The ACE Data Mining System

User's Manual

H. Blockeel L. Dehaspe J. Ramon J. Struyf A. Van Assche C. Vens
D. Fierens

March 9, 2009

Contents

1 Introduction **7**

2 Installing and Running ACE **9**

2.1 Using ACE on Linux 9

2.2 Using ACE on Windows 10

2.3 The ACE Command Prompt 10

2.4 Miscellaneous Commands 10

3 An Example Application **11**

4 Input Files **15**

4.1 The Knowledge Base 15

4.2 The Settings File 17

4.2.1 Specifying the goal of the induction 17

4.2.2 Language Bias (simple) 19

4.2.3 Language Bias (advanced) 20

4.2.4 General Settings 33

4.2.5 Reading and Changing Settings 34

4.2.6 Loading Packages 35

5 General predictive interface **37**

5.1 Building predictive models in ACE 37

5.2 General settings 38

5.3 General output files 38

5.3.1 Detailed result files 38

5.3.2 Summary files 38

5.4 Ensemble methods 40

5.4.1 Constructing ensemble models 40
5.4.2 Settings specific for ensemble methods 40

6 TILDE **43**
6.1 Growing trees with TILDE 43
6.2 Settings specific for TILDE 43
6.2.1 Settings used for all modes of operation 43
6.2.2 Settings specific for classification mode 44
6.2.3 Settings specific for clustering mode 45
6.2.4 Settings specific for regression mode 46
6.2.5 Settings specific for model tree mode 47
6.3 Specific output for TILDE 48

7 The ICL system **51**
7.1 Building rule sets with ICL 51
7.2 Settings specific for ICL 51
7.3 An Example Application 52
7.3.1 The background knowledge (BG file) 53
7.3.2 The example data (KB file) 53
7.3.3 The language file 54
7.3.4 The settings file 54
7.3.5 The output file 55

8 The WARMR System **57**
8.1 The WARMR Algorithm 57
8.2 An Example Application 58
8.2.1 The Frequent Queries 59
8.3 Generating Rules 60
8.3.1 Generating Query Extensions 60
8.3.2 Generating Horn Clauses 62
8.4 Commands 62
8.5 Settings 63

9 The RRL system **65**

9.1 RRL input files 66
9.1.1 Defining the environment 66
9.1.2 Using guidance 67
9.2 Commands 67
9.3 Settings 67

10 Incremental learning systems **69**
10.1 The TG algorithm 69
10.1.1 Introduction 69
10.1.2 Language bias 70
10.1.3 Settings 71
10.1.4 Known issues 72
10.2 The TG-Conv algorithm 72
10.3 The KBR algorithm 72
10.3.1 Introduction 72
10.3.2 Settings 72
10.4 The RIB algorithm 72
10.4.1 Introduction 72
10.4.2 Settings 73
10.5 The TNI algorithm 73
10.6 The IRC algorithm 73

11 Utility Packages **75**
11.1 The Hypothesis Space Package 75
11.2 The Query Package 75
11.3 The Prolog Prompt 76
11.4 Destructive Arrays and Matrices 76
11.5 The Linear Regression Package 77
11.6 Loading .ARFF Files 78

References **79**

Chapter 1

Introduction

ACE is a data mining system that provides a common interface to a number of relational data mining algorithms. Relational data mining is the process of finding patterns in a relational database possibly consisting of multiple tables, and extends classical data mining in the sense that in the latter case only patterns within single tuples are found, whereas patterns found by relational data mining systems may extend over different tuples of different relations.

Currently ACE encompasses the following algorithms.

- Tilde is an upgrade of the decision tree learner C4.5 [35] towards relational data mining; it builds decision trees that allow to predict the value of a certain attribute in a relation from other information in the database.
- Warmr is an upgrade towards relational data mining of the APRIORI algorithm [1]; it looks for frequently occurring patterns in a relational database.
- ICL is a relational rule learner. It is an upgrade of the rule learner CN2 [11] towards relational data mining.
- RegRules is a system for performing linear regression with relational features.
- KBR is a system for learning with first order kernels.
- NLP is a system for learning neural logic programs.
- RIB3 is a relational instance based learning system.
- TG is an incremental version of Tilde.
- RRL is a system for performing reinforcement learning [33, 39] in a relational context. RRL can use the following incremental regression systems: KBR, NLP, RIB3 and TG.

