SMILES v.2.3

A Multi-purpose Learning System*

Technical Report 5-Sep-2002

C. Ferri-Ramírez J. Hernández-Orallo M.J. Ramírez-Quintana

Dep. Sistemes Informàtics i Computació, Univ. Politècnica de València (Spain)
{cferri, jorallo, mramirez}@dsic.upv.es

Abstract

In this paper we describe SMILES, a Stunning Multi-purpose Integrated LEarning System. A machine learning system is useful for extracting models from data and hence it can be used for many applications such as data analysis, decision support or data mining. SMILES is a machine learning system that integrates many different features from other machine learning techniques and paradigms and, more importantly, it presents several innovations in almost all of these features. This report contains four major parts: a description about the system architecture, a user's manual, a more advanced section on how to take the most of the system and, finally, some brief programmer's guidelines. A complete table of all the options provided in the system is also included.

Keywords: Decision tree learning, machine learning, data mining, hypothesis combination, ensemble learning, cost-sensitive learning, ROC analysis.

* This work has been partially supported by CICYT under grant TIC2001-2705-C03-01 and by Generalitat Valenciana under grant GV00-092-14. This system was initiated during two research stays in the Department of Computer Science of the University of Bristol. The stays were granted by Universitat Politècnica de València and by Generalitat Valenciana.

Index

1 Introduction

SMILES (Stunning Multi-purpose Integrated LEarning System) is a machine learning system that integrates many different features from other machine learning techniques and paradigms and presents several innovations in almost all of these features. In particular, it extends classical decision tree learners in many ways (new splitting criteria, non-greedy search, new partitions, extraction of several and different solutions), it has an anytime handling of resources, and has a sophisticated and quite effective handling of costs. In this way, SMILES combines and improves the recent interest in hypotheses combination (e.g. boosting[54]) and cost-sensitive learning (a priori and a posteriori class assignments [11], ROC analysis [48]) outperforming previous systems in many situations.

The origin of SMILES dates back to our previous system FLIP (Functional Logic Inductive Programmer) developed from 1998 until 2001 [32][19][20] with the goal of inducing declarative models from evidence in the form of functional logic programming. The paradigm was called Inductive Functional Logic Programming (IFLP) [31]. As a successful extension of ILP techniques to functional logic programs, it inherited some of the limitations of ILP systems: poor scalability and poor accuracy results with respect to other more general machine learning algorithms.

With the goal of broadening the applicability of machine learning for functional logic programming and other declarative paradigms, we endeavoured the construction of a new system based on a more scalable, flexible and efficient basis, with the premise of generating highly expressive and comprehensible models from data. Although at the present moment the models induced are just a slight extension of classical decision trees, they can still be represented as a restrictive kind of functional logic programs. The next stage that must logically follow this work is the inclusion of more partitions in order to make SMILES more expressive, capable of using background knowledge and capable of generating full functional logic programs with possible higher-order features. It is this long-term goal that shapes the continuity between our previous system FLIP and SMILES (in fact, previous versions of SMILES were called CDTL or FLIP 2.0/3.0) and not the implementation, because, as we will see, they differ drastically in architecture and learning techniques.

This paper tries to present the system in its present form, serving as a short technical description from the machine learning point of view and also as a user's and programmer's guide, which can accompany the source in case that someone wants to use or extend the system.

The paper is then organised as follows. In section 2 the structure of the system is described in a practical and succinct way. SMILES user's manual is presented in section 3, describing separately the general inputs/outputs of the system, the general options, the multitree features, the different ways to extract several solutions, the cost-sensitive and ROC features, the archetype procedure and other facilities. In section 4 we include several results and plenty of know-how about how to take the most from SMILES. Section 5 describes how the system is implemented and how the source code is structured, resembling a very brief programmer's manual. Due to the large number of options that SMILES has, section 6 includes a table with all the system options. Finally, section 7 presents possible future work.

2 The Structure of the System

In this section we briefly present the main algorithm and underlying methods of SMILES. For more details on the structure, we refer to [18][21][22].

Decision tree learning is a very popular kind of machine learning technique. In a decision tree, each node contains a test on an attribute, each branch from a node represents a possible outcome of the test, and each leaf contains a class prediction. A decision tree is usually induced by recursively replacing leaves by test nodes, starting at the root. Classic decision-tree learners such as CART [6], ID3 [49], C4.5 [l] or FOIL [51] have given very good results and are currently used in many applications; however, they do not have flexibility with respect to trading result quality for computational resources.

The main algorithm of SMILES is also based on a greedy search in the decision tree space, such as CART, ID3 or C4.5. However, SMILES is able to obtain more than one solution, looking for the best one or combining them in order to improve the overall accuracy or minimise the classification cost. To do this, once a node has been selected to be split (an AND node) the other possible splits at this point (OR nodes) are suspended until a new solution is required. In this way, the search space is an AND / OR tree [44][42] which is traversed producing an increasing number of solutions for increasing provided time. Since each new solution is built following the construction of a complete tree, our method differs from other approaches such as the boosting method [28][54], which induces a new decision tree for each solution. The result is a multitree rather than a forest; with the advantage that a multitree shares the common parts and the forest does not. We perform a greedy search for each solution, but once the first solution is found the following ones can be obtained taking into consideration a limited computation time. Therefore, our algorithm can be considered *anytime* in a certain way [8].

Apart from the multitree (AND-OR) structure, our system extends the representation language by extending the possible partitions. The final goal, as we have stated in the introduction, is to induce functional logic programs with even higher-order characteristics. In the Functional Logic Programming (FLP) paradigm, conditional programs are sets of rules and, hence, they can also be represented as trees. This allows us to include the type information of the function profile in the split criterion.

The types handled by SMILES are:

Type	***Use***
Nominal	For value sets, Booleans and any non-numerical attribute.
Numerical	For integers, real numbers and any numerical attribute. Ordered nominal attributes such as {low, medium, high} are not directly handled and should be substituted by integers.

At the present moment, our system allows the following partitions, two of them using negation:

Partition	
Partition X=a, X=b, X=c...	Present in ID3, C4.5
Partition X=a, X≠a	
Partition X=Y, X≠Y	
Partition X<c, X≥c	Present in ID3, C4.5

Note that the search of the decision tree space requires the use of several criteria. If the search can be re-activated to explore further solutions then more criteria are needed. The main criteria used by SMILES are:

- Splitting Criterion: among all the possible partitions (split) which one is selected to open the tree.

- Traversal Criterion: from all the given nodes after a split, which nodes are explored first (the rest are suspended).
- Suspended Nodes Forgetting: Should all the suspended nodes retained or some of them can be forgotten?
- Second Tree Opening Criterion: specifies which node to explore for second-best solutions.
- Best Tree Selection Criterion: specifies which of all of these solutions must be shown if the user wants just the best solution.
- kBest Different Solutions Selection criterion: specifies how to select from the AND/OR tree a set of solutions. If k is equal to 1, then it just selects one solution according to the BestTree Selection Criterion.

One important feature of the system is that many of the previous criteria can optionally be defined in terms of the Minimum Description Length (MDL) principle [56]. If all of these criteria are MDL-based then the decision tree is built in a short-to-long way. The MDL principle has previously been used in the induction of decision trees in the post-pruning phase [41] [55]. Also, the MDL principle has been used as a stopping criterion (*pre-pruning*) [45][51], as a measure for globally evaluating discretisations of continuous attributes [46], and for restructuring decision trees [1]. In our approach, the MDL principle is used at the generation phase which is justified because other quality criteria based on discrimination such as the *information gain* [52], the *information gain ratio* [52] or the *Gini heuristic* [6] are not useful for functions that have a recursive definition or that use concepts of the background knowledge. This was one of our premises and, although these functions cannot be learnt with the current version, future versions will hopefully be able to. Another reason is that the guidance of the search by the MDL principle ensures a better use of computational resources following a Levin search [38][57]. We can use the MDL principle as split criterion, as stopping criterion, as pre-pruning criterion and also as solution tree selection criterion. Finally, we derive a measure of confidence for combining multiple solutions. In this way, we can use a uniform framework based on the same measure for constructing the tree, selecting the split, selecting second-best trees to explore and selecting or combining hypotheses.

3 User's Manual

SMILES source-code can be downloaded from http://www.dsic.upv.es/~flip/smiles/. The package includes the C++ sources and some sample datasets. Once the software downloaded and decompressed, follow the *readme* file and run the shell-script for installation on Unix-like machines.

If the installation is successful, you can directly type:

```
./smiles -?
```

and you will have the following usage information:

```
**** SMILES v.2.3.1 (Release Date: 23-August-2002) ****

USAGE:
./smiles file.train [file.test] [file.cost] [file.testcost]
```

That means that the software has been correctly installed.

3.1 SMILES Usage Basics

The current SMILES version is a command-line batch application with little interaction during the learning stage. Almost all interaction is performed through the inputs (mainly the training set, the data set and the options) and the outputs (mainly solution trees and statistics).

The previous usage information suggests that the system must be supplied with some files, at least a training set.

We can run the system by using some of the examples that are provided with the distribution, e.g., the playtennis example. If we type:

```
./smiles samples/playt.train samples/playt.test
```

Using the simplest options (no combination) in the option file (we will discuss on this) the result may be something similar to this:

```
**** SMILES v.2.3.1 (Release Date: 23-August-2002) ****

ftrain: samples/playt.train
ftest: samples/playt.test
fcost:
ftestcost:

Training Set: "samples/playt.train"

No. of Attributes: 4
Cardinality: 14 examples
Class: 0 ("yes"). Distribution: 9
Class: 1 ("no"). Distribution: 5

Valid options.

Creating the multitree.
Learning begins...

The test set: "samples/playt.test" will be used to evaluate the results

Predicting and preparing statistics

Filling new test probabilities of the leaves of the multi-tree with the
Test set
  15 examples done.

Showing below the properties of the best single tree

SOLUTION 0: 5 rules

Statistics over test set of length: 15:
Relative Accuracy: 1
AUC Hand: 1

END OF RESULTS
```

```
Test dataset destroyed successfully
The multitree has been destroyed successfully

====> Time used (for learning): 0.01 secs.

**** Smooth end of SMILES execution ****
```

If we take a look at the results we observe several parts. A first part informs us that the files have been read and tells us about their characteristics: number of attributes of the dataset, number of the examples and class distribution, in the previous example 4 attributes and 14 examples. Finally, it includes statistics on the tree

Let us use a different example: “house-votes”:

```
./smiles samples/house-votes.train samples/house-votes.test
```

If we use an option file that uses combination, the output varies slightly:

```
**** SMILES v.2.3.1 (Release Date: 23-August-2002) ****

ftrain: samples/house-votes.train
ftest: samples/house-votes.test
fcost:
ftestcost:

Training Set: "samples/house-votes.train"

No. of Attributes: 16
Cardinality: 217 examples
Class: 0 ("democrat"). Distribution: 136
Class: 1 ("repub"). Distribution: 81


Valid options.

Creating the multitree.
Learning begins...
 Last Opened Node #1000 learned of 1000
MeanDepth of Second Solution Start Positions: 4.36837


The test set: "samples/house-votes.test" will be used to evaluate the
results

Predicting and preparing statistics

Showing first the results of the combination method of all branches:

COMBINATION RESULTS:
Statistics over test set of length: 218:
Relative Accuracy: 0.963303
AUC (example by example) in Hand & Till's way: 0.990568
```

```
Filling new test probabilities of the leaves of the multi-tree with the
Test set
 218 examples done.

FIRST SOLUTION: (solution as if 1 tree were generated).
  Note: if postpruning is enabled this solution is not the same as with 1
tree.
Num. of Rules: 21

Statistics over test set of length: 218:
Relative Accuracy: 0.949541
AUC Hand: 0.980477


Showing below the properties of the best single tree

SOLUTION 0: 21 rules

Statistics over test set of length: 218:
Relative Accuracy: 0.940367
AUC Hand: 0.976968

END OF RESULTS

Test dataset destroyed successfully
The multitree has been destroyed successfully

====> Time used (for learning): 10.49 secs.

**** Smooth end of SMILES execution ****
```

Now, learning begins exploring not only the first splits but many others (# of opened OR-nodes, in the previous example 1000), in order to obtain a good result. After the learning process, some statistics of the combination all the trees are shown (0.963 accuracy and 0.991 AUC), statistics of the first solution as if the multitree would have only 1 solution (21 rules, 0.950 accuracy and 0.980 AUC) and, finally, a solution which is extracted using a criterion from the multitree (21 rules, 0.940 accuracy and 0.977 AUC).

3.2 Simple Input and Output Format

Let us explain first the format of the training set file. The lines that begin with ! are meant to be directives to the parser. Only the "!TYPES:" directive is mandatory. After several lines of directives the dataset starts with the values of each argument separated by commas. Each example must be put in a different line. The last value of each line is the class of the example and must not be followed by comma.

The meaning of the directives is given in the following table:

Directive	Syntax	Mandatory
!TYPES:	Natural numbers: for nominal types, each number ≥ 1 denotes a different type. For numerical types, a 0 must be used, (0u or OU if missing values are to be taken into account).	Yes
!NAMES:	Argument names separated by commas. Only used for visualising the solutions.	No
!WEIGHTS:	Set of pairs "classname = weight" where weight is a real number. Tries to give more relevance to some classes over others. This will be further explained along with the cost-sensitive options.	No
!RECLASSIFY	Set of pairs "originalclassname>changedclassname;". The semicolon is mandatory even for the last assignment. This options is to change the name of one or more class. This is used for joining two or more classes.	No

Let us illustrate this syntax with an example (playt.train):

```
!TYPES:1,2,3,4,5
!NAMES:sky,temp,humid,wind,play
!WEIGHTS:yes=1,no=2
overcast,hot,high,weak,yes
rain,mild,high,weak,yes
rain,cool,normal,weak,yes
sunny,mild,normal,strong,yes
overcast,mild,high,strong,yes
overcast,hot,normal,weak,yes
sunny,hot,high,weak,no
sunny,hot,high,strong,no
rain,cool,normal,strong,no
sunny,mild,high,weak,no
overcast,cool,normal,strong,yes
sunny,cool,normal,weak,yes
rain,mild,normal,weak,yes
rain,mild,high,strong,no
```

This is a dataset with two classes, where all the attributes are nominal and different (this is why all the integers of the directive !TYPES are different). This training set gives more relevance to class "no" (2) than to class "yes" (1) by using the directive "!WEIGHTS".

Another example could be like this.

```
!TYPES:1,2,2,0,3,0,4
!NAMES:sex,fatherstudlevel,motherstudlevel,IQ,region,fameconlevel,studlev
el
!RECLASSIFY:elementary>no-uni;secondary>no-uni;highschool>no-uni;
university>uni;doctor>uni;
female,secondary,highschool,120,Valencia,20000,doctor
male,university,university,130,Madrid,50000,highschool
female,secondary,elementary,150,Berlin,30000,doctor
male,elementary,elementary,90,Bretagne,40000,secondary
female,secondary,elementary,105,Calabria,15000,university
...
```

In this case we have four nominal and two numerical attributes. Two attributes ("fatherstudlevel" and "motherstudlevel") are of the same type. The class originally had five possible values from the datasets, but the !RECLASSIFY directive has joined the first three into a new class "no-uni" and the remainder two into a new class "uni". This means that this

problem will be treated by SMILES as a two classes problem. Note that when the !RECLASSIFY option is used, the type for the class should not be used for any other attribute (using type 2 for the class in this case could produce some errors).

We can ignore some attributes, either nominal or numerical, (if we think they are not useful for learning) when instead of the type, an 'I' or 'i' is put in the !TYPES directive.

Nominal missing values are represented by "?" and are simply considered as an additional value for an attribute. Consequently, tree partitions may have "?" in some of the branches.

For numerical missing values (which must also be represented by "?"), there are three possible options for handling them: ignore any example that contains numerical missing values (by using the value "0" in the !TYPES directive), treat them as a special value (by using the value "0U" or "0u" in the !TYPES directive) or substitute missing numeric values by a 0. We will see how to change this option later.

For instance::

```
!TYPES:1,2,2,0,I,0U,4
!NAMES:sex,fatherstudlevel,motherstudlevel,IQ,region,fameconlevel,studlev
el
female,secondary,highschool,120,Valencia,20000,doctor
male,university,university,?,Madrid,50000,highschool
female,secondary,elementary,150,Berlin,?,doctor
male,elementary,elementary,?,Bretagne,?,secondary
female,?,elementary,105,Calabria,15000,university
...
```

Now we have some examples with missing numerical and nominal values. We also see that the "region" attribute is ignored and consequently not used for learning. Supposing we have set in the options file that missing values should be discarded and not substituted by zero (we will go back on this), the first example does not contain missing values; the second example is discarded as a missing value appears for an attribute (IQ) that does not accept it; the third example is taken into account because the missing value appears for an attribute (famconlevel) that accepts it; the fourth is discarded as a missing value appears for an attribute (IQ) that does not accept it; the fifth is taken into account and the second attribute (fatherstudlevel) would have an additional value "?".

Numerical partitions with missing values are "trios" of the form (X < a, X >= a, X = ?).

Note: if two nominal arguments are set to be of the same type, it must be ensured that both of them have missing values or none of them. Otherwise, the types would not be identical and there may be an error.

The test set has the same format as the training set, although directives are ignored. If the number or type of the attributes in the test set is different from the training set, an error is produced and SMILES exits. Moreover, it should be noted that in the current version **if the test set presents for a *nominal* type an attribute value that was not present in the training set, an error is produced and SMILES exits**. A very easy way to avoid this is to add a line like the following one in the training set:

```
?,?,?,?,?
```

which adds an *unknown* attribute value "?" for all types. Then the test dataset should be modified in order to substitute any new value by a ?.

3.3 Options File

Before, we saw that the system options are not given through the command line. In order to specify the system options, a configuration system is used, called "**options.cfg**".

In this file, comment lines begin with '%'. The rest are pairs "option=value", where spaces should not be placed around the symbol "=". Most of the options are not mandatory but the file "options.cfg" must be placed in the same directory SMILES is.

The syntax of the options file is illustrated by the following excerpt:

```
%--Expected error method: ways to compute expected error
expected error method=no compute
%
%--frequency error smoothing:use smoothing or not
smoothing method=no smoothing
```

As we have said before, lines that begin with '%' are considered comments and are ignored. The options file is then constituted by assignments, where the left hand side is the option name and the right hand side is the value given to the option. Both the option name and the option value side have a strict syntax and no spaces can be inserted on either side of the '=' sign. A complete list of the available options is given in section 6.

Let us make a change in the options file. If we open it with an editor and modify the "show all k-best solutions" option as follows:

```
%--show all k-best solutions
show all k-best solutions=show
```

and now, following with the playtennis example, we run SMILES again with the same training set and test set, we have that now the output includes one solution in form of rules:

```
**** SMILES v.2.3.1 (Release Date: 23-August-2002) ****

...

Let us show the solution: 0
f(X0, X1, X2, X3) = R :- X0=overcast. [class: yes]
f(X0, X1, X2, X3) = R :- X0=rain, X3=weak. [class: yes]
f(X0, X1, X2, X3) = R :- X0=rain, X3=strong. [class: no]
f(X0, X1, X2, X3) = R :- X0=sunny, X2=high. [class: no]
f(X0, X1, X2, X3) = R :- X0=sunny, X2=normal. [class: yes]

...

**** Smooth end of SMILES execution ****
```

This shows the solution in the form of conditional rules of the type *Head :- Body*.

3.4 General Options

Given the basics of the system, now we are going to describe some other main features and options. All of them are described in Section 6.

One of the first things to choose in a decision tree learning algorithm is the splitting criterion. Different splitting criteria have been implemented: Left first, Gain (entropy), Gain_Ratio and C4.5 [53], CART [6], MDL, DKM [33], Expected_Error and Area Under the ROC Curve [24].

In the options file, the user can choose among all of these:

```
%--splitting criterion: criterion which is used to select the best split
%splitting criterion=left first
%splitting criterion=gain
%splitting criterion=gain ratio
%splitting criterion=c4.5
%splitting criterion=cart
%splitting criterion=mgini
%splitting criterion=desc mdl
%splitting criterion=dkm
%splitting criterion=split expected error
%splitting criterion=local roc area
%splitting criterion=one point local roc area
splitting criterion=mse
%splitting criterion=logloss
%splitting criterion=sqdiff
%splitting criterion=genentropy
%splitting criterion=rocv
%splitting criterion=auch
%splitting criterion=aucs
```

Their description is as follows:

- left first: just chooses the first split.
- gain: Quinlan's information gain [49].
- gain ratio: Quinlan's information gain ratio. [52][55]
- c4.5: same as gain ratio but splits with gain lower than the mean are discarded [52].
- cart: simple implementation of GINI criterion of CART system [6].
- mgini: true implementation of GINI criterion [6].
- desc mdl: a criterion based on MDL (see [18] for more information).
- dkm: An optimisation of mgini [33].
- split expected error: The split with lowest expected error.
- local roc area: the split with greatest area under the ROC curve. Only valid for 2 classes.
- one point local roc area: simplification of "local ROC area". Just computes the are with one point.
- mse: minimum squared errror. It will be explained in section 3.14.
- logloss: It will be explained in section 3.14.
- sqdiff: computed as the square of the difference between probability for class a and probability for class b. Only valid for two classes.
- genentropy: gain and gini can be seen as special cases of a generalised entropy function depending on a power. This is a parametrised split criterion where this exponent can be modified (by program).
- rocv: another (not very successful) extension to the roc split for more than 2 classes.
- auch: Based on Hand and Till's extension of Area Under the ROC Curve for more than two classes [30]. It will be further explained in section 3.14.
- aucs: Fawcett's variant of AUCH.

The best results are usually obtained with the mse, auch and c4.5 criteria. CART only works well for two classes, and MGINI should be used instead. MGINI works well for two classes, so never use CART.

Another important feature to be considered in a decision tree learning algorithm is pruning. Currently, SMILES includes pre-pruning methods and post-pruning. The available pre-pruning criteria can be chosen in the options file as follows:

```
%--pre-pruning method: criterion for pruning when constructing the tree
pre-pruning method=no pruning
%pre-pruning method=proportional
%pre-pruning method=expected error pruning
%pre-pruning method=MDL pruning
%pre-pruning method=stump pruning
%pre-pruning method=pep pruning
%
%--only for stump pruning. Depth Limit.
%--If stump pruning is not active, this option is ignored
stump pruning limit=3
```

It is advisable to use some kind of pruning when some noise is expected in the data. Many of these criteria have some parameters although currently only the stump pruning (depth limit) is modifiable through the options file, whereas the rest of them can be modified in the source code. The effectiveness on accuracy of these criteria is very variable. The MDL pruning criterion can only be used if the MDL splitting criterion has also been selected.

Post-pruning is less efficient (because the entire tree is populated before pruning) but it is easier to use and more effective. Currently, the only post-pruning method implemented is the "Pessimistic Error Pruning" (PEP) introduced by [50] (Quinlan 1987). According to the study in [12], this is the best method that does not modify the tree structure (unlike C4.5 pruning). Although it has a tendency to underprune, we think that it is a quite simple and effective method. The way to activate it

```
%--post-pruning method: criterion for pruning after constructing the tree
(not used for combination)
%post-pruning method=no pruning
post-pruning method=pep pruning
```

It is possible to combine pre-pruning and post-pruning. It could be good results to have a very strict pre-pruning criterion (only pruning very clear cases) and then post-pruning. We have not, though, performed any experiment about this.

Additionally, numerical splits can also be particularised. The problem with numerical splits arises when a dataset has a great number of different values for an attribute. This happens very often with continuous values, such as real numbers. A dataset of thousands of examples and a single numerical attribute could generate a partition with thousands of children. Given n different values, after ordering them, we would have a middle value a_i and a condition ($x<a_i$) for each interval generated by two consecutive different values. This could slow down the system dramatically. For this reason there are some methods for reducing the number of intervals (also called discretisation methods): in a logarithmic way, with a maximum or with no limit. These options can be selected in the options file:

```
%--numeric interval criterion: how numerical attributes are handled
numeric interval criterion=log
%numeric interval criterion=max
%numeric interval criterion=no limit
```

Some of these options have parameters. Currently, these parameters cannot be specified through the options file and must be changed in the source code through program re-compilation.

Finally, the options that correspond to the handling of missing numeric values are:

```
%-- how to handle missing numeric values
missing numeric values=ignore examples with missing numeric values
%missing numeric values=substitute missing numeric values
```

The first one ignores any example that contains a value '?' for a numerical attribute. The second option substitutes them by 0s. Both options are not applicable if we use the types "0U" or "0u" in the "!TYPES" directive of the training set.

Until now, we have described the options that are common to many decision tree algorithms. In the next sections we describe more distinct and advanced features of SMILES.

3.5 Multitree Options

As we have discussed in the introduction, one way to overcome the greedy behaviour of traditional decision tree learning is to explore other splits. Once these alternative trees (OR trees) are grown, then the best tree can be selected from all the open branches or a combined solution of all of them could be built.

The following figure shows a multitree for the playtennis example with a partial exploration of the entire search space:

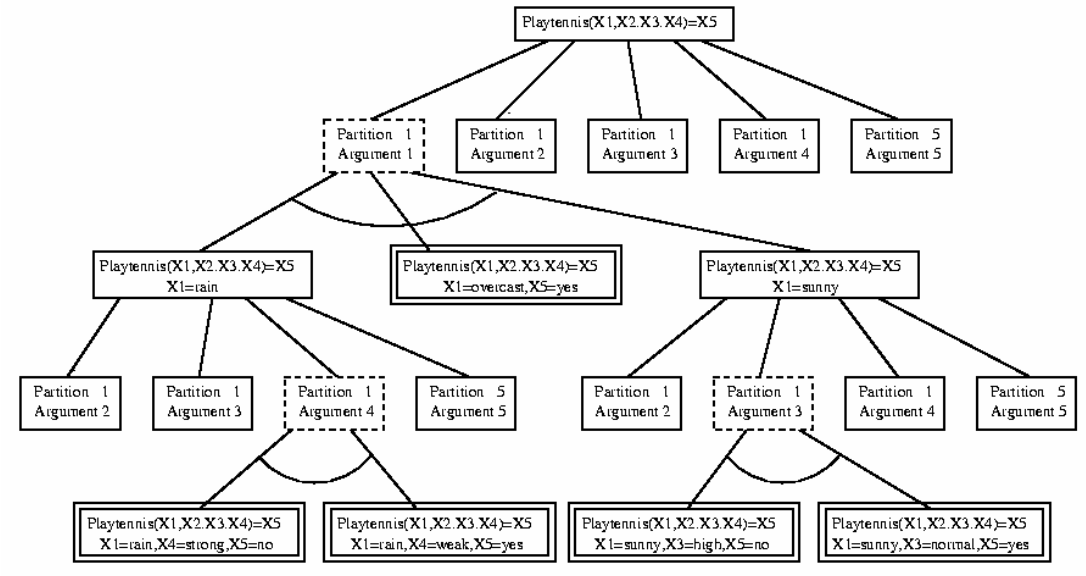


Figure 1: Partial AND/OR tree for the playtennis example

The first thing to specify is the number of trees in the multi-tree that are going to be explored. More precisely, the number correspond to how many different OR-nodes will be explored, which can give many more solutions (all their possible combinations). This is done in the options file as follows:

```
%--multitree: construction: how many or-nodes are opened in the multitree
multitree number of trees=1000
```

Additionally to the number of trees, the way in which second solutions are found has to be also specified. For instance, the following lines would select that the second solution would be the still unexplored node that is *topmost*.

```
%--second tree opening:how to select the 2ond node to explore
%second tree opening=split optimality
%second tree opening=optimality rival ratio
%second tree opening=optimality rival ratio depth
%second tree opening=optimality rival ratio component
%second tree opening=optimality rival ratio component random
second tree opening=second topmost
%second tree opening=second bottommost
%second tree opening=second random
%second tree opening=second random depth
```

These options are more deeply explained in section 6. It is advisable to use topmost to improve accuracy, although it is the one that requires more resources. In case of resource limitations, a good option might be random.

The number of trees (or-nodes) to be explored and how to select them determines the resulting multitree. Now, with all these open branches two things can be done: select a subset of solutions or combine them.

If we decide to select one solution, then we have to choose a selection criterion to specify which of all the possible solutions must be shown. To do this, the options file includes the following alternatives:

```
%--multitree: best tree selection criterion
multitree best tree criterion=occam best
%multitree best tree criterion=test cost best
%multitree best tree criterion=occam and test cost best
%multitree best tree criterion=coverage best
%multitree best tree criterion=cross coverage
%multitree best tree criterion=expected error best
%multitree best tree criterion=split optimality best
%multitree best tree criterion=mdl best
```

For instance, “occam best” selects the shortest solution possible from the multitree. Others such “coverage best”, “expected error best” and “mdl best” choose the tree that gets best results with these measures. These options are explained in sections 3.15 and 6. Later on we will see another method for obtaining a single solution from the set (ensemble) of solutions.

