YaDT: Yet another Decision Tree builder

Salvatore Ruggieri
Dipartimento di Informatica, Università di Pisa
Via F. Buonarroti 2, 56127 Pisa, Italy
ruggieri@di.unipi.it
http://www.di.unipi.it/∼ruggieri

Abstract

YaDT *is a from-scratch* main-memory *implementation of the C4.5-like decision tree algorithm. Our presentation will be focused on the design principles that allowed for obtaining an extremely efficient system. Experimental results are reported comparing YaDT with Weka,* `dti`*, Xelopes and (E)C4.5.*

1. Introduction

The C4.5 decision tree algorithm of Quinlan [10] has always been taken as a reference for the development and analysis of novel proposals of classification algorithms. The survey [5] shows that it provides good classification accuracy and is the fastest among the compared *main-memory* classification algorithms. C4.5 has been further improved in efficiency in [11], where a patch called EC4.5 adds several optimizations in the tree construction phase. Unfortunately, C4.5 (and EC4.5) are implemented in the old style K&R C code. The sources are then hard to understand, profile and extend.

An ANSI C implementation (called `dti`) is available in the Borgelt's software library [1], while object oriented implementations are provided in Java by the Weka environment [12] and in C++ by the Xelopes[1] library [7].

In this paper, we describe a new from-scratch C++ implementation of a decision tree induction algorithm, which yields entropy-based decision trees in the style of C4.5. The implementation is called YaDT, an acronym for *Yet another Decision Tree builder*.

The intended contribution of this paper is to present the design principles of the implementation that allowed for obtaining a highly efficient system. We discuss our choices on memory representation and modelling of data and meta-data, on the algorithmic optimizations and their effect on memory and time performances, and on the trade-off between efficiency and accuracy of pruning heuristics.

1 At the time of writing, however, only ID3 (the precursor of C4.5) is available in the Xelopes C++ library.

2. Meta data representation

A decision tree induction algorithm takes as input a *training set* $\mathcal{TS}$, which is a set of *cases*, or *tuples* in the database terminology. Each case specifies values for a collection of *attributes*.

Each attribute has one the following *attribute types*: *discrete*, *continuous*, *weights* or *class*. The type of an attribute is concerned with its use in the tree construction algorithm, as we will see later. In the training set, there must be one and only one attribute of type *class* (the "target" attribute) and at most one of type *weights*.

The values of an attribute in a case belong to some *data type* including: *integer*, *float*, *double*, *string*. Also, they may include a special value (such as *'?'* or *NULL*), which denotes unknown values.

Summarizing, in YaDT meta data describing the training set $\mathcal{TS}$ can be structured as a table with columns: attribute name, data type and attribute type. Such a table can be provided as a database table, or as a text file such as:

```
outlook,string,discrete
temperature,integer,continuous
humidity,integer,continuous
windy,string,discrete
goodPlaying,float,weights
toPlay,string,class
```

Here, the classic *PlayTennis* example is reported, describing whether we played tennis or not under some outlook, temperature, humidity and windy conditions. The attribute *goodPlaying* is a measure of how good was the choice.

YaDT abstracts each *data type* by a C++ class **datatype** reported in Fig. 1. For each data type, it must be provided a constructor from a string representation of a value, a method

```
class datatype {
public:
  // Constructor
  datatype(const string & s);
  // String representation
  string toString() const;
  // Hashing function
  int hash();
  // Equality operator.
  bool operator ==(const datatype & dt);
  // Is there a total order among values?
  static bool totalOrder();
  // Semisum operator (only if totalOrder())
  datatype semisum(const datatype & dt);
  // Comparison operator (only if totalOrder())
  bool operator <(const datatype & dt);
};
```

Figure 1. A C++ class modelling data types.

to get back to the string representation, a hashing function, and an equality operator. Also, if the data type admits a total ordering (modelled by the **totalOrder()** method), then also a semi-sum operator and a comparison operator should be provided. The **totalOrder()** method is a link between the data type and the attribute type. Classes for which **totalOrder()** returns **true** model data types that can be used for continuous or weights attributes.

