C++ Tools
for
Logical Analysis of Data

Eddy Mayoraz
July 1998

Table of content

Abstract This document describes a software designed to experiment Logical Analysis of Data (LAD). First, while reminding the user what is LAD all about, it gives a complete description of the possibilities of the software. A second part describes how to use the different programs. In a third part (to be completed soon) it gives an insight on the modular structure of this software as well as an understanding of the semantics of its components, in order to provide the reader the possibility to modify the existing code, to add new components or to reuse some modules in different contexts.

Acknowledgement The author developed the basis of the code described in this document in 1994-1995, during a post-doctoral visit at RUTCOR—Rutgers University's Center for Operations Research, New Jersey. During the last three years, several extensions and improvements have been achieved at IDIAP and others are still ongoing. The author is thankful to his colleagues, in particular to Miguel Moreira and Johnny Mariéthoz for their precious collaboration in this work.

Foreword ***This software has been designed for research purpose***. Modularity was a prerequisite, so that each step of Logical Analysis can easily be suppressed, modified or replaced in the processing chain. Moreover, in any combinatorial or logical analysis, there are several mathematical tools that are used constantly. We tried to identify these tools and to implement them in separate modules of general purposes, so that they can be reused easily as often as possible (see for example classes `Matrix`, `binMatrix`, `setCovering`). For the realization of this project we choose the C++ programming language [1] for its popularity and its reasonably high level of abstraction.

This software has been designed for research purpose only. In particular, this means that at any level of the program, it is always assumed that both, the user of the executable and the programmer using parts of this software, know what they are doing. For example, there is no systematic test on erroneous parameters passed to any functions, and in case of misusage of modules or calls in an inap-

propriate sequence, the result is unpredictable. The only tests that are carried out are those that can help the user in tracking errors in his code. These are lower level tests such as checking indices out of range, detecting unexpected null pointers, and are done systematically.

This document contains three parts. *Part 1* and *Part 2* are intended for the user of this software, while *Part 3* is intended to the developer interested in using, but also modifying and extending, some pieces of this software.

- *The first part* presents an overview of the method LAD and the different functionalities of this software.
- *The second part* focuses of the usage of three executable files `bin`, `pat` and `the`, which are programs providing a simple access to most of the components of this software through primitive console-type interfaces. These programs are however not user-friendly, and are meant for research purposes only.
- *The third part* (to be completed soon) presents a description of the main structural components of this software.

In this text, the following terminology and notations are extensively used. A ***database*** is a set of observations. An ***observation*** is a point in a multi-dimensional space. Each dimension of this space is refereed to as an ***attribute***. All the *observations* of a particular database are partitioned into several ***classes***, and the main purpose of this software is the classification of any new *observation* into one of the existing classes. The classes will always be indexed by $c = 1,\ldots,C$, but most of the time this index will be omitted and, instead, it will be mentioned in the text whether the *observations* we consider are from the same class or from different classes. The *observations* and the attributes are indexed by $p = 1,\ldots,P$ and $i = 1,\ldots,I$ respectively.

PART 1

Functionalities

Introduction

General structure of the software

The complete data processing implemented in this software can be divided into three phases:

- binarization of data;
- generation of patterns;
- formation of theory;

accessible through three executables `bin`, `pat` and `the`. A fourth executable `LAD` consists in a sequential call of the first three. *Figure 1* illustrates this structure.

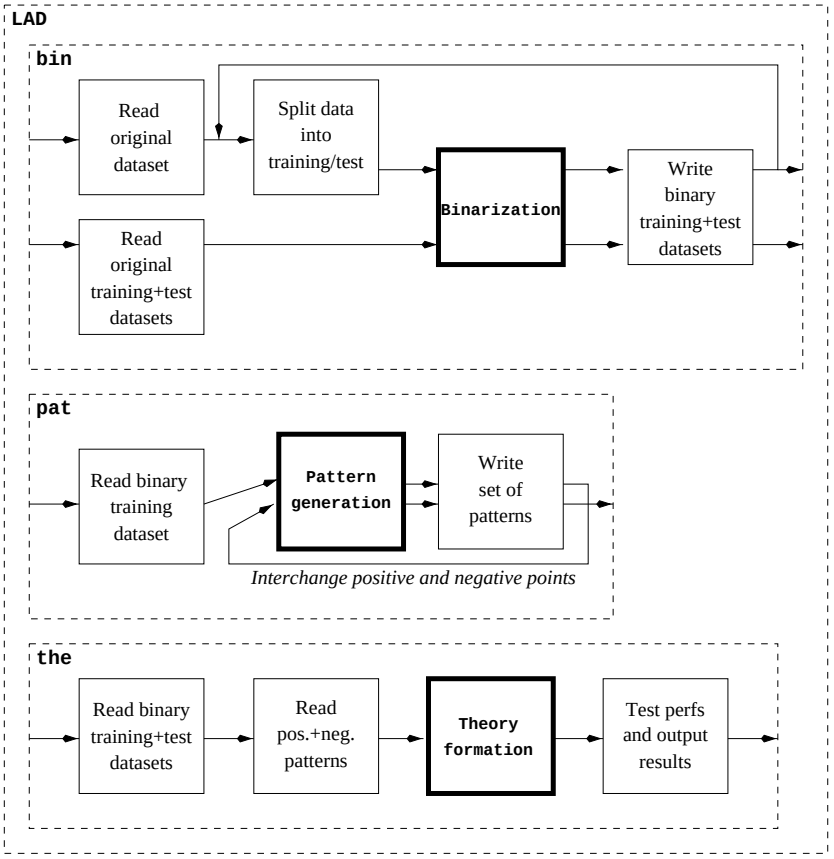

Figure 1. General structure of the software

The generation of positive and negative patterns is produced by two consecutive calls to a unique pattern generation procedure, after interchanging the roles of positive and negative *observations*.

Note The binarization phase is designed to handle multiple classes. On the other hand, the pattern generation and the theory formation are restricted to problems with two classes only.

Characteristic of input data

- The complete analysis is implemented in such a way as to handle missing data. Any missing data is potentially matching any value, with the idea that the "worse" value for our need will always be chosen. For example, when we check whether the dataset is consistent (i.e. whether there is no two identical *observations* in two different classes), two *observations* (1,2,?) and (1,?,3) from two different classes are inconsistent.
- Three types of *attributes* are distinguished:
  - the ***binary attributes*** taking values 0 and 1 or false and true;
  - the ***nominal attributes*** with a small finite set of possible values;
  - the ***continuous attributes*** with values in a continuous interval.

  The different attributes are also divided into ***unordered attributes*** and

ordered attributes. Two values of an *ordered attribute* are comparable, while values of an *unordered attributes* are not. A *binary attribute* is considered as ordered, with order 0 < 1 or false < true. A *Nominal attribute* with more than two possible values is *unordered*, while a two-valued *nominal attribute* is assimilated to a *binary attribute* and thus is considered as *ordered*. Finally, a *continuous attribute* is obviously *ordered*.

- Each *ordered attribute* of the original database can be specified as ***positive***, ***negative*** or ***without monotonicity constraints***. If an attribute is positive (resp. negative), it cannot be used to *discriminate* between a positive and a negative *observation* if the first one has a smaller (resp. larger) value than the second one for this attribute.

Protocols of experiments

The original data can be either already split into training set and test set, or it can be constituted of a single dataset. In the last case, it is often desirable to validate the learning method through some cross-validation processes. Two popular protocols of experiments are available.

The ***N x K-fold cross-validation*** consists in N iterations of the following procedure. The dataset is split into K parts (each class is split as evenly as possible); for k=1,…,K, the training data is composed of every data except those of the k^{th} part, which are used as test data. It is also possible to do it the other way around, i.e. using one fold for training, and the K–1 remaining folds as test.

