2.2	3.5	17.5	21.2
2000	5.3	8.5	24.9	27.9
3000	7.9	12.5	29.1	34.3
4000	9.9	15.5	32.9	38.2
5000	11.6	18.6	36.1	41.4
6000	13.5	21.1	39.0	45.4
7000	14.7	23.0	41.0	48.5
8000	16.6	24.5	43.6	51.6
9000	18.2	26.1	45.5	52.7
10000	19.1	27.4	47.7	55.0
15000	23.0	31.6	55.8	64.9
20000	26.0	34.8	61.4	72.3
25000	27.9	36.3	66.2	77.7

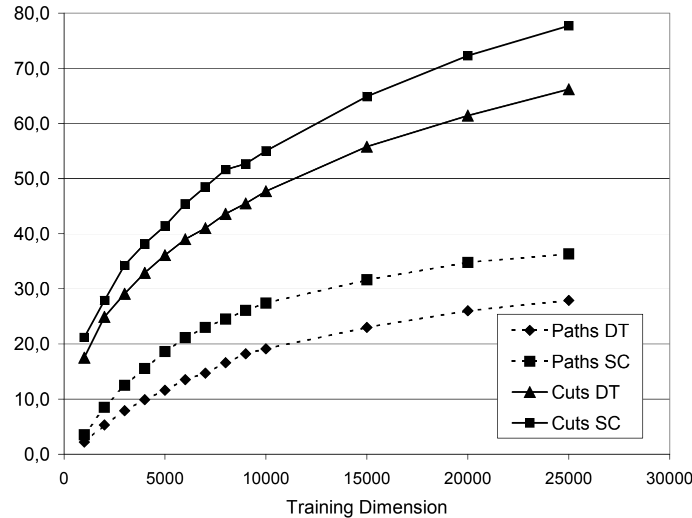

Fig. 7. Average number of paths and cuts extracted by DT and SNN

Once DT and SC are trained, their resulting *ARE* is used to evaluate the network reliability. Table 10 shows the network reliability evaluated using the correct *RE* and the *ARE* obtained by both models, for $P_i = 0.90$; the relative errors are also included for completeness. Both models produce excellent results, but SC errors are significantly lower. On the other hand, for a specific relative error, SC requires a training set with lower size.

Table 10. Average network reliability and relative error using the path sets extracted by DT and SC

Training Set Size	ARE Evaluation		Rel. Error (%)	
	DT	SC	DT	SC
1000	0.64168	0.73433	28.85	18.57
2000	0.79388	0.87961	11.98	2.47
3000	0.85864	0.89315	4.80	0.97
4000	0.88066	0.89637	2.35	0.61
5000	0.88980	0.89830	1.34	0.39
6000	0.89234	0.89972	1.06	0.24
7000	0.89401	0.90034	0.87	0.17
8000	0.89701	0.90078	0.54	0.12
9000	0.89784	0.90087	0.45	0.11
10000	0.89812	0.90115	0.42	0.08
15000	0.89992	0.90180	0.22	0.01
20000	0.90084	0.90194	0.12	0.00
25000	0.90119	0.90200	0.08	0.00

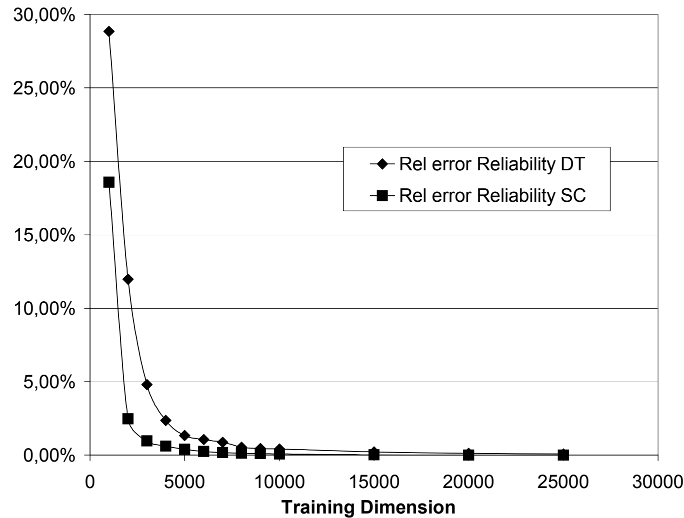

Fig. 8. Reliability relative error using the path sets extracted by DT and SC

6.5 Conclusions

This chapter has evaluated the excellent capability of three machine learning techniques (SVM, DT and SC) in performing reliability assessment, in generating the *Approximate Reliability Expression* (*ARE*) of a system and in determining cut and path sets for a network.