Except for a brief introduction to the systems included in ACE, this manual does not describe all the different applications of these systems, nor how the systems work. Information on the representation of hypotheses, the techniques and algorithms underlying the systems, as well as possible applications can be found in the following publications.

- (Tilde) A brief introduction to Tilde can be found in [4]. Inducing clustering trees with Tilde is described in [5]. A detailed description of the Tilde system can be found in [2].
- (Warmr) Information about Warmr can be found in [15].

- (ICL) Information about ICL is available in [41].
- (RRL) Relational reinforcement learning is introduced in [28]. Different relational regression techniques that can be used in RRL are described in [24, 23, 32]. Guidance [21] can also be used in RRL.
- (Common) Most of the systems in ACE support lookahead and discretization. These techniques are described in [3].
- (Efficiency) ACE includes several techniques for optimizing the efficiency of the induction process. The most important ones are query-packs [6] and query transformations [37, 38].

What this manual does describe, is how to use the ACE system. It gives information about how to prepare the input files, how to interpret the output files, and how to interact with ACE.

Acknowledgements

Tilde was developed by Hendrik Blockeel and Luc De Raedt. Warmr was developed by Luc Dehaspe, Luc De Raedt and Jan Ramon. Many improvements and extensions to both systems were implemented by Jan Ramon, Wim Van Laer and Jan Struyf.

The development of these systems was made possible by

- the Flemish Institute for the Promotion of Scientific and Technological Research in the Industry (IWT), which supported Hendrik Blockeel and Jan Ramon,
- the Fund for Scientific Research of Flanders, which supported Luc De Raedt, Wim Van Laer, Hendrik Blockeel and Jan Struyf.
- the European Community Esprit project 20237, Inductive Logic Programming 2, which supported Luc Dehaspe.
- the European Community Esprit project 28623, Aladin (Applied Logic for Advanced Data Mining in Industry), which supported Wim Van Laer and motivated the development of a more user-friendly interface as well as efficiency improvements.
- the Research Fund of the K.U.Leuven, which supported Hendrik Blockeel and Luc Dehaspe

Many other people have contributed to the development of these systems with comments, suggestions and code. Wim Van Laer's code for the ICL system was in part reused in Tilde, which has accelerated its development significantly. Nico Jacobs, Kurt Driessens, Johannes Fürnkranz, Sašo Džeroski and Bart Vandromme have been very helpful with comments and discussions on the Tilde system. Tilde's rule post-pruning algorithm was designed and implemented by Gerry Pelgrims. Anneleen Van Assche and Celine Vens have designed and implemented first order random forests with complex aggregates.

Chapter 2

Installing and Running ACE

This chapter explains how to install and run ACE. It assumes that you have obtained the version of ACE that matches your operation system from the ACE website[1].

2.1 Using ACE on Linux

The Linux version of ACE comes in a file called "ACE-x.y.z-linux.tar.gz", with x.y.z the version number. To install it, extract this archive file in a directory of your choice. For example, to install ACE in your home directory, issue the following command:

```
tar -xvzf ACE-x.y.z-linux.tar.gz
```

This assumes that you downloaded "ACE-x.y.z-linux.tar.gz" and saved it into your home directory. You should now have a new subdirectory called ACE-x.y.z.

Next, set the environment variable ACE_ILP_ROOT as follows:

```
ACE_ILP_ROOT=$HOME/ACE-x.y.z/linux
export ACE_ILP_ROOT
```

ACE can now be started by running the command "$ACE_ILP_ROOT/bin/ace". Note that ACE needs a settings file to run, otherwise it will print out "No .s file found, exiting.". A full example of running ACE on a small data set is covered in Chapter 3.

If you use the "bash" shell (the default on most Linux systems), then it is useful to place the following in the ".bashrc" settings file, which is located in your home directory.

```
export ACE_ILP_ROOT=$HOME/ACE-x.y.z/linux
alias ACE=$ACE_ILP_ROOT/bin/ace
```

If you now log in again (or type "bash -login"), then you will be able to start ACE with the command alias "ACE".