In the ***N-resampling cross-validation*** protocol, at each of the N iterations, the dataset is split at random into two parts according to a given percentage (each class is split as evenly as possible). The percentage of data used for training can vary between two bounds. This is useful to highlight the dependence between the efficiency of the algorithm and the training size.

Binarization

The purpose of the binarization is the transformation of a database of any type into a Boolean database. This step can be omitted whenever the original database is already fully Boolean. For simplicity, in the present text a *binary attribute* refers to a two-valued attribute of the original database, while a ***Boolean attribute*** denotes a binary attribute resulting from the binarization. A *Boolean attribute* either

(i) is identical to one *binary attribute*,
(ii) is associated to a specific value of one *nominal attribute*,
(iii) corresponds to one ***cut point***, i.e. a critical value along one *continuous attribute*.

In case (*iii*), the *Boolean attribute* takes the value 1 whenever the *continuous attribute* is greater than the cut point. While in case (*ii*), the *Boolean attribute* has the value 1 if and only if the *nominal attribute* has the associated value.

Note The number of *cut points* placed along the same *continuous attribute* is not limited: it can be 0, or it can be as big as necessary.

Note With this binarization of *nominal attributes*, if for a test data a *nominal attribute* takes a value that never occurred in the training dataset, every *Boolean attribute* corresponding to the *nominal attribute* is coded as 0.

The first step of the binarization procedure consists in the generation of a large set of *Boolean attributes* called the ***candidate attributes***. The main stage of the binarization procedure is the extraction of a small subset of *Boolean attributes* from the set of *candidate attributes*. Since the set of *candidate attributes* can be very large and the extraction procedure is time consuming, a facultative step can precede the extraction, in which the *candidate attributes* are ordered according to some criteria, and only a subset of them with high precedence is kept. Finally, the binarization itself takes place, according to the final set of *Boolean attributes* obtained. So, the binarization phase consists of four steps:

- generation of *candidate attributes*;
- ordering and selection of *candidate attributes* with highest precedence;
- extraction of a ‘minimal’ subset of *candidate attributes*;
- construction of the binary data.

Generation of candidate attributes

One *candidate attribute* is generated for each original *binary attribute*. There are V *candidate attributes* generated for each *nominal attribute* taking $V > 2$ distinct values in the training set. Currently, two different methods are implemented for the generation of the candidate *cut points*. The first method, called ***one-cut-per-change***, introduces a *cut point* (t,i) (i.e. of value t along attribute i) if there exist two *observations* ***a*** and ***b*** belonging to two different classes such that $a_i < t = (a_i+b_i)/2 < b_i$ and if there is no *observation* ***c*** with $a_i < c_i < b_i$. The second method introduces a *cut point* $t = (a_i + b_i)/2$ if there exists a pair of *observations* ***a*** belonging to Class c’ and ***b*** belonging to Class c” $> c$’ so that either i is *non-*

monotonic and $a_i \leq b_i$, or i is *positive* and $a_i < b_i$, or i is *negative* and $a_i > b_i$. It will be refereed to as the ***one-cut-per-pair*** method.

The number of *candidate attributes* generated is usually very large it is sometimes better to reduce this set in two steps:

- The *candidate attributes* are sorted and only the best are kept. Different sorting procedures are discussed in Section *Sorting and pre-selection of the candidate attributes*.
- A global optimization procedure discussed in Section *Extraction of a subset of candidate attributes* extracts a small subset of *candidate attributes*.

Extraction of a subset of candidate attributes

A *candidate attribute* d ***discriminates*** a pair of *observations* (***a***,***b***), if the values taken by d for ***a*** and for ***b*** differ. In other words, a *candidate attribute* d associated to a *binary attribute* i ***discriminates*** (***a***,***b***) if and only if $a_i \neq b_i$. A *candidate attribute* d associated to a *nominal attribute* i with value v ***discriminates*** (***a***,***b***) if either $a_i = v$, or $b_i = v$, but not both. A *candidate attribute* d associated to a continuous attribute i with *cut point* value t ***discriminates*** (***a***,***b***) if and only if t is neither smaller nor bigger than both, a_i and b_i. If i is *positive* (resp. *negative*), (t,i) ***discriminates*** between ***a*** belonging to class *c'* and ***b*** belonging to class *c''*>*c'* only if $a_i < t < b_i$ (resp. $a_i > t > b_i$).

A good set of *candidate attributes* should be such that any pair of *observations* from two different classes *is discriminated* by at least one attribute of the set. The original method proposed for the extraction of a small subset of attributes from a given set T determines the smallest subset of attributes with this property by solving the following set covering problem:

$$\begin{aligned} \text{Min} \quad & \sum_{d \in T} z_d \\ \text{s.t.} \quad & \sum_{d \in T} s_{d\,\mathbf{ab}} \cdot z_d \geq 1 \;\forall\; (\mathbf{a},\mathbf{b}) \text{ from different classes} \\ & z_d \in \{0,1\} \;\forall\; d \in T \end{aligned} \qquad (1)$$

where $s_{d\mathbf{ab}} = 1$ if d *discriminates* between ***a*** and ***b***, and $s_{d\mathbf{ab}} = 0$ otherwise.

In the current form of the software, this problem can be solved by various heuristics. This is satisfactory since, in this application, it is not critical to obtain the minimum subset of attributes. Experiment even showed that some larger subsets than the ones provided by our heuristics often led to better final results. Therefore, the current version of this procedure for the extraction of a subset of attributes provides the liberty to specify any positive integer value as the right-hand-side of the constraints in (1).

The measure of pair discrimination of *candidate attributes* associated to *continuous attributes* can be refined if one considers that the larger the gap between t and a_i and b_i the better. For any pair $((\mathbf{a},\mathbf{b}),(t,i))$, the ***discriminating power*** of $d = (t,i)$ between ***a*** and ***b*** is defined as

$$\min\{|t - a_i| , |t - b_i|\} / (\max_{\boldsymbol{a}} a_{\boldsymbol{i}} - \min_{\boldsymbol{a}} a_{\boldsymbol{i}}) \quad (2)$$

if (t,i) *discriminates* between $\boldsymbol{a}$ and $\boldsymbol{b}$, and is 0 otherwise. The choice of the normalization (denominator of expression *(2)*)is arbitrary and it could be replaced for example by the standard deviation along attribute i. With this definition, the maximal *discriminating power* is 0.5, and the *discriminating power* of *candidate attributes* associated to *nominal* or *binary attributes* is arbitrarily set to 0.5, whenever *discrimination* occurs.

In the current procedure for the extraction of a small subset of *cut points*, an alternative is proposed, based on the *discriminating power* instead of the binary discrimination. The integer linear program expressing the previous set-covering problem in *equation (1)* is replaced by a linear program where s_{dab} is the *discriminating power* of d between $\boldsymbol{a}$ and $\boldsymbol{b}$, and the right-hand-side is an arbitrary value representing the required minimal discrimination between two *observations* from different classes. So, we have currently two methods for the extraction of a small subset of *cut points*: the first one is based on a ***binary-discrimination***, while the second one is based on a ***continuous-discrimination***.

For both methods, it can happen that the problem has no solution for a specific right-hand-side. In such cases, for each pair $(\boldsymbol{a},\boldsymbol{b})$ leading to a non satisfiable constraint, all the z_d corresponding to $s_{d\boldsymbol{ab}} > 0$ are set to 1 and the constraint is removed for the system of inequalities.

Sorting and pre-selection of the candidate attributes

Three different ordering criteria are now available. In each of these methods, a weight is associated to each *candidate attribute*, which are then sorted in a weight decreasing sequence. The first method, called ***ordering-by-entropy***, assumes that a good *candidate attribute* contains by itself a lot of information for the global classification. A weight given by

$$\max\{ \Sigma_c p_c^{\,1} \ln(p_c^{\,1}) , \Sigma_c p_c^{\,0} \ln(p_c^{\,0}) \} \quad (3)$$

is associated to a *candidate attribute*, where $p_c^{\,s}$ is the conditional probability that an *observation* $\boldsymbol{a}$ is in class c given that the *candidate attribute* takes value s on $\boldsymbol{a}$. These weights are clearly non positive, and since $\Sigma_c p_c^{\,1} = \Sigma_c p_c^{\,0} = 1$, a weight is 0 if and only if $p_c^{\,s} = 1$ for one c=1,…,C and one s = 0,1.