Taking into account alternative solutions provides better solutions than the first eager, greedy, single solution. This behaviour may require a lot of memory to store all the alternatives (used or not). In fact, most of the alternate trees (the OR nodes) that are suspended never are woken and, consequently, are never explored. A way to do a more efficient use of resources, can be based on not opening all the OR nodes or, seen in other way, to forget part of the suspended nodes. Currently, SMILES presents “suspended nodes forgetting” methods:

```
%--suspended nodes forgetting: must all suspended nodes maintained?
suspended nodes forgetting=maintain all
%suspended nodes forgetting=maintain const random
%suspended nodes forgetting=maintain log random
%suspended nodes forgetting=maintain log random with depth
%suspended nodes forgetting=maintain log random with squared depth
%suspended nodes forgetting=maintain log random with depth adjusted
```

The meaning of these options are:

- "maintain all": as before. Everything is kept in memory.
- "maintain const random": a constant number of children (randomly selected) is maintained.
- "maintain log random": only a log (number of children) are maintained.
- "maintain log random with depth": only a log(number of children/(depth+1)) are maintained.
- "maintain log random with depth adjusted": as "maintain log random with depth" but a constant is added to the number of children. This constant is hardwired in the program.
- "maintain log random with squared depth": similar to the preceding ones, but the formula is now: log(number of children/sqrt(depth+1)).

With these methods, memory and time requirements for the multi-tree can be significantly improved. A compromise between memory and quality of results is obtained through the method "maintain log random with depth adjusted". As we have said, the method "maintain all" just maintains all the suspended nodes and it is the recommended option when there are not memory restrictions.

Finally, although most of the methods maintain a logarithm proportion, in some cases the number of remaining nodes must be too low. In order to avoid this, a minimum number of nodes that must be preserved can be specified. This can be done through the following option.

```
%--suspended nodes maintain const value
suspended nodes maintain const value=2
```

According to the experimental results shown in Section 4, LOG WITH DEPTH is the most economical option, both in memory and in time, and results are not significantly deteriorated. If time is not a problem (and just memory), LOG WITH DEPTH ADJUSTED is also a good option because it even increases accuracy in some cases.

3.6 Combination and Fusion Options

The other way to use the multiplicity in the multi-tree is to combine the results of different branches. There are also several methods to combine the different solutions that can be given in the multi-tree, and they can be selected using the option file.

```
%-- Combination: How to combine several solutions
%multitree solution combination=no combination
%multitree solution combination=cross coverage combination
%multitree solution combination=majority crisp
multitree solution combination=majority absolute stochastic
%multitree solution combination=majority relative stochastic
%multitree solution combination=majority cost stochastic
```

As we will discuss later, the "majority absolute stochastic" method gives the better results and can be combined with other more sophisticated options.

In particular, a decision tree learner can be seen as a soft classifier, taking the proportion of examples that have fallen in each leaf as the probability that a new examples would be assigned to that class.

When we have multiple solutions the question is how these vectors can be "fusioned" in order to give a mixture or ensemble classifier. In our case the fusion is made whenever two or more OR-nodes are found and their predictions are weighted.

At each leaf node, we have a cardinality vector that tells how many examples of the training set have fallen for each class. The first thing that we can decide is to work with the absolute vector or with the relative vector. For this we have the following options:

```
%-- Combination vector: absolute (n. of examples) or relative (frequency)
combination vector=absolute
%combination vector=relative
```

Let us see an example. As we have said, the estimated probabilities assigned to each node depend on the proportion of training examples of each class that have fallen into each node during the training of the decision tree learning. The reliability of each node usually depends on the cardinality of the node. Consider three classes *a*, *b* and *c* and two nodes n1 and n2 with the following train distribution:

n1= { 40, 10, 30 }

n2= { 0, 2, 1 }

If we convert this absolute values to relative values, we would have:

n1= { 0.5, 0.125, 0.375 }

n2= { 0, 0.667, 0.333 }

The prediction of class *a* by n_1 seems less reliable than prediction of class *b* by n_2. However, node n1 has 40 examples supporting the prediction whereas n2 has only 2 examples. Consequently, in this case it seems that the absolute values provide more information.

In either case, in some situations it may be convenient to perform a Laplace smoothing of these vectors. This can be enabled through the following option:

```
%-- Combination smoothing. Use smoothing before combination
%combination smoothing=true
combination smoothing=false
```

The vectors can be left as they are originally or they can be modified in several ways, so affecting the resulting fusion.

```
%-- Combination vector method: how to derive the vector
combination vector method=original
%combination vector method=good loser
%combination vector method=bad loser
%combination vector method=difference
%combination vector method=majority
%combination vector method=squared
```

The exact definition of these transformations is as follows. Consider a vector of values (either absolute or relative) that each classifier assigns for each class and example, $v_{k,,j}(x)$. The transformations are:

original: $v'_{k,j}(x) = v_{k,j}(x)$

difference: $v'_{k,j}(x) = v_{k,j}(x) - \sum_{i \neq j} v_{k,j}(x)$

good loser: $v'_{k,j}(x) = \sum_{j} v_{k,j}(x)$ if $j = \arg\max\left(v_{k,j}(x)\right)$ and 0 otherwise.

bad loser: $v'_{k,j}(x) = v_{k,j}(x)$ if $j = \arg\max\left(v_{k,j}(x)\right)$ and 0 otherwise.

majority: $v'_{k,j}(x) = 1$ if $j = \arg\max\left(v_{k,j}(x)\right)$ and 0 otherwise.

squared: $v'_{k,j}(x) = \left[v_{k,j}(x)\right]^2$

Let us illustrate the previous transformation with an example. Consider four classifiers and three classes with the following values:

$v_{1,,j}(x)$= { 40, 10, 30 }
$v_{2,,j}(x)$= { 7, 2, 10 }
$v_{3,,j}(x)$= { 0, 10, 1 }
$v_{4,,j}(x)$= { 5, 6, 3 }

These transformations would convert the initial vectors into these:

Original:

$v_{1,,j}(x)$= { 40, 10, 30 }
$v_{2,,j}(x)$= { 7, 2, 10 }
$v_{3,,j}(x)$= { 0, 10, 1 }
$v_{4,,j}(x)$= { 5, 6, 3 }

Difference:

$v_{1,,j}(x)$= { 0, -60, -30 }
$v_{2,,j}(x)$= { -5, -15, 1 }
$v_{3,,j}(x)$= { -11, 9, -9 }
$v_{4,,j}(x)$= { -4, -2, -8 }

Good loser:

$v_{1,,j}(x)$= { 80, 0, 0 }
$v_{2,,j}(x)$= { 0, 0, 19 }
$v_{3,,j}(x)$= { 0, 11, 0 }
$v_{4,,j}(x)$= { 0, 14, 0 }

Bad loser:

$v_{1,,j}(x)$= { 40, 0, 0 }
$v_{2,,j}(x)$= { 0, 0, 10 }
$v_{3,,j}(x)$= { 0, 10, 0 }
$v_{4,,j}(x)$= { 0, 6, 0 }

Majority:

$v_{1,,j}(x)$= { 1, 0, 0 }
$v_{2,,j}(x)$= { 0, 0, 1 }
$v_{3,,j}(x)$= { 0, 1, 0 }
$v_{4,,j}(x)$= { 0, 1, 0 }

Squared:

$v_{1,,j}(x)$= { 1600, 100, 900 }
$v_{2,,j}(x)$= { 49, 4, 100 }
$v_{3,,j}(x)$= { 0, 100, 1 }
$v_{4,,j}(x)$= { 25, 36, 9 }

Finally, with these vectors and each time two or more OR-nodes exist at the same level the following fusion methods can be chosen:

```
%-- Combination fusion method: how to combine the vectors
```

```
%combination fusion method=sum
%combination fusion method=prod
%combination fusion method=arithmean
%combination fusion method=geomean
combination fusion method=max
%combination fusion method=min
%combination fusion method=median
```

The meaning of these fusion strategies can be explained by the following definitions of fusion strategies that convert the K classifiers vectors into one combined vector $\Omega_j(x)$:

- sum: $\Omega_j(x) = \sum_k v_{k,j}(x)$
- arithmetic mean: $\Omega_j(x) = \sum_k v_{k,j}(x) / K$
- product: $\Omega_j(x) = \prod_k v_{k,j}(x)$
- geometric mean: $\Omega_j(x) = \sqrt[K]{\prod_k v_{k,j}(x)}$
- maximum: $\Omega_j(x) = \max_k \left(v_{k,j}(x)\right)$
- minimum: $\Omega_j(x) = \min_k \left(v_{k,j}(x)\right)$
- median: $\Omega_j(x) = \underset{k}{median}\left(v_{k,j}(x)\right)$

An important remark to mull over in shared ensembles is that the fusion points may be different. For instance, different predictions can be compared at the bottom of the tree where or-nodes appear whereas this can also happen at the top of the multi-tree. Consequently, a great difference in combined prediction may come out between the sum and the average methods, for instance.

Let us illustrate the application of these fusion methods with the previous example. The four original vectors were:

$v_{1,,j}(x)$= { 40, 10, 30 }
$v_{2,,j}(x)$= { 7, 2, 10 }
$v_{3,,j}(x)$= { 0, 10, 1 }
$v_{4,,j}(x)$= { 5, 6, 3 }

Supposing they are at the same OR level, then we have the following fusioned vectors:

sum:	{ 52, 28, 44 }	$\rightarrow a$
arithmean:	{ 13, 7, 11 }	$\rightarrow a$
product:	{ 0, 1200, 900 }	$\rightarrow b$
geomean:	{ 0, 5.89, 5.48 }	$\rightarrow b$
maximum:	{ 40, 10, 30 }	$\rightarrow a$
minimum:	{ 0, 2, 1 }	$\rightarrow b$
median:	{ 6, 8, 6.5 }	$\rightarrow b$

On the right we see which would be the class predicted if we would compute the maximum value of the resulting fusion vector, which is made at the top of the tree.

We have seen transformation + fusion. Some combination give better results than other or correspond to classical combination politics. For instance, “majority + sum” would give the

typical majority selection. According to the results in [13][14][15], the best method is “absolute + no smoothing + original + max”, although this may depend on the examples.

Another thing to take into account is that because of the very nature of combination (combination avoids overfitting and hence improves accuracy and other quality measures) , pruning is not beneficial. Consequently, pruning is not recommended to be used for combination.

For this reason, since sometimes we may want post-pruning for the best single solution but not for the combination, there exists another option that disables post-pruning for combination despite it can be enabled for single solutions. This is obtained through the following option:

```
%-- Allow post-pruning in combination (if post-prune enabled)
%post-pruning in combination=yes
post-pruning in combination=no
```

Unless the option for combination is “no combination”, SMILES always gives statistics for the combination and statistics for the best single solution. In this way, SMILES tries to give the usually better accuracy results of the combination method jointly with the intelligible results of a single best solution.

3.7 Showing Several Solutions

Up to now we have always produced one best solution from the multi-tree (apart from the combination results that do not provide a comprehensible solution). However, there are some situations where it may be interesting to generate more than one solution. For this, we can determine the number of solutions output in the options file:

```
%--k best: how many solutions are generated in the multitree
k best number of solutions=3
```

The interest in obtaining more than one solution is maintained only if the solutions are different between them. In order to ensure that the solutions are quite different and still are good, SMILES has several “k-best” methods.

```
%--select k best method
%select k best=k-best less visited
%select k best=k-best less visited plus
select k best=k-best less visited then different components
%select k best=k-best different components
%select k best=k-best random
```

The previous methods are based on two main ideas: to avoid the use of the same branches that have been output for other solution (this is possible if several splits have been opened for an OR node), and secondly to avoid the use of the same partitions and attributes (herewith called components) in several solutions. In other words, these two ideas try to avoid repeated parts of solutions. For the second idea, a component matrix is used.

A component matrix describes in a very concise way which components take part in a decision tree. For instance, if we have 3 possible partitions and 4 attributes then a component matrix could be like this:

	At0	***At1***	***At2***	***At3***
Part1	1.5	0	0.5	0.5
Part2	0	0	0	0
Part3	0	0	0	0

A component matrix ignores the values of the conditions. For instance, the previous matrix could represent the following tree:

```
f(X0, X1, X2, X3) = R :- X0=overcast.
f(X0, X1, X2, X3) = R :- X0=rain, X3=weak.
f(X0, X1, X2, X3) = R :- X0=rain, X3=strong.
f(X0, X1, X2, X3) = R :- X0=sunny, X2=high.
f(X0, X1, X2, X3) = R :- X0=sunny, X2=normal.
```

The values are obtained depending on the times an attribute is used and the depth at which it is used. For instance, 1.5 corresponds to 3/2 because the split with X0 has three children at depth 1 (this gives the value 2^1 in the denominator). The value 0.5 corresponds to 2/4 because the split with X3 is used in two splits and at depth 2 (this gives the value 2^2 in the denominator).

SMILES allows the user to see the component matrix of each solution, just by changing the corresponding option in the options file:

```
%--show component matrix of solutions
%show solutions components=no show
show solutions components=show
```

As we have said the component matrix is used for obtaining different solutions. For this reason, a *reference* component matrix has to be updated in order to reflect what kind of partitions and attributes have to be avoided. There are three different ways of how to generate this component matrix between several solutions:

```
%--select k best components generation
select k best components generation=component accumulate
%select k best components generation=component random generated from start
%select k best components generation=component random generated from second
```

More detailed explanation on how this works can be shown in [27].

Let us show an example of how SMILES produces several solutions. If we use the playtennis dataset , 100 open or-nodes in the multi-tree, the “component random generated from second options” and “3 k-best solutions”, we obtain the following output:

```
**** SMILES v.2.3.1 (Release Date: 23-August-2002) ****

ftrain: samples/playt.train
ftest: samples/playt.test
fcost:
ftestcost:

Training Set: "samples/playt.train"

No. of Attributes: 4
Cardinality: 14 examples
Class: 0. Distribution: 9
Class: 1. Distribution: 5

Valid options.

Creating the multitree.
Learning begins...
Tree #100 learned of 100
```

```
MeanDepth of Second Solution Start Positions: 1.77778

The test set: "samples/playt.test" will be used to evaluate the results

Predicting and preparing statistics

Showing first the results of the combination method of all branches:

COMBINATION RESULTS:
Statistics over test set of length: 15:
Relative Accuracy: 1
AUC (example by example) in Hand & Till's way: 1


Filling new test probabilities of the leaves of the multi-tree with the
Test set
 15 examples done.

Let us select the 3 best trees.
From 3 seeked, 3 solutions have been found.
Showing their properties

SOLUTION 0: 5 rules
Component Matrix:
P\R    C 0|  C 1|  C 2|  C 3|
  C 0  1.5|    0|  0.5|  0.5|
  C 1    0|    0|    0|    0|
  C 2    0|    0|    0|    0|

Statistics over test set of length: 15:

Relative Accuracy: 1
AUC (example by example) in Hand & Till's way: 1

Let us show the solution: 0
f(X0, X1, X2, X3) = R :- X0=overcast. [class: yes]
f(X0, X1, X2, X3) = R :- X0=rain, X3=weak. [class: yes]
f(X0, X1, X2, X3) = R :- X0=rain, X3=strong. [class: no]
f(X0, X1, X2, X3) = R :- X0=sunny, X2=high. [class: no]
f(X0, X1, X2, X3) = R :- X0=sunny, X2=normal. [class: yes]

SOLUTION 1: 10 rules
Component Matrix:
P\R    C 0|  C 1|  C 2|  C 3|
  C 0 1.75| 0.75|    0| 1.75|
  C 1    0|    0|    0|    0|
  C 2    0|    0|    0|    0|

Statistics over test set of length: 15:
```

```
Relative Accuracy: 1
AUC (example by example) in Hand & Till's way: 1
Let us show the solution: 1
f(X0, X1, X2, X3) = R :- X0=overcast, X3=weak. [class: yes]
f(X0, X1, X2, X3) = R :- X0=rain, X3=weak. [class: yes]
f(X0, X1, X2, X3) = R :- X0=sunny, X1=hot, X3=weak. [class: no]
f(X0, X1, X2, X3) = R :- X0=sunny, X1=mild, X3=weak. [class: no]
f(X0, X1, X2, X3) = R :- X0=sunny, X1=cool, X3=weak. [class: yes]
f(X0, X1, X2, X3) = R :- X0=overcast, X3=strong. [class: yes]
f(X0, X1, X2, X3) = R :- X0=rain, X3=strong. [class: no]
f(X0, X1, X2, X3) = R :- X0=sunny, X1=hot, X3=strong. [class: no]
f(X0, X1, X2, X3) = R :- X0=sunny, X1=mild, X3=strong. [class: yes]
f(X0, X1, X2, X3) = R :- X0=sunny, X1=cool, X3=strong. [class: yes]


SOLUTION 2: 8 rules
Component Matrix:
P\R    C 0|  C 1|  C 2|  C 3|
  C 01.125|    0|1.375|0.875|
  C 1    0|    0|    0|    0|
  C 2    0|    0|    0|    0|

Statistics over test set of length: 15:

Relative Accuracy: 1
AUC (example by example) in Hand & Till's way: 1

Let us show the solution: 2
f(X0, X1, X2, X3) = R :- X0=overcast, X2=high. [class: yes]
f(X0, X1, X2, X3) = R :- X0=rain, X2=high, X3=weak. [class: yes]
f(X0, X1, X2, X3) = R :- X0=rain, X2=high, X3=strong. [class: no]
f(X0, X1, X2, X3) = R :- X0=sunny, X2=high. [class: no]
f(X0, X1, X2, X3) = R :- X2=normal, X3=weak. [class: yes]
f(X0, X1, X2, X3) = R :- X0=overcast, X2=normal, X3=strong. [class: yes]
f(X0, X1, X2, X3) = R :- X0=rain, X2=normal, X3=strong. [class: no]
f(X0, X1, X2, X3) = R :- X0=sunny, X2=normal, X3=strong. [class: yes]


END OF RESULTS

Calculating mean differences between solutions
Mean Discrepancy: 0.133333
Mean Syntactic Euclidean Distance: 1.49475
Mean Syntactic Manhattan Distance: 2.66667
Mean Accuracy: 0.933333
Maximum Accuracy: 1


Test dataset destroyed successfully
The multitree has been destroyed successfully
```

```
====> Time used (for learning): 0.03 secs.
**** Smooth end of SMILES execution ****
```

Three different solutions have been obtained, which have different components. At the end of the results, SMILES shows some statistics: *mean discrepancy* indicates the mean semantic difference between the solutions (i.e. the percentage of examples in which their predictions differ), two different syntactic distances which measure the difference in components between the solutions, the mean and maximum accuracy from all the solutions. More detailed explanation on these measures can be found in [27].

3.8 Validation Set and Cross-validation

A validation set is a subset of the training set that is not used for constructing the model, but for selecting between or combining the constructed models. A typical use of validation sets is the technique known as cross-validation.

In our case, if the training set is large enough we can "reserve" some part of it and use just a portion of it for training (we will call it subtraining set). This is especially useful with our multitree paradigm, since we can grow the multitree using the subtraining set as always, and then use the remaining dataset (the validation set) to select a good solution or combining between different solutions.

At the present moment, this option can be selected as follows:

```
%--sample training: whether or not a subset of the trainset is to be used
%sample training set=no sample training set
sample training set=sample training set
```

Note that if the "cross coverage" option is selected as the option for the best tree selection or "cross coverage combination" is selected, then the sample training must be activated.

Finally, the proportion (from 0 to 1) of the training set that is devoted for learning is specified through the following option:

```
%--sample training set portion: the portion of the trainset to be sampled
sample training set proportion=0.05
```

In the previous case, 5% of the data would be used for training and 95% of the data would be used for validation.

The use of a validation test can also be quite useful when the number of examples is too large to be handled by SMILES with a reasonable consumption of resources. Note that the sample is random and the rest is used for selection/combination. Consequently, this is better than making a manual sample before feeding the data to SMILES.

Another important feature of SMILES is *cross-validation*. Cross-validation is a powerful method to evaluate the quality of a classifier in a more reliable way. Cross-validation is based on the idea of automatically splitting the whole dataset into two parts: training set and test set. However, this split can be done randomly several times and obtain the average of all the results. This can give a much more accurate result than just evaluating one split (either manually done into training set file and test set file, or automatically).

In order to do cross-validation, the system should be supplied with a training set that contains all the data. In the smiles distribution, this kind of files can be found in the samples directory with names such as "/samples/monk2.all".

The options must be chosen in the following way to do cross-validation:

```
%--sample training: whether or not using a subset of the training set
sample training set=no sample training set
```

```
%sample training set=sample training set
%
%--sample training set portion: the proportion of the training set to be
sampled
sample training set proportion=0.50
%
%-- cross validation: use a different test set file for results or split
the training set
%cross validation=use separate test set
cross validation=cross validation
%cross validation=kfold cross validation
%cross validation=repeated kfold cross validation
%
%-- how many times (if cross validation) the split has to be done
k fold of cross validation=10
%
%-- how many times (if cross validation) we repeat the experiment
repeat kfold=1
```

Note that the “sample training” option must not be enabled. The “sample training set portion”, however, is necessary and tells SMILES that the input data file is going to be split into two datasets (training and test) of equal size (50% and 50%). The next option is the key one: “cross-validation” that must be selected “cross validation”, in order to make SMILES not expect the test set, and let it extract its test set. Finally, the last option “k fold of cross validation” selects the times that different splits have to be done. Note that each fold performs a different and random split of the initial dataset into two different sets (training set and dataset) and learns the multitree, with all the associated process.

For instance, if we use the file “samples/monks2.all” with MSE split and 100 trees, then we would have 10 runs of the algorithm. At the end, a summarised listing of the 10 runs is included and, ultimately, the mean and standard deviation of these 10 runs are computed, giving a portrait as follows:

```
Mean Results:
  N. of susp. nodes explored :          100 +/-            0
  Solutions in the Multitree :  6.31864e+09 +/-  7.02266e+09
 Results for 1st Solution:
  Accuracy of 1st Solution   :     0.701667 +/-    0.0541203
  AUC (by nodes) of 1st Sol  :     0.710585 +/-    0.0698885
  Mean # Rules               :        286.9
 Results for Combination:
  Accuracy of Combination    :     0.768333 +/-    0.0677914
  AUC of Combination         :     0.768977 +/-    0.0550205
 Results for Best Solution:
  Accuracy of Best           :     0.693333 +/-     0.073367
  Mean # Rules               :        275.2
  Accuracy class 0           :     0.756506 +/-    0.0753576
  Accuracy class 1           :     0.566578 +/-    0.0963091
  AUC by Hand                :     0.680732 +/-    0.0821538
  MSE                        :     0.719583 +/-    0.0714988
  LogLoss                    :     0.740103 +/-    0.0827415

 Time Used                   :        0.273 +/-    0.0498999
```

Mean results are computed for a lot of measures. First some statistics on the multitree are shown. Then, the mean results of the first solution are shown: its accuracy and its AUC (Area Under the ROC Curve), as well as the number of rules. Secondly, the same results for combination are shown (except, logically, the number of rules). Finally, for the best solution further measures are shown. The AUC, MSE and LogLoss measures will be discussed later. In the end the mean time used for each iteration is shown.

3.9 k-fold Cross-Validation and repeated k-fold Cross-Validation

K-fold Cross-Validation is the usual way to use cross-validation. The idea is to use all the possible combinations of a partition. For instance if k=10, we can partition the dataset into ten parts. Then, we can select ten different combinations of 9 parts for the training set and 1 part for the test set.

The way to use k-fold Cross-Validation in SMILES is quite simple:

```
%--sample training: whether or not a subset of the training set is to be used
%sample training set=sample training set
%
%--sample training set portion: the proportion of the training set to be sampled
sample training set proportion=0.90
%
%-- cross validation: use a different test set file for results or split the training set
cross validation=repeated kfold cross validation
%
%-- how many times (if cross validation) the split has to be done
k fold of cross validation=10
%
%-- how many times (if cross validation) we repeat the experiment
repeat kfold=1
```

The “sample training set proportion” (lets call it *p*) tells which proportion is used for training, as always. The difference is that now, each iteration a different subset of the *same* partition is used. Note that the meaning of any combination of *p* and *k* is clear when (1-*p*)**k* <= 1. Other combinations are implementation dependent.

Finally, in some cases 10-fold validation could not give a quite reliable information about the quality of a hypothesis. A good idea is to augment k, even to match the number of examples, known as all-to-1 cross-validation.

Another way to augment reliability of the means computed by SMILES is the use of repeated experiments. This option is called “repeated kfold cross validation”. For instance, if we use the previous options with these modifications:

```
%-- cross validation: use a different test set file for results or split the training set
cross validation=repeated kfold cross validation
%-- how many times (if cross validation) we repeat the experiment
repeat kfold=20
```

Then the experiment will be repeated 20 times, i.e. we make 20 partitions and exploit each combination of each partition. In the previous case, we would have 20x10= 200 iterations, from

which the means are computed. For instance, the following output shows the results for monks2 with MSE split and 100 trees, and 10x10= 100 runs of the algorithm.

```
Mean Results:
  N. of susp. nodes explored :          100 +/-            0
  Solutions in the Multitree :  8.07935e+10 +/-  4.18201e+11
 Results for 1st Solution:
  Accuracy of 1st Solution   :        0.683 +/-    0.0511364
  AUC (by nodes) of 1st Sol  :     0.666375 +/-    0.0615097
  Mean # Rules               :       282.79
Results for Combination:
  Accuracy of Combination    :     0.750167 +/-    0.0583391
  AUC of Combination         :     0.741057 +/-    0.0611061
 Results for Best Solution:
  Accuracy of Best           :        0.692 +/-    0.0524634
  Mean # Rules               :       270.71
  Accuracy class 0           :     0.755819 +/-    0.0685639
  Accuracy class 1           :     0.576554 +/-    0.0986817
  AUC by Hand                :     0.669107 +/-    0.0568279
  MSE                        :     0.712333 +/-    0.0493715
  LogLoss                    :     0.743794 +/-    0.0519433

 Time Used                   :       0.2756 +/-    0.0591577
```

As we can see the deviations are now lower than for just 10 cross-validation.

When small datasets are used or low proportions are used for cross-validation, there is a higher possibility than in one partition of the dataset one class wouldn't appear in any example of the validation test dataset. In this case, the evaluation results would not be as accurate, as if this happens. In order to avoid this there is simple (although inefficient) way: if a partition leaves the validation test dataset without examples of any class, than the partition is repeated. This can be enabled through the following options:

```
%--allow (if cross validation) a test dataset with one class without
examples
%allow test without one class=yes
allow test without one class=no
```

3.10 Expected Error and Smoothing Options

The current system includes several ways to compute the expected error. This value can be used or not depending on other options, especially if expected error pre-pruning is active or some other criterion is based on it (split criterion, best tree criterion).

```
%--Expected error method: several ways to compute expected error
%expected error method=no compute
expected error method=relative frequency with majority class
%expected error method=relative frequency with frequency probability
%expected error method=cost with minimum class
%expected error method=cost with frequency probability
%expected error method=cost with cost probability
```

Section 6 includes some details about these options.

The relative frequency used for computing the expected error can be smoothed. Different smoothing criteria can be selected:

```
smoothing method=no smoothing
%smoothing method=laplace
%smoothing method=k-estimate
%smoothing method=m-estimate
%smoothing method=m-estimate uniform
```

Smoothing can also be used for assigning the majority class when using cost information, as we will see in the next subsection, and it can be used in splitting criteria, as we describe next.

The cardinalities of each node in a split are taken into account in some splitting criteria. The derived probabilities can be smoothed in different ways:

```
%--frequency error smoothing:use smoothing or not for node probabilities
in a split
node smoothing method=no smoothing
%node smoothing method=laplace
%node smoothing method=k-estimate
%node smoothing method=m-estimate
%node smoothing method=m-estimate uniform
```

According to our experiments, this smoothing does not improve in general.

Similarly, many splitting criteria are based on the probabilities of the nodes under the split. This probability can be computed from the frequency directly or it can be computed in other more sophisticated ways. These are the current options:

```
%--Probability in splitting criteria
probability in splitting criteria=from frequency no smoothing
%probability in splitting criteria=from frequency smoothing
%probability in splitting criteria=from costs
%probability in splitting criteria=from frequency with stratification
%probability in splitting criteria=from frequency with stratification no
smoothing
```

None of these options is relevant to the Descriptive MDL Splitting Criterion, because this method does not use probabilities. For the Local ROC Splitting Criterion only the two first probabilities (with and without smoothing) can be used and are effective.

Frequency smoothing is especially recommended for CART and DKM splitting criteria. Both of them do not work well for more than two classes. For more than two classes, if you want to use a similar criterion, use MGINI.

3.11 Cost-sensitive and ROC Analysis Features

In many previous sections we have seen some options related to costs. In this subsection we are going to briefly explain cost-sensitive learning [9] and ROC analysis [38] and which features SMILES provides around these items.

Accuracy (or error), i.e., percentage of instances that are correctly classified (respectively incorrectly classified) has been traditionally used as a measure of the quality of classifiers. However, in most situations, not every misclassification has the same consequences. In fact, it is usually the case that misclassifications of minority classes into majority classes (e.g. predicting that a system is safe when it is not) have greater costs than misclassifications of majority classes into minority classes (e.g. predicting that a system is not safe when it actually is). Obviously, the costs of each misclassification are problem dependent, but it is almost never the case that they would be uniform for a single problem. Consequently, accuracy is not generally the best way to evaluate the quality of a classifier or a learning algorithm.

Although there can be other kinds of cost associated with predictions [63] (e.g. test cost that we will address later on), the most relevant ones are misclassification costs, i.e., the cost of classifying an instance of class *a* into class *b*. All these misclassification costs for a specific problem can be arranged in a *c*-dimensional matrix, with *c* being the number of classes. This matrix is called a cost matrix.

