Tilde
Top-down Induction of Logical DEcision trees
User's Manual

Hendrik Blockeel
v1.3, July 21, 1997

Contents

1 Introduction

This is a user's manual for the inductive logic programming system TILDE. It explains how TILDE can be used, but not what it does or how it works. For information on things such as first-order logical decision trees, discretization, ... we refer to [1, 2] which discuss these topics in the context of TILDE and offer further references.

2 Installing TILDE

At this moment TILDE is available for Sun workstations running the SunOS4.1.3 or Solaris (SunOS5.5.1) operating system.

The TILDE system comes in a file called `TILDE_*.tar.gz`. The asterisk is replaced by some characters indicating the version of TILDE it contains (which platform, ...).

Run the following commands:

```
gunzip TILDE_*.tar.gz
tar xvf TILDE_*.tar
```

A directory `TILDE` has now been created in the current directory. From now on we refer to this directory as TILDE's home directory.

3 Running TILDE

The environment variable `TILDE_RT_DIR` must be set to TILDE's home directory.

The script `TILDE_RT` in the directory `$TILDE_RT_DIR` starts TILDE. No parameters are required. `TILDE_RT` must be run in the directory where the data files are. It is recommended to put `$TILDE_RT_DIR` in your path or define an `alias tilde=$TILDE_RT_DIR/TILDE_RT`.

The environment variable `TILDE_BIM_PROLOG_COMPILER` should be set to YES or NO, depending on whether a ProLog-by-BIM compiler is available locally. If this compiler is available and `TILDE_BIM_PROLOG_COMPILER` is set to

YES, background knowledge will be compiled, and the `opt` command (which optimises the knowledge base) becomes available in TILDE. This results in better performance.

If `TILDE_BIM_PROLOG_COMPILER` is not set, a message is written to the screen and the default value NO is assumed.

4 Input Files for TILDE

The directory where TILDE is called, should contain the following files:

app`.kb`
app`.bg`
app`.s`

where *app* is the name of the application.

If more than one *app*`.kb` file is present in the directory where TILDE is run, TILDE will ask the user which application it should run on.

- *app*`.kb` contains all models, in the same format as used by the Claudien [4] and ICL [6] systems. We refer to the Claudien and ICL manuals [7] for more information.
- *app*`.bg` contains background knowledge. This file is optional.
- *app*`.s` contains application-specific settings, the refinement operator, ... This file is discussed in Section 6.

5 The knowledge base

The file *app*`.kb` contains the examples on which TILDE will run. *app*`.bg` contains the background knowledge (which is a Prolog program). TILDE uses the *learning from interpretations* setting [5, 3]. In this setting each example is an interpretation, and therefore in TILDE examples are represented by Prolog programs (which together with the background knowledge generate these interpretations).

An example is represented as follows:

`begin(model(`*name*`)).`
Prolog program
`end(model(`*name*`)).`

The Prolog program will normally contain a fact indicating the class of the example, and may contain other information that strictly spoken does not belong to the interpretation generated by the example (e.g. information on how to partition the data set for cross-validation).

Example 1 *In the* Machines *example application that comes with* TILDE*, the background knowledge file* `mach.bg` *is:*

```
replaceable(gear).
replaceable(wheel).
replaceable(chain).

not_replaceable(engine).
not_replaceable(control_unit).
```

The following two examples are taken from `mach.kb`*:*

```
begin(model(1)).
testnr(1).
sendback.
worn(gear).
worn(engine).
end(model(1)).

begin(model(2)).
testnr(2).
keep.
end(model(2)).
```

The `testnr` *facts are administrative information;* `sendback` *and* `keep` *are class indicators. We will not include them in the interpretations.*

The interpretation generated for example 1 is then:

{*worn(gear), worn(engine), replaceable(gear), replaceable(wheel), replaceable(chain), not_replaceable(engine), not_replaceable(control_unit)*}

and for example 2:

$$\{replaceable(gear),\ replaceable(wheel),\ replaceable(chain),\ not\_replaceable(engine),\ not\_replaceable(control\_unit)\}$$

6 The settings file

The file *app*`.s` contains the definition of the refinement operator, information on discretization, heuristics, lookahead specifications, ...