SVM produce an approximation to the *SF*, which cannot be written in the form of a logical sum-of-products. However, the model generated can be used within a standard Monte Carlo approach to replace the *EF*.

On the other hand, DT and SC are able to generate an approximation to the *SF* in the form of a logical sum-of-products expression, even from a small training set. The expression generated using DT is in disjoint form, which allows to easily obtain the corresponding *ARE*. Expressions generated by SC need to be converted in disjoint form, so as to produce the desired *ARE*. Both DT and SC provide information about minimum paths and cuts.

The analysis of the results on a 21-link network has shown that SC is more stable for all the performance indices evaluated, followed by DT and SVM.

The same set of experiments has been used to evaluate the performance of three ML techniques in two additional networks [33]. The first network

has 20 nodes and 30 double-links. The *EF* is the connectivity between a source node *s* and a terminal node *t*.

The second network analyzed has 52 nodes and 72 double links (the Belgian telephone network) and the success criterion used is the all-terminal reliability (defined as the probability that every node of the network can communicate with every other node through some path). The behavior of SVM, DT and SC for these networks has been similar to the results reported here.

The analysis of different training sets has also shown that SC seems to be more efficient than DT for extracting cut and paths sets: for a specific data set size, SC can produce more sets and therefore, a more precise reliability assessment.

Even if the machine-learning-based approach has been initially developed for approximating binary *SF*, it has been extended to deal also with multi-state systems [32], obtaining again excellent results.

References

1. Aggarwal K.K., Chopra Y.C., Bajwa J.S.: Capacity consideration in reliability analysis of communication systems, *IEEE Transactions on Reliability*, 31, 1982, pp. 177–181.
2. Bevilacqua M., Braglia M., Montanari R.: The classification and regression tree approach to pump failure rate analysis, *Reliability Engineering and System Safety*, 79, 2002, pp. 59–67.
3. Billinton, R. Allan R.N: *Reliability Evaluation of Engineering Systems, Concepts and Techniques (second edition)*, Plenum Press, 1992.
4. Billinton, R. Li W.: *Reliability Assessment of Electric Power System Using Monte Carlo Methods*, Plenum Press, 1994.
5. Breiman L., Friedman J. H., Olshen R. A., Stone C. J.: *Classification and Regression Trees*, Belmont: Wadsworth, 1994.
6. Brayton R. K., Hachtel G. D., McMullen C. T., Sangiovanni-Vincentelli A. L.: *Logic Minimization Algorithms for VLSI Synthesis*, Hingham, MA: Kluwer Academic Publishers, 1984.
7. Campbell C.: An introduction to kernel methods, In Howlett R.J. and Jain L.C., editors, *Radial Basis Function Networks: Design and Applications*, p. 31. Springer Verlag, Berlin, 2000.
8. Cappelli C., Mola F., Siciliano R.: A statistical approach to growing a reliable honest tree, *Computational Statistics & Data Analysis*, 38, 2002, pp. 285–299.
9. Chang C.-C., Lin C.-J.: *LIBSVM: A Library for Support Vector Machines*, software available at http://www.csie.ntu.edu.tw/~cjlin/libsvm/index.html, 2001.