A Cost Matrix (also known as Loss Matrix) indicates the costs for correct and incorrect classifications. An example of a Cost Matrix *C* for three classes {a, b, c} might be as follows:

		Actual		
		a	b	c
Predicted	a	-2.5	4	2
	b	2.1	-3.5	0
	c	1.2	1.3	-4

This example shows the usual portrait, the diagonal of the matrix shows the costs for correct classification (-2.5, -3.5, -4). These values are usually negative or zero, because a correct classification could have benefits instead of costs. The other values represent different cases of misclassification. For instance, the value 2.1 in cell (*b*,*a*) means that classifying incorrectly an '*a*' instance as a '*b*' instance has a cost of 2.1.

The use of cost matrices for the generation of classifiers that minimise the resulting prediction cost instead of the prediction error has been incorporated in a few aspects of a few learning systems by changing some criteria or measures used by these methods [43][5][35]. Nonetheless, it is also common to use a learning system that is not cost-sensitive and to modify the class distribution of the training data set to obtain a classifier that adjusts itself to a specific cost matrix and the class distribution of the test set if known [37][17][9].

However, a change of class distribution is usually done by stratification (or re-balancing), i.e., either by under-sampling or by over-sampling. Stratification presents some problems though (lost of data or redundant data).

The usefulness of cost-sensitive learning does not only apply when the cost matrix is known a priori. If the cost matrix is not known, one or many classifiers can be generated in order to behave well in the widest range of circumstances or contexts as possible. The Receiver Operating Characteristic (ROC) analysis [48][61] provides tools to select a set of classifiers that would behave optimally and reject some other useless classifiers.

Finally, given a classifier, it is usual that its accuracy could be lower than 100%, let us say, for instance, 87,5%. In this case, it may be interesting to know to which class the misclassified 12,5% goes and how this error is distributed. A Confusion Matrix is a very practical and intuitive way of seeing such a distribution. Given 100 test examples and a classifier, an example of a Confusion Matrix *M* for three classes {a, b, c} might be as follows:

		Actual		
		a	b	c
Predicted	a	20	2	3
	b	0	30	3
	c	0	2	40

This matrix is understood as follows. From the hundred examples, 20 were of class 'a' and all were correctly classified, 34 were of class 'b' from which 30 were correctly classified as 'b', 2 misclassified as 'a' and 2 misclassified as 'c'. Finally, 46 were of class 'c' from which 40 were correctly classified as 'c', 3 misclassified as 'a' and 3 misclassified as 'b'. The confusion matrix can be shown by SMILES if costs are active or by modifying an option (see section 3.17).

From the cost matrix and the confusion matrix it is very easy to compute the cost of a classifier for a given dataset, just as the 1 by 1 matrix product, given a Resulting Matrix:

$$R(i,j) = M(i,j) \cdot C(i,j)$$

Our system SMILES incorporates many features that can handle cost information, either given as class weights or as a cost matrix.

The first thing to tell SMILES is how the cost matrix is going to be specified. This once again can be done through the option file:

```
%-- weights method:how the cost matrix is constructed
weight method=no costs
%weight method=uniform weights
%weight method=inverse frequency weights
%weight method=weights from file
%weight method=costs from matrix
```

The first three options do not need any additional information. The next two require some information, either a line in the training set file or a separate cost matrix file.

The "no costs" option is the default option and it is equivalent to the use of "uniform weights", which means a matrix with all equal costs. The difference is just in efficiency, since the first option does not force SMILES to do any cost calculation.

The third option generates class weights according to the class distribution. For instance, if a training set of three classes has distribution (500, 2000, 2500) , then the following weights are generated (c / 500, c / 2000, c / 2500) where *c* is a parameter that can be modified through program (hardwired option). This inverse class distribution assigned weights try to give more relevance to the classes with less cardinality in order to "compensate" the dataset.

As we have discussed before, accuracy is frequently a much too simplified measure of the quality of a classifier. For instance, given a dataset whose distribution of classes is (p_a= 0.85, p_b= 0.1, p_c= 0.05), i.e., most of the examples are of the class '*a*', a simple classifier predicting everything into class '*a*' would have 85% of accuracy.

The weight can also be read from the training file if the option "weights from file" is chosen. Then SMILES would look for a line as follows:

```
!WEIGHTS:yes=1,no=2
```

as we discussed in section 3.1.

For this two latter options, we have talked about weights, whereas we talked about costs elsewhere. How is a weight vector converted into a cost matrix?

For instance, from this weight vector:

	a	b	C
Weights	3	5	2

If we could use over-sampling with these weights, then we would have that the frequency of class '*a*' would be multiplied by 3/2, and the frequency of class '*b*' by 5/2, by conveniently duplicating some of the examples.

It is easy to show that the corresponding cost matrix to this over-sampling would be:

		Actual		
		a	b	c
Predicted	a	-3	5	2
	b	3	-5	2
	c	3	5	-2

Moreover, for two classes it is easy to show (see [27]) that the resulting assignments will be exactly the same that if we put 0s on the diagonal.

Finally, the easiest way to create a matrix is to set the "costs from matrix" option. In this case, SMILES will look for a cost file specified in the command line (the third file), as the usage shows:

```
USAGE:
./smiles file.train [file.test] [file.cost] [file.testcost]
```

Otherwise this file will be ignored. If we do not want to specify the test set, we just place the symbol "-". For instance, the following command line, would just look for a train set file and a cost file.

```
./smiles samples/liver.all - liver.cost
```

The format of the file is just a list of real numbers separated by commas. The last one must also have a comma, such in the following example:

```
% Matrix 4x4 for cars problem
0,     3.2,    1.1,    4.3,
2.5,   0,      10.4,   8.2,
3.2,   17.1,   0,      0.1,
2.3,   8.2,    4.1,    0,
```

As always, lines beginning with '%' are ignored.

Now that we know how to construct the cost matrix in several ways, what can we do with it? The first and most effective thing to do is to change the way in which the classes are assigned to each node. Instead of assigning the majority class (the default option), i.e. the most frequent class, we can label a node with the class that *minimises the cost*. The possible ways are as follows:

```
%--class selection method: how the class of a leaf is assigned
class selection method=majority class
%class selection method=minimum cost selection
%class selection method=minimum cost class without smoothing
%class selection method=stratification class
%class selection method=stratification without smoothing
```

The "minimum cost selection" is the option that assigns the class taking into account the costs, i.e., if we have a leaf vector $V(i)$ then we look for the class i such that:

$$\text{Assigned Class} = \arg\min_i \sum_j C(i,j)\cdot V(j)$$

For instance, if a leaf node has $V = \{20, 10, 22\}$ and we have the following cost matrix:

		Actual a	Actual b	Actual c
	a	0	10	5
Predicted	b	1	0	2
	c	5	3	0

If we assign class *a* to the node we have a cost of 20·0+10·10+22·5=210. If we assign class *b* to the node we have a cost of 20·1+10·0+22·2=64. If we assign class *c* to the node we have a cost of 20·5+10·3+22·0=130. It turns out that despite that *c* is the majority class, *b* is the less costly class.

Note that when pruning is not active all the leaf nodes are pure and hence, the options "minimum cost selection" and "minimum cost class without smoothing" are equal to the majority class provided the cost matrix is normalised (all values are positive and only 0s in the

diagonal). The stratification methods are usually worse because they use a simplified version of the cost matrix. They are useful for comparing with other methods and are not recommended.

The class selection method is the most important option in the sense that it has a very effective impact on cost minimisation. Other less effective (or even with no provable good effects) are based on modifying some criteria used during learning.

One first idea that turned out to be poorly successful was to modify the splitting criteria taking costs into account. Since most of them are based on probabilities, the idea was to modify these probabilities taking cost into account. Although we saw the probability in the splitting criteria before, the three last possibilities can now be understood as ways to compute this probability based on expected costs instead of expected frequencies.

```
%--Probability in splitting criteria
probability in splitting criteria=from frequency no smoothing
%probability in splitting criteria=from frequency smoothing
%probability in splitting criteria=from costs
%probability in splitting criteria=from frequency with stratification
%probability in splitting criteria=from frequency with stratification
with stratification
```

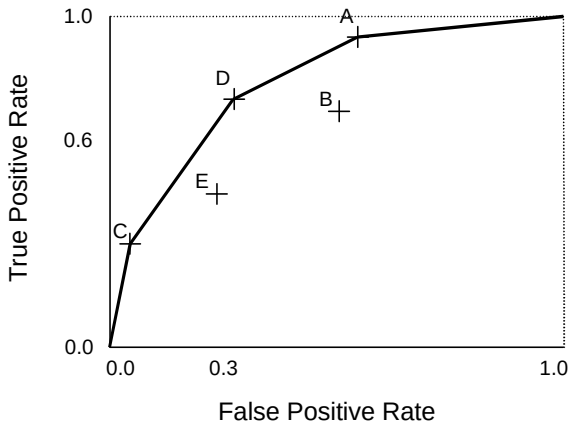
In the case that the third one is chosen ("from costs"), then there are two ways of deriving this probability, that can be selected through the option file:

```
%--cost derived probability method
cost derived probability=direct
%cost derived probability=with smoothing
```

These options are explained in Section 6.

There are also cost-sensitive options in how to compute the expected error, which would turn into an expected cost. This would be used in any other option that uses expected error. Costs can also be used in "multitree solution combination" options.

Finally, there are some facilities related to ROC analysis. These are possible for problems with 2 classes (in the following sections we will see measures that are applicable for more than 2 classes). A ROC plot of several points is a convex hull as illustrated in the following figure:



Example of a ROC curve

First of all we can compute and show the ROC points obtained by the optimal assignments of a classifier. Note that these points are obtained from a single "soft" classifier, not as usual, when we obtain a ROC curve from many classifiers. These points represent different assignments of the same trees, that, in fact, give different classifiers:

```
%--compute ROC points
compute ROC points=no
%compute ROC points=yes
%
%-- show ROC points
show ROC points=no
%show ROC points=yes
```

From these points we can draw and compute the area by using these options:

```
%-- compute ROC area
compute ROC area=no
%compute ROC area=yes
%
%-- generate ROC curve file
%generate ROC curve file=no
generate ROC curve file=yes
```

The outputs of each solution are generated into a postscript file called “ROCtstN.ps” for the training set, “ROCtotN.ps” when the training set is used for ordering the nodes and the test set to compute the leaves probabilities, and “ROCtstN.ps” for the test set, where N is the number of the solution. If there is only one solution, then N is just 0.

These ROC features are thoroughly explained in [24].

3.12 AUC Evaluation

We have just described that different curves can be output: wrt. the training set, using the training set for the ordering and the test set to compute the leaves probabilities and the entire curve with the test set. From these curves we can compute the area under the ROC curve, which is very useful to estimate the quality of a classifier. In particular the AUC measures used and output by SMILES are:

- AUC0: the curve is constructed with the order derived from the training set and with the node distributions given by the training set. If pruning is not active, it is usually 1. Consequently this measure is not much too informative.
- AUC1: the curve is constructed with the order derived from the test set and with the node distributions given by the test set. This way to evaluate a classifier seems cheating, because the order is derived from the test, which could not be performed in general. What this measure tells is that if this value is high, there will be good labellings of the tree that could obtain high accuracies for different cost matrices, but it does not tell that we are necessarily going to be able to *use* these optimal labellings.
- AUC2: the curve is constructed with the order derived from the training set and with the node distributions given by the test set. This is a very useful measure. What this measure tells is that if this value is high, there will be good labellings of the tree that could obtain high accuracies for different cost matrices, and since we are using the order from the training set, we can know that order and we will be able to *use* these optimal labellings.
- AUC4 (1P-AUC): it is a simplification of AUC2 that only uses one point for computing the curve. The interest of this measure is that can be used for more than 2 classes (currently it is not implemented for more than 2 classes).

All the AUC measures have singular values when FPR=0 or TPR=0. We have assumed the following assignments. If FPR=0 then AUC=TPR, and if TPR=0 then AUC=FPR.

Theoretically, AUC1<AUC0, AUC2<AUC1 and AUC4<AUC1. These measures are shown with cross-validation when ROC options are active (option "compute ROC points=yes"). For monks2, for instance, this could be a possible result (only the excerpt for the best solution is shown):

```
Results for Best Solution:
  Accuracy of Best           :        0.692 +/-    0.0524634
  Mean # Rules               :       270.71
  Accuracy class 0           :     0.755819 +/-    0.0685639
  Accuracy class 1           :     0.576554 +/-    0.0986817
  AUC0 (train)               :     0.185172 +/-     0.014343
  AUC1 (test)                :     0.991017 +/-   0.00788894
  AUC2 (train-order + test)  :     0.669008 +/-    0.0594674
  AUC4 (train-1-lab + test)  :     0.666187 +/-    0.0528094
  AUC by Hand                :     0.669107 +/-    0.0568279
  MSE                        :     0.712333 +/-    0.0493715
  LogLoss                    :     0.743794 +/-    0.0519433
```

If possible, AUC2 should be used instead of accuracy to evaluate classifiers when class distributions or costs might change when the model were to be applied. This is the one that is shown by default (computed in Hand and Till's way) and shown as "AUC by Hand". AUC by Hand and AUC2 are not exactly the same because of the ordering functions. When two or more nodes have the same ratio, then their precise order is not defined, and hence small variations can occur if both methods order them differently (this is solved since version 2.1.7) he next section we better explain how AUC2 is computed in Hand and Till's way.

3.13 Multi-class AUC Evaluation. AUCH, MSE and LogLoss Measures

The problem of previous AUC measures and ROC analysis is that they are only applicable to problems with two classes and not valid for multi-class problems. Only the 1-point AUC measure is extensible. However this measure does not take into account the possible labelling that can be done in a decision tree, turning it into a soft classifier.

Fortunately there are some extensions and approximations of the AUC measure for more than two classes. The first one is the Hand and Till M Function (that we will call AUCH), which is the most popular extension, and the other two are traditional measures adapted for this purpose. All of them try to consider that not all the errors have the same consequences and that more compensated solutions are preferable from those that would be selected by using accuracy.

3.13.1 Hand and Till M *Function*

Hand and Till present a generalisation of a particular AUC measure [30]. It has been shown that for two dimensions the AUC measure is equivalent to the GINI measure (not that the GINI measure is not the GINI splitting criterion used in the CART algorithm).

The idea is that in the AUC measure for 2 dimensions, they use the estimated probabilities of an example x_i pertaining to the class 0, denoted by $p_0(\cdot)$, (estimated from the training set), to rank the pairs $\{g_k, f_k\}$ where g_k and f_k are defined as $g_i = p_0(x1_i)$ and $f_j = p_0(x0_j)$ where $x1_i$ are the examples from the test set of class 1 and $x0_j$ are the examples from the test set of class 0. Note that instead of ordering nodes they order examples.

For instance consider the following two nodes for the training set:

Node 1: (4,1) --> class 0 with prob= 4 / 5 = 0.8

Node 2: (2,3) --> class 1 with prob= 2 / 5 = 0.4

And now consider that the test set is distributed in the following way over the decision tree:

Node 1: (6,4)

Node 2: (4,11)

with n_0= 10 elements of class 0 and n_1= 15 elements of class 1. Then we have:

- 6 of class 0 with $p_0(\cdot) = 0.8$
- 4 of class 1 with $p_0(\cdot) = 0.8$
- 4 of class 0 with $p_0(\cdot) = 0.4$
- 11 of class 0 with $p_0(\cdot) = 0.4$

From here, we can rank them as described in [30]. Let us denote with r_i the rank of the i^{th} class 0 test set point. Let us denote $S_0 = \Sigma r_i$. The they derive the area as:

$$\hat{A} = \frac{S_0 - n_0(n_0+1)/2}{n_0 n_1}$$

This area, although is an AUC (and it has the equivalence Gini + 1 = 2 × A), has two main differences with respect to usual ROC curves and also to a similar proposal in [18]:

- It is a step-like (or a stairs-like) area (no diagonals between the points are computed)).
- It is not convex, because the order is given by the training set and the examples are given by the test set.

Apart from this, the most relevant novelty of Hand and Till paper is that they understand *A* as "an overall measure of how well separated are the estimated distributions of $p_0(\cdot)$ for class 0 and class 1", i.e., A(i,j) could be computed for whatever pair of classes *i* and *j*.

This interpretation allows what they call "a simple generalisation of the AUC for multiple class classification problems". They define a new measure M as:

$$M = \frac{1}{c(c-1)} \sum_{i \neq j} \hat{A}(i,j) = \frac{2}{c(c-1)} \sum_{i<j} \hat{A}(i,j)$$

3.13.2 Other Measures MSE and LogLoss

The AUC measure tells "how well separated are the estimated distributions of $p(x)$ for class 0 and class 1" (Hand & Till 2001). Why do not we develop other measures that try to approximate how well separated two distributions are?

One measure of this kind is the well-known Mean Squared Error measure.

$$MSE = \frac{\sum_{i=1..m} \sum_{j=1..c} (f(i,j) - p(i,j))^2}{m \cdot c}$$

Where $f(i,j)$ is the actual probability of example *i* to be of class *j* and $p(i,j)$ is the estimated probability of example *i* to be of class *j*. The denominator gives a normalised MSE between 0 and 1. For classification problems, $f(i,j)$ will be always 0 or 1, depending on the class.

Another measure is the log-loss, which is claimed to be a measure of the goodness of probability estimates (Bernardo & Smith 1993) (Mitchell 1997).

$$LogLoss = \frac{-\sum_{i=1..m}\sum_{j=1..c}(f(i,j)\log_2 p(i,j))}{m}$$

In the case p(i,j)=0, we use a forced smoothing in order to avoid a negative infinite value.

Note that these two measures are closely related to AUCH. As we have said, the AUCH measure tells "how well separated are the estimated distributions of $p(x)$ for class 0 and class 1" (Hand & Till 2001). This is quite the same of what is measured by the expression $(f(i,j)-p(i,j))^2$ or by the expression $(f(i,j)\log p(i,j))$.

These two measures have the advantage that are much easier to be understood and computed. Obviously, they have no problems of generalisation for more than 2 classes.

3.14 ROC-based Splitting Criteria

In [24], the first ROC-based splitting criteria was defined as follows:

AUCsplit: Given several splits s_j, each one formed by n_j leaves $\{l^j_1, l^j_{2,}..., l^j_{nj}\}$, then the best split is the one that maximises:

$$AUCsplit(s_j) = \sum_{i=1..n_j} A(P^j_{i-1}, P^j_i)$$

where the points $P_i{}^j$ are obtained in the usual way (sorting the leaves of each split by local positive accuracy), and A(p1, p2) means the area of the trapezoid between these two points.

The first question that arises with a new splitting criterion is how it differs from other criteria previously proposed. To answer this question, let us review the general formula of other well-known splitting criteria, such as Gini [6], Gain, Gain Ratio and C4.5 criterion [52] and DKM [33]. These splitting criteria find the split with the lowest $I(s^j)$, where $I(s_j)$ is defined as:

$$I(s) = \sum_{j=1..n_j} p_j \cdot f(p^+_j, p^-_j)$$

where p_j is the probability of being sorted into that node in the split (cardinality of child node divided by the cardinality of parent node). Using this general formula, each splitting criterion implements a different function f, as shown in the following table:

CRITERION	$f(a,b)$
ACCURACY (EERROR)	$\min(a,b)$
GINI (CART)	$2ab$
ENTROPY (GAIN)	$a\cdot\log(a)+b\cdot\log(b)$
DKM	$2(a\cdot b)^{1/2}$

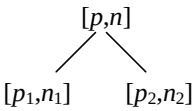
These functions $f(a,b)$ are impurity functions, and the function $I(s)$ calculates a weighted average of the impurity of the children in a split. In general, we need to compare this weighted average impurity of the children with the impurity of the parent, if we are comparing different splits of different nodes.

Consider for instance the following two splits:

The children have the same weighted average impurity in both cases. In order to see that the left is a better split than the right (assuming $a?b$), we need to take the impurity of the parent into account. In contrast, AUCsplit evaluates the quality of the whole split (parent + children) and

cannot be reduced to a difference in impurity between parent and children. The left split has AUCsplit=$a/(a+b)$ (assuming $a>b$), while the right split has AUCsplit=0.5, indicating that nothing has been gained in ROC space with respect to the default diagonal from (0,0) to (1,1).

An interesting relationship can be established with the Gini index. Consider the following binary split:



If the left child has higher local positive accuracy, then we have:

$$AUCsplit = \frac{1}{2}\left(\frac{p_1}{p} - \frac{n_1}{n} + 1\right) = \frac{p_1 n - p n_1 + pn}{2pn} = \frac{p_1 n + p n_2}{2pn}$$

It is interesting to note that the denominator of this expression is the Gini index of the parent, and the enumerator could be called a mutual Gini index of the children given the parent.

This splitting criterion should be used when we want to maximise AUC instead of maximising accuracy.

In the previous section we have discussed that this measure can only be applied to problems with two classes, and, consequently, so can the splitting criterion. Nonetheless, in the previous section, we have presented measures that are valid for more than two classes. Let us see the corresponding splitting criteria. These are developed in [25]. Let us begin with Hand and Till's M function (AUCH).

The first problem is that the previous formulation of Hand and Till's *M* function is based on a ranking of examples. Consequently it has cost O(m·log$m$), where m is the number of examples. If this has to be done for the c classes, this can be intractable. This high computational complexity would become an important hindrance for using it as splitting criterion.

Fortunately, this complexity could be reduced if we know that for all the examples under the same node the rank would be the same. Consequently, we only have to rank the nodes, as we made in the two-classes case, and the complexity of the ordering would be O(n·log$n$) where n is the number of nodes. Given a set of nodes S, let us define the area just considering two classes a and b:

$$AUC_{ab}(S) = \sum_{i=1..|s|} A_{ab}(P_{i-1}^{ab}, P_i^{ab})$$

where the points P_i^{ab} are obtained in the usual way (sorting the leaves of each split by local positive accuracy, just taking into account classes a and b).

And now, we can redefine the *M* function as M-AUC in the following way:

$$MAUC(S) = \frac{2}{c(c-1)} \sum_{a<b} AUC_{ab}(S)$$

Finally, we can easily define the splitting criterion in the following way:

M-AUCsplit. Given several splits s_j, each one formed by n_j leaves $\{l_1^j, l_2^j, ..., l_{nj}^j\}$, then the best split is the one that maximises MAUC(s_j).

The other two measures MSE and LogLoss are much easier to be used as splitting criteria. Consider a partition with n nodes. Since for classification problems, $f(i,j)$ will be always 0 or 1, depending on the class, we have the following equation:

$$MSEsplit = \frac{\sum_{i=1..n} \sum_{j=1..c} card(i,j) \cdot \sum_{k=1..c} if(k=j)(1-p(i,j))^2 else(0-p(i,j))^2}{n \cdot c}$$

where *card*(*i*,*j*) is the number of examples of class *j* in the node *i*, and *p*(*i*,*j*) is the estimated probability of class *j* for the node *i*.

The formula for logloss is simpler, because many cases are just removed when *f*(*i*,*j*)=0. Consequently, we have:

$$LogLossSplit = \frac{\sum_{i=1..n} \sum_{j=1..c} card(i,j) \cdot (f(i,j) \log_2 p(i,j))}{n}$$

The cost is just O(*n*·*c*). Finally, it is important to note that, in the same way expected error is not a very good splitting criterion for obtaining low global errors, it is possible that M-AUCsplit is not the best formulation for minimising M-AUC, MSEsplit is not the best formulation for minimising MSE and LogLossSplit is not the best formulation for minimising LogLoss.

According to our experiments (which we will show in section 4), the best criterion seems to be MSE.

3.15 Test Cost

Apart from misclassification costs, there is another kind of costs that can be extremely important in some applications, especially in medical diagnosis. Consider an imaginary diagnosis problem for three different diseases (DISEASE1, DISEASE2 and DISEASE3) as follows:

BP-Min (Minimum Blood Pressure): numerical.
BP-Max (Maximum Blood Pressure): numerical.
Div_End (Diverticulities_through_Endoscopy). Nominal: pos / neg.
Cysts_Scopy (Cysts through Colonoscopy) . Nominal: pos / neg
Meningitis_Lumbar (Meningitis through Lumbar Puncture). Nominal: pos / neg
Cysts_Echo (Cysts through Echography). Nominal: pos / neg
Glucose_BA (Glucose concentration through Blood Analysis): numerical
Leucocytoses_Urine (Leucocytoses through Urine Analysis) Nominal: pos / neg

The test costs (taking into account economic, risk and pain issues) have been determined as follows:

BP-Min: 1 cost units.
BP-Max : 1 cost units.
Div_End: 30 cost units.
Cysts_Scopy: 50 cost units.
Meningitis_Lumbar: 200 cost units.
Cysts_Echo: 15 cost units.
Glucose_BA: 15 cost units.
Leucocytoses_Urine: 10 cost units.

And now consider that we have three decision trees for this problem:

DECISION TREE 1:

Disease (BP-Min, BP-Max, Div_End, Cysts_Scopy, Meningitis_Lumbar, Cysts_Echo, Glucose_BA, Leucocytoses_Urine) = R
{DT1-Node1} :- Cysts_Scopy=neg, Glucose_BA>=120 [class: DISEASE1]
{DT1-Node2} :- Cysts_Scopy=neg, Glucose_BA<120, Leucocytoses_Urine=neg [class: DISEASE1]
{DT1-Node3} :- Cysts_Scopy=neg, Glucose_BA<120, Leucocytoses_Urine=pos [class: DISEASE2]
{DT1-Node4} :- Cysts_Scopy=pos [class: DISEASE3]

DECISION TREE 2:

Disease (BP-Min, BP-Max, Div_End, Cysts_Scopy, Meningitis_Lumbar, Cysts_Echo, Glucose_BA, Leucocytoses_Urine) = R

{DT2-Node1} :- BP_Min >= 100, Cysts_Echo = pos [class: DISEASE3]

{DT2-Node2} :- BP_Min >= 100, Cysts_Echo = neg, Meningitis_Lumbar=neg [class: DISEASE1]

{DT2-Node3} :- BP_Min >= 100, Cysts_Echo = neg, Meningitis_Lumbar=pos [class: DISEASE2]

{DT2-Node4} :- BP_Max >= 150, Cysts_Echo = pos [class: DISEASE3]

{DT2-Node5} :- BP_Max >= 150, Cysts_Echo = neg, Meningitis_Lumbar=neg [class: DISEASE1]

{DT2-Node6} :- BP_Max >= 150, Cysts_Echo = neg, Meningitis_Lumbar=pos [class: DISEASE2]

{DT2-Node7} :- BP_Min < 100, BP_Max < 150, Div_End = pos [class: DISEASE3]

{DT2-Node8} :- BP_Min < 100, BP_Max < 150, Div_End = neg [class: DISEASE1]

DECISION TREE 3:

Disease (BP-Min, BP-Max, Div_End, Cysts_Scopy, Meningitis_Lumbar, Cysts_Echo, Glucose_BA, Leucocytoses_Urine) = R

{DT3-Node1} :- Leucocytoses_Urine=neg, Cysts_Echo = pos [class: DISEASE3]

{DT3-Node2} :- Leucocytoses_Urine=neg, Cysts_Echo = neg [class: DISEASE1]

{DT3-Node3} :- Leucocytoses_Urine=pos [class: DISEASE2]

The three previous trees use different attributes to make a diagnosis. If we do not have any additional information apart from their accuracy we would have to select the most accurate one or the one with highest AUC.

However, let us consider that we have determined, by using e.g. the training set, what is the frequency that an example falls into each node:

DECISION TREE 1:	**DECISION TREE 2:**	**DECISION TREE 3:**
{DT1-Node1} (0.2)	{DT2-Node1} (0.1)	{DT3-Node1} (0.4)
{DT1-Node2} (0.3)	{DT2-Node2} (0.03)	{DT3-Node2} (0.5)
{DT1-Node3} (0.1)	{DT2-Node3} (0.05)	{DT3-Node3} (0.1)
{DT1-Node4} (0.4)	{DT2-Node4} (0.15)	
	{DT2-Node5} (0.02)	
	{DT2-Node6} (0.05)	
	{DT2-Node7} (0.15)	
	{DT2-Node8} (0.45)	

From the previous information we can compute the mean test cost of an example.

MEAN TEST COST DECISION TREE 1:

{DT1-Node1} $0.2 \cdot [50 + 15] = 12$

{DT1-Node2} $0.3 \cdot [50 + 15 + 10] = 22.5$

{DT1-Node3} $0.1 \cdot [50 + 15 + 10] = 7.5$

{DT1-Node4} $0.4 \cdot [50] = 20$

TOTAL: **62 cost units**.

DECISION TREE 2:

{DT2-Node1} $0.1 \cdot [1 + 15] = 1.6$

{DT2-Node2} $0.03 \cdot [1 + 15 + 200] = 6.48$

{DT2-Node3} $0.05 \cdot [1 + 15 + 200] = 10.8$

{DT2-Node4} $0.15 \cdot [1 + 15] = 2.4$

{DT2-Node5} $0.03 \cdot [1 + 15 + 200] = 6.48$

{DT2-Node6} $0.05 \cdot [1 + 15 + 200] = 10.8$

{DT2-Node7} $0.15 \cdot [1 + 1 + 30] = 4.8$

{DT2-Node8} $0.45 \cdot [1 + 1 + 30] = 14.4$

TOTAL: **57.76 cost units**.