6.1 Miscellaneous Specifications

The following specifications are optional, i.e. if they are not present, default values will be used. For most of these values it is best not to change them, unless you know what you are doing.

- `classes(`*classlist*`).`
  Classlist is a list specifying into which classes the examples have to be classified. Each class has to be a ground atom. **Default:** `[pos, neg]`.

- `output_options(`*optionlist*`).`
  optionlist is a list that specifies what should be written to the output file, and in what format. The user may wish to see a lot of output information, or only the most important things (e.g. classification accuracy). This can be controlled with this setting. A more detailed description is given in Section 8, where the output file is discussed. **Default:** `[c45,prolog]`.

- `minimal_cases(`n`).`
  This is the minimal number of examples that a leaf of the tree should cover. **Default: 2.**

- `heuristic(`*heur*`).`
  heur = `gain` | `gainratio`
  This specifies the heuristic that is to be used. `Gainratio` is C4.5's default heuristic; it usually yields better results than `gain`. **Default:** `gainratio`.

- `confidence_level(`*cl*`).`
  cl is a number between 0 and 1, that is used in the post-pruning phase for estimating the prediction error. **Default: 0.25.**

- `accuracy(`*a*`).`
  a is a number between 0 and 1, and specifies the minimal accuracy that has to be reached in order to stop splitting a node any further. **Default: 1.**

- `talking(`*t*`).`
  t is an integer between 0 and 5 (included). It controls the amount of output TILDE writes to the screen. With $t = 0$, no output is written. With $t = 4$ all clauses that are tested are written to the screen. $t = 5$ is only useful for debugging purposes. **Default: 2.**

6.2 Refinement Operator Specification

In order to compute a split for a node, TILDE takes the query that was used to generate this node (i.e. all examples covered by this node, and only those, are models for that query), and tries to refine it by adding a conjunction to it, in such a way that an optimal split between examples of different classes is obtained.

The refinement operator (and consequently, the hypothesis language) is specified using facts of the following form:

`rmode(`*N*`:` *conj*`).`

Such a fact specifies that a refinement step can consist of adding the conjunction of literals *conj* to the query to be refined. This can happen at most N times on the basis of this fact. However, if other specifications allow *conj* to be added as well, they are counted independently.

For the variables occurring in the conjunction, modes can be specified. 3 modes are available:

- $+X$ means that the variable is an input variable; i.e. it has to be bound when adding this literal. To this end, the variable is unified with a variable already occurring in the query.

- $-X$ means that the variable can, but need not be bound (unification with other variables is possible but not mandatory).

- X means this is a new variable; no unification is performed with already existing variables (though variables that are introduced later on may be unified with this variable)

If a variable occurs several times in one conjunction, its mode should be indicated only once.

Example 2 *Suppose the following* `rmode`*-facts (among others) are given:*

```
rmode(3: p(+X)).
rmode(3: q(+X, Y)).
rmode(3: (r(-X), s(X)).
```

Then a query `?- a(X), b(Y)` *can be refined into:*

```
?- a(X), b(Y), p(X).          % p's argument has to be bound
?- a(X), b(Y), p(Y).
?- a(X), b(Y), q(X,Z).        % q's first argument has to be bound,
?- a(X), b(Y), q(Y,Z).        % its second argument is a new variable.
?- a(X), b(Y), r(X), s(X).    % r's argument can but need not be bound
?- a(X), b(Y), r(Y), s(Y).    % and s has the same argument as r
?- a(X), b(Y), r(Z), s(Z).
```

Another form of the `rmode`-specification is

`rmode(`N`: #(`n`*`m`*`V`: `*conj1*`, `*conj2*`)).`

where n and m are numbers, and V is a variable or a structure (list or other) containing variables. V shares variables with both *conj1* and *conj2*.

Just before the actual refinements are computed, the conjunction *conj1* is called. It can be called for at most N examples (each call occurs in the context of a different example). For each example, at most m answer substitutions for the variables that are shared with V are stored. Each answer substitution of V generates an instantiation of *conj2*, and each such instantiation is considered for addition to the current clause.

This construction is useful for generating literals that contain constants.