In principle, data types other than the basic ones (integer, float, strings) can be added to the system, provided that the interface of Fig. 1 can be designed for them. As an example, a *datetime* data type readily fits the interface. As a more interesting example, we could design a variant of the *float* data type, let us call it *dfloat*, that takes into account a non-uniform distribution of values, e.g. a normal one. Specifically, the **semisum()** operator for the *dfloat* data type does not return the semi-sum of two floats (which is the float equi-distant from two given ones under the uniform distribution), but the float equi-distant from two ones under the given distribution.

3. Data representation

From a logical point of view, the training set is a table whose column names, data types and attribute types are those described in meta data. Training data can be provided in YaDT as a database table or as a (possibly compressed) text file. As an example, training data for *PlayTennis* may include the following cases:

```
sunny,85,85,false,1,Don't Play
sunny,80,90,true,1,Don't Play
overcast,83,78,false,1.5,Play
rain,70,96,false,0.8,Play
overcast,64,65,true,2.5,Play
...
```

Since YaDT is a main-memory algorithm, training data is loaded into memory. While the choice of a data structure for storing the training set is not terrifically relevant for performance as for out-of-core algorithms, it is still important to accurately consider memory occupation. Let us review some approaches.

C4.5 models an attribute value by a `union` structure to distinguish discrete from continuous attributes.

```
typedef  union  {
        short  discr;
        float  cont;
     } AttValue;

typedef  AttValue **Table;
```

Distinct values of discrete attributes are stored in a specific array and the attribute value actually refers the position in such array – let us say that we store the *id*-value. Values of continuous attributes are stored directly in the `union` structure. A table is represented as a matrix where the first dimension is the *case number* and the second one is the *attribute number*. In other words, the table is stored *by rows*. Summarizing, at least $|\mathcal{TS}| \cdot |\mathcal{A}| \cdot sizeof(\textbf{float})$ bytes are required to store the training set, where $\mathcal{A}$ is the set of attribute names. Also, accessing an attribute value (e.g., `Table[3][2].cont`) requires two accesses in memory (the one to `Table[3]` and the one to `Table[3][2]`).

As C4.5, Weka stores the table *by rows*, using id-codes for discrete attributes. Both id-codes and continuous values are represented by a **double** data type. Since typically $sizeof(\textbf{double}) = 2 * sizeof(\textbf{float})$, Weka requires twice the memory needed by C4.5.

EC4.5 stores continuous values as discrete ones, i.e. it stores *id*-values both for continuous and discrete attributes. While this is useful for algorithmic optimizations, it does not improve on the memory requirements of C4.5, since *id*'s range over **int** and typically $sizeof(\textbf{int}) = sizeof(\textbf{float})$.

Xelopes store the table *by columns*, i.e. each attribute is represented as a vector of values (always of type **double**). As in C4.5, discrete attributes store *id*-values and continuous values are represented directly. While the memory requirements are the same of Weka, scanning the values of an attribute for a set of cases (which will be a common task of the algorithm) is now faster since each value can be retrieved with a single access in memory.

Finally, Borgelt's `dti` approach is in the middle between C4.5 and Xelopes, since it stores the table *by columns* (as Xelopes), but values are represented with the `union` structure (as in C4.5). Therefore, the memory occupation is the same as C4.5.

Let us present the YaDT solution. As in EC4.5, we store *id*-values both for discrete and continuous attributes (and also for the class attribute). As in `dti`, we store the table *by columns*. Differently from EC4.5 and `dti`, we can now observe that for n distinct attribute values (plus, possibly, the *unknown* value), $\lceil log(n+1) \rceil$ bits are sufficient to code *id*-values. Since coding *id*-values at bit-level compromises efficiency, YaDT uses the minimal integral data type (**bool**, **unsigned char**, **unsigned short**, **unsigned int**) that is represented with at least $\lceil log(n+1) \rceil$ bits.