DECISION TREE 3:

{DT3-Node1} :- $0.4 \cdot [10 + 15] = 10$

{DT3-Node2} :- $0.5 \cdot [10 + 15] = 12.5$

{DT3-Node3} :- 0.1 · [10] = 1

TOTAL: **23.5 cost units**.

As we can see the third decision tree has an average test cost quite lower than the other two trees, and in this regard, it is preferable over the rest. We can also see that the frequency of each node is very relevant. For instance, decision tree 2 uses the most expensive test (Meningitis_Lumbar). However, it uses it quite infrequently and turns out to be, in overall terms, a decision tree which is less expensive than the first one. Note that a combination of the three trees, by using any fusion method, would have a cost of ths um of all the single costs, i.e.: 143.26.

Provided that the three decision trees have similar accuracy, it is then much more preferable to use the third decision tree, because test cost would be minimised. Consequently, economic, risk and pain issues are minimised.

SMILES provides tools to compute test cost and to use it in a reasonable way, in order to obtain trees with less cost or to select from a pool of trees (the multitree) the tree with less test cost. Let us review the facilities that SMILES offers in this regard.

The first obvious thing to be done is to allow SMILES to read the test cost information. From this there is an optional additional file that can be specified through the command line:

```
USAGE:
./smiles file.train [file.test] [file.cost] [file.testcost]
```

If we do not want to specify neither a test set nor a cost matrix, we just place the symbol "-". For instance, the following command line, would just look for a train set file and a testcost file.

```
./smiles samples/liver.all - - liver.testcost
```

The format of the file is just a list of real numbers separated by commas. The last one must also have a comma, such in the following example:

```
% Test costs from UCI liver bupa problem
7.27, 7.27, 7.27, 7.27, 9.86, 1
```

As always, lines beginning with '%' are ignored. However, if this file is specified in the command line we must also enable one option in the options file, that tells SMILES to read test costs from files.

```
%--test cost method: how the vector of attribute test costs is
constructed
%test cost method=no test costs
%test cost method=uniform test costs
test cost method=test costs from file
```

SMILES shows the information has read. For instance, for the following command line:

```
./smiles samples/liver.all - - liver.testcost
```

SMILES outputs the following after reading the training set:

```
...
Test Cost Vector to be used:
Argument 0: 7.27
Argument 1: 7.27
Argument 2: 7.27
Argument 3: 7.27
Argument 4: 9.86
Argument 5: 1
...
```

As we can see there are two other options: “no test costs” which does not take this kind of cost into account and “uniform test costs”, which, when we do not know any information about the test costs, we can assume that these are uniform, and SMILES will look for trees that minimise the average number of tests per example, in a similar way as is made in ROC analysis. In other words, if AUC can be computed when the real cost matrix is not know, the uniform test cost vector can be used when the test cost vector is not known.

Now that we know how to read a particular test cost vector or to assume a uniform one, let us explain how SMILES can take advantage of it.

When constructing each tree, a splitting criterion is used to assign different degrees of optimality to each split. This optimality can be modified taking testcost into account. The testcost is computed using the cardinality of each node, i.e., multiplying the examples that fall into each branch by the cost of the tests (attributes) used until that node. Once the testcost is computed, there are three ways of treating test cost information for modifying splitting criteria:

```
%--test cost use: how the vector of attribute test costs is used
test cost use=test costs no use
%test cost use=test costs linear plus1 without repetition
%test cost use=test costs linear plus1 with repetition
```

The first option logically makes no use of this information. The second and third options use this formula to modify the existing splitting optimality (Opt):

$$Opt \; / \; (testcost + 1)^{w}$$

A new weight that tells how much the testcost is used to inversely modify splitting criteria. The exponent w is used to give more relevance to the testcost. This factor is 1 by default and can only be modified in the program sources through the TestCostRelevanceInSplitting option that must be between 0 and infinite. Note that the relevance of testcost can also be modified quite easily by augmenting the absolute value of the testcost vector file.

The difference between the two last options (without or with repetition) is that numerical attributes can be used more than once in the same branch. Consider e.g. a condition X < 3.2; this could be followed by an additional partition below on the same attribute in the same branch suc as X < 1.3 and X >= 1.3. It is not sensible to compute the test cost of attribute X more than once. For this reason, it is more accurate the use of the option “without repetition” that only takes into account this cost once. This gives an exact measurement of testcost.

This modification provides a way to construct trees and muli-trees that have lower average testcosts per example. Obviously, if testcost is given too much importance then accuracy can be affected. For a deeper explanation of this effect and some results, we refer to [16].

Despite the effect and good results of the previous approach, it is not much wise to use a testcost sensitive criterion and then select the best tree using non-testcost sensitive criteria. For this, SMILES also includes two methods for extracting the best solution from the tree taking testcosts into account.

```
%--multitree: best tree selection criterion
%multitree best tree criterion=test cost best
%multitree best tree criterion=occam and test cost best
```

Apart from the “occam best” and others, SMILES provides “test cost best” and “occam and test cost best”. The first method selects the tree with the lowest testcost (repeated use of the same attribute in the same branch are discarded). The “occam and test cost best” combines “occam best” and “test cost best” through the use of a weight factor. The weight of each is determined through the following formula:

$$test\cos t^{a} \cdot numrules^{(1-a)}$$

Where α is a factor modifiable by program by the new option value TestCostRelevanceInSelectBest that must be from 0 to 1.

More information of how effective this selection methods are can be found in [16].

Finally, testcost vectors can be used to give more relevance to some attributes than others, e.g. when the user thinks that some attributes can be more useful or comprehensible than others.

Let us see an example with monk2. Its test costs are given in the file "monks.testcost":

```
% Test costs for monks problems
1.0, 0.0, 10.0, 5.0, 0.0, 100.0
```

If we run SMILES with MSE splitting criterion, and a multi-tree with 100 open nodes and with 10x10 cross validation, but without using testcost, we have:

```
Mean Results:
  N. of susp. nodes explored :          100 +/-            0
  Solutions in the Multitree :  8.07935e+10 +/-  4.18201e+11
 Results for 1st Solution:
  Accuracy of 1st Solution   :        0.683 +/-    0.0511364
  AUC (by nodes) of 1st Sol  :     0.666375 +/-    0.0615097
  Mean # Rules               :       282.79
  Test Cost per Example      :       96.038 +/-      9.27396
Results for Combination:
  Accuracy of Combination    :     0.750167 +/-    0.0583391
  AUC of Combination         :     0.741057 +/-    0.0611061
 Results for Best Solution:
  Accuracy of Best           :        0.692 +/-    0.0524634
  Mean # Rules               :       270.71
  Accuracy class 0           :     0.755819 +/-    0.0685639
  Accuracy class 1           :     0.576554 +/-    0.0986817
  AUC by Hand                :     0.669107 +/-    0.0568279
  MSE                        :     0.712333 +/-    0.0493715
  LogLoss                    :     0.743794 +/-    0.0519433
  Test Cost per Example      :      103.333 +/-      8.65866

  Time Used                  :       0.2765 +/-    0.0581599
```

As we can see now, we have "Test Cost per Example" information for the 1st solution and for the Best Solution. For the first solution we have 96.038 testcost units, and for the best solution we have 103.333.

Now we can see the results if we enable

```
test cost use=test costs linear plus1 without repetition
```

Then we have:

```
Mean Results:
  N. of susp. nodes explored :          100 +/-            0
  Solutions in the Multitree :   1.2661e+14 +/-   6.4715e+14
 Results for 1st Solution:
  Accuracy of 1st Solution   :     0.687667 +/-    0.0535297
  AUC (by nodes) of 1st Sol  :     0.607398 +/-    0.0676029
  Mean # Rules               :       324.36
  Test Cost per Example      :      46.2625 +/-      6.58806
 Results for Combination:
```

```
 Accuracy of Combination    :     0.736667 +/-    0.0566558
 AUC of Combination         :     0.671598 +/-    0.0699609
Results for Best Solution:
 Accuracy of Best           :     0.682667 +/-    0.0624922
 Mean # Rules               :       309.23
 Accuracy class 0           :      0.74696 +/-    0.0755163
 Accuracy class 1           :     0.566277 +/-    0.0993084
 AUC by Hand                :     0.648622 +/-    0.0827158
 MSE                        :     0.689375 +/-    0.0669407
 LogLoss                    :       0.8372 +/-    0.0476209
 Test Cost per Example      :      77.3888 +/-      17.9072

Time Used                   :       0.2716 +/-    0.0625344
```

We can see that testcost is dramatically reduced, with a slight loss in accuracy. We see that the best solution has greater cost than the first solution. This is so because the best solution is defined “Occam best”, in order to obtain the shortest one, not the one with lowest testcost.

We can even reduce the test cost of the best solution by using the option:

```
multitree best tree criterion=test cost best
```

And now we have:

```
Mean Results:
  N. of susp. nodes explored :          100 +/-            0
  Solutions in the Multitree :   1.2661e+14 +/-   6.4715e+14
 Results for 1st Solution:
 Accuracy of 1st Solution    :     0.687667 +/-    0.0535297
 AUC (by nodes) of 1st Sol   :     0.607398 +/-    0.0676029
 Mean # Rules                :       324.36
 Test Cost per Example       :      46.2625 +/-      6.58806
Results for Combination:
 Accuracy of Combination     :     0.736667 +/-    0.0566558
 AUC of Combination          :     0.671598 +/-    0.0699609
Results for Best Solution:
 Accuracy of Best            :     0.685667 +/-    0.0546399
 Mean # Rules                :       323.39
 Accuracy class 0            :     0.753444 +/-    0.0629829
 Accuracy class 1            :     0.561628 +/-    0.0993743
 AUC by Hand                 :     0.610403 +/-    0.0687935
 MSE                         :     0.685792 +/-    0.0541103
 LogLoss                     :     0.845749 +/-    0.0451748
 Test Cost per Example       :      46.6472 +/-      6.5868

Time Used                    :       0.2716 +/-    0.0648654
```

We see that the test cost for the best solution is now significantly lower than before. Unlike this example, in general, with these options is usually lower than the one for the first solution.

3.16 Archetype Solution

The use of a multi-tree allows combination to obtain more accurate results. However, comprehensibility is lost with combined hypothesis. Moreover, combined solutions explored many alternatives and, consequently, they have an extremely high test cost. Consequently they

are not useful when “test-cost” is to be taken into account. This makes combination methods useless for many applications, such as medicine, where test are expensive, and may also be risky and painful. It is not sensible to make all the existing tests to a patient, because a machine learning algorithm requires a multiplicity of different solutions.

But we have described before that a single solution can be extracted from the pool of solutions (with several best single solution methods), the solution is usually not as good (by far) as the combined solution.

An original idea introduced in SMILES is the notion of “archetype” or “representative”. Combined solutions are usually much better than single solution. Combined solutions have a behaviour that is different from any single solution, but, in many cases, one or more solutions are quite closer (in a semantic way) to the combined solution. Why not choosing the single solution that is semantically closer to the combined solution? This is the idea of the *archetype* or *representative* of a group: the individual that best represents the group.

Another original idea of SMILES is that it does not require an additional dataset to compute which is the solution that is most similar to the combined one. SMILES can construct an invented dataset for this. The first thing we must tell SMILES is to extract this archetype to the use of an invented dataset. This is done through the “combination to single solution”:

```
%-- Combination to Single Solution Options
%combination to single solution method=no extraction
combination to single solution method=invented dataset
```

Once determined the construction of an invented dataset, we must tell the size of this invented dataset.

```
%-- Length of the invented random dataset
invented dataset length=10000
```

The greater the invented dataset the better the estimation of the archetype will be (and it will be slower too). Finally, the last thing to specify is how the similarity is to be computed. For this SMILES provides three similarity functions to be applied to the invented dataset.

```
%-- Similarity function used for the selection of a single solution from
the combination
similarity method for combination to single=kappa
%similarity method for combination to single=kappa1
%similarity method for combination to single=qstat
```

According to our experiments in [23], the best similarity method is “kappa”.

Let us see an example. If we run “monks2” in SMILES with MSE splitting criterion, and a multi-tree with 100 open nodes and with 10x10 cross validation, and select the previous options, as well as the use of the testcost (“monks2.testcost”), the option that restricts archetype criteria to just similarity, i.e.:

```
%-- Combination to Single Solution (Archetype) Use of Other Criteria
archetype similarity importance factor=1.0
archetype occam importance factor=0.0
archetype test cost importance factor=0.0
```

Then we would have:

```
Mean Results:
  N. of susp. nodes explored :          100 +/-            0
  Solutions in the Multitree :  4.12571e+15 +/-  3.88401e+16
 Results for 1st Solution:
  Accuracy of 1st Solution   :     0.687667 +/-    0.0535297
  AUC (by nodes) of 1st Sol  :     0.607398 +/-    0.0676029
```

```
 Mean # Rules                :       324.36
 Test Cost per Example       :      46.2625 +/-      6.58806
Results for Combination:
 Accuracy of Combination     :     0.737833 +/-    0.0522155
 AUC of Combination          :     0.676091 +/-    0.0692213
Results for Archetype:
 Accuracy of Archetype       :     0.720333 +/-    0.0520392
 AUC (by nodes) of Archetype:     0.670872 +/-    0.0742581
 Mean # Rules                :       321.89
 Test Cost per Example       :      63.6257 +/-       16.246
Results for Best Solution:
 Accuracy of Best            :     0.685167 +/-    0.0593121
 Mean # Rules                :       308.93
 Accuracy class 0            :      0.75152 +/-    0.0752203
 Accuracy class 1            :     0.564675 +/-    0.0977124
 AUC by Hand                 :     0.648288 +/-    0.0775428
 MSE                         :     0.692875 +/-    0.0626703
 LogLoss                     :      0.83394 +/-    0.0513109
 Test Cost per Example       :      77.5215 +/-      17.3678

Time Used                    :       0.2738 +/-    0.0758358
```

As we can see the combination has an accuracy and AUC of 0.738 and 0.676 respectively, which are significantly higher than the accuracy and AUC of the First and Best Solutions. However, we have obtained an archetype whose accuracy and AUC are quite closer to the combination, and is a single solution.

Finally, we have seen several methods to extract a single solution from the multi-tree: Occam-best, Testcost-best, a combination of both, and, finally, we have just seen the semantic archetype. It makes sense to be able to obtain a single solution that combines the Occam criterion, Testcost criterion and the semantic criterion. SMILES allows this through three options:

```
%-- Combination to Single Solution (Archetype) Use of Other Criteria
archetype similarity importance factor=15.0
archetype occam importance factor=4.0
archetype test cost importance factor=1.0
```

Considering these factor f1, f2 and f3, the exact formula for this combination is:

$$similarity^{f1} \cdot \left(1/numrules^{f2}\right)\left(1/test\,cost^{f3}\right)$$

These factors affect how the Archetype is extracted. If all the factors except similarity are left to 0, then it is just a semantic extraction

But if these factors are used, then we can extract an archetype taking into account its length and its testcost too.

This permits the user to extract of the best single solution according to the user's relevance: comprehensible and shortness (Occam), accuracy and AUC (similarity) and testcost (test cost factor).

For the previous example (monks2), if we use weights 15, 4 and 1, now we would have:

```
Mean Results:
  N. of susp. nodes explored :          100 +/-            0
  Solutions in the Multitree :  4.12571e+15 +/-  3.88401e+16
```

```
Results for 1st Solution:
 Accuracy of 1st Solution   :     0.687667 +/-    0.0535297
 AUC (by nodes) of 1st Sol  :     0.607398 +/-    0.0676029
 Mean # Rules               :       324.36
 Test Cost per Example      :      46.2625 +/-      6.58806
Results for Combination:
 Accuracy of Combination    :     0.737833 +/-    0.0522155
 AUC of Combination         :     0.676091 +/-    0.0692213
Results for Archetype:
 Accuracy of Archetype      :     0.715833 +/-    0.0527778
 AUC (by nodes) of Archetype:     0.647856 +/-    0.0750104
 Mean # Rules               :        318.3
 Test Cost per Example      :      57.2522 +/-       9.0557
Results for Best Solution:
 Accuracy of Best           :     0.685167 +/-    0.0593121
 Mean # Rules               :       308.93
 Accuracy class 0           :      0.75152 +/-    0.0752203
 Accuracy class 1           :     0.564675 +/-    0.0977124
 AUC by Hand                :     0.648288 +/-    0.0775428
 MSE                        :     0.692875 +/-    0.0626703
 LogLoss                    :      0.83394 +/-    0.0513109
 Test Cost per Example      :      77.5215 +/-      17.3678

Time Used                   :       0.2743 +/-    0.0758555
```

Now we can see that the archetype has lost some similarity to the combination (and consequently accuracy and AUC are reduced) but the number of rules and the testcost is also reduced, that is what we wanted, a single solution that takes into account the similarity with the combination, a short number of rules and a low testcost.

The weights assigned to each criteria are problem dependent, but according to the usual absolute amounts of each factor, the values 15, 4 and 1 represent a compromise of the three criteria. The user can change these values in order to give more relevance to each desired characteristic of the model.

3.17 Other Facilities

There are other options mainly related to output. They are self-explained by their description in the options file, although a more complete description can be found in Section 6:

```
%--output: syntax to show the rules:
show rules mode=functional-logic
%
%--output class dist.: show the trainset examples falling in each rule.
show class distribution=no
%show class distribution=yes (not implemented)
%
%--show all multitree rules
show all multitree rules=no show
%show all multitree rules=show
%show all multitree rules=to file
%
%--show all k-best solutions
```

```
show all k-best solutions=no show
%show all k-best solutions=show
%show all k-best solutions=to file
%
%--show component matrix of solutions
show solutions components=no show
%show solutions components=show
%
%--show confusion matrix
%show confusion matrix=no show
%show confusion matrix=show only if costs
show confusion matrix=show
%
%--how to show statistics
%show statistics=absolute statistics
%show statistics=relative statistics
%show statistics=both statistics
show statistics=just accuracy
%
%--show number of possible solutions in the multitree
show number of multitree possible solutions=yes
%show number of multitree possible solutions=no
```

At the present version of SMILES there are no other options (from the ones already explained) that could be modified through the options file. We will introduce in Section 5 other hardwired options for programmers or expert users.

4 SMILES Expertise

In this section, we take a more practical view on how to use the options described in the previous section, and we explain, mainly through experiments, when it is better to use some options over others.

4.1 Experimental comparison of splitting criteria

One of the things that affect the resulting accuracy of a decision tree is the splitting criterion used for learning.

First of all, we are going to study the splitting criteria wrt. two-class problems and then with multi-class problems. We use 25 datasets extracted from the UCI repository [2]. All of them have two classes, either originally or by selecting one of the classes and joining all the other classes. Table 1 shows the dataset (and the class selected in case of more than two classes), the size in number of examples, the nominal and numerical attributes and the percentage of examples of the minority class.

Table 1. Datasets used for the experiments.

#	DATASET	SIZE	ATTRIBUTES		%MIN
			NOM	NUM	CLASS
1	MONKS1	566	6	0	50
2	MONKS2	601	6	0	34.28
3	MONKS3	554	6	0	48.01

4	TIC-TAC	958	8	0	34.66
5	HOUSE-VOTES	435	16	0	38.62
6	AGARICUS	8124	22	0	48.2
7	BREAST-WDBC	569	0	30	37.26
8	BREAST-WPBC	194	0	33	23.71
9	IONOSPHERE	351	0	34	35.9
10	LIVER	345	0	6	42.03
11	PIMA	768	0	8	34.9
12	CHESS-KR-VS-KP	3196	36	0	47.78
13	SONAR	208	0	60	46.63
14	BREAST-CANCER	683	0	9	34.99
15	HEPATITIS	83	14	5	18.07
16	THYROID-HYPO	2012	19	6	6.06
17	THYROID-SICK-EU	2012	19	6	11.83
18	TAE [{0}]	151	2	3	32.45
19	CARS [{UNACC}]	1728	6	0	29.98
20	NURSERY [{NR}]	12960	8	0	33.33
21	PENDIGITS [{0}]	10992	0	16	10.4
22	PAGE-BLOCKS [{0}]	5473	0	10	10.23
23	YEAST [{ERL}]	1484	0	8	31.2
24	LETTER [{A}]	20000	0	16	3.95
25	OPTDIGITS [{0}]	5620	0	64	9.86

First we compare the most commonly used splitting criteria: Gain Ratio (only considering splits with at least average gain as is done in C4.5), Gini (as used in CART), DKM and Expected Error.

Table 2. AUC values for different splitting criteria.

SET	GAIN RATIO	GINI	DKM	EERR
1	81.5 ± 14.0	79.8 ± 11.9	79.8 ± 11.9	82.2 ± 5.3
2	60.6 ± 10.4	57.7 ± 8.4	55.5 ± 7.9	69.8 ± 4.1
3	98.8 ± 1.6	98.7 ± 1.7	98.7 ± 1.7	95.4 ± 2.6
4	81.3 ± 8.0	80.6 ± 7.5	79.8 ± 8.1	76.4 ± 5.6
5	96.9 ± 2.5	96.9 ± 2.5	96.9 ± 2.5	96.9 ± 2.5
6	1 ± 0	99.9 ± 0.2	1 ± 0	1 ± 0.1
7	91.1 ± 6.6	90.9 ± 5.8	95.7 ± 5.3	93.6 ± 3.7
8	58.1 ± 24.4	66.4 ± 18.3	54.9 ± 18.6	51.2 ± 3.5
9	88.8 ± 10.2	56.1 ± 13.6	90.8 ± 5.0	59.0 ± 15.1
10	65.1 ± 6.7	63.4 ± 8.2	65.6 ± 8.4	59.9 ± 9.4
11	78.0 ± 5.2	27.8 ± 3.5	69.3 ± 25.7	30.5 ± 39.8
12	99.7 ± 0.4	99.3 ± 0.4	99.7 ± 0.3	98.3 ± 0.8
13	60.6 ± 10.2	69.7 ± 10.4	72.7 ± 6.8	68.1 ± 12.8
14	95.5 ± 2.5	95.2 ± 2.7	96.8 ± 2.1	94.8 ± 2.9
15	92.9 ± 12.4	65.4 ± 24.4	72.9 ± 26.3	65 ± 24.2
16	83.2 ± 16.5	48.6 ± 51.2	96.9 ± 5.7	34.8 ± 41.1
17	93.6 ± 3.2	49.7 ± 46.1	65.8 ± 45.5	3.7 ± 11.3
18	50.5 ± 25.9	48.9 ± 27.1	52.5 ± 24.5	21.5 ± 21.4
19	98.1 ± 0.7	98.2 ± 0.8	98.1 ± 0.8	97.8 ± 1.1
20	1 ± 0	1 ± 0	1 ± 0	1 ± 0
21	99.7 ± 0.6	98.2 ± 0.7	99.7 ± 0.3	96.3 ± 2.1
22	93.7 ± 3.7	81.7 ± 4.9	66.6 ± 21.6	50 ± 0
23	73.7 ± 3.1	66.6 ± 9.9	73.5 ± 4.3	51.0 ± 4.0
24	98.7 ± 1.0	95.9 ± 2.4	99.4 ± 0.5	85.7 ± 0.5
25	98.1 ± 2.3	95.9 ± 3.3	98.0 ± 2.6	96.0 ± 3.3
M	85.53	77.26	83.19	71.12

Although all methods behave very similarly in terms of accuracy (as has been shown in the machine learning literature and by our own experiments not listed here), the differences in AUC are very noticeable, especially in datasets 9, 11, 15, 16, 17, 22, 23. There is no apparent relationship with any dataset characteristic except the minority class proportion, which will be analysed at the end of this section.

The worst methods according to the AUC measure are clearly Gini and Expected Error. Better and more similar results are given by GainRatio and DKM. If we select Gain Ratio as the best classical method, we can compare its results with AUCsplit results. In order to make comparisons significant, we have repeated 10-fold cross validation 10 times, making a total of 100 learning runs for each pair of dataset and method. These new results are shown in Table 3.

Table 3. Accuracy and AUC for Gain Ratio and AUCsplit.

	GAIN RATIO		AUCSPLIT		BETTER?	
SET	ACC.	AUC	ACC.	AUC	ACC.	AUC
1	90.7±6.6	83.6±11.8	96.5±3.9	94.3±6.7	✓	✓
2	57.7±6.5	61.1±7.9	56.0±6.2	56.7±8.0	x	x
3	97.6±7.8	97.4±8.5	99.1±1.1	99.1±1.4	✓	✓
4	78.9±4.6	79.8±7.2	77.6±4.7	76.9±6.5	x	x
5	95.8±2.6	95.2±3.1	95.8±2.6	95.2±3.1		
6	1±0	1±0	1±0	1±0		
7	92.5±4.1	91.5±6.1	92.9±3.7	94.7±4.6		✓
8	72.1±10.2	61.3±16.9	69.5±10.6	59.3±16.2	x	
9	92.0±4.7	90.4±7.0	89.6±5.0	89.7±6.7	x	
10	62.6±8.8	64.2±10.6	64.0±9.0	65.8±10.1		
11	73.3±5.7	76.6±6.9	72.5±5.1	76.7±6.0		
12	99.1±2.3	99.5±1.6	99.2±0.6	99.5±0.6		
13	68.2±10.2	67.4±11.9	71.0±10.4	73.6±11.0	✓	✓
14	95.4±2.5	96.3±2.5	96.2±2.5	97.6±2.1	✓	✓
15	86.4±14.2	85.1±17.9	83.4±14.0	63.5±22.3		x
16	98.0±10.9	84.6±13.1	98.6±0.8	94.8±5.6	✓	✓
17	95.2±1.4	92.6±3.5	96.7±1.2	95.1±3.1	✓	✓
18	71.4±12.4	61.5±20.8	68.9±11.6	59.8±21.3		
19	95.0±1.8	98.2±0.9	94.8±1.9	98.1±1.0		
20	1±0	1±0	1±0	1±0		
21	99.6±0.3	99.6±0.5	99.6±0.2	99.4±0.6		
22	96.8±0.9	93.3±4.7	96.8±0.2	95.1±6.9		✓
23	70.4±3.9	72.2±4.9	71.1±3.6	73.3±4.0		✓
24	99.5±0.2	98.9±1.4	99.5±0.1	99.3±0.7	✓	✓
25	98.9±1.8	94.2±19.4	99.5±0.3	98.5±1.8	✓	✓
M.	87.49	85.78	87.55	86.24		

Table 3 lists the accuracy of the chosen labelling and the AUC values of the whole set of optimal labellings. The first thing that can be observed is that the differences in accuracy are smaller than in AUC. In some cases it happens that Gain Ratio is better than AUCsplit in terms of accuracy, but not significantly in terms of AUC.

Since means of different datasets are illustrative but not reliable we compare dataset by dataset if one method is better than the other. The 'Better?' column represents if AUCsplit

behaves better (✓) or worse (x) than Gain Ratio. These marks are only shown when the differences are significant according to the *t*-test with level of confidence 0.1. This gives 8 wins, 13 ties and 4 loses for accuracies and 11 wins, 11 ties and 3 loses for AUC.

Now let us analyse the splitting criteria used for more than two classes. We have performed experiments with the same methodology as before for 14 multi-class datasets with the following characteristics:

Table 4. Datasets used for the experiments.

#	DATASET
1	balance
2	Cars
3	derm
4	echocardiogram
5	newt
6	nursery_3c
7	page
8	pendigits
9	tae
10	iris
11	opt-digits
12	sat
13	segmentation
14	wine

We examine now the results with the GainRatio, M-AUCsplit, MSEsplit, LogLsplit and the GINI criterion, without and with pruning, also showing the results for the 25 two-class problems and the 14 multi-class problems:

NOPRUNING

Two-class

GEOMEANS	GainRatio	M-AUCsplit	MSEsplit	LogLsplit	GINI
Accuracy of Best	0,864903	0,865027	0,867024	0,867124	**0,868102**
M-AUC	0,888052	0,884429	**0,889705**	0,887637	0,872447
Rules	59,98738	58,18974	**53,35602**	56,46755	108,2007
Time Used	0,711908	0,573635	0,538084	**0,535376**	1,646817

Multi-class

GEOMEANS	GainRatio	MAUCsplit	MSEsplit	LogLsplit	GINI
Accuracy of Best	0,833811	0,817345	**0,838144**	0,834983	0,8343
M-AUC	0,911666	0,896301	0,911093	**0,912881**	0.9074
Rules	144,9268	213,5072	**130,7308**	162,9093	257,69

PRUNING

Two-class

GEOMEANS	GainRatio	M-AUCsplit	MSEsplit	LogLsplit	GINI
Accuracy of Best	**0,874515**	0,871911	0,870511	0,87014	0,8651
M-AUC	0,874204	**0,880809**	0,87984	0,879037	0.7901
Rules	23,27337	21,18863	22,99033	21,28136	**12,30577**

	Time Used	0,710292	0,556105	0,54099	**0,529881**	1,624467

Multi-class

GEOMEANS	GainRatio	MAUCsplit	MSEsplit	LogLsplit	GINI
Accuracy of Best	0,809011	0,802879	**0,831174**	0,809857	0,7969
M-AUC	0,893041	**0,901847**	0,900887	0,897177	0.8583
Rules	74,48508	75,61664	68,26054	82,5315	**42,54716**

While it seems that MAUC is the best one with pruning, and both MSE and LogLoss are quite good without pruning, it seems that the MSE is a quite good option for all the situations. It also gives very short trees. Only GINI with pruning gives shorter trees, but this may be related to the king of postpruning used (PEP pruning).