Example 3 *Suppose the current query to be refined is* `?- a(X), b(Y,Z).` *Then the following specification*

```
rmode(1: #(1*10*C: member(C, [1,2,3,4,5,6,7,8,9,10]), +X = C)).
```

gives rise to these refinements:

```
?- a(X), b(Y,Z), X=1.
?- a(X), b(Y,Z), X=2.
...
?- a(X), b(Y,Z), X=10.
?- a(X), b(Y,Z), Y=1.
?- a(X), b(Y,Z), Y=2.
...
?- a(X), b(Y,Z), Y=10.
?- a(X), b(Y,Z), Z=1.
?- a(X), b(Y,Z), Z=2.
...
?- a(X), b(Y,Z), Z=10.
```

The specification

```
rmode(1: #(1000*3*C: p(C), p(C))).
```

yields, for example:

```
?- a(X), b(Y,Z), p(2.4).
?- a(X), b(Y,Z), p(1.8).
?- a(X), b(Y,Z), p(1.1).
?- a(X), b(Y,Z), p(1.5).
?- a(X), b(Y,Z), p(2.3).
...
```

In each example (with a maximum of 1000), 3 values for p's argument that occur in that example are chosen.[1] *These constants will occur in the possible refinements. For instance, in the above example, it might be that the first model (example) contained the facts* `p(2.4)`, `p(1.8)`, *the second* `p(1.1)`, `p(1.5)`, `p(2.3)`, `p(2.8)` *(with only the first 3 of these 4 selected), and so on.*

Remark: due to the Prolog syntax definition, there has to be a space between the colon and the `#`.

[1] It is not specified how the examples are chosen.

6.3 Constraints on Variables

Note: This feature is still in a test phase.

It is possible to explicitly specify constraints that must hold for variables that occur in certain places. These constraints are akin to the use of modes and types, but offer more flexibility.

Constraints can be specified in the following way:

`constraint(`*conj*`,` *constr*`).`

This specification means that whenever *conj* is added to a clause, *constr* should not be violated.

Example 4 *Suppose* `bond` *literals can be added under the following conditions:*

```
rmode(5:bond(+X, +Y, Z)).
constraint(bond(X, Y, Z), X \== Y).
```

According to the `rmode` *specification,* `bond(`X`,`Y`,`Z`)` *can be added to a clause if* X *and* Y *are unified with some variable in that clause. If* X *and* Y *have the same type, they might be unified with one and the same variable* V*, such that* `bond(V,V,Z)` *is added to the clause. However, the constraint specification prohibits this.*

The second argument of the `constraint` specification can be any Prolog query. Moreover, two extra predicates can be used: `occurs/1` and `not_occurs/1`. These check whether a predicate already occurs in a clause.

Example 5 *The specification*

```
constraint(p(X,Y), not_occurs(p(X,Z))).
```

tells TILDE *that p should not be added with a first argument that already occurs as first argument of a p-literal. Another interesting constraint is*

```
rmode(p(+X, +Y)).
constraint(p(X,Y), occurs(p(_, X))).
```

This can e.g. be used to construct "chains" of p-literals, such as `p(A,B)`, `p(B,C)`, `p(C,D)`*, avoiding the combinatorial explosion of possible unifications that would occur if each variable could be unified with any variable already occurring.*

6.4 Types

If a typed language is used, variables can only be unified if their types correspond (i.e. the queries must be type-conform). A typed language can be specified by putting the following fact into the settings:

```
typed_language(yes).
```

If this fact is present, then type specifications *have* to be present for each predicate. If a predicate is untyped, this can still be indicated by using anonymous variables where normally type names would be put.

A type specification looks as follows:

`type(`*pred*`(`$targ_1$`, ..., `$targ_n$`)).`

Pred is the name of the predicate, $targ_i$ is a constant denoting a type, or a variable.

Example 6 *These are some type specifications:*

```
type(info(string, number)).
type(number < number).
type(X = X).
type(member(_, list)).
```

Variables that have the same type can always be compared using the equality operator, but the operator $<$ *will only be used with variables that have the type* `number`*. The* `member` *predicate is partially untyped: its second argument must be a* `list`*, but its first argument can be anything.*

Note that types only influence the way in which variables are unified, and nothing else. Constants are always considered to be untyped. For instance, if the type declaration `type(p(a,b))` is present, a literal such as `p(1,1)` might be added if the `rmode` declarations allow it.