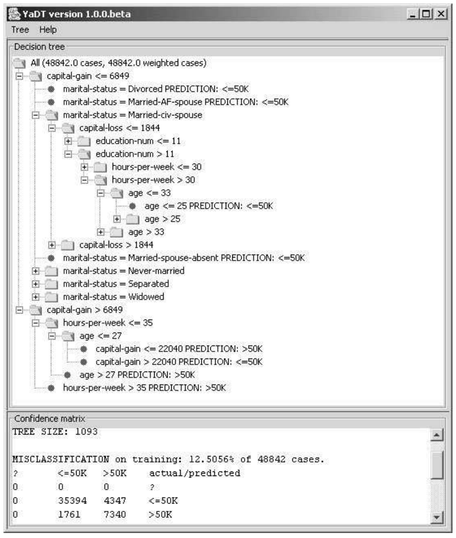
The major benefit of the approach is the following. Consider an attribute such as *age*. Since there are at most 256 distinct values for it (actually, much less), we can use an array of **unsigned char** to store the attribute indexes. Since $sizeof($**unsigned char**$) = 1$ and – on most machines – $sizeof($**float**$) = 4$, this means that storing the attribute requires $1/4$ of the space required by (E)C4.5 and `dti`. The same reasoning can be done with attributes with only two values (an array of **bool** suffices) and with at most 65536 values (an array of **unsigned short** suffices). Implementing such a parametric approach in C++ is quite natural and efficient by means of templates. As an example, let us consider the real world dataset `Adult` from the UCI Machine-Learning Repository [2]. It consists of 15 attributes reporting people age, workclass, education, race, sex, etc. The memory occupation of the dataset is 3Mb for (E)C4.5, 2.8Mb for `dti`, 6Mb for Weka and Xelopes, and only 1.1Mb for YaDT.

As a drawback of the chosen representation, two scans of the input training set are now required. First pass collects the distinct values of each attribute. These values are sorted and maintained in memory. Second pass reads values, lookups their position in the distinct value array and stores the position as the *id*-value.

4. Tree induction algorithms

A *decision tree* is a tree data structure consisting of *decision nodes* and *leaves*. A leaf specifies a class value. A decision node specifies a *test* over one of the attributes, which is called the attribute *selected* at the node. For each possible outcome of the test, a child node is present. In particular, the test on a discrete attribute A has h possible outcomes $A = d_1, \ldots, A = d_h$, where $d_1, \ldots d_h$ are the known values for attribute A. The test on a continuous attribute has two possible outcomes, $A \le t$ and $A > t$, where t is a value determined at the node, and called the *threshold*.

A decision tree is used to *classify* a case, i.e. to assign a class value to a case depending on the values of the attributes of the case. In fact, a path from the root to a leaf of the decision tree can be followed based on the attribute values of the case. The class specified at the leaf is the class *predicted* by the decision tree. A performance measure of a decision tree over a set of cases is called *classification error*. It is defined as the percentage of *mis-classified* cases, i.e. of cases whose predicted class differs from the actual class.



```
All (48842.0 cases, 48842.0 weighted cases)
  capital-gain <= 6849
    marital-status = Divorced PREDICTION: <=50K
    marital-status = Married-AF-spouse PREDICTION: <=50K
    marital-status = Married-civ-spouse
      capital-loss <= 1844
        education-num <= 11
        education-num > 11
          hours-per-week <= 30
          hours-per-week > 30
            age <= 33
              age <= 25 PREDICTION: <=50K
              age > 25
            age > 33
      capital-loss > 1844
    marital-status = Married-spouse-absent PREDICTION: <=50K
    marital-status = Never-married
    marital-status = Separated
    marital-status = Widowed
  capital-gain > 6849
    hours-per-week <= 35
      age <= 27
        capital-gain <= 22040 PREDICTION: >50K
        capital-gain > 22040 PREDICTION: <=50K
      age > 27 PREDICTION: >50K
    hours-per-week > 35 PREDICTION: >50K
```

```
TREE SIZE: 1093

MISCLASSIFICATION on training: 12.5056% of 48842 cases.
?       <=50K   >50K    actual/predicted
0       0       0       ?
0       35394   4347    <=50K
0       1761    7340    >50K
```

Figure 2. A decision tree built with YaDT.