4.2 Comparison of criteria to extract a solution from the multitree

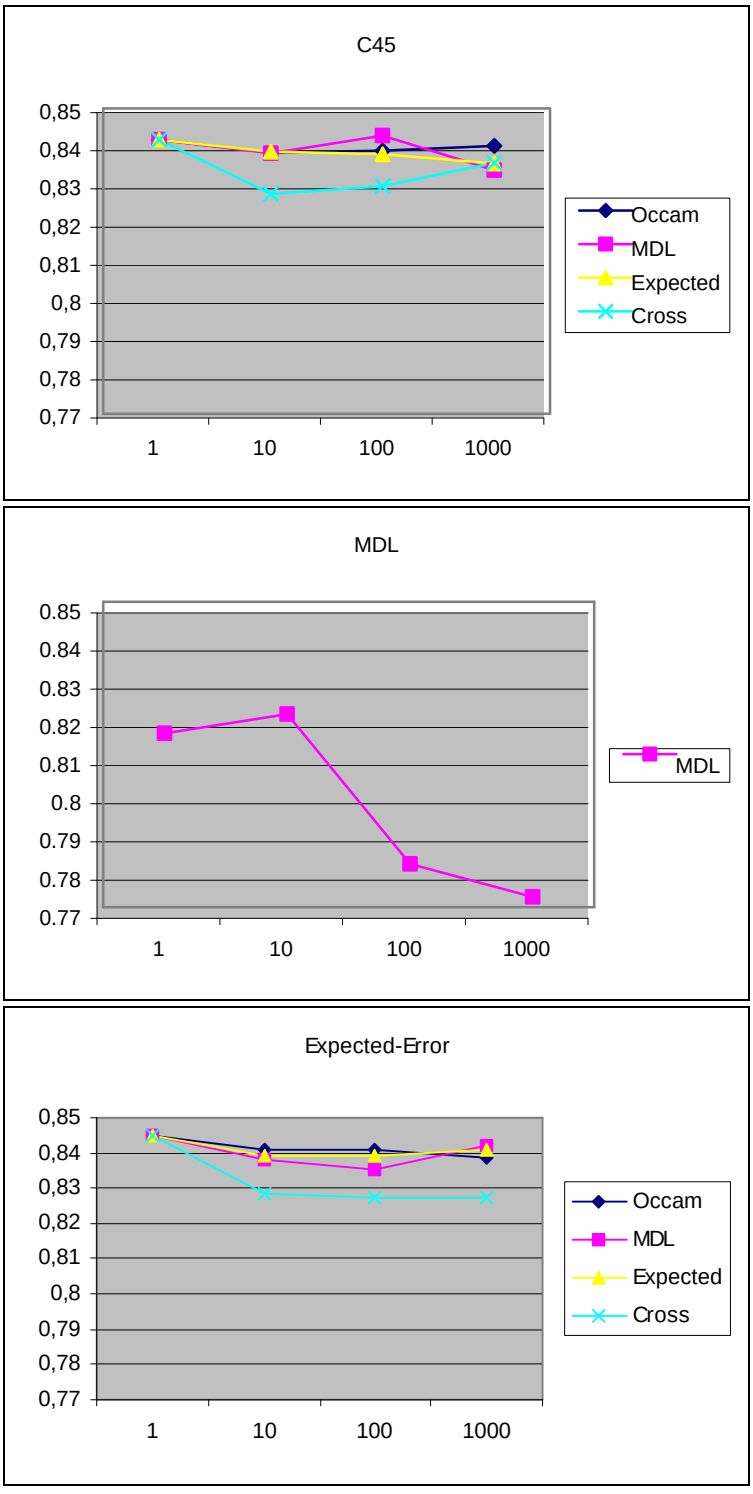
Once the multitree has been built there are several criteria to extract a solution from the multitree. In the current implementation of the system, these are the possible options:

```
%--multitree: best tree selection criterion
multitree best tree criterion=occam best
%multitree best tree criterion=test cost best
%multitree best tree criterion=occam and test cost best
%multitree best tree criterion=coverage best
%multitree best tree criterion=cross coverage
%multitree best tree criterion=expected error best
%multitree best tree criterion=split optimality best
%multitree best tree criterion=mdl best
```

We are not going to analyse the combinations with test cost. We are going to analyse all the rest.

The cross-coverage option splits the train data set into two independent parts and requires that the “same training set” option is active: the first part is used in the construction of the multitree, and the second part (named validation data set) is used for the selection of the solution. Concretely, we select the solution that has more accuracy w.r.t. the validation data set. We often use approximately the 20% of the original train data set for the validation data set and the rest 80% for the new train data set. Note that this technique is very useful to handle problems with a huge amount of examples and/or arguments.

We have made experiments on the behaviour of some of the previous criteria in the learning of 10 problems. These problems contain both numerical and continuous arguments and noisy data. The following figures show the average of the accuracy of the programs obtained with the C45, MDL, and Expected-Error splitting criteria to build the multitree depending on the size of the multitree and the technique applied for the selection of the solution. The number of solutions is varied from 1 to 1000.



With one solution we only can examine the splitting criterion. The best one, as we said, seems to be Expected-Error (although the difference is not significant). The performance of the four techniques of selecting the solution is very similar apart from the bad behaviour of MDL using MDL as splitting criterion. When C45 or Expected-Error criteria are used for the split criterion, a

further population of the multitree does not give the impression to improve the accuracy. Cross-coverage seems to decrease.

This may suggest that further study on the best tree selection method must be performed. However, it must be said that these results are obtained with RivalRatio instead of Topmost, being the latter much better to increase accuracy. The next section tries to give more light on this issue.

4.3 Evolution of Best Solution Accuracy for Increasing Number of trees

One of the great advantages of SMILES is that the multitree can be further explored to obtain better solutions. However, is it always true that further populating the multitree gets better and better solutions?

The next figure shows the number of rules and accuracy for increasing number of solutions. From these solutions just one comprehensible solution is obtained with the "best tree"="Occam", i.e., the shortest solution is selected. All the results are obtained for "second tree opening criterion" = "rival ratio". Moreover, pre-pruning is active and split MDL)

Numtree	1		10		100		1000	
Example	Rules	Accuracy	Rules	Accuracy	Rules	Accuracy	Rules	Accuracy
cars	126	85.53	126	85.53	101	85.65	69	84.03
house-votes	71	86.70	71	86.70	53	93.11	49	89.90
tic-tac-toe	346	65.55	297	70.35	263	75.99	252	74.94
nursery	471	91.34	467	91.37	408	91.77	364	92.37
monks1	17	94.90	17	94.90	7	100	7	100
monks2	100	69.90	97	69.90	89	79.16	61	79.62
monks3	35	88.42	35	88.42	28	87.26	22	88.19
drugs	134	92.09	132	92.90	131	92.72	129	93.00
tae	41	57.33	40	60.00	38	60.00	37	61.33
mean		**81.31**		**82.23**		**85.07**		**84.82**

Results with SMILES 0.5. Second Rival Ratio and Occam Best

It is clear (as expected) that if the best tree selection criterion (Occam) tries to select the shortest solution, the number of rules cannot increase and, it is shown, it usually decreases. The results are also generally positive for accuracy. Except in two cases, the accuracy with 1000 is higher than with 1. One interesting thing that can be observed is that, in the average, the maximum is not obtained with 1000 but with 100. This may suggest that there is a point from which Occam criterion selects much too short solutions.

As we said before better results may be obtained with other second-best criteria, such as topmost.

Let us make a comparison using more typical options, such as "MSE splitting criterion", "no pruning", "second tree opening criterion=random", and let us study the shortest solution again (i.e. Occam), with the current version of SMILES, but now with more datasets:

#	Dataset	Size	Classes	Nom.Attr.	Num.Attr.
1	Balance-scale	625	3	0	4
2	Cars	1728	4	5	0
3	Dermatology	358	6	33	1
4	Ecoli	336	8	0	7
5	Iris	150	3	0	4
6	House-votes	435	2	16	0
7	Monks1	566	2	6	0
8	Monks2	601	2	6	0
9	Monks3	554	2	6	0
10	New-thyroid	215	3	0	5
11	Post-operative	87	3	7	1
12	Soybean-small	35	4	35	0
13	Tae	151	3	2	3
14	Tic-tac	958	2	8	0
15	Wine	178	3	0	13

	1	10	100	1000
#	1st	Occ	Occ	Occ
1	76.82	76.81	76.74	76.77
2	89.01	89.03	89.08	89.11
3	89.80	90.09	90.20	90.69
4	77.55	77.79	78.36	77.48
5	93.63	93.93	93.63	93.93
6	94.67	94.73	94.40	94.80
7	92.25	96.47	100.00	100.00
8	74.83	73.75	72.53	70.43
9	97.55	97.60	97.55	97.62
10	92.62	92.57	92.95	93.67
11	60.88	60.00	62.25	62.13
12	97.25	97.50	96.75	96.25
13	62.93	61.93	62.07	61.13
14	78.22	78.27	78.58	79.47
15	93.12	92.94	93.24	94.41
gmeans	83.87	83.93	84.28	84.19

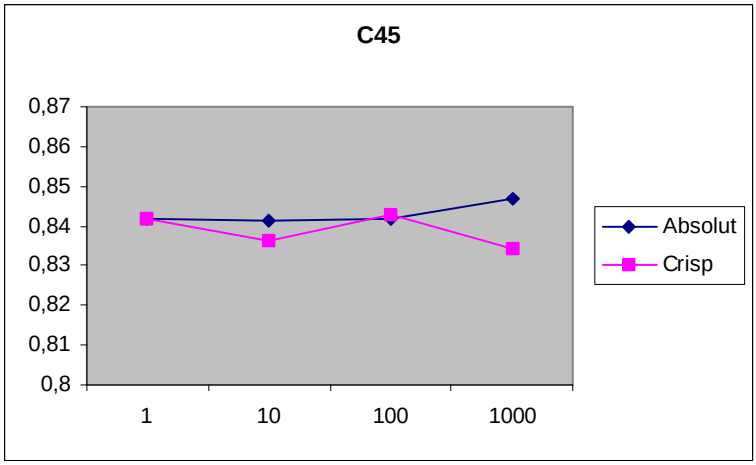
Now we can see that this increase arrives to a saturation point (in general around 100) and then begins to decrease slowly. As we will see, without the use of combination or archetype, we do not fully exploit the possibilities of the multitree.

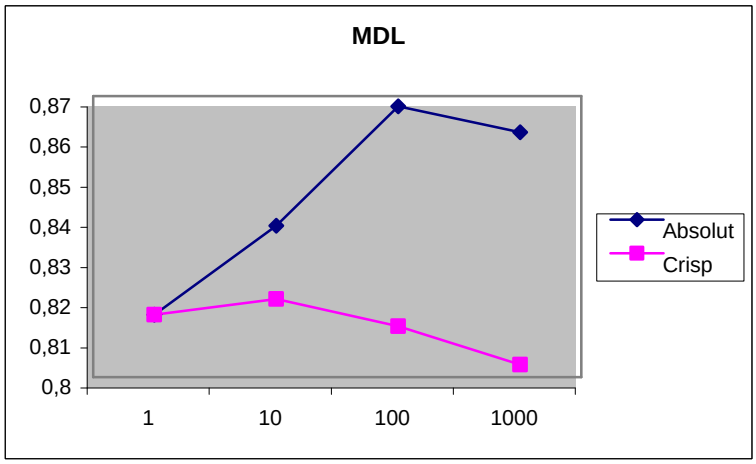
4.4 Comparison of Combination

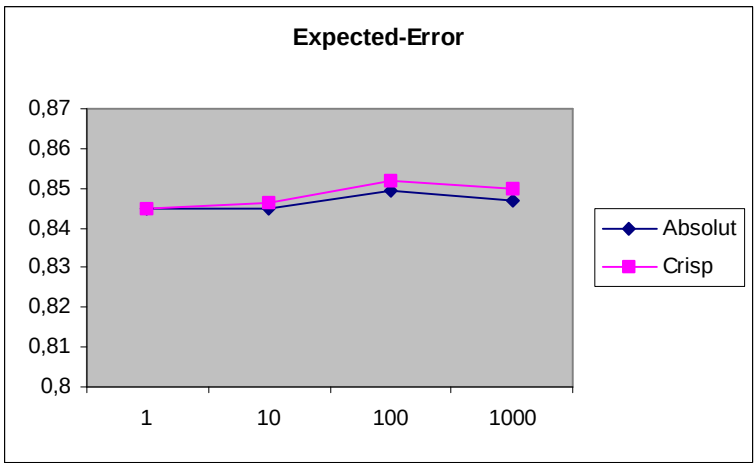
In this section, we present some results on the combination of the hypotheses once the multitree has been created. The possible options are:

```
%-- Combination: How to combine several solutions
%multitree solution combination=no combination
%multitree solution combination=cross coverage combination
%multitree solution combination=majority crisp
multitree solution combination=majority absolute stochastic
%multitree solution combination=majority relative stochastic
%multitree solution combination=majority cost stochastic
```

We are going to explore the two ways with better results: *majority crisp* and *majority absolute stochastic*. The following pictures compare these two methods on the learning of 10 problems. The figures present the average of accuracy obtained using the two methods depending on the size of the multitree and the split criterion. All the results are obtained for “second tree opening criterion” = “rival ratio”.







There are some interesting points to conclude from these results:

- The use of hypotheses combination usually improves the accuracy although it is important to note that the comprehensibility of the solution is lost.

- The improvement of accuracy does not grow linearly with the size of the multitree. A multitree populated in excess might even affect negatively in the accuracy because later solutions may have lower accuracy than the first ones.
- The majority_absolute_stochastic technique seems to obtain better results than majority_crisp.
- The best improvement has been obtained with the MDL split criterion, although this criterion has worse results initially.

Finally, we analyse in more detailed the results for one splitting criterion (MDL) and *majority_absolute_stochastic* combination criterion.

Numtree	1	10	100	1000
Example	Accuracy	Accuracy	Accuracy	Accuracy
cars	85.53	85.53	90.16	92.12
house-votes	86.69	88.99	92.66	92.20
tic-tac-toe	65.55	82.46	83.71	85.39
nursery	91.34	91.45	92.98	93.98
monks1	94.90	94.90	100	91.90
monks2	69.90	69.91	79.17	79.63
monks3	88.43	90.05	92.59	86.80
drugs	92.09	92.45	93.09	93.09
tae	57.33	57.33	57.33	58.67
mean	**81.31**	**83.67**	**86.85**	**85.97**

Figure 2. Results with SMILES 0.5. Second Rival Ratio and Combination (pre-pruning active)

If we compare these results with the results with the selection of one solution with Occam best criterion, we see that combination results are slightly better (around a 1% better in accuracy). In our opinion, only in quite limited situations is preferable to have this slight increase in accuracy with the loss of comprehensibility that combined solutions have.

Finally, although we are not showing results, we realised that in order to improve accuracy for combination, pruning must not be used with combination when the multitree number is high (>100).

4.5 Fusion Methods

To study the different fusion methods, we are going to use the following datasets:

#	Dataset	Size	Classes	Nom.Attr.	Num.Attr.
1	Balance-scale	625	3	0	4
2	Cars	1728	4	5	0
3	Dermatology	358	6	33	1
4	Ecoli	336	8	0	7
5	Iris	150	3	0	4
6	House-votes	435	2	16	0
7	Monks1	566	2	6	0
8	Monks2	601	2	6	0
9	Monks3	554	2	6	0
10	New-thyroid	215	3	0	5
11	Post-operative	87	3	7	1
12	Soybean-small	35	4	35	0
13	Tae	151	3	2	3
14	Tic-tac	958	2	8	0
15	Wine	178	3	0	13

Datasets used in the experiments

For the following experiments, we used GainRatio as splitting criterion and we chose a random method for populating the shared ensemble (after a solution is found, a suspended OR-node is woken at random). Pruning is not enabled.

Since there are many sources of randomness, we performed the experiments by averaging 10 results of a 10-fold cross-validation. This makes a total of 100 runs for each pair composed of a method and a dataset.

The following table shows the mean accuracy and the standard deviation using the different fusion techniques introduced in Section 3.6 for each dataset. We summarise the results with the geometric means for each technique. The techniques studied are *sum*, *product*, *maximum*, *minimum*, and *arithmetic mean*, all of which use the original vectors. In the table, we do not include the experiments with *geometric mean* because they are equivalent to the results of *product*. The multi-tree was generated by exploring 100 suspended OR-nodes, thus giving thousands of possible hypotheses (with much less required memory than 100 non-shared hypotheses). According to the experiments, the best fusion technique was *maximum*. Thus, we will use this fusion method to study the effect of applying the transformations on the vector.

	Arit.		Sum.		Prod.		Max.		Min.	
#	Acc.	Dev.	Acc.	Dev.	Acc.	Dev.	Acc.	Dev.	Acc.	Dev.
1	80.69	5.01	81.24	4.66	76.61	5.04	83.02	4.76	76.61	5.04
2	91.22	2.25	91.25	2.26	83.38	3.65	90.90	2.09	83.38	3.65
3	94.17	4.06	94.34	3.87	89.06	5.19	94.00	4.05	89.06	5.19
4	80.09	6.26	79.91	6.13	76.97	7.14	80.09	6.11	76.97	7.14
5	95.63	3.19	95.77	3.18	93.28	3.71	95.93	2.81	93.28	3.71
6	94.53	5.39	94.20	5.66	94.00	5.34	94.47	5.45	94.40	5.34
7	99.67	1.30	99.71	1.18	81.00	8.60	99.89	0.51	81.00	8.60
8	73.35	5.86	73.73	5.82	74.53	5.25	77.15	5.88	74.53	5.25
9	97.87	2.00	97.91	1.80	97.58	2.45	97.62	1.93	97.58	2.45
10	94.52	4.25	93.76	5.10	92.05	5.71	92.57	5.43	92.05	5.71
11	62.50	16.76	63.25	16.93	61.63	17.61	67.13	14.61	61.63	17.61
12	97.50	8.33	97.50	9.06	97.75	8.02	94.75	11.94	97.75	8.02
13	63.60	12.59	64.33	11.74	62.00	12.26	63.93	12.03	62.00	12.26
14	81.73	3.82	82.04	3.78	78.93	3.73	82.68	3.97	78.93	3.73
15	94.06	6.00	93.88	6.42	91.47	7.11	92.53	6.99	91.47	7.11
Geomean	85.83	4.72	85.99	4.71	82.53	5.93	86.40	4.52	82.55	5.93

Comparison between fusion techniques

The next table illustrates the results for accuracy using the original vector and the good loser, bad loser, majority and difference transformations. According to these experiments, all transformations get very similar results, except from majority. We will use the combination "max + difference" in the following experiments.

	Max + Orig		Max + Good		Max + Bad		Max + Majo.		Max + Diff.	
#	Acc.	Dev	Acc.	Dev	Acc.	Dev	Acc.	Dev	Acc.	Dev
1	83.02	4.76	83.02	4.76	83.02	4.76	67.84	6.61	83.02	4.76
2	90.90	2.09	90.90	2.09	90.90	2.09	81.48	3.22	90.90	2.09
3	94.00	4.05	94.00	4.05	94.00	4.05	79.97	7.98	94.00	4.05
4	80.09	6.11	80.09	6.11	80.09	6.11	78.21	6.07	80.09	6.11
5	95.93	2.81	95.93	2.81	95.93	2.81	89.44	4.84	95.93	2.81
6	94.47	5.45	94.47	5.45	94.47	5.45	91.47	6.90	94.47	5.45
7	99.89	0.51	99.89	0.51	99.89	0.51	77.58	6.29	99.89	0.51
8	77.15	5.88	77.15	5.88	77.15	5.88	83.42	5.06	77.15	5.88
9	97.62	1.93	97.62	1.93	97.62	1.93	90.40	4.02	97.62	1.93
10	92.57	5.43	92.57	5.43	92.57	5.43	89.14	6.74	92.57	5.43
11	67.13	14.61	67.13	14.61	67.13	14.61	68.25	15.33	67.00	14.60
12	94.75	11.94	94.75	11.94	94.75	11.94	50.75	28.08	94.75	11.94
13	63.93	12.03	63.87	12.14	63.93	12.03	60.93	11.45	65.13	12.53
14	82.68	3.97	82.68	3.97	82.68	3.97	68.26	4.35	82.68	3.97
15	92.53	6.99	92.53	6.99	92.53	6.99	78.41	11.25	92.53	6.99
Gmean	86.40	4.52	86.39	4.53	86.40	4.52	76.11	7.19	86.49	4.54

Comparison between vector transformation methods

4.6 Combination Accuracy as Multi-tree is Bigger

Let us study now the influence of the size of the multi-tree, varying from 1 to 1,000 explored OR-nodes. The results have been obtained with the combination "max + difference". The next table shows the accuracy obtained using the shared ensembles depending on the number of OR-nodes opened.

	1		10		100		1000	
#	Acc.	Dev.	Acc.	Dev.	Acc.	Dev.	Acc.	Dev.
1	76.82	4.99	77.89	5.18	83.02	4.76	87.68	4.14
2	89.01	2.02	89.34	2.20	90.90	2.09	91.53	2.08
3	90.00	4.72	91.43	4.67	94.00	4.05	94.00	4.05
4	77.55	6.96	78.58	6.84	80.09	6.11	80.09	6.11
5	93.63	3.57	94.56	3.41	95.93	2.81	95.56	2.83
6	94.67	5.84	94.27	5.69	94.47	5.45	95.00	5.14
7	92.25	6.27	96.45	4.15	99.89	0.51	100.00	0.01
8	74.83	5.17	75.33	5.11	77.15	5.88	82.40	4.52
9	97.55	1.89	97.84	1.86	97.62	1.93	97.75	1.92
10	92.62	5.22	93.43	5.05	92.57	5.43	90.76	5.89
11	60.88	17.91	63.00	15.88	67.00	14.60	68.13	15.11
12	97.25	9.33	96.00	10.49	94.75	11.94	95.50	10.88
13	62.93	12.51	65.00	12.19	65.13	12.53	65.33	12.92
14	78.22	4.25	79.23	4.03	82.68	3.97	84.65	3.34
15	93.12	6.95	93.29	6.31	92.53	6.99	92.99	5.00
Gmean	83.88	5.52	84.91	5.30	86.49	4.54	87.47	4.47

Influence of the size of the multitree

The results indicate that the greater the population of the multi-tree the better the results of the combination are. A saturation point is not arrived in most of the datasets (at least for 1000 open second trees).

4.7 The relevance of Second Tree Opening Criterion

Some of the previous results were obtained for “second tree opening criterion” = “rival ratio”. and some other with “second tree opening criterion” = “random”. This was the criterion used in the first versions of SMILES.

However, later on, we designed new second tree opening criteria and realised that many other criteria behave much better that “rival ratio”. As we have shown in Section 3.5, these are the possible other options:

```
%--second tree opening: how to select the 2ond node to explore
%second tree opening=split optimality
%second tree opening=optimality rival ratio
%second tree opening=optimality rival ratio depth
%second tree opening=optimality rival ratio component
%second tree opening=optimality rival ratio component random
second tree opening=second topmost
%second tree opening=second bottommost
%second tree opening=second random
%second tree opening=second random depth
```

The first results were surprising. Simple criteria such as TopMost or Random were much better than rival ratio with lower number of trees, both for select best solution (Occam) or for combination. We did the experiments with just one dataset: cars.

Method	1	10	100	500	1000	5000	10000
Rival Ratio - Occam Best cars1-	0.894	0.894	0.898	0.896	0.896	0.898	0.898
TopMost - Occam Best cars1-	0.894	0.883	0.883	0.884	0.884	0.889	0.890
Bottom Most- Occam Best cars1-	0.894	0.890	0.902	0.894	0.898	0.895	0.895
Rival Ratio - Combination cars1-	0.894	0.902	0.896	0.897	0.904	0.925	0.933
BottomMost - Combination cars1-	0.894	0.903	0.899	0.890	0.894	0.918	0.907
TopMost - Combination cars1-	0.894	0.925	0.934	0.934	0.931	0.925	0.913
Random - Combination cars1-	0.894	0.895	0.912	0.924	0.92	0.911	0.91

Figure 3. Effect on accuracy depending on several second tree opening methods

Figure 5 shows that the best single solution results are obtained for BottomMost at 100 with Occam: 0.902 accuracy, whereas for combination the topmost at 100 also gets the maximum with TopMost: 0.934. Rival Ratio seems to have the maximum later, at 10000 opened trees.

If we restrict to just combination and 1000 second trees for different datasets, we have a clearer portrait of the several second tree opening methods (RivalRatio, Front (LIFO), Back (FIFO), BottomMost, TopMost and Random), as we can see in the following table:

Dataset	RivalR	time	Front	time	Back	time	Bottm	time	TopMost	time	Rand.	time
cars	0.904	0.69	0.932	22.9	0.904	0.6	0.894	0.56	0.931	83.84	0.920	4.8
nursery	0.951	3.4	0.967	101.2	0.926	1.0	0.962	0.92	0.970	13245	0.967	21.3
tae	0.6	3.8	0.6	84.5	0.6	0.3	0.6	0.34	0.613	316.8	0.613	28.9
monks1	0.991	0.7	1	5.4	0.97	0.34	0.956	0.26	1	12.2	1	2.1
monks2	0.734	1.58	0.771	18.64	0.773	0.31	0.724	0.26	0.773	54.1	0.773	4.6
monks3	0.919	0.94	0.949	5.39	0.891	0.26	0.926	0.27	0.947	12.3	0.947	2.3
means	0.850	1.85	0.870	39.7	0.844	0.47	0.844	0.44	0.872	2287	0.870	10.7

Figure 4. Effect on combination accuracy depending on several second tree opening methods

The best results are obtained with TopMost. However, the required time is much higher than for the other methods. In fact, with TopMost we lose the shared parts between trees and the multitree method is similar to a forest method. After this result, it seems that the “Random” option obtains high accuracy relatively quick. “RivalRatio” also seems a compromise between accuracy and time. Note that these results depend, in the end, on the mean depth at which second tree openings are performed.

Finally, we can corroborate the previous results if we make a selection of methods and study them with variable number of trees (1 tree, RivalRatio 1000, TopMost 100 and 1000, and Random 100, 1000 and 10000):

Dataset	Sol-1	RiR1000	TM100	TM1000	R100	R1000	R10000
cars	0.824	0.904	0.934	0.931	0.912	0.920	0.907
nursery	0.924	0.951	0.969	0.970	0.948	0.967	0.969
tae	0.6	0.6	0.6	0.613	0.613	0.613	0.6
monks1	1	0.991	1	1	1	1	0.977
monks2	0.741	0.734	0.766	0.773	0.766	0.773	0.762
monks3	0.866	0.919	0.947	0.947	0.944	0.947	0.928
means	0.826	0.850	0.869	0.872	0.864	0.87	0.857

Figure 5. Effect on combination accuracy depending on several second tree opening methods

Again the best results are obtained with TopMost and, again, the required time is much higher than random. In fact, we have only been able to arrive to 10.000 for all datasets with the random option. However, the accuracy has not been increased further.

A more technical discussion about how all these methods work and more results can be found at [27].

4.8 Archetype Expertise

The archetype is one of the new and innovative features in SMILES that better take advantage of the multi-tree structure. With it, SMILES is able to obtain a highly accurate and, at the same time, comprehensible hypothesis.

We are going to illustrate how to generate good archetypes. For the experiments, we used GainRatio as splitting criterion. We chose a random method for populating the shared ensemble (after a solution is found, a suspended OR-node is woken at random) and we used the maximum fusion strategy for combination. As usual, we used several datasets from the UCI dataset repository. The following table shows the dataset name, the size in number of examples, the number of classes, the nominal and numerical attributes.

#	Dataset	Size	Classes	Nom.Attr.	Num.Attr.
1	monks1	566	2	6	0
2	monks2	601	2	6	0
3	monks3	554	2	6	0
4	tic-tac	958	2	8	0
5	house-votes	435	2	16	0
6	post-operative	87	3	7	1
7	balance-scale	625	3	0	4
8	soybean-small	35	4	35	0
9	dermatology	358	6	33	1
10	cars	1728	4	5	0
11	tae	151	3	2	3
12	new-thyroid	215	3	0	5
13	ecoli	336	8	0	7

Since there are many sources of randomness, we have performed the experiments by averaging 10 results of a 10-fold cross-validation. This makes a total of 100 runs (each one with a different multi-tree construction, random dataset and hypothesis selection process) for each pair of method and dataset.

In the experiments, we will use the following notation:

- First Solution: this is the solution given by just one hypothesis (the first hypothesis that is obtained). This is similar to C4.5.
- Combined Solution: this is the solution given by combining the results of the ensemble (in our case, the multi-tree, as described in the previous section).
- Archetype Solution: this is the single solution which is most similar to the combined solution.
- Occam Solution: this is the single solution with the lowest number of rules, i.e., the shortest solution.

It is not our purpose now to evaluate the improvement of the *Combined Solution* over the *First Solution* using shared ensembles. We have done that in previous section. We have not included the results using post-pruning because it does not improve the performance of any of the four kinds of solutions. Our goal is to show that a significant gain can be obtained from the *First*

Solution to the *Archetype* and *Occam* methods as long as the size of the ensemble increases. Another question to be answered is to determine which method to extract a single solution from an ensemble is better: *Archetype* or *Occam*.