6.5 Lookahead

When refining queries, TILDE can look ahead in the refinement lattice, in those cases where that is allowed explicitly by the user. Lookahead is computationally expensive, but in some cases the quality of a refinement can

be assessed better. Therefore it is important to allow lookahead where it is useful, but not more than necessary.

Typically, lookahead is useful when a conjunction is added that in itself will always succeed (i.e. it does not yield any gain), but introduces new variables that may be important for the classification. The advantage of adding such conjunctions would otherwise be underestimated.

Lookahead-specifications look as follows:

`lookahead(`*pattern*`,` *conj*`).`

Such a specification indicated that, whenever a conjunction is added that matches with *pattern*, the conjunction *conj* can (but need not) be added as well.

Example 7 *Consider the following lookahead specifications:*

```
lookahead(next_to(X,Y), large(Y)).
lookahead((on(X,Y), next_to(Y,Z)), on(X,Z)).
```

They tell TILDE *that whenever* `next_to(A,B)` *can be added, the addition of* `next_to(A,B), large(B)` *(in one refinement step) also has to be considered. And whenever a conjunction* `on(A,B), next_to(B,C)` *can be added,* `on(A,B), next_to(B,C), on(A,C)` *should be tried as well.*

Lookahead can be allowed recursively, e.g.:

```
lookahead(next_to(X,Y), next_to(Y,Z)).
```

allows a chain of `next_to` literals to be introduced in one refinement step. In order to avoid infinite recursion, a maximal lookahead depth can be given by means of

`max_lookahead(`n`)`

n is the maximal number of lookahead levels that is allowed.

6.6 Discretization

Discretization is a technique used by symbolic learners to handle numeric data. A continuous domain is transformed into a discrete domain by introducing discrete values that correspond to intervals in the continuous domain. The discrete domain can be characterised by the thresholds between the intervals. For instance, the continuous domain $[0, 1]$ could be discretized into three discrete values `{small,average,large}` corresponding to the intervals $[0, 0.25)$, $[0.25, 0.75)$, $[0.75, 1]$. This particular discretization is characterised completely by the thresholds 0.25 and 0.75.

The question is, then, how to find suitable values for these thresholds. TILDE uses ICL's discretization algorithm [12] which is based on Fayyad and Irani's paper [10, 9]. In this manual, we do not discuss the algorithm, but focus on how the user can control the discretization process.

The user can indicate that a variable has a continuous domain and has to be discretized, by means of:

`discretization(bounds(`*n*`)).`
`to_be_discretized(`*pred*`, `*varlist*`).`
`to_be_discretized(`*pred*`, `*n*`, `*varlist*`).`

n is the number of thresholds discretization is allowed to yield at most. It can be specified for all predicates by means of `discretization(bounds(`*n*`))`, but this value can be overridden for any specific predicate by explicitly adding it to the `to_be_discretized` specification. *Varlist* contains the variables that are to be discretized.

Example 8 *Let us take a look at the following specifications:*

```
discretization(bounds(3)).
to_be_discretized(employee(Name, Address, ID, Age), [Age]).
to_be_discretized(wage(ID, Wage), 5, [Wage]).
```

Ages of employees are discretized into four discrete values (i.e. there are three thresholds). Wages are discretized with five thresholds.

Discretization is performed one time, before the induction itself starts. After discretization is done, a predicate `discretized` is available that contains the results of discretization in the following format:

`discretized(`*pred*, *varlist*, *constlist*`)`

Pred and *varlist* should correspond with the *pred* and *varlist* arguments of a `to_be_discretized` fact. The result of the discretization of the variable is the list *constlist*, which contains all thresholds that have been found.

Example 9 *For the specification of example 8, the following results might have been obtained:*

```
discretized(employee(Name, Address, ID, Age), [Age], [21,35,50]).
discretized(wage(ID, Wage), [Wage], [35000, 45000, 50000, 70000, 85000]).
```

A typical use of these results would be:

```
rmode(#(1*10*C: (discretized(employee(_, _, _, X), [X], L),
                 member(C, L)),
        +Age < C)).
rmode(#(1*10*C: (discretized(wage(ID, X), [X], L), member(C, L)),
        +Wage < C)).
```

The `#`-construct indicates that some age or wage variable should be compared with a constant that occurs in the list of discretization thresholds of ages or wages.