[1] http://www.cs.kuleuven.be/~dtai/ACE

2.2 Using ACE on Windows

The Windows version of ACE comes in a file called "`ACE-x.y.z-windows.exe`", with `x.y.z` the version number. To install it, start it by double clicking it in Windows Explorer. This will start a wizard, which will guide you through the installation procedure.

After ACE is successfully installed, you should have a new entry in your start menu for ACE. This entry includes, among others, an icon for opening the Windows command prompt. ACE can be started by opening this command prompt and issuing the command "`ACE`". Note that ACE needs a settings file to run, otherwise it will print out "No .s file found, exiting.". A full example of running ACE on a small data set is covered in Chapter 3.

2.3 The ACE Command Prompt

After starting ACE, it will show the following prompt

```
Starting interactive session
****************************

For list of commands : h/0 or help/0

ace>
```

which can be used to give commands to the ACE system. This includes commands for running the learning systems such as Tilde and Warmr. We discuss each of these commands in the chapters describing these systems. To obtain a list of all available commands, use the " `help`" command.

- `help -- h`
  `help(`*topic*`) -- h(`*topic*`)`
  `help` shows a list of available commands and settings. Detailed information about one of these can be obtained with `help(`*topic*`)`.

2.4 Miscellaneous Commands

- `help -- h`
  `help(`*topic*`) -- h(`*topic*`)`
  `help` shows a list of available commands and settings. Detailed information about one of these can be obtained with `help(`*topic*`)`.
- `quit -- q`
  terminates the program.
- (expert) `prompt -- p`
  allows the user to leave the ACE interactive session and go to the Prolog prompt.

Chapter 3

An Example Application

Distributed together with the ACE system is a dataset called Machines. The Machines dataset consists of a description of a number of machines, each of which have a number of components. Some of these components may be worn, while others are still in good shape. The dataset also contains for each machine an action that is to be undertaken: it should either be fixed, sent back to the manufacturer, or else no action need be undertaken.

On this Machine data set a number of learning tasks can be defined. For instance, a first task could be to derive a criterion that allows one to decide for a new machine whether it should be fixed, sent back, or left alone based on the components in it that are worn. In other words, the task consists of learning a classifier, a function that classifies machines into one of three classes. The function should of course be consistent with the data set. The Tilde algorithm could be used to solve this task.

A second possible task is to find out which kind of components are often worn, or possibly which combinations of components are often together worn. This is what we call a search for frequent patterns, and for this kind of task the Warmr algorithm is useful. Related to the discovery of frequent patterns is the discovery of association rules; association rules are rules of the form "when pattern p occurs, pattern q usually also occurs" with p and q to be filled in by the discovery system. This kind of tasks, too, can be performed by Warmr.

Example 1. *Assume the following data are given:*

Background*:*

```
/* information about which components are replaceable and which are not */
replaceable(gear).
replaceable(wheel).
replaceable(chain).
not_replaceable(engine).
not_replaceable(control_unit).
```

Examples*:*

```
Machine 1: worn(gear), worn(engine), sendback. /* gear & engine are worn */
Machine 2: ok.                                 /* nothing is worn here */
Machine 3: worn(gear), fix.                    /* gear is worn, fix machine */
Machine 4: worn(engine), sendback.
Machine 5: worn(gear), worn(chain), fix.
```