A decision tree built with YaDT can be exported in text format, in an internal binary format, in XML format compliant to the Predictive Modelling Markup Language (PMML) specification [6]. Also, trees are navigable with a simple Java graphic user interface as shown in Fig. 2.

4.1. C4.5-like algorithm

The C4.5 algorithm constructs the decision tree with a *divide and conquer* strategy. Each node in a tree is *associated* with a set of cases. Also, cases are assigned *weights* to take into account unknown attribute values. At the beginning, only the root is present, with associated the whole training set $\mathcal{TS}$ and with all case weights equal to 1.0 (or, if present, to the value of the attribute with type *weights*). The following *divide and conquer* algorithm is executed, trying to exploit the locally best choice, with no backtracking allowed.

At each node, the *information gain* [10] of each attribute is calculated with respect to the cases at the node. For discrete attributes, the information gain is relative to the splitting of cases in T into sets with distinct attribute values. For continuous attributes, the information gain is relative to the splitting of T into two subsets, namely cases with attribute value *not greater than* and cases with attribute value *greater than* a certain *local threshold*, which is determined during information gain calculation.

The attribute with the highest information gain is selected for the test at the node. Moreover, in case a contin-

				No. of attributes			Elapsed time			
N	$\mathcal{TS}$ name	$\|\mathcal{TS}\|$	NC	Disc.	Cont.	Tot.	Weka	`dti`	EC4.5	YaDT
1	*Thyroid*	3,772	3	15	6	21	0.39s	0.90s	0.08	0.08s
2	*Statlog Satel.*	4,435	6		36	36	3.4s	2.6s	0.7s	0.5s
3	*Musk Clean2*	6,598	2	2	166	168	16.8s	33s	4.8s	1.5s
4	*Letter*	20,000	26		16	16	21s	10s	1.4s	1.1s
5	*Adult*	48,842	2	8	6	14	36s	11s	4.3s	2.6s
6	*St. Shuttle*	58,000	7		9	9	17s	12.2s	2.4s	0.6s
7	*Forest Cover*	581,012	7	44	10	54	∞	31m35s	4m53s	1m20s
8	*SyD106*	1,000,000	2	3	6	9	16m	5m46s	2m10s	1m24s
9	*KDD Cup 99*	4,898,431	22	7	34	41	∞	2h7m	19m05s	4m19s
10	*SyD107*	10,000,000	2	3	6	9	∞	2h26m	24m42s	10m32s

Table 1. ***Datasets used in experiments and elapsed time for building a decision tree (∞ means out of 1Gb main memory). Processor: Pentium IV 1.8Ghz. OS: Red Hat Linux 8.1.***

uous attribute is selected, the *threshold* is computed as the greatest value of the *whole* training set that is below the local threshold. The *divide and conquer* approach consists of recursively applying the same operations on a partition of cases (actually, cases with unknown value of the selected attribute are replicated in all child nodes) with proportional weights.

The classification error of a node is calculated as the sum of the errors of the child nodes. If the result is greater than the error of classifying all cases at the node as belonging to the most frequent class, then the node is set to be a leaf, and all sub-trees are removed.