6.7 Specifying an Alternative Root Query

By default, the root query (this is the query associated with the root of the tree, i.e. the query from which TILDE starts refining by adding literals to it) is *true*. In some cases it may be desirable to have an alternative root query. Typically, this is the case when there are determinate literals that with certainty are known to be relevant for classification. In such cases it is not necessary to let the system find out for itself that these literals are relevant, the more so because relevance is not always obvious in the case of determinate literals.

It is possible to specify an alternative root query by means of the `root(RootQuery)` specification. The default setting is equivalent to `root(true)`.

Example 10 *For a certain classification task involving molecules, it is known that the weight of a molecule is relevant. One way of defining the refinement operator would be:*

```
rmode(1:weight(W)).
rmode(5: #(1*10*C: (discretized(weight(W), [W], L), member(C, L)),
          +Weight < C).

lookahead(weight(Weight),
          #(1*10*C: (discretized(weight(W), [W], L), member(C, L)),
            Weight < C).
```

The tests on weights can only be added after `weight(W)` *has been added, and since* `weight(W)` *itself never yields any gain, a separate lookahead specification is necessary. All this can be done more easily by putting* `weight(W)` *in the root query:*

```
root(weight(W)).

rmode(5: #(1*10*C: (discretized(weight(W), [W], L), member(C, L)),
          +Weight < C).
```

This way, a variable indicating the weight of the molecule will always be available for testing.

The advantages of the latter approach become more obvious when more variables are introduced by the root query.

In the extreme case where the root query introduces a set of variables S and none of the nodes can contain any other variables than the ones in S, the decision tree becomes a propositional one.

7 Commands in TILDE

- `monitor.`
  starts a separate window in which the progress of the induction is shown. During the induction process this window is updated every few seconds, showing how many examples have already been covered by leaves. Also, the part of the tree that has already been formed is shown. Users with lots of time to waste can watch the tree grow.

- `go(`*output_file*`).`
  `go.`

starts induction (possibly preceded by discretization), generating a logical decision tree. Output is written to *output_file*. If `go` is used without an argument, the output file is called `output`. Exactly what is written to the output file, is discussed in Section 8. With this command, induction is always performed on the whole dataset.

- `nfold(`n`).`
  performs an n-fold cross-validation. A random partition of n sets is first created, and subsequently n runs of the induction algorithm are performed. For each run a different set of the partition has been left out of the training data, so training is done on the examples of $n-1$ sets, and the remaining set is used as a test set. Output of these runs is written to files with fixed names: `ur1`, `ur2`, ..., `ur`n.

  In order to make cross-validation possible, it is necessary that a fact `testid(`*modelname*`)` is available in each model, that explicitly indicates the name of the model. This name has to be the same name as the name occurring in the `begin(model(`*name*`))` and `end(model(`*name*`))` specifications.

  Remark: the respecification of the modelname is in fact redundant. In the most recent versions of TILDE (from v1.3 onwards) it is not required anymore.

- `nfold(`n`, `s`).`
  does the same as `nfold(`n`)`, but s is a seed used to create the random partition. If in consecutive cross-validations the same value for s is used, each cross-validation will be done on the same partition. This is useful with respect to reproduceability of results.

- `leave_one_out`.
  runs an m-fold cross-validation, with m the total number of examples; each time $m-1$ examples are used for training, and one example for testing. Output is written to `ul1`, `ul2`, ..., `ul`m.

  For the leave_one_out procedure to work, a fact `testnr(`i`)` has to be available in each model, assigning a unique number i from 1 to m to the model.

  Remark: This command will probably be removed in later versions of

TILDE. It can be simulated by using `nfold(`n`)` with n the number of examples.

- `leave_one_out_from_list(`*list*`)`
  assumes a predicate `testid` to be available, on which cross-validation will be based. For each constant c in *list*, the data are partitioned in those examples where `testid(`c`)` succeeds (these will form the test data), and those where it does not succeed (these will form the training set). This gives rise to an l-fold cross-validation, with l the length of the list (assuming that in each model `testid` succeeds for one constant only).