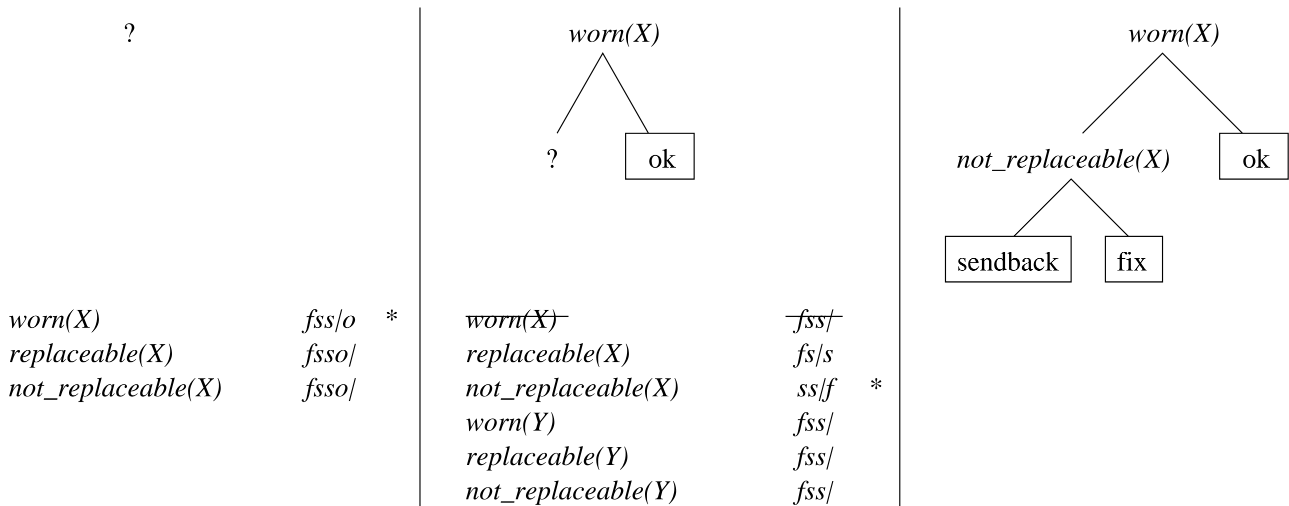

Figure 3.1: Decision tree corresponding to the classification rule in Example 1

What is the relationship between the action to be undertaken (sendback, fix or ok) and the worn components? The TILDE *system, when confronted with the above data, might build a classifier of the following form:*

if the machine contains some component X that is worn and not replaceable
then `sendback`
else if it contains some component X that is worn {*but none that are worn and not replaceable*}
then `fix`
else {*it contains no worn components*} the machine is `ok`.

Classifiers induced by TILDE *are represented in the form of a so-called first order logical decision tree. The tree corresponding to the above rule is the rightmost one in Figure3.1.*

The WARMR *system could be used on these data to check which patterns frequently occur. For instance, it might detect that in 60% of the cases the gear is worn; that in 60% of the cases some replaceable component is worn (be it gear, chain or wheel); that in 40% of the cases a machine has to be sent back; that in 40% of the cases the engine was worn; and that in 40% of the cases both the engine was worn and the machine had to be sent back. From the last three statements it follows e.g. that in all cases where the machine had to be sent back, the engine was worn. This is an association rule.*

In order to be able to use TILDE and WARMR, it is useful to understand to some extent how they work. Both systems search for hypotheses (patterns, classifiers, or whatever) by starting with one or more trivial (and obviously wrong) working hypotheses and continuously refine these working hypotheses until correct hypotheses are obtained. This refinement process is to some extent controlled by the user of the system.

For instance, when building the above-mentioned rule TILDE might first notice that the occurrence of worn components has some influence on the classification of the machine, so it builds a first approximation of the classifier :

if the machine contains worn components **then** ? **else** the machine is `ok`

The **else** part of the rule is already correct, but the **then** part needs to be refined. After the following refinement step TILDE will arrive at the classifier mentioned above. The gradual construction of the

final theory is easiest to understand when looking at the theories in tree format; this is illustrated in Figure 3.1.

Similarly, WARMR starts with simple patterns and combines them into more complex patterns. For instance, it would first detect that in 40% of the cases a machine contains a worn engine and that in 40% of the cases it needs to be sent back, and only afterwards combine these two patterns into a new pattern: in 40% of the cases a machine both has a worn engine and needs to be sent back.

As mentioned, the process of refining patterns, rules etc. until good ones are obtained, is controlled by the user. Basically, the user tells the system how patterns can be extended to form more complex patterns. In its simplest form this just means that the user tells ACE which properties are allowed to occur in the patterns. Experienced users can control the refinement process in a much more detailed manner. This is exactly what a large part of this manual is about.