4.2. YaDT optimizations

EC4.5 [11] implements several optimizations, mainly related to the efficient computation of information gain. At each node, EC4.5 evaluates information gain of attributes by choosing the best among three strategies. All the strategies adopt a binary search of the threshold in the whole training set starting from the local threshold computed at a node. The first strategy computes the local threshold using the algorithm of C4.5, which in particular sort cases by means of the *quicksort* method. The second strategy also uses the algorithm of C4.5, but adopts a *counting sort* method. The third strategy calculates the local threshold using a main-memory version of the RainForest [4] algorithm, which does not need sorting. The selection of the strategy to adopt is performed accordingly to an analytic comparison of their efficiency. We refer the reader to [11] for further details.

YaDT inherits from EC4.5 the same optimizations. In addition, it implements the approach of Fayyad and Irani [3], which speeds up finding the local threshold for continuous attributes by considering splittings at *boundary* values. v is a boundary value if there exist two cases at the node with attribute value v and with distinct class value, or if all cases with attribute value v at the node have the same class which is not the class of all cases with the successor attribute value.

As a further optimization, let us now consider the way a tree is built. After splitting a node, a (weighted) subset of cases are "pushed down" to each child node. How to represent then weighted subsets and the "pushing down" method?

(E)C4.5 maintains an array of weighted case indexes. After splitting a node, for each child the cases that must be pushed down are rearranged at the beginning of the array, and their weights updated (by a factor computed at the node). A depth-first strategy is necessarily adopted to build the tree. After a child tree has been completely built, the weights of cases are rolled back.

On the contrary, YaDT builds a weighted array for each node. On the one hand, the roll-back of weights is not necessary anymore. On the other hand, any building strategy can be adopted, since each node maintains its own private data. We experimented both a depth-first and a breadth-first growing strategy.

The depth-first strategy is slightly faster, since the following optimization can be implemented. Consider a node with n childs and assume that after building the first child tree the resulting error is greater than the one of making the node a leaf. In this case, the algorithm would cut all the child sub-trees. Therefore, we can prevent building child nodes 2 to n at once.

The breadth-first strategy has a better memory occupation performance, requiring to maintain arrays of weights and cases indexes for a total of at most $2 \cdot |\mathcal{TS}|$ elements, i.e. for all cases that may appear in at most two levels of the decision tree. With a depth-first strategy this upper bound can be much higher, especially when tests do not split cases uniformly among child nodes. For this reason, the default strategy in YaDT is the breadth-first one.

		YaDT simpl			YaDT+C4.5simpl		
		Time	Mem	Error	Time	Mem	Error
1	*Thyroid*	0.07s	181Kb	0.35%	0.07s	181Kb	0.35%
2	*Statlog Satel.*	0.38s	345Kb	36.7%	0.45s	455Kb	36.7%
3	*Musk Clean2*	1.2s	3.1Mb	0.45%	1.2s	3.1Mb	0%
4	*Letter*	0.8s	1.4Mb	14%	1.0s	1.9Mb	13.96%
5	*Adult*	1.8s	3.6Mb	13.89%	2.1s	5.4Mb	13.86%
6	*St. Shuttle*	0.46s	2.2Mb	0.057%	0.55s	3.9Mb	0.057%
7	*Forest Cover*	1m00s	30.9Mb	32.40%	1m29s	91.5Mb	32.71%
8	*SyD106*	57s	54.4Mb	0.76%	1m05s	59.6Mb	0.75%
9	*KDD Cup 99*	3m46s	341Mb	14.1%	4m10s	421Mb	14.1%
10	*SyD107*	8m13s	451Mb	0.31%	9m20s	549Mb	0.307%

Table 2. ***Time, memory and classification error comparisons between YaDT default simplification and YaDT with the C4.5 simplification procedures (datasets split into 70% training, 30% test; error is on test set).***

4.3. Some experiments on efficiency

The relevant characteristics of the training sets used in experiments are reported in Table 1. Each row contains the name of the training set ($\mathcal{TS}$ name), the number of cases ($|\mathcal{TS}|$), the number of class values (NC), the number of discrete attributes, the number of continuous attributes, and the total number of attributes. Training sets $(1-7)$ are taken from the UCI Machine-Learning Repository [2], while (9) is from the *KDD Cup Competition 1999* [8] and $(8, 10)$ are synthetic datasets generated by the Quest Generator [9] using function 5.