  Output is written to files `uL`c, with c the constant from the list on which the run was based.

  An example of where this command could be used, is the Mesh dataset [8], where cross-validation is often done based on the 5 structures that appear in the data.

- `tenfold.`
  will probably be removed in the future. This runs a tenfold cross-validation based on a `testnr` predicate, where in the i-th cross-validation the models with a number n such that $n \bmod 10 = i$ are used as test set. This can be simulated with `leave_one_out_from_list`. Output files are called `ut`i with $i = 0 - 9$.

- `opt.`
  is only available if a ProLog-by-BIM compiler is available and the environment variable `TILDE_BIM_PROLOG_COMPILER` is set to YES. This command causes TILDE to generate an optimised (compiled) version of the knowledge base. A file *app*`.kb.O.wic` is created containing the optimised code. Whenever TILDE is subsequently run, it will use the optimised version of the knowledge base if it is not older than the unoptimised version. This will result in much better performance (both faster loading of the knowledge base and faster induction).

  Note that TILDE has to be restarted for the optimisation to take effect. The `opt` command generates an optimised version of the knowledge base, but does not change the internal database of TILDE. Only by

exiting TILDE and restarting it, the optimised version of the data will be loaded.

Also note that the optimisation process takes a while. Dots appear on the screen indicating the progress. There is one dot for 100 examples.

8 Output files

8.1 Result Files

The results of each run of the induction algorithm are written to a file. The filename is the argument of the `go` command, or `ur`*i* for a random n-fold cross-validation, etc.

The output of a run contains the following statistics:

- CPU-times for discretization and induction itself
- Complexity, accuracy on training set, accuracy on test set, and global accuracy (both sets together) are shown:
  - for the original tree
  - for a more compact but equivalent version of the above tree
  - for the pruned tree (TILDE uses a post-pruning algorithm that is based on the algorithm used in Quinlan's C4.5 system [11].)

Moreover, extra information can be written according to several output options. These output options are specified by means of

`output_options(`*list*`).`

with *list* a list containing one or more of the following constants:

- `c45`
  the pruned tree is written in C4.5-like output format, i.e. the tree's root is to the left, *yes* and *no* branches are drawn, and for each leaf the total number of examples covered, and the number of examples *correctly* predicted are shown between brackets.

- `c45c`
  writes the compact version of the original tree, in the same format as above.

- `c45e`
  writes the pruned tree in the same format as `c45`, but also writes for each leaf the list of examples that are covered by that leaf.

- `c45ce`
  writes the compact version of the original tree with examples shown

- `lp`
  writes the logic program corresponding to the pruned tree

- `prolog`
  writes the Prolog program corresponding to the pruned tree

- `elaborate`
  writes a lot of information: the thresholds found for the discretization, very elaborate representations of trees, ... — this is mainly used for debugging purposes, and makes the output much less readable.

By default, `output_options([c45,prolog])` is assumed.

8.2 Summary Files

When performing cross-validations, TILDE not only gives detailed reports on each run, but also generates a summary of the whole cross-validation. Two files are generated: `CT.`x and `SUMMARY.`x, where x = `ul`, `uL`, `ut` or `ur`, depending on which kind of cross-validation is performed (this is consistent with the names of the output files).

- `CT.`x contains a contingency table of real vs. predicted classes. For this table, Cramer's coefficient is reported. This coefficient is defined as

$$V = \sqrt{\frac{\chi^2}{n(q-1)}}$$

with

$$\chi^2 = \sum_{i=1}^{q} \sum_{j=1}^{q} \frac{(x_{ij} - e_{ij})^2}{e_{ij}}$$

where

n is the total number of examples
x_{ij} is the number of examples in row i and column j
$e_{ij} = \frac{r_i \cdot c_j}{n}$, the number of examples expected in row i and column j
r_i is the total number of examples in row i
c_j is the total number of examples in column j
q is the number of classes

While the χ^2-statistic gives an idea of how significantly the prediction differs from random prediction, V scales it to a number between 0 and 1 and can therefore be used as some sort of correlation coefficient. For $q = 2$, V equals the classical correlation coefficient for 2×2-tables, i.e. for a table $\boxed{\begin{array}{cc} A & B \\ C & D \end{array}}$, $\varphi = \frac{AD-BC}{\sqrt{(A+B)(A+C)(B+D)(C+D)}}$.