The first thing we are going to study is the similarity metric. As we saw, there are three possibilities:

```
similarity method for combination to single=kappa
%similarity method for combination to single=kappa1
%similarity method for combination to single=qstat
```

The next table shows the accuracy for each pair composed of a dataset and a method and the geometric means for each method. The methods studied are First, Combined and Archetype. The latter uses three different similarity metrics kappa, theta (kappa2) and Q (qstat). The multi-tree has been generated exploring 100 suspended OR-nodes.

#	1st	Comb	Arc. κ	Arc. θ	Arc. Q
1	92.3	100	100	100	100
2	74.8	77.4	76.1	76.2	75.8
3	97.5	97.5	97.6	97.6	97.6
4	78.2	82.7	78.2	78.3	78.5
5	93.6	96.0	94.4	93.9	94.2
6	60.9	66.3	63.8	64.3	61.9
7	76.8	83.1	80.1	80.1	79.8
8	97.3	96.5	96.5	91.0	47.0
9	89.8	93.6	90.6	89.9	74.3
10	89.0	91.0	89.6	89.6	89.3
11	62.9	64.5	61.9	62.9	49.8
12	92.6	92.6	92.8	92.9	91.4
13	77.5	79.9	79.4	78.9	76.7
gmeans	82.41	85.45	83.78	83.45	76.24

As expected, hypothesis combination improves the accuracy w.r.t. the first single tree. The use of the archetype method also obtains good results. On the other hand, the results show that the Archetype method is very dependent on the measure of similarity used: kappa seems to be the best metric and Q the worst (it even obtains lower accuracy than the first single hypothesis).

The next thing we are going to study is the size of the invented dataset. Similarity is approximated through the use of an invented dataset. Let us study the influence of its size, varying from 10 to 100,000 examples. The similarity metric and the size of the multi-tree are fixed to kappa and 100 alternative opened OR-trees, respectively.

		10	100	1000	10000	100000
#	Comb	Arc	Arc	Arc	Arc	Arc
1	99.8	72.3	93.3	99.8	100	99.9
2	77.3	64.6	61.0	75.2	76.1	76.2
3	97.6	82.9	94.5	97.6	97.6	97.6
4	82.9	65.9	70.3	78.0	78.2	78.6
5	95.8	73.7	92.4	94.4	94.4	93.8
6	67.5	69.1	63.6	63.9	63.8	63.5
7	83.0	62.5	75.4	79.4	80.1	79.9
8	95.0	68.8	93.3	95.0	96.5	96.5
9	93.6	45.6	84.7	90.5	90.6	89.9
10	91.0	71.0	75.4	88.1	89.6	89.8
11	63.7	44.3	54.3	59.1	61.9	61.2
12	92.5	73.8	89.3	91.3	92.8	92.6
13	80.0	46.8	73.9	77.9	79.4	79.0
gmeans	85.36	63.57	77.40	82.88	83.78	83.57

The previous table shows that in order to obtain a good archetype hypothesis, the similarity metric has to be computed as accurately as possible. Although it depends on the dataset, a size of 10,000 invented examples seems to be sufficient.

Now, we are going to study the influence of the size of the multitree. The effect of the size of the multi-tree is evaluated in the following table.

	1	10				100				1000			
#	1st	Comb	Arc	Occ	#Sol	Comb	Arc	Occ	#Sol	Comb	Arc	Occ	#Sol
1	92.3	96.1	96.0	96.5	107	100	100	100	8.7×10^8	100	100	100	1.6×10^{19}
2	74.8	74.9	74.3	74.3	148	77.4	76.1	72.5	2.6×10^{10}	82.3	82.1	70.4	3.2×10^{20}
3	97.5	97.7	97.7	97.6	46	97.5	97.6	97.5	80×10^4	97.7	97.7	97.6	7.1×10^{14}
4	78.2	79.0	78.1	78.3	257	82.7	78.2	78.6	2.7×10^{12}	84.6	79.8	79.5	3.1×10^{38}
5	93.6	94.9	94.2	93.9	63	96.0	94.4	93.6	26×10^5	95.7	94.1	93.9	5.6×10^{11}
6	60.9	63.8	61.8	60.0	55	66.3	63.8	62.3	59674	68.5	65.9	62.1	2.1×10^9
7	76.8	77.9	77.2	76.8	131	83.1	80.1	76.7	3.4×10^8	88.0	83.5	76.8	1.2×10^{18}
8	97.3	97.0	98.0	97.5	23	96.5	96.5	96.8	38737	95.0	93.3	96.3	1.8×10^{18}
9	89.8	91.3	90.6	90.1	92	93.6	90.6	90.2	3.3×10^7	93.8	91.1	90.8	1.2×10^{10}
10	89.0	89.6	89.1	89.0	151	91.0	89.6	89.1	1.7×10^9	91.6	90.0	89.1	2.8×10^{24}
11	62.9	62.5	62.3	61.9	97	64.5	61.9	62.1	1.5×10^6	64.5	60.9	61.1	4.6×10^{14}
12	92.6	93.2	92.6	92.6	26	92.6	92.8	93.0	3392	90.7	92.6	93.7	6.1×10^7
13	77.5	79.1	77.6	77.8	57	79.9	79.4	78.4	1134750	80.3	78.2	77.0	3.8×10^8
gm.	82.41	83.49	82.85	82.55	78.31	85.45	83.78	82.91	4.3×10^7	86.44	84.49	82.65	6.2×10^{14}

In this table, we show the accuracy of the first single solution and the accuracy of the combination, the archetype solution and the Occam solution for multi-trees created by exploring 10, 100, and 1000 alternative OR-nodes[1]. We also include the geometric average number of solutions in the multi-tree (#Sol). Note that with 100 OR-nodes, we obtain millions of solutions with much less required memory than 100 non-shared hypotheses.

The results are quite encouraging: by simply exploring 10 OR-nodes, the archetype solution surpasses the first solution and the Occam solution. This difference is increased as long as the multi-tree is populated. This is mainly due to the improvement in the accuracy of the combined solution and the fact that the archetype hypothesis can actually get close to it. The Occam solution does not seem to be improved by larger multi-trees. Nevertheless, the Occam

[1] The experiments for datasets 9 and 13 have been performed exploring only 300 and 500 alternative OR-nodes, respectively.

hypothesis can also be regarded as a way to obtain more and more compact solutions without losing accuracy.

Finally, it should be said that we can combine the archetype with occam and with testcost criteria. This possibility has not been fully evaluated.

4.9 Forgetting Suspended Nodes

As we introduced in section 3.5, SMILES presents "suspended nodes forgetting" methods:

```
%--suspended nodes forgetting: must all suspended nodes maintained?
suspended nodes forgetting=maintain all
%suspended nodes forgetting=maintain const random
%suspended nodes forgetting=maintain log random
%suspended nodes forgetting=maintain log random with depth
%suspended nodes forgetting=maintain log random with squared depth
%suspended nodes forgetting=maintain log random with depth adjusted
```

The following table shows how the previous methods affect on accuracy, time and memory (all results are with C4.5 splitting criterion with smoothing, 100 trees and 10x10 cross-validation):

DATASET	**MAINTAIN:**	ALL	CONST RANDOM (2)	LOG RANDOM	LOG WITH DEPTH	LOG WITH DEPTH SQRT	LOG WITH DEPTH ADJ. (+8)
MONKS2	Memory	3900K	2400K	2300K	450K	850K	2500K
	Time (sec)	0.38	0.27	0.36	0.095	0.17	0.43
	First Sol.	74.83	74.83	74.83	74.83	74.83	74.83
	Comb	77.15	76.87	77.47	75.85	76.2	77.83
	Best	72.8	72.47	72.67	74.88	74.68	72.8
WINE	Memory	32000K	2100K	300K	1800K	2100K	1800K
	Time (sec)	3.30	2.43	0.83	0.84	2.91	3.25
	First Sol.	93.12	93.12	93.12	93.12	93.12	93.12
	Comb	92.53	92.76	93.05	92.77	92.35	93.17
	Best	93.29	93.24	93.12	93.53	93.29	93.24
BALANCE	Memory	10000K	3000K	3000K	4500K	3700K	3800K
	Time (sec)	1.30	0.52	72.07	2.24	1.56	1.65
	First Sol.	76.82	76.82	76.82	76.82	76.82	76.82
	Comb	83.01	81.47	82.14	85.24	84.92	85.03
	Best	76.87	76.55	76.55	76.76	76.79	76.72
POSTOPER	Memory	3500K	1800K	1800K	400K	1300K	2000K
	Time (sec)	0.33	0.19	0.24	0.12	0.36	0.46
	First Sol.	60.88	60.88	60.88	60.88	60.88	60.88
	Comb	67.0	66.25	66.88	65.25	67.63	67.5
	Best	60.88	60.88	60.88	60.75	62.75	62.75
GEOMEAN	**Memory**	**8130**	**2284**	**1389**	**1099**	**1712**	**2418**
	Time (sec.)	**0.86**	**0.5**	**1.51**	**0.38**	**0.73**	**1.01**
	First Sol.	**75.56**	**75.56**	**75.56**	**75.56**	**75.56**	**75.56**
	Comb	**79.38**	**78.76**	**79.33**	**79.09**	**79.73**	**80.32**
	Best	**75.08**	**74.91**	**74.94**	**75.6**	**76.12**	**75.61**

According to these experimental results, LOG WITH DEPTH is the most economical option, either in memory and in time, and results are not significantly deteriorated. If time is not a

problem (and just memory), LOG WITH DEPTH ADJUSTED is also a good option because it even increases accuracy in some cases. However, more datasets and types of combination should be studied in order to make a more reliable conclusion about these options.

4.10 Comparison with other systems

To conclude this section we compare the more popular thing of machine learning algorithms: their top accuracy. Although we have argued that comparing AUC, comprehensibility, test costs or other issues is at least equally important, accuracy and resources are usually used in the literature to compare some systems.

The following results use SMILES with a C4.5 splitting criterion, random population of the multi-tree, original + max fusion and no pruning.

	Bagging			Boosting			Multi-tree		
#	10	100	300	10	100	300	10	100	300
balance-scale	82.24	82.76	82.87	78.72	76.00	75.60	77.89	83.02	85.50
cars	93.89	94.36	94.29	95.92	97.07	97.15	89.34	90.90	91.53
dermatology	96.40	97.19	97.30	96.62	96.51	96.65	91.43	94.00	95.71
ecoli	83.90	85.15	85.56	83.66	84.29	84.20	78.58	80.09	79.64
house-votes	95.40	95.73	95.77	95.19	95.24	95.42	94.56	95.93	96.21
iris	94.20	94.27	94.53	94.20	94.53	94.53	94.27	94.47	94.47
monks1	99.95	100.00	100.00	99.46	99.46	99.46	96.45	99.89	100.00
monks2	65.52	67.51	67.94	76.67	82.17	83.40	75.33	77.15	79.37
monks3	98.76	98.88	98.88	97.96	97.92	97.92	97.84	97.62	97.65
new-thyroid	94.33	94.66	94.81	94.98	95.31	95.22	93.43	92.57	92.71
post-operative	63.11	64.89	64.78	59.67	59.00	59.00	63.00	67.00	67.75
soybean-small	97.75	97.95	97.95	97.95	97.95	97.95	96.00	94.75	95.75
tae	60.10	61.11	61.05	64.81	64.81	64.81	65.00	65.13	65.40
tic-tac	83.06	84.05	83.91	82.11	82.62	82.55	79.23	82.68	83.72
wine	94.90	95.90	96.35	95.90	96.85	96.57	93.29	92.53	91.94
GeoMean	85.77	86.59	87.03	86.61	86.99	86.70	84.91	86.49	87.16

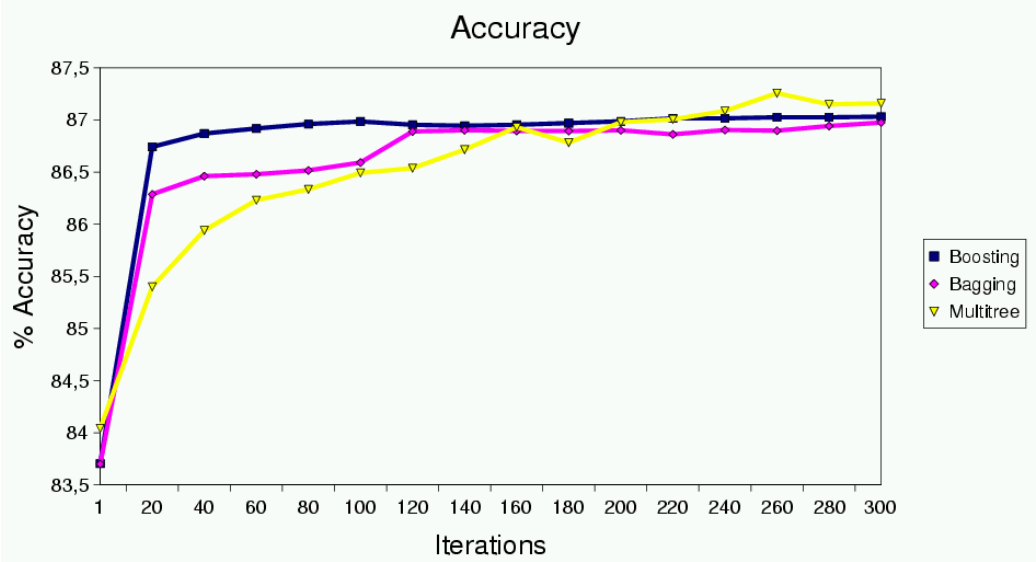
Accuracy comparison between ensemble methods.

The previous table presents a comparison of accuracy between our system (multi-tree), boosting and bagging, depending on the number of iterations. We have employed the Weka implementation (http://www.cs.waikato.ac.nz/~ml/weka/) of these two ensemble methods.

For all the experiments we have used GainRatio as splitting criterion, and we have chosen a simple random method for populating the multi-tree and a fusion strategy based on selecting the branch that gives a maximum cardinality for the majority class.

The datasets have been extracted from the UCI repository [2]. The experiments were performed with a Pentium III-800Mhz with 180MB of memory running Linux 2.4.2. Since there are many sources of randomness, we have performed the experiments by averaging 10 results of a 10-fold cross-validation (1500 runs in total). The results present the mean accuracy for each dataset, and finally the geometric mean of all the datasets. Although initially our method obtains lower results with a few iterations, with a higher number of iterations it surpasses the other systems.

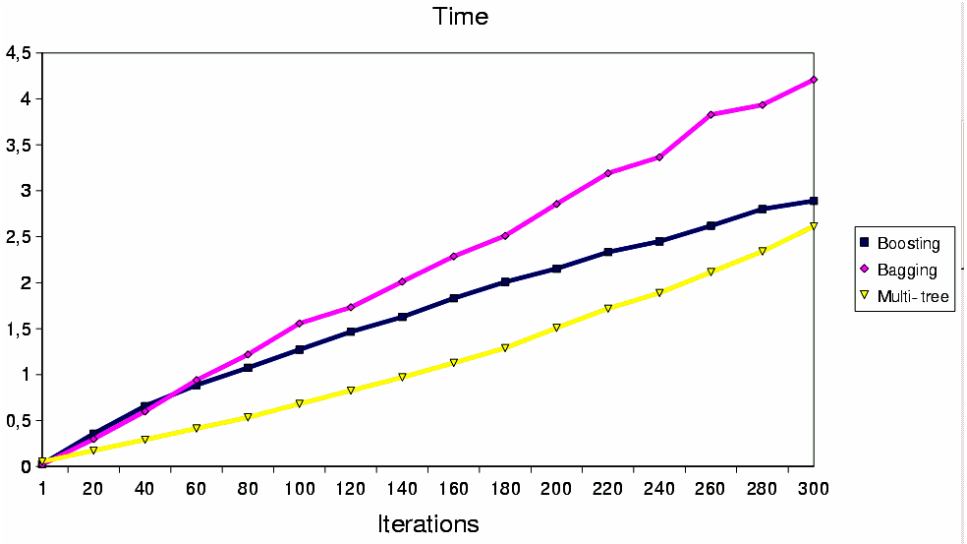
A similar portrait is shown graphically below. The fusion method is “original + max”:



Accuracy comparison between ensemble methods

Nevertheless, the major advantage of the method is appreciated by looking at the consumption of resources. We have argued that a multitree has a great advantage over a forest because the former shares the common parts between several trees whereas the latter codes and stores the several trees separately. This, in theory, must have consequences on both space and time requirements. As we will see in the following graphs, this is the case.

The following figure shows the average training time depending on the number of iterations (1-300) for the three methods. Note that the time increase of *Bagging* is linear, as expected. *Boosting* behaves better with high values because the algorithm implemented in Weka trickily stops the learning if it does not detect a significant increasing of accuracy. Finally, SMILES presents a sub-linear increase of required time due to the sharing of common components of the multi-tree structure.



Time comparison between ensemble methods

We have performed the comparison where this can be done. There are lot of features in SMILES that do not exist in other systems: archetype, size minimisation, misclassification cost minimisation, test cost minimisation, ROC analysis, ROC-inspired splitting criteria, etc.

5 Short Programmer's Manual

SMILES has been implemented on the C++ Programming Language [60][31]. There are some reasons for this: C++ is a powerful language, standardised and portable, efficient and with a lot of mathematical and ML-related software.

5.1 Summary of source files: classes and functions

The following table shows the different files that compose the system:

file			*description*	*classes or structs*	*functions*
components	.h	.cpp	for constructing and handling component matrices	component_matrix	get_partition
cost	.h	.cpp	for computing MDL-related measures	vindex, tindex	compute_cost, calcula_partv
criteria	.h	.cpp	for computing some criteria and defines structures related to these options	split_selection_criteria, split_partition, ...	
estructures	.h		some basic definitions about types	argument, variable, discrete, numeric	
evaluate	.h	.cpp	for the statistics for evaluating trees including cost and confusion matrix	class_matrix, evaluation_statistics	
exemples	.h		for defining and handling (sampling) datasets	example, dataset	randomGenerator
getopt	.h		part of the GNU C library for handling command-line arguments		
main		.cpp	main program: reads options and arguments, creates the multitree, learns and evaluates results.		main
options	.h	.cpp	defines many options and the options class	options	default_options, hardwired_options, trau_opcions
options-file	.h	.cpp	for parsing the option file		ompli_options
parser	.h	.cpp	parser for the dataset files		ompli_memoria
roc	.h	.cpp	ROC facilities		compute_ROC_points, compute_ROC_AREA, genera_ps
rules	.h	.cpp	defines basics of rules	condition, rule	
temps	.h	.cpp	utilities for handling time		start_st, stop_st, pulsos_a_segs, vore_st
trees	.h	.cpp	largest source file with the main learning functions	tree, and_tree, or_tree, multi_tree	
utils	.h	.cpp	some utilities: mathematical and output functions		Pause, log2, logbase, Sqr, RandUniform, RandNormal

5.2 Main source files

The most basic definitions and types are in the file "estructures.h"

```
typedef unsigned char byte;
typedef unsigned short int word;
typedef unsigned long word4;
typedef float numeric;  // used for numerical attributes
typedef byte discrete;  // used for nominal attributes
typedef int variable;   // used for variables
```

```
typedef word4 indext;   // used for indexes in datasets

typedef enum { EMPTY, CONST_EQ, CONST_NEQ, VAR_EQ, VAR_NEQ, NUM_LT, NUM_GE } cond_op;
// kind of operators in conditions

typedef enum { NUMERIC_KIND, DISCRETE_KIND, VARIABLE_KIND } kind;
// kinds of arguments

typedef vector <string> argtable;     // argument values name
typedef vector <argtable> exetablet;  // table of several arguments values names
typedef byte indargtable;
typedef word argoffsetv;              // offset of arguments in tables
typedef argoffsetv argoffsett;
typedef word sizeexamplet;            // size of example
typedef vector<indext> vdistt;        // class distribution
typedef vector<kind> typest;          // vector of kinds

class argument;                       // argument
```

From these definitions the main components of a solution (conditions and rules) can be easily defined:

```
class condition {
  // constructs a condition with an operator and an argument
  condition(cond_op o, argument v);

  // checks if two conditions are added over the same argument
  bool Specialises(const condition &c2) const;
};

class rule {
  // constructs a rule of n arguments (n+1 if we consider the class)
  rule(int n);

  // copy constructor
  rule(const rule &r);

  // adds a condition to argument a of the rule
  bool Add_Condition(int a, const condition &c);

  // converts the rule to string (to be shown, for instance)
  string ToStringUnfolded(char * FName, const dataset & DataSet, bool Short= false);
};
```

In the files “examples.h” and “examples.cpp” the class dataset is implemented. These are the main methods:

```
// constructs a training set from a file
dataset(const string & file)

// constructs a test set from a file and maintaining the structure of a training set
dataset(const string &file, const dataset &dtrain)
:vdist(dtrain.NumClasses())

// shows a dataset
void show()

// shows class distribution
void show_vdist()

// Returns the nth argument (narg) of example nex
argument Nth_arg(indext nex ,int  narg) const

// Obtains a set of the real values (numeric). Used for intervals in partitions
void OrderedSetOfRealsOf(int narg, set<numeric> &s, const vector<indext> &Covered)
const

// Computes if argument i of example e follows condition c
bool FollowsCondition(int i, const condition & c, indext e) const

// Obtains the classes of all examples in a vector
void Obtain_Results(vector<discrete> &results)

// Returns the class of example i
inline discrete ClassOf(indext i) const

// Returns the kind (numeric, discrete) of an argument.
kind KindOf(int a) const
```

```
// Returns the type of an argument
int TypeOf(int a) const

// Return the number of arguments of the dataset (without including the class)
int NumArguments() const

// Returns the number of constants of argument a
discrete NumConstants( int a) const

// Returns the number of classes of the dataset
discrete NumClasses() const

// Returns if rule r covers example e
bool Covers(const rule & r, indext e) const

// Returns the cardinality of the dataset
indext Cardinality() const

// Returns the class distribution of the dataset
vdistt ClassDistribution()  const

// Returns the majority class of the dataset
discrete MajorityClass() const

// Returns the frequency of class C
float FreqOfClass(int C) const

// Returns the name of value v of argument a
string NameArgumentValue(int a, discrete v) const

// Returns the name of class with value v
string NameClassValue(discrete v) const

// Returns the name of class with value v
string GetName(discrete v) const

// Returns the weight of class v
numeric GetWeight(discrete v) const

// Set weights of classes to 1
void SetWeightsTo1()

// Set weights to inverse frequency of the classes using Param
void SetWeightsToInvFreq(float Param)

// Recovers the value of an argument from the argument and the name
discrete GetIndex(word numarg,char * arg) const

// Extracts a % of the examples randominly putting them in a new dataset New
void RandomExtract(numeric pcn, dataset  & New)
```

From here, learning structures can be further constructed. In this sense, probably the most important and complex source file is “trees.h”. A lot of methods of the tree, and_tree, or_tree and multitree classes are defined in this file because they are inline.

The structure of the multitree is defined in the files “trees.h” and “trees.cpp”. The trees can be traversed bi-directionally. The classes “and_tree” and “or_tree” are specialisations of tree such that the parent and children of an and_tree are or_trees and the parent and children of an or_tree are and_trees. With this construction, a multitree is a class that contains a root node that is an and_tree.

Let us take a look to the public methods of the multitree:

```
// Constructs the multitree
multitree(const dataset & Train, options &Opt)

// Learns the multitree
void Learn(const dataset &Train,const dataset &Validation_Test)

// Obtains N solutions: a vector of vector of nodes
void ObtainNBestDifferent(int n, vector<vector<and_tree *> > &Solutions)

// Unmarks and marks a multitree according to a solutions (set of leaf nodes)
void Mark(const vector<and_tree *> Sol)

// Shows the solutions of a multitree given as argument
void Show(const vector<and_tree *> Sol, char *FunctionName)
```

```
// Obtains a vector of class results using the marks in the OR nodes
void Result_as_Marked(const dataset &Test, const options &Options, vector<discrete>
&Results)

// Predict the class using the marks in the OR nodes
discrete PredictMarked(and_tree *Actual_AND_Node, indext i, const dataset & Test)

// Fill all leaf nodes with their "TestDistribution"
void FillTestDistribution(const dataset &Test)

// Show all Rules of Multitree
string ShowAllRules(char *FunctionName, const dataset &DataSet, bool Short) const;

// Show tree in the form of rules. Obsolete
string ShowTreeBest(char *FunctionName, const dataset &DataSet, const dataset
&Validation_Data,bool Short) const

// Obtains the prediction of the best tree for example i of dataset Test
discrete PredictBest(indext i, const dataset & Test)

// Obtain the prediction of a combined solution with for example i of dataset Test
// Three function with three different combination methods
discrete PredictComb(indext i, const dataset & Test)
discrete PredictCombVect(indext i, const dataset & Test)
discrete PredictCrossCoverage(indext i, const dataset & Test)

// Evaluates the best solution. Obsolete
void Evaluate(const dataset & Test, evaluation_statistics & Eval,options &Options)
```

From here, the `main` function in "`main.cpp`" just reads options and arguments, creates the multitree, learns and evaluates results. The main file has the following (simplified) structure:

Reads Options
Begins Loop for Cross-Validation
  Reads Training Set
  Splits Training Set (if cross-validation or sampling)
  Reads Cost Information
  Creates and Learns Multitree
  Shows Rules (optional)
  Prepares (or reads) Test Set
  Evaluates Combined Results
  Begins Loop for Different k Single Solutions
    Obtains 1 Solution
    Evaluates it
    Shows it (optional)
    ROC Analysis (optional)
  Ends Loop
  Computes Means for Different k Single Solutions
  Accumulates Some Results
  Destroys Dataset and Multitree
Ends Loop
Computes and Shows Cross-Validation Results (optional)

5.3 Default and hardwired options

Options structures and attributes (some of them defined in options.h and others in criteria.h) are grouped together in a structure named "options". The most important attributes of this structure are:

```
struct options {
  missing_numeric_values MissingNumericValues;
  reliability_calculation ReliabilityCalculation;
  class_probability_vector_calculation ClassProbabilityVectorCalculation;
  show_mumber_solutions ShowNumberSolutions;
```

```
    expected_error_options ExpectedError;
    smoothing_options Smoothing;
    smoothing_options NodeSmoothing;
    probability_in_splitting_criteria ProbabilityInSplittingCriteria;
    output_options Output;
    multitree_options Multitree;
    solution_combination Comb;
    split_selection_criteria SSC;
    pruning_criterion PC;
    post_pruning_method PostPruningMethod;
    numerical_interval_criterion IntervalCriterion;
    vector<bool> Enabled_Partitions;

    weights_options Weights;
    cost_derived_probability_method Cost_Derived_Probability_Method;
    class_matrix CostMatrix;
    vector<double> WeightVector;

    vector<double> TestCostVector;  // vector of attribute test costs.
    test_cost_method TestCostMethod;
    test_cost_use TestCostUse;
    double TestCostRelevanceInSplitting; // must be from 0 to infinite.
    double TestCostRelevanceInSelectBest; // must be from 0 to 1.

    component_matrix Components;
    component_matrix AccComponents;
    component_matrix RandomComponents;
    bool Components_Reckoning;  // if this is true the previous matrices are computed
    components_in_split_criterion Components_In_Split_Criterion;
    float ComponentsRandomFactor;

    k_best_selection SelectKBest;

    sample_training_set SampleTrainingSet;
    cross_validation CrossValidation;
    float SampleTrainingSetProportion;
    int kFoldValidation;
    int RepeatKFold;
    bool AllowTestWithoutOneClass;

    ...
  }
```

These options are assigned default values by the function "`default_options`" in "options.h". In case that the options file (usually called "options.cfg") does not contain any entry about a particular option, this default option will be used in SMILES.

However, not every option can be modified through the options file, especially some parameters and the enabled partitions. In this case, only by modifying the program it is possible to change one of these options. These options can be recognised in the function "`hardwired_options`". It is not difficult to change them by changing the sources, even with little idea of C++ programming. The file to be modified is "options.cpp". This is an excerpt of this function which gives an idea of how easy it can be to modify one option:

```
  void hardwired_options(options &Options) {
     Options.ComponentsRandomFactor= 0.5;

     ...

     Options.Smoothing.m= 5;
     Options.Components_In_Split_Criterion.Met= NO_USE_OF_COMPONENT_IN_SPLIT;
     Options.SelectKBest.FactorForRandomComponents= 0.25;

     ...
```

```
    vector<bool> Part(4);  // AÇÒ SERIA MILLOR UN CONJUNT.
    Part[PARTITION_MANY_CONST_EQ]= true;            // Partition X=a, X=b, X=c...
    Part[PARTITION_ONE_CONST_EQ]= false;            // Partition X=a, X!=a
    Part[PARTITION_ONE_VARIABLE_EQ]= true;          // Partition X=Y, X!=Y
    Part[PARTITION_ONE_CONST_DISEQ]= true;         // Partition X<c, X>=c

    Options.Enabled_Partitions= Part;
}
```

The following section presents a summary of all the options and distinguish between the ones that can be modified through the options file and the hardwired (only modifiable by program).

6 Options Summary

The following table shows all the options that the current system has, their description, whether they are implemented or not, if there are some additional parameters and if they can be modified through the options file (Y:yes, P:partially, N:no).