The second method, ***ordering-by-minimal-discrimination***, associates to a *candidate attribute* d a weight proportional to its smallest non-zero *discriminating power* over all possible pairs of *observations* from two different classes. This weighting measures the robustness of an attribute and it clearly favors the ones associated to nominal or *binary attributes*.

The motivation for the *ordering-by-minimal-discrimination* method is that a *cut point* with low *discriminating power* for some pairs of *observations* should

be avoided. Instead of the minimal *discriminating power*, the third method, ***ordering-by-total-discrimination***, associates to each attribute the sum of the *discriminating power* for all possible pairs of *observations* from different classes. This third weighting method has also some similarity with the first one. For example, an attribute associated to an original *binary attribute* has a *discriminating power* of either 0 or 0.5 for each pair of *observations*, therefore its weight depends on the number of pairs it *discriminates*.

Binarization and confidence interval

In the final stage of the binarization procedure, the original database is replaced by a new one with one *Boolean attribute* for each remaining *candidate attribute*.

In the case the extracting method is based on *continuous-discrimination*, we do care about the *discriminating power* of a *cut point* for each pair of *observations*. However, it may happen that a *cut point* which has been selected for its high *discriminating power* between some pairs, has a poor *discriminating power* between some other pairs, and we would like to avoid relying on this *cut point* to distinguish the latter pairs of *observations*. A natural way to model this is to define a confidence interval θ for all the *cut points* (t,i), and to define the binarized coefficient as being 1 if $a_i > t+\theta$, 0 if $a_i < t-\theta$ and ***unknown*** if $|t-a_i| \leq \theta$.

In the current implementation, we have the possibility to set up this confidence parameter θ, which is used through the whole binarization procedure. If this confidence is non-zero, the notions of *discrimination* and of *discriminating power* as well as the weighting methods for the *cut points* used in the previous steps are modified as expected.

Goodness of the binarization

In most utilization of this software, the whole database available is first split into two parts (see Section *Protocols of experiments*): the ***training set*** is used for the construction of the classifier, and the ***testing set*** is used to measure the quality of the classifier. This quality depends on each stage of the analysis, and at the end of the binarization procedure it is possible to measure what will be the best result that could ever be achieved, given this binarization.

Indeed, after a binarization of the training set and the validation set according to the same rule, it might happen that an *observation* of one class in the training set is identical to an *observation* of another class in the validation set. Assuming that the classifier elaborated in the next stages classifies correctly each *observation* of the training set, we can determine a list of *observations* in the validation set that will be surely incorrectly classified. Another source of unavoidable wrong classification is due to non-coherent binarized validation set, i.e. containing identical *observations* in different classes. A procedure is available to count the total number of unavoidable wrong classifications on the validation set, assuming that the classifier commits no mistakes on the training set.

Pattern generation

The second phase of logical analysis consists in the generation of patterns. A ***pattern*** is a term covering at least one positive *observation* and none of the negative ones. In contrast with the binarization phase, the pattern generation is designed for databases with two classes only. As illustrated in . *General structure of the software*, the same pattern generation procedure is called twice: once when the *observations* of class 1 play the role of positive *observations* while those of class 2 are the negative ones, and once when these roles are reversed.

For simplicity, we will describe this procedure only for the case where positive *observations* have to be covered by patterns; the case where negative *observations* are covered is similar, when positive and negative *observations* are exchanged. Another difference between the two cases occurs when monotonicity is involved. A positive (resp. negative) Boolean variable can not appear as a negative (resp. positive) literal in a *pattern* in the first situation, while it is the other way around in the second situation.

We first generate a large set of *patterns* of small degree, then some additional *patterns* are produced to cover the positive *observations* not covered by any small *patterns*, finally different strategies are proposed to reduce the number of *patterns* while keeping the most interesting ones.

Prime patterns of small degree

The present procedure for the generation of patterns of small degree is a breadth-first-search that explores the whole set of terms up to a given maximal degree. A breadth-first-search is slower and more space consuming than a depth-first-search, but is has the advantage to yield the exhaustive list of *patterns* up to a certain degree *d*.

Beside the maximal degree of terms, several other parameters can control this generation of patterns. The minimal number of positive *observations* covered by each interesting *pattern* can be set to higher values than 1.

The satisfactory coverage of each positive *observation* can be any positive integer. Setting this parameter to a low value will allow the procedure reducing the number of positive *observations* along the way, by suppressing those that have been sufficiently covered, and this can sensitively improve the computational time. Note that this suppression of *observations* is done after the completion of the exploration of each new depth in the tree of terms. Therefore, the only *patterns* that will be omitted due to this optimization are *patterns* covering only *observations* already heavily covered by *patterns* of smaller degree.

By definition, a *pattern* is ***prime*** if none of its literals can be dropped without violating this property. Consequently, a *prime pattern* has a minimal Hamming distance of exactly 1 from the set of negative *observations*. In some occasions, it might be interesting to rely on *patterns* of higher degree but more distant from the negative *observations*. A positive integral parameter allows us

specifying this minimal distance which is 1 by default.

On the other hand, in some other cases we may want to relax the property that none of the negative *observations* is covered, since a term covering a large number of positive *observations* and just one or two negative ones may contain a lot of information about our classification problem. The parameter ξ has been introduced for that purpose with the meaning that a term covering p positive *observations* is allowed to cover

$$\xi \, p \, N^{-}/N^{+} \tag{4}$$

negative *observations*, where N^{-} and N^{+} denote the number of positive and negative *observations*.

Patterns covering specific observations

The previous procedure has the advantage to enumerate all the *prime patterns* of small degree. However, it suffers from a combinatorial explosion and if the number of *Boolean attributes* is large, this breadth-first-search can not be carried out beyond a very small degree. It may thus happen that some *observations* are covered by too few *patterns* or no *pattern* at all.

In this case, it can be desirable to find out *patterns* focusing on the coverage of each of these *observations*. Thus, the pattern generation module incorporate a second procedure, optional, for the coverage of uncovered *observations*.

Suppression of subsumed patterns

Even if all the *patterns* generated by the procedure described in the previous section are incomparable on the whole hypercube (as they are all *prime*), it might happen that the set of *observations* from the training set covered by a *pattern* P_1 is a subset of the set of *observations* covered by another *pattern* P_2. In this case, *pattern* P_2 is said to ***subsume*** P_1. An optional procedure is provided to rule out the *subsumed patterns*. However, in the present implementation, a *pattern subsumed* only by *patterns* of larger degree is not suppressed. Moreover, when two *patterns* cover the same set of *observations*, the one of larger degree is suppressed, and if the two degrees are identical, both are kept.

Theory formation

The previous stage produces two sets of *patterns*, one for the positive *observations*, and the other for the negative *observations*. In the last stage of this analysis, to each *pattern* is associated a weight, and the classifier will be represented by a combination of the two pseudo-Boolean functions corresponding to the positive and negative *observations*. However, even after the suppression of *subsumed patterns*, the set of remaining *patterns* is still quite big. For practical reasons, it was convenient to include at the beginning of this last stage (instead of at the end of the previous stage), another possibility to extract a smaller subset of interesting *patterns*.

Extraction of small subsets of patterns

The suppression of *subsumed patterns* turned out to erase a large number of *patterns* in many applications. Nevertheless, one of the main advantages of LAD versus other approaches, is that the interpretation of the results of the analysis is simple and clearly understandable for any expert in the field the classification problem comes from. To make this interpretation feasible, it is important to have a very small number of *patterns*, even if the prediction accuracy may slightly drop. For that purpose, a second facultative procedure is provided for the extraction of a small number of *patterns*. The minimal subset of *patterns* covering the same set of positive *observations* is given in a natural way by a set-covering problem. As for the binarization (see Section*Extraction of a subset of candidate attributes*), the right-hand-side (minimal coverage) can be set to any value and different heuristics are available for the resolution of this NP-Hard problem.