- `SUMMARY.`*x* contains a summary of the induction times, tree complexities and predictive accuracies of all the runs in the cross-validation, as well as their average values and standard deviations.

8.3 Other Output Files

Some other output files are generated that may be of use.

- The file *app*`.progress` is updated by TILDE each time a leaf of the tree has been created (i.e. it is continuously updated during the induction process). It contains the number of leaves created up till now, as well as the number of examples that have been covered by these leaves (both as an absolute number and as a percentage of the total number of examples). As such, it gives the user some idea of how far the induction process has proceeded. TILDE's monitor accesses this file.
- The file *app*`.ptree` is updated by TILDE each time a test or leaf is added to the tree. It contains the partial tree induced up till now, in c45-format. This file, too, is accessed by TILDE's monitor.

- The file **pruned_ct** contains a couple (*real class*, *predicted class*) for each example in the dataset. These data are used for computing the contingency table.

9 Acknowledgements

TILDE was developed by Hendrik Blockeel and Luc De Raedt. Its development was made possible by a grant from the Flemish Institute for the Promotion of Scientific and Technological Research in the Industry (IWT), which supports Hendrik Blockeel, by the Fund for Scientific Research of Flanders, which supports Luc De Raedt, and by the European Community Esprit project 20237, Inductive Logic Programming 2.

Many thanks are due to Wim Van Laer and Luc Dehaspe, whose code for the ICL and Claudien systems was in part reused in TILDE; this has significantly accelerated TILDE's development. Luc Dehaspe, Wim Van Laer, Nico Jacobs, Johannes Fürnkranz, Sašo Džeroski and Bart Vandromme have been very helpful with comments and discussions on the TILDE system.

References

[1] H. Blockeel and L. De Raedt. Experiments with top-down induction of logical decision trees. Technical Report CW 247, Dept. of Computer Science, K.U.Leuven, January 1997. Also in Periodic Progress Report ESPRIT Project ILP2, January 1997. **http://www.cs.kuleuven.ac.be/-publicaties/rapporten/CW1997.html**.

[2] H. Blockeel and L. De Raedt. Lookahead and discretization in ILP. In *Proceedings of the 7th International Workshop on Inductive Logic Programming*. Springer-Verlag, 1997.

[3] L. De Raedt. Induction in logic. In R.S. Michalski and Wnek J., editors, *Proceedings of the 3rd International Workshop on Multistrategy Learning*, pages 29–38, 1996.

[4] L. De Raedt and L. Dehaspe. Clausal discovery. *Machine Learning*, 26:99–146, 1997.

[5] L. De Raedt and S. Džeroski. First order jk-clausal theories are PAC-learnable. *Artificial Intelligence*, 70:375–392, 1994.

[6] L. De Raedt and W. Van Laer. Inductive constraint logic. In *Proceedings of the 5th Workshop on Algorithmic Learning Theory*, volume 997 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 1995.

[7] L. Dehaspe, L. De Raedt, and W. Van Laer. Claudien : a clausal discovery engine: a user manual. Technical report, KUL, 1994.

[8] B. Dolšak and S. Muggleton. The application of Inductive Logic Programming to finite element mesh design. In S. Muggleton, editor, *Inductive logic programming*, pages 453–472. Academic Press, 1992.

[9] J. Dougherty, R. Kohavi, and M. Sahami. Supervised and unsupervised discretization of continuous features. In A. Prieditis and S. Russell, editors, *Proc. Twelfth International Conference on Machine Learning*. Morgan Kaufmann, 1995.

[10] U.M. Fayyad and K.B. Irani. Multi-interval discretization of continuous-valued attributes for classification learning. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence*, pages 1022–1027, San Mateo, CA, 1993. Morgan Kaufmann.

[11] J. Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann series in machine learning. Morgan Kaufmann, 1993.

[12] W. Van Laer, S. Džeroski, and L. De Raedt. Multi-class problems and discretization in ICL (extended abstract). In *Proceedings of the MLnet Familiarization Workshop on Data Mining with Inductive Logic Programming (ILP for KDD)*, 1996.