Issue	Variants	Impd	Parameters	M
Expected Error Method. This option selects between different ways to compute the expected error. The expected error is not needed if neither pruning nor the selection criterion (split and best solution) are based on it.	NO_COMPUTE: It doesn't compute it. Best option if the Expected Error is not going to be used.			Y
	RELATIVE_FREQUENCY_WITH_MAJORITY_CLASS: Expected Error just as 1-freq of majority class.			Y
	RELATIVE_FREQUENCY_WITH_FREQ_PROB: Expected Error just as = $\Sigma(p_i)(1-p_i)$.			Y
	COST_WITH_MINIMUM_CLASS: Computing the cost instead of error by assuming that the class with minimum cost has been assigned to the node.			Y
	COST_WITH_FREQ_PROB: Computing the cost instead of error by multiplying the probability of assigning a class with the cost associated to that case. The probability is based on frequency.			Y
	COST_WITH_COST_PROB: Computing the cost instead of error by multiplying the probability of assigning a class with the cost associated to that case. The probability is based on costs.			Y
	COST_WITH_COST_PROB_REL_FREQ_SECOND: The same as before but in case that two or more classes have the same cost use the one with less expected error.	NO		N
Frequency Error Smoothing: Use smoothing or not for every calculation of frequencies. There are two options: "smoothing method" (for frequencies of each class). "node smoothing" (for frequencies of each node).	NO_SMOOTHING: the probability of a class is just computed as the frequency of the examples of that class under a particular node.			Y
	LAPLACE: Laplace correction of relative frequency, i.e., $p(c) = (n(c) + 1) / (n + NumClasses)$			Y
	K-ESTIMATE: aberration of Laplace correction: $p(c) = (n(c) + k) / (n + k)$			Y
	M_ESTIMATE: M-estimate correction of relative frequency using the frequencies of dataset, i.e., $p(c) = (n(c) + freq_c \cdot m) / (n + m)$ where $freq_c$ is the frequency of class c for all the dataset.		m	P
	M_ESTIMATE_UNIFORM: M-estimate correction of relative frequency assuming uniform distribution of dataset, i.e., $p(c) = (n(c) + freq \cdot m) / (n + m)$ where freq = 1 / NumClasses. If m=NClasses then M-ESTIMATE = LAPLACE.		m	P

Class Selection Method: How the class of a leaf node is selected	MAJORITY_CLASS: the class with more instances in a node is selected. When two or more classes have the same number of instances, the most common one in the dataset is selected. If the node has cardinality zero, then the majority class of the dataset is selected.			Y
	MAJORITY_CLASS_WITH_SMOOTHING: With Laplace smoothing it is the same to use smoothing or not, but with other smoothing methods may be different.	NO		N
	MIN_COST_CLASS: the class which minimises the cost is selected. When two classes have the same cost, the majority class is selected. When cardinality is zero the one which minimises the cost (assuming a uniform distribution) is selected.	YES, but check		Y
	MIN_COST_CLASS_WITHOUT_SMOOTHING: the same as before but does not use smoothing (in the case it is activated).	NO		Y
	STRATIFICATION_CLASS: the class which maximises the benefit (using the weight vector) is selected. When two classes have the same benefit, the majority class is selected. When cardinality is zero, the one which maximises the benefit (assuming a uniform distribution) is selected.	YES, but check		Y
Prediction reliability calculation methods: Now this reliability is not shown.	FREQUENCY RELIABILITY			Y
	LAPLACE RELIABILITY			Y
Prediction class probability vector calculation methods (this is used for the AUC example by example)	FREQUENCY CLASS PROBABILITY VECTOR			Y
	LAPLACE CLASS PROBABILITY VECTOR			Y
Partitions: which partitions are active in the learning process.	PARTITION_MANY_CONST_EQ // Partition X=a, X=b, X=c...			N
	PARTITION_ONE_CONST_EQ // Partition X=a, X!=a			N
	PARTITION_ONE_VARIABLE_EQ // Partition X=Y, X!=Y			N
	PARTITION_ONE_CONST_DISEQ // Partition X<c, X>=c			N
	PARTITION_BACKGROUND // Partition X=f(...), X !=f(..)	NO		N
Best_Partition_Set: way of using the partitions.	FIXED: Generate all the possible nodes using the active partitions.			N
	ADJUST: selects the partitions according to the problem by using a first round adjusting the active partitions.	NO		N
Partitions Restriction / Priorisation: methods on how to restrict some partitions on some attributes using association or correlation information.	ASSOCIATION-CORRELATION	NO	SUPPORT / CONFIDENCE	N
Numeric Interval Criterion: how continuous attributes are handled in each node in order to generate the thresholds (and how many)	NO_LIMIT: all intervals given by the middle points of the values falling under the node.			Y
	CLASS_INTERVAL (C4.5). It just differs in efficiency with NO_LIMIT. Doesn't generate a threshold if the classes on both sides are the same.	NO		N
	MAX(a): maximum number of intervals. It just orders the values and split it in *a* segments…			Y
	LOG(a,b): like the previous one but the number of intervals is obtained by: $(a + \log_b n)$.			Y

	Discretisation based on reduction on Variance (see "Investigation and reduction of discretization variance in decision tree induction" P. Geurts, L. Wehenkel. Proc. of ECML2000, Barcelona, Spain, May 2000, @springer-verlag) • Classical (C4.5) • Kolmogorov-Smirnov (J.H. Firedman "A recursive partitioning decision rule for nonparametric classifier) IEEE Transactions on Computers, C-26:404-408, 1977. • Median. (implemented MAX(1))	NO		N
Probability in splitting criteria: only applicable to GAIN and derivatives, GINI and DKM.	FROM_FREQUENCY_NO_SMOOTHING			Y
	FROM_FREQUENCY_SMOOTHING			Y
	FROM_FREQUENCY_FROM_COSTS			Y
	FROM_FREQUENCY_WITH_STRATIFICATION			Y
	FROM_FREQUENCY_WITH_STRATIFICATION_NO_SMOOTHING			Y
Splitting Criterion: criterion which is used to select the best split. This sets the SPLIT_OPTIMALITY value used by other issues.	LEFT_FIRST: Selects the first node.			Y
	GAIN: Entropy. Quinlan's Gain.			Y
	GAIN_RATIO: Quinlan's Gain Ratio.			Y
	C4.5 : Quinlan's Gain Ratio for those with gain greater than the mean.			Y
	ADJGAIN : C4.5 but with numerical splits improvement.	NO		N
	CART: Simplified Breiman et al. GINI heuristic.			Y
	MGINI: Correct Breiman et al. GINI heuristic.			Y
	DESC_MDL: The description cost of the examples following under the split nodes + the cost of the partition.			Y
	PRED_MDL: Just the class of the examples are described.	NO		N
	DKM: Kearns & Mansour modification of CART.			Y
	SPLIT_EXPECTED_ERROR: Uses expected error (and its selected method to compute it, maybe using cost).			Y
	WEIGHTED_GAIN: to be defined how to combine it with costs.	NO		N
	WEIGHTED_MDL: to be defined how to combine it with costs.	NO		N
	LOCAL_ROC_AREA: Selects the split with greatest area under the ROC curve. Only takes into account the nodes in that split.			Y
	GLOBAL_ROC_AREA: Selects the split with greatest area under the ROC curve. Takes into account all the open nodes in the tree.	NO		N
	ONE_POINT_LOCAL_ROC_AREA: simplification of "local ROC area". Just computes the are with one point.			Y
	MSE: minimum squared errror.			Y
	LOGLOSS: logloss metric.			Y
	SQDIFF: computed as the square of the difference between probability for class *a* and probability for class *b*. Only valid for two classes			Y
	GENENTROPY: gain and gini can be seen as special cases of a generalised entropy function depending on a power. This is a parametrised split criterion where this exponent can be modified (by program).			Y
	ROCV: another (not very successful) extension to the roc split for more than 2 classes.			Y
	AUCH: Based on Hand and Till's extension of Area Under the ROC Curve for more than two classes			Y
	AUCS: Fawcett's variant of AUCH.			Y

Components in Split Criterion: use of a component matrix (reflecting the partitions and attributes that have been used).	NO_USE_OF_COMPONENT_INFORMATION			N
	ADAPTATIVE COMPONENT: use an accumulative component matrix to avoid components (partitions and attributes) that have been used in previous solutions in order to obtain different solutions.			N
	RANDOM_COMPONENTS: generate the component matrix randomly also in order to obtain different solutions.		alpha	N
Pre-pruning: criterion for pruning when constructing the tree.	NO_PRUNING			Y
	PROPORTIONAL: prune when the number of examples under a node are less than alpha and the proportion of correctly classified is greater than beta.		alpha + beta	P
	EXPECTED ERROR: prune using EXPECTED ERROR (and its associated method that may take cost into account). The parameter alpha is used as a factor between the expected error of the parent and the expected error of the children.		alpha (1 is the default and as greater it is the more it prunes).	P
	SIGNIFICANCE	NO	alpha: significance degree	N
	MDL_PRUNING: prune if the description cost of children (and partition) is greater than the parent node.			Y
	STUMP_PRUNING: prune the tree at a constant depth.		Stump Pruning Limit (depth)	Y
	PEP-pruning: Uses “Pessimistic Error Pruning”:			Y
	MDL_PRUNING2: as MDL_PRUNING but measuring exceptions separately.	NO		N
Post-pruning	NO_POSTPRUNING: doesn’t use post-pruning			Y
	PEP_PRUNING: Uses “Pessimistic Error Pruning”: This pruning is based on *marking* nodes as pruned, but doesn’t delete them.			Y
Multitree: how many and how the several trees are generated	ONE_ONLY: just generate the first tree. (Greedy search). Never used. If MaxNumTree=1 then ONE_AND_FORGET is used.			N
	ONE_AND_FORGET: just generate the first tree. However, unused or nodes are deleted when constructing the tree. This is the option that requires lower memory resources.		If MaxNumTree =1.	Y
	MANY_AND_MAINTAIN: generate and maintain MaxNumTrees.		MaxNumTrees	Y
	MANY_AND_FORGET: when only one solution is required, bad solutions can be forgotten. A h() function is necessary as a A* search algorithm.	NO		N
Stopping Criterion for the overall algorithm.	CONSTANT: The algorithm stops when NumTree OR-nodes have been explored.			N
	MAXTIME: The algorithm stops when time is finished.	NO		N
	STALLED: The algorithm stops when accuracy has not been incremented in the last *m* iterations.	NO		N
	HEURISTICALLY: The number of iterations is guessed accordingly to number of attributes, types and number of examples.	NO		N
BestTree Selection Criterion: How to select the best solution if only one of all has to be shown (comprehensible model)	OCCAM_BEST: the one with lower number of rules.			Y
	TEST_COST_BEST: the one that minimises testcost.			Y
	OCCAM_AN_TEST_COST_BEST: a combination of the preceding two. The relevance of each one through internal option TestCostRelevanceInSelectBest that must be from 0 to 1			P

	EXPECTED_ERROR_BEST: the one with lower expected error (or cost if it is included).			Y
	COVERAGE BEST:			Y
	CROSS_COVERAGE: The training set is split in two parts. Only one is used for learning and the other one is used for selecting the best tree (the one with more accuracy).			Y
	SPLIT_OPTIMALITY_BEST: the one with higher optimality (using the optimality as is computed by the splitting criterion).	NO		Y
	MDL_BEST: the one with lower description cost.			Y
SelectKBest: how the k best solutions are seleted.	K_Best_Less_Visited: from all the possible solutions, selects the best according to the "BestTree Selection Criterion". For the second tree, tries to avoid the branches selected by the first and so on.			Y
	K_Best_Less_Visited_Plus.	NO		Y
	K_Best_Less_Visited_Different_Components from all the possible solutions, selects the best according to the "BestTree Selection Criterion". For the second tree tries to avoid the branches selected by the first and so on. When the same branch has to be passed again, the one with different components (according to the accumulated component matrix).			Y
	K_Best_Different_Components: tries to select solutions with components different to the *reference* component matrix. In case of tie then the "besttree selection criterion is used".			Y
	K_Best_Random: selects each solution randomly combining the all possible branches. Note that repeated solutions might be obtained.			Y
K Best Component Generation: how the *reference* component matrices are generated.	Components_Accumulate: the reference component matrix accumulates the resulting component matrix of previous solution matrices. Initially the matrix is filled with 0s.			Y
	Components_Random_Generated_From_Start: the reference matrix is generated randomly.			Y
	Components_Random_Generated_From_Second: This is a variant of the previous one that ensures that the first solution is always the same as if only one solution (the best one) were generated.			Y
K Best Number of Solutions:	How many solutions are generated from the multitree.			Y
Second Tree Opening Criterion: once the first solution is found, how to select a second node to explore for the second solution.	SPLIT_OPTIMALITY: the node with best absolute splitting optimality of all the tree structure.			Y
	OPTIMALITY_RIVAL_RATIO: the node with best relative absolute splitting optimality of all the tree structure.			Y
	OPTIMALITY_RIVAL_RATIO_DEPTH: balancing optimality_rival_ratio with the depth of the node.		alpha	P
	OPTIMALITY_RIVAL_WITH_COMPONENT: balancing optimality_rival_ratio with an accumulated component matrix.			Y
	OPTIMALITY_RIVAL_WITH_COMPONENT_RANDOM: balancing optimality_rival_ratio with a random component matrix.			Y
	SECOND_BOTTOMMOST: the bottommost node is selected.			Y
	SECOND_TOPMOST: the topmost node is selected.			Y
	SECOND_FRONT: the node most time suspended is selected. A FIFO order.			Y
	SECOND_BACK: the node most time suspended is selected. A LIFO order.			Y
	SECOND_RANDOM: selects a node pseudo-randomly with a uniform distribution.			Y

	SECOND_RANDOM_DEPTH: selects a node pseudo-randomly with a distribution that takes also into account the depth of the node.		weight to ponder the random result and depth	P
	ROC-DISTANCE	NO		N
	ROC-SAMPLING	NO		N
Suspended Nodes forgetting: must all suspended nodes maintained?	MAINTAIN ALL			
	MAINTAIN CONST RANDOM		The const is specified through the option "suspended nodes maintain const value"	Y
	MAINTAIN LOG RANDOM			Y
	MAINTAIN LOG RANDOM WITH DEPTH			Y
	MAINTAIN LOG RANDOM WITH SQUARED			Y
	MAINTAIN LOG RANDOM WITH DEPTH ADJUSTED			Y
Change of dataset weights between different solutions:	NO_CHANGE	NO		N
	BOOSTING_ADJUST_OF_COST_MATRIX	NO	alpha	N
Weights (Cost) Method: How the cost matrix is constructed.	NO_COSTS / UNIFORM_WEIGHTS: don't use cost information (or just construct the matrix with all values equal).			Y
	WEIGHTS_FROM_FILE: get a weight vector from the dataset file and convert it into a cost_matrix, understanding the vector as a stratification.	NO		Y
	INVERSE_FREQUENCY_WEIGHTS: assume the weight vector as the inverse of the frequencies on the dataset.	NO		Y
	COST_FROM_MATRIX: cost matrix given by the user.		Cost_Matrix	Y
Cost Derived Probability Method	DIRECT: no use smoothing to compute cost derived probability.			Y
	WITH_VECTOR_SMOOTHING: use somoothing.			Y
TESTCOST: how the vector of attribute test costs is constructed	no test costs			Y
	uniform test costs			Y
	test costs from file			Y
TEST COST USE: how the vector of attribute test costs is used ("without repetition" better than "with repetition")	test costs no use			Y
	test costs linear plus1 without repetition			Y
	test costs linear plus1 with repetition			Y
TestCostRelevanceInSplitting	In the case that TEST COST USE is different from "no use" then the option TestCostRelevanceInSplitting can be used to give it more or less importance. Its value must be from 0 to infinite.			N
Combination: how to combine several solutions	NO COMBINATION: no combination is used.			Y
	MAJORITY CRISP: when an OR node is found, just assign the majority class of the children.			Y
	MAJORITY ABSOLUTE STOCHASTIC (CLASS DISTRIBUTION): Instead of propagating the assigned class, a class distribution is propagated. At root, the majority class of that vector is selected.			Y
	MAJORITY RELATIVE STOCHASTIC (CLASS DISTRIBUTION): Instead of using a class distribution.	NO		Y
	CROSS_COVERAGE_COMBINATION: Using the partition of the test set into two datasets (one for training and one for combining). Uses the same technique as "Absolute Stochastic".			Y

	MINIMAL COST: selects the OR with less cost.	NO		N
	MAJORITY_COST_STOCHASTIC			Y
	COMPREHENSIBLE RULE SELECTION: tries to select a set of rules from all the possible solutions.	NO	MaxNumRules MinAccuracy Min%WholeAccuracy	N
	MDL	NO		N
Combination smoothing. Use smoothing before combination	YES/NO			Y
Allow post-pruning in combination (in post-prune enabled)	YES/NO			Y
Combination vector: absolute (n. of examples) or relative (frequency)	ABSOLUTE/RELATIVE			Y
Combination vector method: how to derive the vector	ORIGINAL			Y
	GOOD LOSER			Y
	BAD LOSER			Y
	DIFFERENCE			Y
	MAJORITY			Y
	SQUARED			Y
Combination fusion method: how to combine the vectors	SUM			Y
	PROD			Y
	ARITHMEAN			Y
	GEOMEAN			Y
	MAX			Y
	MIN			Y
	MEDIAN			Y
ARCHETYPE: Similarity function used for the selection of a single solution from the combination	KAPPA			Y
	KAPPA1			Y
	QSTAT			Y
Combination to Single Solution Method	NO EXTRACTION			Y
	INVENTED DATASET			Y
Length of the invented random dataset				Y
Combination to Single Solution (Archetype) Use of Other Criteria. These factors affect how the Archetype is extracted. If all the factors except similarity are left to 0, then it is just a semantic extraction. 15, 4, 1 are compensated values.	ARCHETYPE SIMILARITY IMPORTANCE FACTOR			Y
	ARCHETYPE OCCAM IMPORTANCE FACTOR			Y
	ARCHETYPE TEST COST IMPORTANCE FACTOR			Y
SAMPLE_TRAINING: splits the training set into a subtraining set and a validation (fake) test.	NO_SAMPLE / SAMPLE: whether or not to generate a percentage of the training set to use it for pre-pruning, selection criteria and so on. It depends on the following options.			Y

SAMPLE TRAINING SET PROPORTION	Proportion of the training set to be used (the rest is for validation if cross coverage options are active or the rest is simply ignored). The selection is made randomly.			Y
CROSS VALIDATION: use separate files for train/test or split the training set	CROSS_VALIDATION: splits the training set into two sets: one for learning (real training set) and other for validation (test set)			Y
	USE_SEPARATE_TEST: reads two different tests: training set file and test set file.			Y
	KFOLD CROSS VALIDATION: uses the same partition several times, until all the combinations have been used.			Y
	REPEATED KFOLD CROSS VALIDATION: repeats the experiments n times (as option REPEAT KFOLD below).			Y
KFOLD OF CROSSVALIDATION:	Times that the cross validation is to be performed.			Y
REPEAT KFOLD	how many times (if cross validation) we repeat the experiment			Y
allow (if cross validation) a test dataset with one class without examples	YES/NO			Y
OUTPUT OF RULES	FUNCTIONAL LOGIC: Functional Logic Programming without simplification of conditions (constraints).			Y
	SIMPLE FLP: Functional Logic Programming but simplifies the constraints and eliminates empty rules.	NO		N
	IF-THEN-ELSE	NO		N
SHOW CLASS DISTRIBUTION	SHOW: show the proportion of each class (for the training set).	NO		Y
	NO SHOW			Y
SHOW ALL MULTITREE RULES: show all rules of the multitree (used or not)	NO SHOW			Y
	SHOW			Y
	TO FILE: outputs them to a file.	NO		Y
SHOW ALL K BEST SOLUTIONS: show the rules of the k best solutions	NO SHOW			Y
	SHOW			Y
	TO FILE: outputs them to a file.	NO		Y
SHOW COMPONENT MATRIX OF SOLUTIONS	SHOW/NO SHOW: whether or not to show the component matrix corresponding to each solution.			Y
SHOW CONFUSION MATRIX	NO SHOW: Never shows the confusion matrix.			Y
	SHOW ONLY IF COSTS: Only if costs are used.			Y
	SHOW: Shows the confusion matrix for combined solution and for K best solutions.			Y
SHOW STATISTICST MODE	ABSOLUTE STATISTICS: Shows results per class and total, in an absolute way (number of examples).			Y
	RELATIVE STATISTICS: Shows results per class and total, in a relative way (proportion of examples).			Y
	BOTH STATISTICS: absolute and relative statistics.			Y
	JUST ACCURACY: Just shows accuracy.			Y
ROC AREA WITH	TRAINING SET (YES/NO)			N
	TEST SET (YES/NO)			N
	TRAINING ORDER TEST	NO		N
COMPUTE ROC POINTS	YES/NO			Y
SHOW ROC POINTS	YES/NO			Y
COMPUTE ROC AREA	YES/NO			Y
GENERATE ROC CURVE FILE	YES/NO			Y

show number of possible solutions in the multitree	YES/NO			Y
HOW TO HANDLE MISSING VALUES	IGNORE EXAMPLES WITH MISSING NUMERIC VALUES			Y
	SUBSTITUTE MISSING NUMERIC VALUES			Y
Use Fake Test for Combination	Future feature.	NO		N
Use Fake Test for Second Tree Opening	Future feature.	NO		N
Use Fake Test for Selecting the Best Solution	Future feature.	NO		N
Null-argument and missing argument treatment	Future feature.	NO		N
Post-pruning	Future feature.	NO		N

As we can see in the previous table, some options are not yet implemented. These were once considered as imminent future work to be done and, jointly with the issues discussed in the next section.

7 Future work

In this section, we describe some improvements or new things that could be added in new versions of SMILES.

A first thing strongly related to implementation is that the second tree opening criterion looks into a list of suspended nodes to see which one has the highest optimality to be opened. This requires the traversal of the entire list every time a new second tree is required. One alternative could be to insert in an ordered way into the list of suspended nodes. This has advantages and disadvantages: faster retrieval but slower insertion of new suspended nodes that are found throughout the multitree construction.

Another relevant thing would be to extend the stop criteria. Now the number of multitrees just depends on a constant. It would be better to make some mechanism in order to detect when further trees are not improving significantly the solution. This could be done through the use of an additional validation set, a reserved part of the training set or an invented dataset.

At the current implementation, partitions cannot be activated or deactivated through the "options.cfg" file. The intention was to include options to make this automatic, i.e. the system should have heuristics to use the appropriate ones for each problem, but at the moment it can only be done modifying the source code. Moreover, new partitions are envisaged and optionally a *types file*. Consequently, all this could vary a great deal in the future. This is also the reason why the directive `!NAMES` in the training set file has not been implemented either.

Related to the previous extension, it would be interesting to accept *ordered nominal values* such as {low, medium, high}. There is also a restriction in the number of different possible values for a nominal attribute: 256. Although this is maintained in this way to reduce space, it precludes SMILES from handling some datasets. In a similar way, a further treatments of unknown numerical values (now denoted by '?') could be implemented. Currently there are two things to be done: to ignore the examples or to consider an additional branch (X='?'), whose appropriateness has not been analysed.

One important thing to improve user friendliness is to improve the outputs. The output of rules is in if-then-else form, not showing rules with 0 coverage, etc, would be interesting.

Outputs to file should also be improved: such as saving and loading the entire multi-tree, best k-solutions, ROC points, etc. In the future, inputs and outputs should be in XML standards.

Moreover, now SMILES integrates the generation of the multi-tree structure (the real learning stage) and the combination and the extraction of solutions. It would be very interesting to run both parts independently, and to be able to export and import multi-trees, to extract archetypes, to apply them, to visualise them, etc.

The handling of testcosts now does not take into account "test groupings", i.e. that two or more attributes can be obtained in the same real tests, e.g., the concentration of sugar in blood and the leucocytes level can be performed by the same real test (a blood analysis) and its cost should only be reckoned once. The best form to do is through the modification of the ".testcost" files, through additional information for the joining of attributes. For instance,

3, 50, 3, 50, 1, 0, 50, 4

1->3

2->7

2->4

Means that tests on attributes 1 and 3 should be added only once, and tests on attributes 2, 4 and 7 should be added only once.

Now SMILES learns models but doesn't give any additional support to apply them. In fact, SMILES can compute the reliability of each prediction, but this reliability is not shown. SMILES (or an additional application) should be able to open existing models, apply them, even edit them, change their format, etc.

The pre-pruning methods are quite limited and are hardwired with quite arbitrary parameters. More standard pre-pruning and, especially, post-pruning methods should be implemented.

Some other things of a more novel hue that could be implemented are:

- New Occam fusion method: weight each of the hypotheses to be combined depending on their size. Reorganise all the "MAJORITY" fusion options, because in practice just one is used.
- New fusion method and splitting criterion based on the measure of confidence (support/confidence), which are closely related to the AUC measure.
- Now the archetype just uses an invented dataset. Make it possible to use part of the training dataset or an additional external dataset. Allow that the archetype would be constructed with an external Oracle (just an additional training set) instead of the combination.
- Design a kind of pruning based in combination.
- Design a kind of pruning based in AUCH, MSE and LogLoss.
- Study cross-coverage again and compare with archetype.
- Pursue with the suspended or forgetting methods: add a children limit per node, a global limit, new forget method (forget the similar ones).
- Use of background knowledge and more expressiveness.

Finally, other more ambitious extensions may include a visual interface and a modification to convert SMILES in a regression system, a clustering system or even an association rules discoverer.

Acknowledgments

The name of the system has been borrowed from “Smiles Brewery” (http://www.smiles.co.uk), a famous Bristol brewery.

References

1. Berkman, N.C.; Utgoff, P.E.; Clouse, J.A. “Decision tree induction based on efficient tree restructuring” *Machine Learning*, 29(1):5--44, 1997.
2. Blake, C.; Merz, C. “UCI repository of machine learning databases” (http://www.ics.uci.edu/~mlearn/MLRepository.html). University of California, Dept of Computer Science, 1998.
3. Blockeel, H.; Struyf, J. “Frankestein classifiers: Some experiments on the Sisyphus data set”, ECML/PKDD2001 Workshop on *Integrating Aspects of Data Mining, Decision Support and Meta-Learning*, 2001.
4. Boissonat, J.D.; Yvinec, M. *Algorithmic Geometry*. Cambridge University Press, 1998.
5. Bradford, J.; Kunz, C.; Kohavi, R.; Brunk, C; and Brodley, C. “Pruning decision trees with misclassification costs” in *Proc. of the European Conference on Machine Learning*, pp. 131-136, 1998.
6. Breiman, L.; Friedman, J.H.; Olshen, R.A. and Stone, C.J. *Classification and regression trees*, Belmont, CA, Wadsworth, 1984.
7. Clementine Data Mining System, http://www.spss.com/clementine.
8. Dean, T.; Boddy, M. “An analysis of time-dependent planning” Proc. of the 7th National Conference on Artificial Intelligence, pp. 49-54, 1988.
9. Domingos, P. “Metacost: A general method for making classifiers cost-sensitive” *Proc. of the Fifth International Conference on Knowledge Discovery and Data Mining*, pp. 155-164, New York, ACM, 1999.
10. Drummond, C.; Holte, R.C. “Exploiting the cost (in)sensitivity of decision tree splitting criteria”, *Proc. of the Seventeenth International Conference on Machine Learning, pp. 239-246*, 2000.
11. Elkan, C. “The Foundations of Cost-Sensitive Learning”, *Proc. of the Seventeenth International Joint Conference on Artificial Intelligence*, IJCAI’01, 2001.
12. Esposito, F., Malerba, D. & Semeraro, G. (1997). A Comparative Analysis of Methods for Pruning Decision Trees. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 19, No. 5.
13. Estruch, V. Ferri, C.; Hernández-Orallo, J.; Ramírez-Quintana, M.J. “SMILES. A Multi-purpose Learning System” 8th European Conference on Logics in Artificial Intelligence” JELIA, Lecture Notes in Computer Science (LNCS) , to appear, 2002.
14. Estruch, V. Ferri, C.; Hernández-Orallo, J.; Ramírez-Quintana, M.J. “Share Ensembles using Multi-trees” 8th Ib. Conf on A.I., Iberamia 2002.
15. Estruch, V. Ferri, C.; Hernández-Orallo, J.; Ramírez-Quintana, M.J. “Re-designing Cost-Sensitive Decision Tree Learning” Workshop on Learning and Data Mining, 8th Ib. Conf on A.I., Iberamia 2002.
16. Estruch, V. Ferri, C.; Hernández-Orallo, J.; Ramírez-Quintana, M.J. “Making Combination Methods Comprehensible and Cost-Sensitive using Shared Ensembles. Applications in Medicine” Submitted to Artificial Intelligence in Medicine 2002.
17. Fan, W.; Stolfo, S.; Zhang, J. and Chan, P.H. "AdaCost: Misclassification Cost-sensitive Learning" Proceedings of the Sixteenth International Conference on Machine Learning (ICML'99), pp.97-105, Bled, Slovenia, June 1999.
18. Ferri-Ramírez, C.; Hernández-Orallo, J. ; Ramírez-Quintana, M.J. “Learning MDL-guided Decision Trees for Constructor-Based Languages”, in WIP track of 11th Intl. Conf. on Inductive Logic Programming, ILP01, pages 39-50, 2001.
19. Ferri-Ramírez, C.; Hernández-Orallo, J.; Ramírez-Quintana, M.J. “FLIP: User’s Manual” Technical Report, Dpto. de Sistemas Informáticos y Computación, Valencia, TR: II-DSIC-24/00, pp. 8, 2000.
20. Ferri-Ramírez, C.; Hernández-Orallo, J.; Ramírez-Quintana, M.J. “Incremental Learning of Functional Logic Programs” in Kuchen, H.; Ueda, K. (eds.) "Fifth International Symposium on Functional and Logic Programming", Lecture Notes in Computer Science (LNCS) series, Vol. 2024, pp. 233-247, Springer, 2001.