10. Colbourn Ch.: *The Combinatorics of Network Reliability*, Oxford University Press, 1987.
11. Cristianini N., Shawe-Taylor J.: *An Introduction to Support Vector Machines*, Cambridge University Press, 2000.
12. Dietmeyer D. L., *Logical Design of Digital Systems (third edition)*, Boston, MA: Allyn and Bacon, 1988.
13. Duda R. O., Hart P. E., Stork D. G.: *Pattern Classification*, John Wiley & Sons, 2001.
14. Dubi A.: Modeling of realistic system with the Monte Carlo method: A unified system engineering approach, *Proceedings of the Annual Reliability and Maintainability Symposium*, Tutorial Notes, 2001.
15. Grosh D.L.: *Primer of Reliability Theory*, John Wiley & Sons, New York, 1989
16. Heidtmann K. D.: Smaller sums of disjoint products by subproducts inversion, *IEEE Transactions on Reliability*, 38, 1989, pp. 305–311.
17. Hong S. J., Cain R. G., Ostapko D. L.: MINI: A heuristic approach for logic minimization, *IBM Journal of Research and Development*, 18, 1974, pp. 443–458.
18. Muselli M., Liberati D.: Training digital circuits with Hamming Clustering, *IEEE Transactions on Circuits and Systems*, 47, 2000, pp. 513–527.
19. Muselli M., Liberati D.: Binary rule generation via Hamming Clustering, *IEEE Transactions on Knowledge and Data Engineering*, 14, 2002, pp. 1258–1268.
20. Muselli M., Quarati A.: Reconstructing positive Boolean functions with Shadow Clustering, *ECCTD 2005 – European Conference on Circuit Theory and Design*, Cork, Ireland, 2005.
21. Muselli M.: Switching Neural Networks: A new connectionist model for classification, *WIRN '05 – XVI Italian Workshop on Neural Networks*, Vietri sul Mare, Italy, 2005.
22. Murthy S., Kasif S., Salzberg S.: A system for induction of oblique decision tree, *Journal of Artificial Intelligence Research*, 2, 1994, pp. 1–32.
23. Papadimitriou C. H., Steiglitz K.: *Combinatorial Optimisation: Algorithms and Complexity*, Prentice Hall, New Jersey, 1982.
24. Pereira M. V. F., Pinto L. M. V. G.: A new computational tool for composite reliability evaluation, *IEEE Power System Engineering Society Summer Meeting*, 1991, 91SM443-2.
25. Pohl E. A., Mykyta E. F.: Simulation modeling for reliability analysis, *Proceedings of the Annual Reliability and Maintainability Symposium*, Tutorial Notes, 2000.
26. Portela da Gama J. M.: *Combining Classification Algorithms*, PhD. Thesis, Faculdade de Ciências da Universidade do Porto, 1999.
27. Quinlan J. R.: *C4.5: Programs for Machine Learning*, Morgan Kaufmann Publishers, 1993.
28. Rai S, Soh S.: A computer approach for reliability evaluation of telecommunication networks with heterogeneous link-capacities, *IEEE Transactions on Reliability*, 40, 1991, pp. 441–451.

29. Reingold E., Nievergelt J., Deo N.: *Combinatorial Algorithms: Theory and Practice*, Prentice Hall, New Jersey, 1977.
30. Rocco C. M., Moreno J. M.: Fast Monte Carlo reliability evaluation using Support Vector Machine, *Reliability Engineering and System Safety*, 76, 2002, pp. 239–245.
31. Rocco C. M.: A rule induction approach to improve Monte Carlo system reliability assessment, *Reliability Engineering and System Safety*, 82, 2003, pp. 87–94.
32. Rocco C. M., Muselli M.: Approximate multi-state reliability expressions using a new machine learning technique, *Reliability Engineering and System Safety*, 89, 2005, pp. 261–270.
33. Rocco C. M., Muselli M.: Machine learning models for reliability assessment of communication networks, submitted to *IEEE Transactions on Neural Networks*.
34. Shawe-Taylor J., Cristianini N.: *Kernel Methods for Pattern Analysis*, Cambridge University Press, 2004.
35. Stivaros C., Sutner K.: Optimal link assignments for all-terminal network reliability, *Discrete Applied Mathematics*, 75, 1997, pp 285–295.
36. Vapnik V.: *Statistical Learning Theory*, John Wiley & Sons, 1998.
37. Veropoulos K., Campbell C., Cristianini N.: Controlling the sensitivity of Support Vector Machines, *Proceedings of the International Joint Conference on Artificial Intelligence*, Stockholm, Sweden, 1999, pp. 55–60.
38. Wahba G.: *Spline Models for Observational Data*, SIAM, 1990.
39. Yoo Y. B., Deo N.: A comparison of algorithm for terminal-pair reliability, *IEEE Transaction on Reliability*, 37, 1988, pp. 210–215.