Patterns weighting

When any training *observation* is covered by at least one *pattern*, each of these two pseudo-Boolean functions is 0on one set of *observations* and positive on the other. Therefore, a simple way to combine the two pseudo-Boolean functions is by a ***majority vote***, i.e. for each new *observation*, the guessed class is given by the pseudo-Boolean function with higher value.

Several methods for weighting the *patterns* have been implemented in the current version. The simplest one associates a constant value to each of them. For several others, the weight is function of the number of *observations* covered by the *pattern* (linear, quadratic, cubic or exponential are available). Since small *patterns* might be more desirable than large ones, another weighting method associates a weight 2^{-d} to a *pattern* of degree d.

The next weighting method is a combination of two previous ones. In this case, it is assumed that the weight of a *pattern* should be proportional to the probability that one of the true *observations* of the *pattern* is in the list of our *observations*. Therefore, a *pattern* of degree d covering p *observations* will have a weight $p2^{-d}$.

Finally, a fifth weighting method tends to determine the weights of *patterns* in order to increase the minimal non-zero value of each pseudo-Boolean function in the set of training *observations*. Two different cases are considered. In the first one, the weights of each of the two sets of *patterns* are set independently by solving the following linear program:

$$\begin{array}{ll} \max & k \\ \text{s.t.} & \boldsymbol{A}\boldsymbol{x} \geq k \\ & \sum_q x_q = 1 \\ & x_q \geq 0\ , \end{array} \tag{5}$$

where x_q is the weight of the q^{th} *pattern* and $\boldsymbol{A}$ is a 0-1 matrix with one column per *pattern* and one row per *observation* in the class covered by the *patterns*: $a_{i,q} = 1$ if and only if the q^{th} *pattern* covers the i^{th} *observation*. By opposition, in the second case, the weights $\boldsymbol{x}$ and $\boldsymbol{y}$ for the *patterns* of the two pseudo-Boolean functions are fixed simultaneously by the solution of:

$$\begin{array}{ll} \max & k \\ \text{s.t.} & \boldsymbol{A}\boldsymbol{x} \geq k \\ & \boldsymbol{B}\boldsymbol{y} \geq k \\ & \sum_q x_q + \sum_r y_r = 1 \\ & x_q, y_r \geq 0\ , \end{array} \tag{6}$$

where $\boldsymbol{A}$ and $\boldsymbol{B}$ are two 0-1 matrices associated to the two sets of *observations* and of *patterns*.

Combination of pseudo-Boolean functions

For many applications, there is no reason to believe that a majority vote is the best combination of the two pseudo-Boolean functions f^+ and f^- (for the positive class and the negative class respectively). For example, if the sets of positive and negative *observations* are very unbalanced and so are the two sets of *patterns*, it would be reasonable to apply the majority rule after a normalization of the weights. The present version provides an option where each weight of positive *pattern* is divided by the sum of the weights of the positive *patterns* and similarly for the negative *patterns*.

Beside a normalization of the pseudo-Boolean functions, we might also consider a shift (addition of a constant value) of one of them, before applying the majority rule. The present version of the software also proposes a procedure that adjusts two parameters: α for the normalization and β for the shift: $\alpha f^+ + \beta$ will be compared to f^-. For a better result, some *observations* should be excluded from the training set for the *Pattern generation* phase, and reintroduced for the adjustment of α and β. The two parameters are presently chosen as follows. Each positive and negative *observation* $\boldsymbol{a}$ of the training set is represented by the pair $(f^+(\boldsymbol{a}), f^-(\boldsymbol{a}))$. Thus, they correspond to points in the plane, and the goal is to find the half-plane of the equation $\alpha x^+ + \beta \geq x^-$ containing as many points representing positive *observations* and as few points corresponding to negative *observations*. If the two sets of points in the plane are linearly separable, we will pick α and β from

the solutions of

$$\begin{array}{lll} \max & k & \\ \text{s.t.} & \alpha f^+(\boldsymbol{a}) + \beta - f^-(\boldsymbol{a}) \geq k & \forall \text{ positive } \textit{observation } \boldsymbol{a} \\ & \alpha f^+(\boldsymbol{a}) + \beta - f^-(\boldsymbol{a}) \leq -k & \forall \text{ negative } \textit{observation } \boldsymbol{a}. \end{array} \quad (7)$$

When the two sets of points in the plane are not linearly separable, α and β are chosen to minimize the following non-negative piece-wise linear expression:

$$\Sigma_a \left| \alpha f^+(\boldsymbol{a}) + \beta - f^-(\boldsymbol{a}) \right| c(\boldsymbol{a}, \alpha, \beta) , \quad (8)$$

where $c(\boldsymbol{a}, \alpha, \beta)$ is 1 if $\boldsymbol{a}$ is a positive (resp. negative) *observation* and $\alpha f^+(\boldsymbol{a}) + \beta - f^-(\boldsymbol{a})$ is negative (resp. positive), otherwise $c(\boldsymbol{a}, \alpha, \beta) = 0$.

PART 2

User Guide

Introduction

The current version of the executable files `bin`, `pat` and `the`, or `LAD`, enable the user to apply the complete chain of transformations and analyses of data pictured in. *General structure of the software*.

The main input of this program is a file containing the database in a format specified in Section *Input data file*. The basic output of this program is the table of results of a sequence of experiments, for which several information are reported, as well as some statistics (means and standard deviations) for each element of information. However, when a single problem is solved in a session of the program, many additional outputs are possible, providing much more details on this particular run. Each of these possible outputs will be discussed in Section *Output files*.

The next section enumerates the sequence of questions asked by the program at the beginning of each session, and describes their meanings and effects.

How to run the program

In this section, the sequence of questions asked to the user at each step of the program is detailed. This is subdivided into three subsections, one for each of the three modules `bin`, `pat` and `the`. The executable `LAD` is essentially a concatenation of the former three programs and it takes entries into a file where each parameter can be preceded on the same line by the text of the corresponding question.

As already mentioned, the program has two slightly different behaviors, according to the fact that a single problem is executed (***single-run***), or a sequence of problems are executed (***multiple-runs***). A multiple-run is characterized either by the execution of many problems for one particular size of training set, or by the experimentation of different sizes of training set in the same session. The sequence of questions varies slightly in the single-run mode or in the multiple-run mode, and this will be mentioned along the way.

Binarization

In each of the three programs, the first question allows selecting the debug mode.

```
Q1  Trace level {1=normal, 2=debug} (default 1) :
```

In fact, a third level of debug extremely verbose is also available. It is not recommended to use some information level 2 or 3 for a session with multiple experiments, since the amount of information displayed might be gigantic.

Input / output file names

All the files generated by the binarization module will have a common prefix entered at the following question.

```
Q2  Prefix for the output files :
```

The split between training and test data can either be generated at random from a common database by setting A3 to 'no' (A3 denotes the answer to Question Q3), or two data files are available as input (A3 = yes).

```
Q3  Read separate training and testing data files {y,n} :
```

Then the input file name is expected. Only its prefix must by entered in Q4 (if A3 = yes) or in Q5 (if A3= no).

```
Q4  Prefix X of the files (X.tra X.tes) with original data :
Q5  Prefix X of the file (X.all) with original data :
```

Sequencing the experiments

If A3 = yes, there will be clearly only one experiment with the given training dataset. Otherwise, the protocol of experiments (i.e. number of experiments and the way the dataset is split between training and test) has to be selected using questions Q6 to Q10.

```
Q6  Size K of the K-folding (enter 1 for resampling) :
```

For regular $N \times K$-fold cross-validation, set A6 to $K \geq 2$ and A10 to N. If A6 ≤ -2,

the protocol is a $N \times K$-fold cross validation, except that for each experiment, one fold is used as training, and the $K-1$ others are used for test. This is useful when very large dataset are available.