21. Ferri-Ramírez, C.; Hernández-Orallo, J. ; Ramírez-Quintana, M.J. “AND / OR Trees for the Learning of Functional Logic Programs” *in 2001 Joint Conference on Declarative Programming,*APPIA-GULP-PRODE 2001, pages 329-341, 2001.
22. Ferri-Ramírez, C.; Hernández-Orallo, J. ; Ramírez-Quintana, M.J. “Induction of Decision Multitrees using Levin Search”, International Conference on Computational Science, ICCS2002, Amsterdam, April 2002, Springer LNCS.
23. Ferri-Ramírez, C.; Hernández-Orallo, J. ; Ramírez-Quintana, M.J. “From Ensemble Methods to Comprehensible Models”, Discovery Science 2002, Springer LNCS, to appear.
24. Ferri-Ramírez, C.; Flach, P. Hernández-Orallo, J. “Learning Decision Trees using the Area Under the ROC Curve” Intl. Conf. On Machine Learning, ICML’ 2002.
25. Ferri-Ramírez, C.; Flach, P. Hernández-Orallo, J. “Multi-class Decision Tree Splitting Criteria for Maximising the Area under the ROC Curve” Technical Report 2002.
26. Ferri-Ramírez, C.; Flach, P. Hernández-Orallo, J. “Multi-dimensional ROC Analysis with Decision Trees” Technical Report, Dep. of Computer Science, University of Bristol, 2002.
27. Ferri-Ramírez, C.; Flach, P. Hernández-Orallo, J. “Learning Multiple and Different Hypotheses” Technical Report, Dep. of Computer Science, University of Bristol, 2002.
28. Freund, Y. ; Schapire, R.E. “Experiments with a new boosting algorithm” *Proceedings of the Thirteenth International Conference on Machine Learning* (ICML'1996), pages 148--156. Morgan Kaufmann, 1996.
29. Hand, D.J. *Construction and assessment of classification rules*. Chichester: Wiley, 1997.
30. Hand, D.J.; Till, R.J. “A Simple Generalisation of the Area Under the ROC Curve for Multiple Class Classification Problems” *Machine Learning*, 45, 171-186, 2001.
31. Hernández, E.; Hernández, J.; Juan, M.C. “C++ Estándar” Paraninfo Thomson Learning 2001.
32. Hernández-Orallo, J.; Ramírez-Quintana, M.J. “A Strong Complete Schema for Inductive Functional Logic Programming” in Dzeroski, S.; Flach, P. (eds) “Inductive Logic Programming” Lecture Notes in Artificial Intelligence (LNAI) series, Vol. 1634, pp. 116-127, Springer 1999.
33. Kearns, M. and Mansour, Y. “On the boosting ability of top-down decision tree learning algorithms” *Proceedings of the Twenty-Eighth ACM Symposium on the Theory of Computing*, pp. 459-468, New York, ACM Press, 1996.
34. Knoll, U.; Nakhaeizadeh, G.; Tausend, B. “Cost-sensitive pruning of decision trees” *Proc. of the Eight European Conference on Machine Learning*, ECML-94, pp. 383-386, Berlin, Germany, Springer-Verlag, 1994.
35. Kukar, M.; Kononenko, I. “Cost-sensitive learning with neural networks” *Proc. of the Thirteenth Conference on Artificial Intelligence*, Chichester, NY, Wiley, 1998.
36. Lane, T. “Extensions of ROC Analysis to Multi-Class Domains”, ICML-2000 Workshop on cost-sensitive learning, 2000.
37. Lavrac, N., Gamberger, D. and Turney, P. “Cost-sensitive feature reduction applied to a hybrid genetic algorithm.” In Proc. Seventh International Workshop on Algorithmic Learning Theory, pp. 127-134, Springer, Berlin, 1996.
38. Levin, L.A. “Universal Search Problems” *Problems Inform. Transmission*, 9:265--266, 1973.
39. Margineantu, D.; Dietterich, T.G. “Learning Decision Trees for Loss Minimization in Multi-Class Problems”, Technical Report 99-30-03, Department of Computer Science, Oregon State University, 1999.
40. Margineantu, D.; Dietterich, T.G. “Bootstrap Methods for the Cost-Sensitive Evaluation of Classifiers”, Proceedings of the Seventeenth International Conference on Machine Learning (ICML-2000), pp.583-590, Morgan Kaufmann, San Francisco, CA, 2000.
41. Mehta, M.; Rissanen, J.; Agrawal, R. "MDL-Based Decision Tree Pruning" *Proceedings of the First International Conference on Knowledge Discovery and Data Mining* (KDD'95)}, pages 216--221, 1995.
42. Nilsson, N.J. “ Artificial Intelligence: A New Synthesis” Morgan Kaufmann 1998.
43. Pazzani, M. J., Merz, C. J., Murphy, P., Ali, K., Hume, T., and Brunk, C. “Reducing Misclassification Costs” In Proceedings of the 11th International Conference of Machine Learning, Morgan Kaufmann, 217-225, 1994.
44. Pearl, J. *Heuristics: Intelligence search strategies for computer problem solving*, Addison-Wesley, 1985.

45. Pfahringer, B. “A new {MDL} measure for robust rule induction” In N. Lavraç and S. Wrobel, editors, Proceedings of the 8th European Conference on Machine Learning, volume 912 of LNAI, pages 331--334, Berlin, 1995. Springer.
46. Pfahringer, B. “Compression-based discretization of continuous attributes” in Proc. 12th International Conference on Machine Learning, pages 456--463. Morgan Kaufmann, 1995.
47. Provost, F.J. “Goal-directed inductive learning: Trading off accuracy for reduced error cost” *AAAI Spring Symposium on Goal-Driven Learning*, 1994.
48. Provost, F. and Fawcett, T. “Analysis and visualization of classifier performance: Comparison under imprecise class and cost distribution” in *Proc. of The Third International Conference on Knowledge Discovery and Data Mining (KDD-97)*, pp. 43-48, Menlo Park, CA: AAAI Press, 1997.
49. Quinlan, J.R. “Induction of Decision Trees” *Readings in Machine Learning*, Morgan Kaufmann, 1986.
50. Quinlan, J.R. « Simplifying Decision Trees”. *International Journal Man-Machine Studies*, vol. 27, pp. 221-234, 1987.
51. Quinlan, J.R. “Learning Logical Definitions from Relations” *Machine Learning*, 5(3):239--266, 1990.
52. Quinlan, J.R. *C4.5. Programs for Machine Learning*, San Francisco, Morgan Kaufmann, 1993.
53. Quinlan, J.R. “Improved use of continuous attributes in C4.5” *Journal of Artificial Intelligence Research*, 4, 77-90, 1996.
54. Quinlan, J.R. “Bagging, Boosting and C4.5” in Proc. of the Thirteenth Nat. Conf. on A.I. and the Eighth Innovative Applications of A.I. Conference, pages 725--730. AAAI Press / MIT Press, 1996.
55. Quinlan, J.R.; Rivest, R.L. “Inferring Decision Trees Using The Minimum Description Length Principle” *Information and Computation*, 80:227--248, 1989.
56. Rissanen, J. “Modelling by shortest data description” *Automatica*, 14:465--471, 1978.
57. Schmidhuber, J.; Zhao, J. and Wiering, M. “Shifting Inductive Bias with Success-Story Algorithm, Adaptive Levin Search, and Incremental Self-Improvement”, *Machine Learning*, 28:105--130, 1997.
58. Smith, S. P., and Jain, A.K. “Testing for Uniformity in Multidimensional Data” IEEE Trans. on Pattern Analysis and Machine Intelligence, 6 (1984), pp. 73-81.
59. Srinivasan, A. “Note on the Location of Optimal Classifiers in N-dimensional ROC Space” Technical Report PRG-TR-2-99, Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford.
60. Stroustrup, B. “The C++ Programming Language” Third Edition, Addison-Wesley 1997.
61. Swets, J., Dawes, R., and Monahan, J. “Better decisions through science” *Scientific American*, October 2000, 82-87.
62. Turney, P.D. “Cost-Sensitive Classification: Empirical Evaluation of a Hybrid Genetic Decision Tree Induction Algorithm”, *Journal of Artificial Intelligence Research* 2, pp. 369-409, 1995.
63. Turney, P.D “Types of Cost in Inductive Concept Learning”, Workshop on Cost-Sensitive Learning at the Seventeenth International Conference on Machine Learning (WCSL at ICML-2000), pp 15-21, 2000.
64. University of California, UCI Machine Learning Repository Content Summary, http://www.ics.uci.edu/~mlearn/MLSummary.html.
65. Weiss, G. and Provost, F. "The Effect of Class Distribution on Classifier Learning: An Empirical Study" Technical Report ML-TR-44, Department of Computer Science, Rutgers University, 2001.

Appendix A: Program History

v.0.1	October 2001:	New system design begins as a successor of FLIP (FLIP3.0)
v.0.5	November 2001:	Basic Facilities: multi-tree structure, parsing utilities
v.0.6	November 2001:	First Evaluation measures: testset
v.0.7	December 2001:	Cost-sensitive facilities: Use of cost and confusing matrices
v.0.9	December 2001:	Different Solutions: component matrices
v.1.0	January 2002:	Options file. General improvement of inputs and outputs
		The system receives its current name: SMILES
		FIRST RELEASED VERSION.
v.1.1	January 2002:	ROC Facilities
v.1.2	January 2002:	Cross-validation and basic missing numeric values handling
V.1.2.5	January 2002:	K-Fold Cross-validation implemented
v.1.3	January 2002:	First Post-pruning method implemented (PEP Method)
v.1.4	January 2002:	Classes can be made equivalent in order to reduce no. of classes
v.1.4.1	January 2002:	Repeated k-fold cross-validation
v.1.4.2	January 2002:	Some memory leaks are fixed.
v.1.5	January 2002:	When only one tree is learnt the new option "ONE_AND_FORGET" is used.
		This options deletes or_nodes that are not further used and memory is freed.
		SECOND RELEASED VERSION.
v.1.6	February 2002:	MSE measure and MSE split criterion implemented
v.1.6.1	February 2002:	LogLoss split criterion implemented
v.1.6.2	February 2002:	All memory leaks corrected.
v.1.6.3	February 2002:	Some correlations are computed using the Compute_Correlation function.
		The new source file "utils.cpp" is added to the project.
v.1.6.4	February 2002:	SqDiff split criterion implemented
v.1.6.5	February 2002:	MGINI_SPLIT and GENENTROPY_SPLIT criteria implemented
v.1.6.6	February 2002:	Split Nodes Smoothing implemented (with new option "Nodes Smoothing")
v.1.6.7	February 2002:	New Smoothing Method: k-estimate
v.1.6.8	February 2002:	GINI Criterion fixed (CART) for more than 2 classes.
v.1.6.9	February 2002:	MSE_SPLIT remade with better results!
v.1.7.0	March 2002:	M AUC measure (Hand's measure) called AUCH implemented.
v.1.7.1	March 2002:	New split criterion: AUCH_SPLIT using Hand's measure.
v.1.8.0	March 2002:	Invented Datasets created. "Comb2Single" option with Kappa similarity method.
		Two bugs fixed: SECOND_RANDOM sometimes selected NULL or_trees and
		Setvdist() in "exemples.h" used word instead of long and caused problems with
		long datasets;
v.1.8.1	March 2002:	Combination Accuracy and Comb2Single Accuracy (means and StdDev)
		are computed when cross-validation.
		First Solution is also shown when combination is enabled.
v.1.8.2	March 2002:	Combination Methods: majority crisp and majority absolute stochastic corrected
v.1.9.0	March 2002:	New Combination Methods: MAX, MIN, GEOMEAN, ...
v.1.9.1	March 2002:	Smoothing in Combination
v.1.9.2	March 2002:	Postpruning now enabled also for combination (a new function PostPrune)
v.1.9.3	March 2002:	Combination options arranged.
v.1.9.4	April 2002:	First Solution (shown when combination) corrected.
		However, it doesn't work with post-pruning.

v.1.9.5	April	2002:	New options: PruneInCombination, SIMILARITY MEASURES for Comb2Single and AllowTestWithoutOneClass
v.1.9.6	April	2002:	Mean NumRules are also output of each kind of solution.
v.1.9.7	April	2002:	The memory of the random datasets were not freed. Two bugs: A temp object delete missing in the dataset constructor and no delete in main. There was also a similar memory link bug in "tracta_exemple".
v.1.9.8	May	2002:	Computed the number of possible solutions of the multitree. The AUC value can be computed by Hand or by our way.
v.1.9.9	June	2002:	Treat missing numeric values using !TYPES directive, '0U' or '0u' (extend numeric partition with an extra leaf for unknown value)
v.1.9.10	June	2002:	Now SMILES ignores some attributes when an 'I' or 'i' is put in the !TYPES.
v.1.9.11	June	2002:	New vector combination method: squared.
v.1.9.12	June	2002:	Suspended Nodes Forgetting. New options enabled to free memory from suspended nodes that probably will never be woken.
v.2.0B	June	2002:	Version 2.0 Beta release. **THIRD RELEASED VERSION.**
v.2.0	June	2002:	Version 2.0 stable release.
v.2.1	June	2002:	New way to compute AUC for more than 2 classes: Fawcett's method. New splitting criterion AUCS
v.2.1.1	July	2002:	Reliability is now calculated for each prediction. Two ways are implemented:
v.2.1.2	July	2002:	Reliability is used to compute 2-class AUC for combined and other solutions!!!!
v.2.1.3	July	2002:	Now SMILES accepts a ".testcost" file
v.2.1.4	July	2002:	AUC of comb. for 2 classes fixed. Now it matches AUC computed with nodes.
v.2.1.5	July	2002:	AUC of combination for more than 2 classes is implemented.
v.2.1.6	July	2002:	Prediction now works better for combination that uses "DIFFERENCE" or other methods that mangle the probability vector.
v.2.1.7	July	2002:	AUC by Hand (computed by nodes) is now implemented in a way that this measure is independent of the casual ordering of two or more nodes that have the same ratio.
v.2.1.8	July	2002:	In "roc.cpp" a double type is used for cardinalities and now smoothing works better. This has the following consequences: MSE and LogLoss can be slightly different if Options.ProbabilityInSplittingCriteria == FROM_FREQUENCY_SMOOTHING. AUC by nodes with LAPLACE SMOOTHING and AUC by examples with LAPLACE SMOOTHING now match.
v.2.2	July	2002:	testcost learning (modification of optimality measure implemented). For computing the total test cost, we check the repeated use of the same argument, and it is not reckoned, giving an exact measurement of testcost.
v.2.2.1	July	2002:	new way of selecting the Best Tree from the multitree: TEST_COST_BEST
v.2.2.2	July	2002:	the way in which TEST_COST_BEST now is weighted with the cardinality of each node. Now it computes the real TEST_COST
v.2.2.3	July	2002:	A new Best Tree Method: OCCAM_AND_TEST_COST_BEST that combines both OCCAM and TEST_COST through the use of a factor. The weight of each is determined by the new value TestCostRelevanceInSelectBest that must be from 0 to 1.
v.2.2.4	July	2002:	A new weight that tells how much TEST_COSTS are used to modify splitting criteria. TestCostRelevanceInSplitting that must be from 0 to infinite.
v.2.2.5	July	2002:	Two bugs fixed: "test cost method=no test costs" didn't work and there was a problem with numeric attributes with unknown values.
v.2.2.6	July	2002:	More friendly messages are output when the cost and testcost files are not provided correctly. The good way is something such as: ./smiles samples/liver.all.train - - liver.testcost
v.2.2.7	July	2002:	Some minor improvement in the parser. Now if a line doesn't contain the class, SMILES gives a proper message and exits.

v.2.2.8	August 2002:	The standard deviation of kfold crossvalidation for FirstAUC fixed. Testcost is also shown for Archetype solution.
v.2.2.9	August 2002:	There was a bug in the archetype extraction. Suspended Nodes were considered. Although it seems it didn't affect on results, it did on efficiency.
v.2.3.0	August 2002:	Implementation of the Archetype Use of Other Criteria.
v.2.3.1	August 2002:	Minor improvements to the interface.
v.2.3.2	September 02:	Test Costs not shown in statistics if they don't exist. Now AUC computed and shown for archetype. **FOURTH RELEASED VERSION**

Appendix B: Datasets in SMILES format

Along with the SMILES distribution, a lot of datasets can be downloaded from the SMILES webpage. These datasets are mostly taken from the UCI repository [2] and they are by no means an alternative repository but just a format adaptation. Some of them have been partially modified, so users of other systems should use the original datasets.

In the following pages we show the following information:

- name of the dataset
- usability (whether it is fast, modifications performed, ..)
- whether AUC calculation is feasible (there are a minimum of examples for each class)
- the name of the file without numeric missing treatment and with numeric missing treatment (when there are no numerical missing values, just the name in the first column is shown),
- whether or not is about medical domain,
- whether misclassification cost and test cost files are provided,
- the number of classes
- the size of all the examples and the percentage of the least frequent class,
- the size of all the examples without missing values and the percentage of the least frequent class only for the examples without missing values,
- the number of nominal and numerical attributes.

The experiments performed and shown in this manual have been done with these datasets.

Dataset	Usability	AUC Calc	Miss Val	File without numeric missing treatment	file with numeric missing treatment	Me d?	MisCl Costfile	Test Costfile	#c	Size All	%minc All	Size no-miss	%min c no-miss	attribs	
														nom	num
Monks1	Fast	**OK**	No	monks1.all + .train + .test		No	.cost	.testcost	2	566	50	566	50	6	0
Monks2	Fast	**OK**	No	monks2.all + .train + .test		No			2	601	34.28	601	34.28	6	0
Monks3	Fast	**OK**	No	monks3.all + .train + .test		No			2	554	48.01	554	48.01	6	0
Tic-tac	Fast	**OK**	No	tic-tac.all + .train + .test		No			2	958	34.66	958	34.66	8	0
House-votes	Fast	**OK**	No	house-votes.all +.train+.test		No			2	435	38.62	435	38.62	16	0
Agaricus-Lepiota	Fair	**OK**	No	agaricus.all		No			2	8124	48.2	8124	48.2	22	0
Breast-wdbc	Fair	**OK**	No	breast-wdbc.all		Yes			2	569	37.26	569	37.26	0	30
Breast-wpbc	Fair	**OK**	Yes	breast-wpbc.all	breast-wpbc-UM.all	Yes			2	198	23.74	194	23.71	0	33
BreastCancerWisc	Fast	**OK**	Yes	breast-cancer-wisc.all	breast-cancer-wisc-UM.all	Yes			2	699	34.48	683	34.99	0	9
Ionosphere	Fair	**OK**	No	ionosphere.all		No			2	351	35.9	351	35.9	0	34
Liver-bupa	Fast	**OK**	No	liver.all		Yes		.testcost	2	345	42.03	345	42.03	0	6
Pima Diabetes	Fast	**OK**	No	pima.all		Yes		.testcost	2	768	34.9	768	34.9	0	8
Chess-kr-vs-kp	Fair	**OK**	No	chess-kr-vs-kp.all		No			2	3196	47.78	3196	47.78	36	0
Sonar	Slow	**OK**	No	sonar.all		No			2	208	46.63	208	46.63	0	60
Hepatitis	Fast	**OK**	Yes	hepatitis.all	hepatititis-UM.all	Yes		.testcost	2	155	20.65	83	18.07	14	5
Thyroid-hypo	Slow	**OK**	Yes	thyroid-hypo.all	thyroid-hypo-UM.all	Yes			2	3163	4.63	2012	6.06	19	6
Thyroid-sick-eu	Slow	**OK**	Yes	thyroid-sick-eu.all	thyroid-sick-eu-UM.all	Yes			2	3163	9.26	2012	11.83	19	6
tae [{0}]	Fast. Reduced to 2 classes.	**OK**	No	tae2c.all		No			2	151	32.45	151	32.45	2	3
cars [{unacc}]	Fast. Reduced to 2 classes.	**OK**	No	cars2c.all		No			2	1728	29.98	1728	29.98	6	0
nursery [{nr}]	Fair. Reduced to 2 classes.	**OK**	No	nursery2c.all		No			2	12960	33.33	12960	33.33	8	0
pendigits [{0}]	Slow. Reduced to 2 classes.	**OK**	No	pendigits2c0.all		No			2	10992	10.4	10992	10.4	0	16
page-blocks [{0}]	Slow. Reduced to 2 classes.	**OK**	No	page-blocks2c0.all		No			2	5473	10.23	5473	10.23	0	10
yeast [{ERL}]	Fair. Reduced to 2 classes.	**OK**	No	yeast2c.all		No			2	1484	31.2	1484	31.2	0	8
letter [{a}]	Slow. Reduced to 2 classes.	**OK**	No	letter2c.all		No			2	20000	3.95	20000	3.95	0	16
optdigits [{0}]	Very Slow. Reduced to 2 c.	**OK**	No	optdigits2c0.all		No			2	5620	9.86	5620	9.86	0	64
Spect	Fast	**OK**	No	spect.all		No			2	267	25.94	267	25.94	22	0
SpectF	Slow	**OK**	No	spectf.all		No			2	349	27.22	349	27.22	0	44
balance	Fast	**OK**	No	balance-scale.all		No			3	625	7.84	625	7.84	0	4
cars	Fast	**OK**	No	cars.all		No	.cost		4	1728	3.76	1728	3.76	5	0
dermatology	Fast	**OK**	Yes	dermatology.all	dermatology-UM.all	Yes			6	366	5.46	358	5.59	33	1
echocardiogram	Fast	**OK**	Yes	echocardiogram.all	echocardiogram-UM.all	Yes			3	132	18.18	107	16.82	1	6
thyroid-new	Fast	**OK**	No	new-thyroid.all		Yes			3	215	4.65	215	4.65	0	5
nursery_4c	Fast. Reduced to 4 classes.	**OK**	No	nursery4c.all		No			4	12957	2.53	12957	2.53	8	0
page-blocks	Slow	**OK**	No	page-blocks.all		No			5	5473	5.12	5473	5.12	0	10
pendigits	Slow	**OK**	No	pendigits.all		No			10	10992	9.60	10992	9.60	0	16
tae	Fast	**OK**	No	tae.all + .train + .test		No		.testcost	3	151	32.45	151	32.45	2	3
iris	Fast	**OK**	No	iris.all		No			3	150	33.33	150	33.33	0	4
opt-digits	Very Slow	**OK**	No	optdigits.all		No			10	5620	9.80	5620	9.80	0	64
sat	Very Slow. No cross-valid!	**OK**	No	sat.all		No			6	6435	9.73	6435	9.73	0	36
image-segment	Fair	**OK**	No	segmentation.all		No			7	2310	14.29	2310	14.29	0	14
wine	Fast	**OK**	No	wine.all		No			3	178	26.97	178	26.97	0	13
Post-Operative	Fast	NO	Yes	post-operative.all	post-operative-UM.all	Yes			3	90	2.22	87	1.15	7	1
HeartDis-cleve	Fast	**Diff**	No	heart-disease-cleveland.all		Yes		.testcost	5	303	4.29	303	4.29	8	5

HEARTDIS-LONG	Fast	VDiff	Yes	heart-disease-longbeach.all	...-UM.all	YES		.TESTCOST	5	200	5.00	137	4.38	8	5
HEARTDIS-HUNG	Fast	**OK**	Yes	heart-disease-hungarian.all	...-UM.all	YES		.TESTCOST	2	294	36.05	270	37.41	8	5
HEARTDIS-SWITZ	Fast	NO	Yes	heart-disease-switzerland.all	...-UM.all	YES		.TESTCOST	5	123	4.07	117	4.27	8	5
HEARTDIS-ALL	Fair	**OK**	Yes	heart-disease-alltogether.all	...-UM.all	YES		.TESTCOST	5	920	3.04	827	2.90	8	5
SOYBEAN-SMALL	Fast	**Diff**	No	soybean-small.all		YES			4	47	21.28	47	21.28	35	0
AUTOSDRIVEWHEELS	Fast	**VDiff.**	Yes	autos-drivewheels.all	autos-drivewheels-UM.all	NO			3	205	4.39	160	5	9	16
ANNEALING	Fast	**Diff**	No	anneal.all		NO			5	898	0.89	898	0.89	32	6
GLASS	Fast	**VDiff**	No	glass.all		NO			6	214	4.21	214	4.21	0	9
CONNECT4	Very very slow.	OK	No	connect-4.all		NO			3	67557	9.55	67557	9.55	42	0
SOLAR FLAREC	Fast	NO	No	flarec.all		NO			3	323	2.17	323	2.17	10	0
SOLAR FLAREM	Fast	NO	No	flarem.all		NO			4	323	0.62	323	0.62	10	0
HAYES-ROTH	Fast	**OK**	No	hayes-roth.all		NO			3	160	19.38	160	19.38	4	0
WAVEFORM	Slow	**OK**	No	waveform.all		NO			3	5000	32.94	5000	32.94	0	21
CMC	Fast	**OK**	No	cmc.all		YES			3	1473	22.61	1473	22.61	7	2
ECOLI4C	Fast. Similar classes joined.	**OK**	No	ecoli4c.all		NO			4	336	7.44	336	7.44	0	7
PAGE-BLOCKS	Slow	**OK**	No	page-blocks.all		NO			5	5473	0.51	5473	0.51	0	10
YEAST	Fair.	NO	No	yeast.all		NO			10	1484	0.34	1484	0.34	0	8
YEAST-ERL	Fair. Yeast except ERL class	**Diff**	No	yeast-noERL.all		NO			9	1379	1.45	1379	1.45	0	8
LETTER	Very very slow.	OK	No	letter.all		NO			25	20000	3.67	20000	3.67	0	16
THYROID-ALLBP	Slow	**Diff**	Yes		thyroid-allbp-UM.all	YES			3	0	-	3772	0.37	22	7
THYROID-ALLHYPER	Slow	NO	Yes		thyroid-allhyper-UM.all	YES			5	0	-	3772	0.03	22	7
THYROID-ALLHYPER	Slow, “Second” cl. removed	Diff	Yes		thyroid-allhyper-sec-UM.all	YES			4	0	-	3771	0.27	22	7
THYROID-ALLHYPO	Slow	NO	Yes		thyroid-allhypo-UM.all	YES			4	0	-	3772	0.05	22	7
THYROID-ALLHYPO	Fair, “Second” cl. removed	**OK**	Yes		thyroid-allhypo-sec-UM.all	YES			3	0	-	3770	2.52	22	7
THYROID-ALLREP	Slow	**OK**	Yes		thyroid-allrep-UM.all	YES			4	0	-	3772	0.90	22	7
THYROID-ANN	Fair	**OK**	No	ann-thyroid.all		YES		.TESTCOST	3	7200	2.31	7200	2.31	15	0
LUNG-CANCER	Fast	NO	No	lung-cancer.all		YES			3	32	28.13	32	28.13	56	0
HRS-COLIC-OUTCOME	Fair. With Outcome as class	**OK**	Yes		horse-colic-outcome-UM.all	YES			3	366	14.21	20	20.00	14	8
HRS-COLIC-SURGICAL	Fair. With Surgical as class	**OK**	Yes		horse-colic-surgical-UM.all	YES			2	366	5.46	20	20.00	14	8
ARRHYTMIA2C	Slow. Class Normal vs. rest	**OK**	Yes	arrhythmia2c.all	arrhythmia2c-UM.all	YES			2	68	29.41	452	45.80	212	67
HABERMAN-BREAST	Fast	**OK**	No	haberman-breast.all		YES			2	306	26.47	306	26.47	0	3
POST-OPERATIVE-2	Fast. Class I removed	**OK**	Yes	post-operative-I.all	post-operative-I-UM.all	YES			2					7	1
ECOLI	Fast	NO	No	ecoli.all		NO			8	336	0.60	336	0.60	0	7
ECOLI6C	Fast. 3 less freq cl. joined	VDiff	No	ecoli6c.all		NO			9	336	2.68	336	2.68	0	7
NURSERY	Fast	NO	No	nursery.data		NO			5	12960	0.02	12960	0.02	8	0
SPAM	Very slow	OK	Yes	spam.all		NO			2	4601	39.40	4601	39.40	0	57
ADULT	Very slow	OK	No	adult.all		NO			2	48842	23.93	48842	23.93	8	6
CYL-BANDS	Fair. Two id values ignored.	OK	Yes	cyl-bands.all	cyl-bands-UM.all	NO			2	365	36.99	540	42.22	17	19
PLAYTENNIS	Very Fast. Toy Problem	No	No	playt.train + playt.test		NO			2					4	0
INVENTED	Very Fast. Toy Problem	No	No	invented.train +.test		NO			2					5	0
WATER	Very Fast. Toy Problem	No	No	water.train + water.test		NO			3					1	1
DRUG	Fast. From Clementine	Diff	No	drug.train + drug.test		YES			5	1100	7.00	1100	7.00	3	3