Table 1 reports the elapsed time of building a C4.5-like decision tree on the mentioned training sets for the Weka, `dti`, EC4.5 and YaDT systems. The elapsed time includes data loading and tree construction, but not tree simplification (see next section for this issue). The trees built are nearly the same, but not exactly the same mainly due to different arithmetical rounding errors. From Table 1, we derive the following observations:

Weka has critical memory limitations that lead to disk swapping for datasets (7,9,10). The problem is due to the use of the **double** data type for representing attribute values. In most cases, boolean, small integers or integers would have been sufficient. When not exceeding memory, Weka performs the worst. Looking inside the Weka source code, we note that it does linear search of threshold when the selected attribute is continuous. As noted in [11], this is the main source of C4.5 efficiency limitations and should be replaced with a binary search (obviously, this requires maintaining an ordered list of attribute values or a similar appropriate data structure).

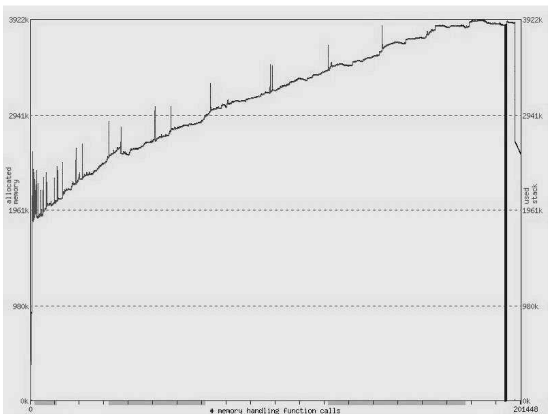
The `dti` system does not run out of memory, due to the use of the **float** data type for representing attribute values (instead of **double** as in Weka). Also, it prevents linear search of thresholds by setting the threshold equal to the local threshold. This is somewhat departing from the C4.5 algorithm. Moreover, there is no particular optimization in computing the information gain of continuous attributes. As a result, execution times become higher and higher as the number of continuous attributes increases (training sets (3,7-10)).

EC4.5 is a patch to C4.5 that performs several optimizations to the computation of information gain of continuous attributes. While the memory requirements are the same of C4.5 and `dti`, those optimizations allow for speeding up the execution time up to 75-80% for the medium-large training sets (9,10).

In addition to the optimizations of EC4.5 (and some further ones), YaDT maintains minimal data structures to store in memory the training set. This allows for building decision trees on larger training sets. For instance, the memory required by YaDT for storing the training set (9) in memory is about 250Mb against 860Mb required by EC4.5. Summarizing, YaDT is at least twice faster than EC4.5 and allows for reasoning on larger training sets.

5. Pruning decision trees

Decision tree are commonly pruned to alleviate the over fitting problem. The C4.5 system adopts an *error-based pruning* (EBP), which consists of a bottom-up transversal of the decision tree. At each decision node a pessimistic estimates is calculated of: (1) the error in case the node is turned into a leaf; (2) the sum of errors of child nodes in case the node is left as a decision node. If (1) is lower or equal than (2) then the node is turned into a leaf. In addition, C4.5 estimates also: (3) the error of *grafting* a child sub-tree in place of the node. More in detail, given the child node N with the maximum number of cases associated, (3) is calculated by "moving downwards" the cases associated to the node towards the child N and its sub-trees.