Chapter 4

Input Files

The directory where ACE is run should contain files called *app*`.kb`, *app*`.bg` and *app*`.s` where *app* is the name of the application. The use of these files is as follows:

- *app*`.kb` : this file contains the examples, e.g., the descriptions of the machines in our running example. Both training and test data are included in this file.
- *app*`.bg` : this file, which is optional, contains background knowledge. In our running example, the information on replaceability would be put here.
- *app*`.s` : this is called the settings file; it allows the user to control certain parameters of the algorithms incorporated in ACE. This file is discussed in Section 4.2.

4.1 The Knowledge Base

The knowledge base is assumed to be in the files *app*`.kb` (which should contain example-related information) and *app*`.bg` (which should contain background knowledge about the domain). A predicate or relation can be considered to be background knowledge if adding an example to the set of examples does not change the definition of that predicate.

The *appl*`.kb`, *appl*`.bg` and *appl*`.s` files are prolog files. They can be created using a text editor or by a small prolog program translating you favorite database format into prolog.

There are two different formats available for the knowledge base. In one format, which we call the *models* format, each individual example is described by a so-called model, a block beginning with `begin(model(`*name*`))` and ending with `end(model(`*name*`))`. All facts in between the model delimiters are considered to describe properties of the single example.

The second format, which we refer to as the *key* format, is closer to normal Prolog syntax and can actually be seen as just a Prolog program. In this format, there is a less clear distinction between example descriptions and background knowledge. Individual examples are referred to by a certain identifier, and properties of a single example are given by listing facts that refer to this identifier.

Example 2. *The examples of the Machines dataset mentioned above can be represented using the models format as follows:*

```
begin(model(machine1)).
worn(gear).
```

```
worn(engine).
sendback.
end(model(machine1)).

begin(model(machine2)).
ok.
end(model(machine2)).

begin(model(machine3)).
worn(gear).
fix.
end(model(machine3)).

begin(model(machine4)).
worn(engine).
sendback.
end(model(machine4)).

begin(model(machine5)).
worn(gear).
worn(chain).
fix.
end(model(machine5)).
```

In this format the background knowledge can be represented as follows:

```
replaceable(gear).
replaceable(wheel).
replaceable(chain).
not_replaceable(engine).
not_replaceable(control_unit).
```

When using the key format, facts about specific examples need an extra argument to indicate which example they refer to. The distinction between background and example predicates becomes somewhat blurred in this case. One could say that background predicates are those predicates that do not refer to any specific example. In the context of our Machines dataset this distinction is obvious; in some situations however it may not be so clear-cut.

```
machine(machine1, sendback).
worn(machine1, gear).
worn(machine1, engine).

machine(machine2, ok).

machine(machine3, fix).
worn(machine3, gear).

machine(machine4, sendback).
worn(machine4, engine).

machine(machine5, fix).
```

```
worn(machine5, gear).
worn(machine5, chain).

replaceable(gear).
replaceable(wheel).
replaceable(chain).
not_replaceable(engine).
not_replaceable(control_unit).
```

Note that the models format imposes a higher modularity on the data than the key format, and may be preferable when different examples do not share much relevant information. The key format is closer to the relational database format and may be preferable if the data are originally stored in such a database.

The ACE command `knowledge_info` (shorthand `ni`) can be used to display some information about the knowledge base that is currently loaded.

4.2 The Settings File

The file *app*`.s` contains a number of settings that influence the way in which ACE works. These settings can mainly be divided in two kinds: settings that define the language bias (the kind of patterns that can be found), and settings that control the system in some other way.

The language bias can be defined in two ways. For beginning users there are the `warmode`-settings; these are simple to use and allow to define a good language bias very quickly. `warmode`-settings are automatically translated by the system to a lower level consisting of `rmode`, `type` and other settings. The user can also specify the language directly at this lower level, which offers better control of the way in which the program traverses the search space but is more complicated.

In the following sections we first discuss the features allowing a simple language bias specification, then the advanced features, and finally the remaining (not language-related) settings are explained.

4.2.1 Specifying the goal of the induction

The most important thing to tell ACE is the goal of the induction process; i.e., what Tilde has to predict, or what Warmr has to count. The way in which this is done is different for the two available formats.

The Key Format

In Tilde, the user indicates the goal of the induction by means of the `predict` declaration. The `predict` predicate takes as argument an atom of the predicate where each argument is `+` or `-` (these are mode declarators; see further). `-` symbols indicate the arguments that are to be predicted.