If A6 = 1, N-resampling cross-validation is used. In that case, questions Q7 to Q9 allows specifying the lower bound, upper bound and interval of the percentage of training data.

```
Q7  Training set's size (in %) from (default 50) :
Q8  Training set's size (in %) to :
Q9  Interval in training set's size (in %) :
Q10 #iterations of each experimentation :
```

The seed of the random generator

Fixing the seed of the random generator allows replaying an experiment in exactly the same setting. This can be done with Q11.

```
Q11 Seed:
```

However, when many experiments are iterated in the same run for cross-validation purposes, it may happen that only one particular experiment has to be replayed. Therefore, in this program, the seed of the random generator is used in two different ways, depending whether there is only one or more than one experiment. In the first case, i.e. when

$$\text{A3} = \text{yes} \text{ or } (\text{A6} = 1 \text{ and } \text{A10} = 1 \text{ and } \text{A7} + \text{A9} > \text{A8}), \qquad (9)$$

the seed is fixed to A11 before any call to the random generator. On the other hand, when *equation (9)* does not hold, there is say $M > 1$ experiments ($M = NK$ or $M = N \cdot \text{floor}((\text{A8} - \text{A7}) / \text{A9})$). In this case, the seed is fixed to A11 and then M random numbers are drawn and stored in a table. At the beginning of the m^{th} experiment, $m = 1,\ldots M$, the seed is fixed to the m^{th} element before any call to the random generator. Moreover, these seeds are printed in the log file. Thus, if one particular experiment has to be replayed, it suffices to get from the log file the seed effectively used for the experiment and to rerun the program requiring a single experiment and specifying this seed to A11.

Steps of the binarization method

As far as continuous attributes are concerned, the binarization method can be based either on binary-discrimination, or on continuous-discrimination. Q12 allows choosing among these two possibilities.

```
Q12 Binarization method {1=binary, 2=continuous} (default 2) :
```

The complexity of the heuristic used to solve the set-covering problem is linear in the number of pairs of different classes. It is possible to reduce this list of pairs by the simple following rule. If ***a*** and ***b*** are two *observations* from different classes, and if there is a point ***e*** included in the hyper-box delimited by ***a*** and ***b***, then the separation of (***a***, ***b***) will be at least as good as the separation either of the pair (***a***, ***e***) or of the pair (***b***, ***e***). Thus, the pair (***a***, ***b***) can be dropped from the list of pairs to be separated.

```
Q13 Apply point-in-a-box to reduce the # of pairs of pts {y,n} :
```

In practice, it turned out that for some databases, this technique allows the suppression of up to 40% of the rows, while for others very few rows are suppressed. Since this operation is quite costly, especially when the number of attributes is large, it is worse doing some preliminary experiments on each new database in order to decide whether this optimization is worth it or not.

The parameter θ discussed in Section *Binarization and confidence interval* is set in question Q14. To have a unique θ for all continuous attributes, these are considered as normalized such that their minimum and maximum on the training set are 0 and 1. Therefore, θ is usually very small, typically around 0.01. In some databases, the ideal value for this parameter was around 0.008, while in others, a confidence interval up to 0.05 seemed more adequate.

```
Q14 Confidence interval around each cut point [0 , 0.1] :
```

Question Q15 allows the choice of the method for generating the candidate *cut points* : A15 =0 corresponds to one-cut-per-change, while A15 = 1 indicates one-cut-per-pair.

```
Q15 Cut points generation method {0=each change, 1=each pair} :
```

The user should be aware that the second method generates in general much more pairs and thus it is recommended to sort the *candidate attributes* and keep only the first ones, before extracting a minimal subset. This is feasible through the questions Q17 and Q18, when answering yes to Q16.

```
Q16 Filter cut points according to a specific order {y,n} :
Q17 Ordering method {1=entropy, 2=min-discr, 3=total-discr} :
```

of A12 = 1

```
Q18 Minimal # of CA separating each pair of pts (filter) :
```

else if A12 = 2

```
Q19 Minimal separability of each pair of pts (filter) :
```

The ordering methods ordering-by-entropy, ordering-by-minimal discrimination and ordering-by-total-discrimination, discussed in Section*Sorting and pre-selection of the candidate attributes*, are selected through Q17. Question Q18 or Q19 allows determining the amount of *candidate attributes* kept according to this order. When some filter is used, the *candidate attributes* are ordered according to the ordering criterion specified, and then, the first *k* are selected and the others are suppressed, where *k* is the minimal number so that the *k* first candidates are sufficient to achieve the required global separability. If this global separability is too high, this requirement is readjusted to the maximal global separability (when all the cut-points are present) and this modification of requirement is notified in the log file.

The last group of questions Q20 to Q23 concerns the final extraction of a small subset of *candidate attributes* (see Section *Extraction of a subset of candidate attributes*).

```
Q20 Minimize # of cut points {y,n} :
```

if A12 = 1,

```
Q21 Minimal # of CA separating each pair of pts (optim) :
```

else if A12 = 2,

```
Q22 Minimal separability of each pair of points (optim) :
```

Again, if the required minimal separability cannot be achieved it is readjusted and this is noticed in the log file. For the sack of efficiency of the pattern generation process, it may be important to bound the number of *candidate attributes* finally produced. This is possible with Q23.

```
Q23 Maximal number of cut points {0=unbounded} :
```

However, the user must be aware that a too small bound introduced in Q23 may result into a set of *candidate attributes* which does not fulfil the criterion specified in Q21 or Q22.

Pattern generation

Input-output file names and sub-sampling

The first questions have the same purpose than those in the `bin` module.

```
Q24 Trace level {1=normal, 2=debug} :
Q25 Prefix for the output files :
Q26 Prefix X of the files (X.tra) containing the training data :
Q27 Training set's size (in %) from :
Q28 Training set's size (in %) to :
Q29 Interval in training set's size (in %) :
Q30 #iterations of each experimentation :
```

Note that the files resulting from experiments with $N \times K$-fold cross-validation are named the same way as those of N- resampling. For example, if a 4-fold was used in the binarization module, the names will be similar than if 75% of the data was used as training. To use these with the `pat` module, just answer 75% and 75% to Q27 and Q28.

In case one would like to run (or rerun) the pattern generation module on a single problem out of many that have been binarized. Say that this problem is the 6^{th} of the ones with 66% training, answer 6 to Q31.

```
Q31 Index of the single iteration to do :
```

The seed of the random generator works in the same way than in `bin`.

```
Q32 Seed :
```

As discussed in *this Paragraph* of Section *Combination of pseudo-Boolean functions*, it is sometimes desirable to sub-sample the training set for the pattern generation module, in order to keep some unseen data for the theory formation module. For this purpose, A33 should be set to less than 100%.

```
Q33 Percentage of training sample used for pattern generation :
```

Depth-first-search

In the current implementation, there is no procedure for the depth-first-search generation of patterns, so Q34 should be answered negatively and Q36 to Q39 will not be asked.

```
Q34 Generate patterns by depth-first-search {y,n} :
Q35 Satisfactory coverage of each positive point in DFS :
```

```
Q36 Satisfactory coverage of each negative point in DFS :
Q37 Literal evaluation method for positive patterns :
Q38 Literal evaluation method for negative patterns :
```

Breadth-first-search

The main module for pattern generation proceeds by a breadth-first-search.

```
Q39 Generate patterns by Breadth-First-Search {y,n} :
```

It consists (when A39 is yes) into two consecutive calls to the same function, once with the positive and negative *observations* taken as such, and another time when their roles are reversed. This is why every parameter is doubled. The first one concerns the maximal depth (i.e. degree of the terms) of the breadth-first-search exploration.

```
Q40 Generate positive patterns of degree up to :
Q41 Generate negative patterns of degree up to :
```

To avoid the generation of too many *patterns*, it is often desirable to focus on *patterns* covering sufficiently many *observations*.

```
Q42 Minimal coverage of each positive pattern {neg number -> %} :
Q43 Minimal coverage of each negative pattern {neg number -> %} :
```

When A42 (resp. A43) is negative, the given value is considered as a percentage of the total number of positive (resp. negative) *observations* to be covered by positive (resp. negative) *patterns*. For example, if there are 40 positive *observations*, answering −5 or +2 to A42 is equivalent and implies that only positive *patterns* covering at least 2 positive *observations* will be considered.