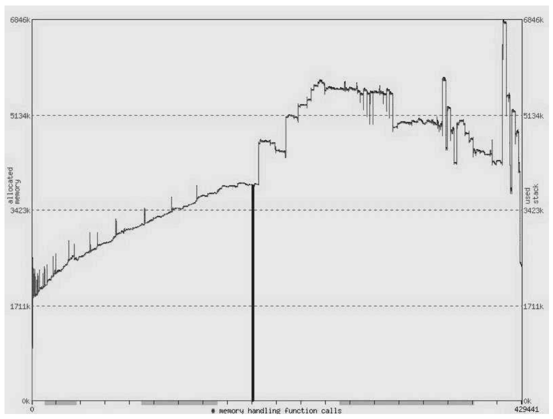


Figure 3. ***Memory usage over time for YaDT (left) vs YaDT with the full C4.5 simplification procedure (right) on the adult dataset. The vertical line denotes the end of the construction phase and the beginning of the pruning phase. Note that the X, Y scales of the two plots are different.***

It turns out that (3) is a time and memory consuming phase. In fact, (1+2) requires for each node to compute its error and to pass it upwards to the father node. (1+2+3) requires for each node to compute, in addition, the error of a whole sub-tree. By default, YaDT does not perform (3) – yet being an option to include it (as in Weka and `dti`). Table 2 reports the time, memory and error of trees simplified by default YaDT (i.e., (1+2)) and by YaDT with the C4.5 pruning procedure (i.e., (1+2+3)). In most cases, the error rates are the same. However, as the size of dataset increases, including step (3) turns into a much more demanding time and memory requirements.

Even more interesting is Figure 3, showing memory allocation over time. Default YaDT starts requiring memory for the dataset, then for each node of the tree being built. At the end of tree construction, the pruning steps (1+2) does not require significative additional memory or time. In contrast, steps (1–3) require a considerable amount of total time and the repeated allocation/release of large amounts of memory.

6. Conclusions

We have presented the design principles of YaDT on meta-data representation, data representation, algorithmic optimizations and tree pruning heuristics. We believe that those principles may be of general help in the design of old and new algorithms for decision trees induction, and, more in general, of main-memory divide-and-conquer AI algorithms.

References

[1] C. Borgelt. A decision tree plug-in for DataEngine. In *Proc. 6th European Congress on Intelligent Techniques and Soft Computing*, volume 2, pages 1299–1303, 1998. Verlag Mainz. `dti` version 3.12 from `http://fuzzy.cs.uni-magdeburg.de/~borgelt`.

[2] E. K. C. Blake and C. Merz. UCI repository of machine learning databases `http://www.ics.uci.edu/~mlearn/mlrepository.html`, 2003.

[3] U. M. Fayyad and K. B. Irani. On the handling of continuous-valued attributes in decision tree generation. *Machine Learning*, 8:87–102, 1992.

[4] J. E. Gehrke, R. Ramakrishnan, and V. Ganti. RainForest — A framework for fast decision tree construction of large datasets. *Data Mining and Knowledge Discovery*, 4(2/4):127–162, 2000.

[5] T. Lim, W. Loh, and Y. Shih. A comparison of prediction accuracy, complexity, and training time of thirthy-tree old and new classification algorithms. *Machine Learning Journal*, 40:203–228, 2000.

[6] Predictive Model Markup Language (PMML). Version 2.0. `http://www.dmg.org`.

[7] Prudsys AG. The XELOPES library (eXtEnded Library fOr Prudsys Embedded Solutions) v. 1.1 for C++, May 2003. `http://www.prudsys.com`.

[8] KDD Cup Competion Data Sets. On-line documentation, 1999. `http://www.epsilon.com/new/1datamining.html`.

[9] Quest synthetic data generation code. On-line documentation, Visited in May 2003. `http://www.almaden.ibm.com/software/quest`.

[10] J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, San Mateo, CA, 1993.

[11] S. Ruggieri. Efficient C4.5. *IEEE Transactions on Knowledge and Data Engineering*, 14:438–444, 2002.

[12] I. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations.* Morgan & Kaufmann, 2000. Weka version 3.2.3 from `http://www.cs.waikato.ac.nz/ml/weka`.