The processing time of the breadth-first-search procedure depends on the number of positive and negative *observations*. If this number can be reduced on the way, the processing time can decrease significantly. When some positive *observations* have already been covered by many *patterns*, they can safely be suppressed from the list. The next parameter to be entered at Q44 and Q45 provides the threshold coverage value for a point to be suppressed from the list. If this value is 10, for example, it does not mean that every positive point will be covered by 10 *patterns*, but that whenever a point is covered by 10 *patterns*, we do not consider it any more for the generation of further *patterns*. In practice, this suppression of widely covered *patterns* is done only after the completion of the exploration of each new depth of the search.

```
Q44 Satisfactory coverage of each positive point :
Q45 Satisfactory coverage of each negative point :
```

The purpose of Questions Q46 and Q47 is to get the minimal distance from a term to the set of negative *observations*, so that this term is considered as *pattern* (see Section *Prime patterns of small degree*, *Paragraph*).

```
Q46 Minimal distance from a positive pattern to an opposite point :
Q47 Minimal distance from a negative pattern to an opposite point :
```

For a *prime pattern*, this distance is 1. It can however be increased to 2 (or more, but the experience has shown that this parameter is very sensitive), meaning that only *patterns* at distance at least 2 from any negative point are considered.

The next questions are related to the relaxation of the concept of *patterns*, allowing some conjunctions covering many positive *observations* and very few negative ones to be also considered as *patterns* (see Section *Prime patterns of small degree*, *Paragraph*). The parameter ξ in *equation (4)* is entered as A48 and A49.

```
Q48 * A conjunction covering C+ (resp. C-) points among the N+ (N-)
     total positive (negative) points
    is a positive pattern if (C-/C+)(N+/N-) is at most :
Q49 is a negative pattern if (C+/C-)(N-/N+) is at most :
```

Patching

The next two questions allow choosing whether a second pattern generation procedure must be activated in order to cover the *observations* uncovered by the *patterns* generated so far.

```
Q50 Generate extra patterns to cover uncovered pos. points {y,n} :
Q51 Generate extra patterns to cover uncovered neg. points {y,n} :
```

Cleaning the sets of patterns

Finally, at the end of the `pat` module, the user has the choice to reduce the potentially large set of *patterns* generated by suppressing the *subsumed patterns* (Section *Suppression of subsumed patterns*), before the *patterns* found are stored on files.

```
Q52 Suppress subsumed patterns {y,n} :
```

Theory formation

Input-output file names

The first questions have the same purpose than those in the `bin` and the `pat` modules (see Section*Input-output file names and sub-sampling*).

```
Q53 Trace level {1=normal, 2=debug} :
Q54 Prefix for the output files :
Q55 Testing theory(ies) on test data {y,n} :
Q56 Prefix X of the files (X.tra) with the training data :
Q57 Prefix X of the files (X.pos, X.neg) with the patterns :
Q58 Training set's size (in %) from :
Q59 Training set's size (in %) to :
Q60 Interval in training set's size (in %) :
Q61 #iterations of each experimentation :
Q62 Index of the single iteration to do :
Q63 Seed : 12345
```

Weighting the patterns

Before associating weights to the *patterns*, one still have the option to extract a subset of them chosen so that each point is covered by at least A64 *patterns* (see Section*Extraction of small subsets of patterns*).

```
Q64 Extract a subset of patterns with minimal point coverage of
    {0 = keep all patterns} :
```

If some *observations* are covered by less *patterns* (when all *patterns* are considered) than the specified number, all the *patterns* covering these *observations* are necessarily placed in the subset and this fact is mentioned in the log file.

The selection of some of the weighting techniques discussed in Section *Patterns weighting* is done through Q65

```
Q65 Weighting method (0>cst, 1>Cov, 2>Cov/FSize, 3>FSize, 6>Cov^2,
    7>Cov^3, 8>1.2^Cov:
```

where `Cov` stands for coverage (number of *observations* covered) and `Fsize` is proportional to the size of the face of the hypercube represented by the *pattern* (`Fsize` = 2^d for a *pattern* of degree d). Methods 6, 7 and 8 correspond to weight growing respectively as a quadratic, a cubic or an exponential (basis 1.2) function of the coverage. The last two methods discussed in Section *Patterns weighting* are not yet implemented.

As mentioned at the beginning of Section *Combination of pseudo-Boolean functions*, it is often interesting to balance the total contribution of positive and of negative *patterns*. This is the purpose of Q66. If Q66 = yes,the weights associated to the *patterns* according to the chosen method are normalized, so that the sum of the weights of negative *patterns* is equal to the sum of the weights of positive *patterns* and is equal to 1.

```
Q66 Normalize weights so that sum of neg = sum of pos = 1 {y,n} :
```

A finer normalization as well as a shift of the threshold for the final decision is obtained by learning the two parameters α and β described in Section *Combination of pseudo-Boolean functions*.

```
Q67 Readjust threshold and proportion between pos/neg {y,n} :
```

In the evaluation of a classification system, it is often interesting to distinguish between a wrong answer and no answer. Using the sign of the pseudo-Boolean function $f^+ - f^-$ (or $\alpha f^+ + \beta - f^-$) for the final decision, whenever the result of this function is close to 0, it is wise not to take a decision. The parameter ε entered as A68 means that whenever the result of the decision function is between $-\varepsilon$ and ε, the answer of the classifier is “I don’t know”.

```
Q68 Half size of the range around threshold leading to unknown :
```

In the output statistics of the `the` module, the rates of errors and of unknowns are first distinguished and then, in the total error rates, all the unknowns are counted as errors.

Input and output files

Input data file

The formalism used to describe the syntax is the EBNF, which is as follows:

MetaSymbol	*Meaning*
→	is defined to be
(X)	1 instance X
[X]	0 or 1 instance X
{X}	0 or more instance X
X Y	X followed by Y
X | Y	Either X or Y
x	Non-terminal symbol
`x`	Terminal symbol

Formal description

In what follows, `EOF`, `EOL`, `TAB` and `SPACE` represent the end-of-file, end-of-line, tabular and space respectively. The input data file must fulfil the following syntax.

InputDataFile → *HeaderOrInclude Data* `EOF`

HeaderOrInclude → (*Header* | *Include*)

Include → `include` *FileName* `EOL`

FileName is sequence of characters satisfying the file name's syntax. There must exist a file with this name containing a *Header* .

Header →
[*Identifier* `EOL`]
Attribute { `;` { *Comment* } { `EOL` } *Attribute* } `.`
{ *Comment* } { `EOL` }

Comment → `//` { any character except `EOL` } `EOL`

Attribute → *Identifier* `:` *AttributeDescr*

Identifier →
(`A` | ... | `Z` | `a` | ... | `z`)
{ any character except `.` `,` `:` `;` `(` `)` `/` `SPACE` `TAB` `EOL` }

AttributeDescr → (*RegularAttribute* | *SpecialAttribute*)

RegularAttribute → (*NonOrderedAttribute* | *OrderedAttribute*)

NonOrderedAttribute → *Identifier* `,` *Identifier* `,` *Identifier* {`,` *Identifier* } [`(target)`]

OrderedAttribute →
(`continuous` | (*Identifier* `,` *Identifier*))
[*Monotonicity* | `(target)`]

Monotonicity → `(+)` | `(-)`

SpecialAttribute → (`multiplicity` | `label` | `ignored`)

Data → *OneDatum* { *DataSeparator OneDatum* }

OneDatum → (*Numerical* | *Identifier* | `?`)

DataSeparator → { `SPACE` } (`SPACE` | `TAB` | `EOL` | `,` | `;`) { `SPACE` }

Simple example of input data

This syntax is illsutrate through *Example 1*.

```
Mushrooms

name: label;
toxicity: eatable, poisonous (target);
density: continuous;
pH:  continuous (+); // means that if pH increases,
                     // toxicity cannot decrease
cap-color: n, b, c, g, r, p, u, e, w, y;
bruises: yes, no; // note that here, yes=0 and no=1!
veil:  absent, present (-).

lepiote                 eatable         2.352     7.4    3      0      1
chanterelle             0               4.01      6.7    2      1      0
amanite-panthere        poisonous       3.5       6.2    3      1      1
```

Example 1

Constraints and semantic

The *Header* (which can be in a separate file, using **`include`**) contains a description of each attribute of the dataset. The total number of *OneDatum* in *Data* must be a multiple of the number of *Attribute* in the *Header*.

As discussed in Section *Characteristic of input data*, *nominal attributes* are either *nonOrderedAttribute*s or two-valued *orderedAattribute*s. In the data, the values of a *nominal attribute* can be given either by their names or in a numerical form. In the latter case, the order will be the one of the list of values in the description of the attribute, starting at 0.

One *regularAttribute* must be specified as **`target`**. If more than one *Attribute* is specified as target, the first one will be the effective target.

Whenever an *orderedAttribute* is the target, other *orderedAttribute*s can have *monotonicity constraints*. Monotonicity constraints will be ignored when the target is a *nonOrderedAttribute*.

The **`label`** attribute is used to give a name to each data. After some preprocessing, it may occur that some data correspond to several original data. This information is very important, especially when counting the coverage of the *patterns*. The attribute **`multiplicity`** is used on this purpose. If there is more than one label (resp. multiplicity) attribute, the first one will be considered as the effective label (resp. multiplicity) and the other label (resp. multiplicity) attributes will be ignored. The data corresponding to a label attribute can be either a *Numerical* value or an *Identifier*. The data corresponding to a multiplicity attribute must be *Numerical*. If there is no label attribute, then each data is labeled by its order in the file (starting with 1). If there is no multiplicity attribute, then each multiplicity is set to 1. If one value of the multiplicity (resp. the label) attribute is set to “unknown” (i.e. **`?`**), then the multiplicity is arbitrarily set to 1 (resp. the label is set to the character “?”).

Output files

Outputs of the binarization

The binarization module take as input a file with the dataset in the form described in Section *Formal description*. It generates several files named

(*Prefix* **-bin.** *Suffix0* | *Prefix* **-** *Perc* **-** *Iter* **.** *Suffix1* | *Prefix* **.** *Suffix2*)

Files of the last form are generated only in case of a *single run*, i.e. when the number of iterations is 1.

Prefix → any sequence of alphanumeric (given as parameter)

Suffix0 → (**out** | **log** | **tmp**)

Perc → one 2 digits number (except for 100) specifying the percentage of the whole data used for training

Iter → one 2 digits number, giving the iteration (when an experiment with the same percentage is repeated several times)

Suffix1 → (**tra** | **tes**)

Suffix2 → **b_a**

Example of output of the binarization

A single run of `bin` on the database 'Heart Disease' of the Irvine repository, with 50% data for training will produce the following files, when the given prefix is HD:

```
HD-bin.out
HD-bin.log
HD-bin.tmp
HD-50%001.tra
HD-50%001.tes
HD.b_a
```

Example 2

The file with suffix **out** contains all the statistical results of the binarization. The file with suffix **log** is the log file and contains information related to problems occurred during the binarization as well as the seeds used at the beginning of each experiment (useful to rerun one particular experiment). The file with suffix **tmp** is a temporary file. It is used to follow the progress of the binarization procedure, or in case the program is interrupted, partial results are stored in this file. The files with suffix **tra** and **tes** contain the training and testing data in the binary form and according to the syntax described in Section *Formal description*.

The file with suffix **b_a** is created only if the number of iterations is 1. It contains the list of *Boolean attributes* and thus is useful to associate each *Boolean attribute* to the original attributes. For example, the file presented in is produced

by the previous run of `bin`.

```
total_nb_of_original_attributes 15
nb_of_cut_points 21
v 1: s= 47.00  1: 54.5  2: 55.5  3: 56.5
v 2: s= 1.00  4: 0.5
v 3: s= 3.00  5: 1.5  6: 2.5
v 4: s= 80.00  7: 133.0
v 5: s=251.00  8: 242.0  9: 243.5 10: 255.5 11: 280.0
v 6: s= 1.00 12: 0.5
v 8: s= 1.00 13: 0.5
v 9: s=131.00 14: 154.5 15: 170.5
v10: s= 1.00 16: 0.5
v11: s= 44.00 17: 10.5
v12: s= 2.00 18: 1.5
v13: s= 4.00 19: 0.5 20: 1.5
v14: s= 2.00 21: 0.5
```

Example 3

The first two lines recall the total number of original and *Boolean attributes*. Then, every original attribute associated to at least one *Boolean attribute* is listed. Each original attribute start with a new line and they are indexed **v1**, **v2**, etc. (starting from 1). After this index and a column, **s=***N* indicates the 'span' used for this original attribute, which was just the max value minus the min value found on the training set, but this is for internal use and can be ignored at a macro level. Then, the *Boolean attributes* associated to the original attribute are listed, with their index (starting from 1), a column and the value of the *cut point*.

Outputs of the pattern generation

The pattern generation module uses essentially only the files

Prefix **-** *Perc* **-** *Iter* **.tra**

containing the binarized training data. It creates the files

(*Prefix* **-pat.** *Suffix0* | *Prefix* **-** *Perc* **-** *Iter* **.** *Suffix1*)

Prefix → any sequence of alphanumeric, given as parameter,

Suffix0 → (**out** | **log**)

Perc → one 2 digits number (except for 100) specifying the percentage of the whole data used for training

Iter → one 2 digits number, giving the iteration (when an experiment with the same percentage is repeated several times)

Suffix1 → (**pos** | **neg**)

Example of outputs of the pattern generation module

A single run of `pat` on data 'Heart Disease' with 50% data for training will produce the files listed in *Example 4* when the given prefix is HD:

```
HD-pat.out
HD-log.log
HD-50%001.pos
HD-50%001.neg
```

Example 4

The file with suffix **out** contains all the statistical results of the pattern genera-

tion. The file with suffix **log** is the log file and contains information related to problems occurred during the pattern generation. The files with suffix **pos** and **neg** contain the lists of positive and negative *patterns*. An example of such a file is presented in *Example 5*:

```
total_nb_of_attributes = 21
nb_of_patterns = 14
max_degree = 6
c: 25  | 16 21
c: 10  | 16 20
c: 3  | 1 -2
c: 25  | 1 -6 16
c: 23  | -6 16 19
c: 22  | 9 16 -18
c: 3  | -5  8 17
c: 1  | 14 -4 -18 -21
c: 9  | -7 -13 -14 -18 -20
c: 6  | 19 -4 -9 -12 -21
c: 3  | 11 -4 -13 -15 -21
c: 2  | 18 -4 -14 -17 -21
c: 1  | 5 11 13 18 -19
c: 4  | -4 -5 -10 -15 -18 -20
```

Example 5

The first three lines recall the total number of *binary attributes*, the total number of *patterns* as well as the degree of the longest *pattern*. Then each *pattern* is listed on one line according to the syntax OnePattern:

OnePattern → **c:** *Coverage* [**w:** *Weight*] **|** *Literal* { *Literal* } **EOL**

Coverage is an integer representing the number of *observations* in the training data covered by this *pattern.* Weight is a the weight of the *pattern* given as a real number. If this is not present, all the *patterns* are supposed to be of the same weight 1.0. Literal specifies one literal of the *pattern* and is given as an integer whose absolute value is the index (starting from 1) of the *binary attribute* and whose sign specifies whether the literal occurs as such or negated. In the above example, the third *pattern*

```
c: 3  | 1 -2
```

is the Boolean conjunction (x_1 and not(x_2)) and covers three *observations* in the training data.

Outputs of the theory formation

The third module uses the four files

Prefix **-** *Perc* **-** *Iter* (**.tra** | **.tes** | **.pos** | **.neg**)

Based on the training data, it eventually prunes the lists of positive and negative *patterns*, then it associates weights to each remaining *patterns* and finally, it tests the obtained theory on the testing dataset.

The files generated by the theory formation module are the following

(*Prefix* **-the.** *Suffix0* | *Prefix* **.** *Suffix1*)

Files of the last form are generated only in case of a single run, i.e. when the number of iterations is 1.

Prefix = any sequence of alphanumeric, given as parameter

Suffix0 = (**out** | **log**)

Suffix1 = (**pat** | **tr** | **te** | **tre** | **tee**)

A single run of bin on data 'Heart Disease' with 50% data for training produces the following files when the given prefix is HD:

```
HD-the.out
HD-the.log
HD.tr
HD.te
HD.tre
HD.tee
HD.pat
```

Example 6

The file with suffix **out** contains all the statistical results of the performances of the theory. The file with suffix **log** is the log file and contains information related to problems occurred during the theory formation. The files with suffix **tr** and **te** contain information related to the performances of the theory on the training and testing data and their format is illustrated in *Example 7*.

```
1 137 1 0.14694 0.13845 0.00849
1 179 1 0.38571 0.00712 0.37859
1 270 1 0.00000 0.04747 -0.04747
1 185 1 0.05714 0.10680 -0.04966
1 102 1 0.28367 0.01820 0.26548
0 42 2 0.17722 0.00408 0.17313
0 287 1 0.04035 0.16939 -0.12904
0 111 1 0.13687 0.08980 0.04707
0 178 1 0.00000 0.00000 0.00000
0 246 1 0.47389 0.00000 0.47389
```

Example 7

Results related to each data is on one line. The first number is the class (0=false, 1=true). The second number is the label identifying the data point. The third number is the multiplicity. The next two numbers are the results of the pseudo-Boolean functions f^+ and f^- discussed in Section *Combination of pseudo-Boolean functions*. If the *observation* is of class 1, the forth column is f^+ and the fifth is f^-, the order is reversed if the point belongs to class 0. The last column is the difference of the previous two. If this last value is positive, then the point is correctly classified, if it is negative, is it wrongly classified, and if it is 0 or very close to 0, then it is not classified.

The files with suffix **tre** and **tee** give more details about the errors. *Example 8* illustrates the information that can be found in the file for each misclassified

observation.

```
point 301, from class 1 0.01633 0.17484 -0.15852
Positive firing patterns
c: 6 w: 0.012  | 19 -4 -9 -12 -21
c: 2 w: 0.004  | 18 -4 -14 -17 -21
Negative firing patterns
c: 26 w: 0.021  | -5 -11 -21
c: 22 w: 0.017  | -1 -5 -21
c: 22 w: 0.017  | -2 -5 -21
c: 22 w: 0.017  | -3 -5 -21
c: 17 w: 0.013  | -1 -5 18
c: 17 w: 0.013  | -2 -5 18
c: 17 w: 0.013  | -3 -5 18
c: 15 w: 0.012  | -5 -7 -21
c: 11 w: 0.009  | -5 13 -21
c:  9 w: 0.007  | -1 -5 13
c:  9 w: 0.007  | -3 -5 13
c:  9 w: 0.007  | -2 -5 13
c:  7 w: 0.006  | -5 19 -21
c:  3 w: 0.002  | -1 18 20
c:  3 w: 0.002  | -3 -17 20
c:  3 w: 0.002  | -3 18 20
c:  3 w: 0.002  | -2 -17 20
c:  3 w: 0.002  | -2 18 20
c:  3 w: 0.002  | -1 -17 20
```

Example 8

The first line recalls information about the observation: label and class followed by the result of the pseudo-Boolean functions f^+ and f^- (or f^- and f^+) and the difference of these two values. Then all positive and negative *pattern* covering this observation are listed according to the same syntax as in the files with extensions **pos** and **neg** (see *Example 5* in Section *Outputs of the pattern generation*).

Finally, the file with suffix **pat** gives a information of the behavior of the theory detailed by *patterns* instead of by *observations in a form illustrated by Example 9*.

```
          Training data  |                Test data  |  positive patterns

 +/+ +/? +/- -/- -/? -/+  |   +/+ +/? +/- -/- -/? -/+  |
  69   0   0  82   0   0  |    55   0  14  58   4  21  |  <-- total

  25   0   0   0   0   0  |    27   0   1   1   0   5  |  c: 25   w: 0.051      |   16    21
  10   0   0   0   0   0  |    14   0   0   0   0   0  |  c: 10   w: 0.020      |   16    20
   3   0   0   0   0   0  |     2   0   0   3   0   0  |  c:  3   w: 0.006      |    1    -2
  25   0   0   0   0   0  |    25   0   0   0   0   7  |  c: 25   w: 0.051      |    1    -6    16
  23   0   0   0   0   0  |    25   0   0   0   0   2  |  c: 23   w: 0.047      |   -6    16    19
  22   0   0   0   0   0  |    17   0   0   0   0   1  |  c: 22   w: 0.045      |    9    16   -18
  19   0   0   0   0   0  |    13   0   0   0   0   0  |  c: 19   w: 0.039      |    8    16    17
...
          Training data  |                Test data  |  negative patterns
 +/+ +/? +/- -/- -/? -/+  |   +/+ +/? +/- -/- -/? -/+  |
  69   0   0  82   0   0  |    55   0  14  58   4  21  |  <-- total

   0   0   0  23   0   0  |     2   0   0  15   0   0  |  c: 23   w: 0.018      |   -1     4
   0   0   0   3   0   0  |     1   0   1   1   0   2  |  c:  3   w: 0.002      |    6    16
   0   0   0   3   0   0  |     1   0   0   2   0   1  |  c:  3   w: 0.002      |   12    15
   0   0   0   2   0   0  |     1   0   1   2   0   0  |  c:  2   w: 0.002      |   15    16
   0   0   0  35   0   0  |     0   0   3  31   0   0  |  c: 35   w: 0.028      |   -1    14   -21
   0   0   0  35   0   0  |     0   0   3  34   0   0  |  c: 35   w: 0.028      |   -2    14   -21
   0   0   0  35   0   0  |     0   0   3  36   0   0  |  c: 35   w: 0.028      |   -3    14   -21
...
```

Example 9

The file is split into two parts, one for the positive *patterns* and the other for the negative *patterns*. At the beginning of each part, a header gives the legends of

each columns as well as one special row denoted as "total". The set of columns is split into three parts, one for training data, one for testing data and one specifying the *pattern* according to the same syntax as in Section *Outputs of the pattern generation*, *Example 5*, for the files with suffixes **pos** and **neg**. The first two parts are made of 6 columns of integers. These columns are labeled T/E, where T is the target output '+' or '–' and E is the effective output '+', '–' or '?' (in case of no classification). The value in column T/R and in the row 'total' gives the total number of *observations* of the (training/testing) dataset, of class T and classified as E. The value in column T/R and a row corresponding to *pattern* P gives the number of *observations* of the (training/testing) dataset, of class T, classified as E and for which the *pattern* P is firing.

Bibliography

[1] Bjarne STROUSTRUP, *The C++ Programming Language*, Third Edition, Addison-Wesley, 1997.