For instance, if the 3rd argument of a predicate p of arity 4 is to be predicted from the other information in the database, the declaration is

```
predict(p(+,+,-,+))
```

The arguments to be predicted can be numeric or symbolic; depending on this the regression, classification of clustering subsystems of TILDE will be used. If a typed language is used (see further), types can be added to the arguments of the predicate, behind the mode declarators. The above example then becomes, e.g.,

```
predict(p(+name,+address,-phone,+fax))
```

In WARMR, the atom that will be used to identify examples is indicated with the `warmode_key` declaration.

`warmode_key(`*pred*`(`$-$*targ*$_1$`, ..., `$-$*targ*$_n$`)).`

All queries will have the atom specified with `warmode_key` in the front. Each substitution of the variables of this atom will be associated with an example. The total number of such substitutions for which the query succeeds will determine the frequency of the query.

The Models Format

In the models format, it is unnecessary to tell WARMR what it has to count: it automatically counts models. For TILDE, the user still needs to specify what the model that is going to be built should predict. This can be done in two ways: one can use the predict setting as in the key format, but with the key arguments dropped, or one can use the approach outlined hereafter.

In the classification setting, a list of classes needs to be given. A class in this context is just a nullary predicate that succeeds for a given example if and only if the example belongs to the class. The list of classes is given using the `classes` setting. It defaults to `[pos, neg]`.

In the regression setting of Tilde, the target attribute is a numeric argument of a predicate. Which argument of which predicate is to be predicted is indicated using the `euclid` setting. E.g.,

```
euclid(person(ID, Name, Age), Age).
```

indicates that the Age of persons is to be predicted. Each example is supposed to contain one `person` literal; the third argument of this literal is the one that should be predicted.

Multiple `euclid` facts may be given, indicating that there are multiple target variables (i.e., a vector of numbers is to be predicted instead of a single number).

Regression, Classification, Clustering

TILDE actually consists of several instantiations of a generic "top-down induction of decision trees" algorithm; these different instantiations allow it to perform regression, classification or clustering. The `tilde_mode` setting indicates which instantiation of TILDE should be used; possible values for it are `classify`, `regression` and `cluster`.

Example 3. *For the Machines example, the classes are* `fix`*,* `sendback` *and* `ok`*. This is indicated by putting in the settings file*

```
classes([sendback,fix,ok]).
```

In the key setting, a `predict` *setting would be used:*

```
predict(machine(+,-)).
```

TILDE *automatically determines which values occur for the argument that is to be predicted.*

Assume we would want to predict a cost for the reparation of machines; this cost is a number. We then need to set TILDE *in regression mode, which is done by specifying*

```
tilde_mode(regression).
```

(For classification no `tilde_mode` *need be mentioned because classification is the default.)*

Assuming the cost of examples in the key setting is indicated by facts such as `cost(machine, 10)`*, then the* `predict` *setting becomes*

```
predict(cost(+,-)).
```

whereas in the models setting, assuming the cost is indicated by facts such as `cost(10)`*, a* `euclid` *setting needs to be given:*

```
euclid(cost(X), X).
```

4.2.2 Language Bias (simple)

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

`warmode(`*pred*`(`tm_1`,`tm_2`,...,`tm_n`)).`

Such a fact specifies that a refinement step can consist of adding the literal *pred/n* to the query to be refined. The *tm*'s are type and mode declarations for variables (explained below) or constants.

According to variant

`warmode(N:`*pred*`(`tm_1`,`tm_2`,...,`tm_n`)).`

the literal can be added at most N times. By default, N is infinite.

The arguments in the `warmode` declaration are mode declarators, possibly extended with types. Three basic modes are available:

- + 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.
- − 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)
- \ means that a new variable is to be put here, and a constraint should be added to the clause stating that the value of the variable has to be different from the values of any other variable of the same type.

The basic modes can be combined, yielding the additional mode operators `+-`, `+\`, `-\`, and `+-\`. For instance, `+-` means that the variable can, but need not be bound (unification with other variables is possible but not mandatory).

Example 4. *Suppose the following* `warmode`*-facts (among others) are given:*

```
warmode(p(+)).
warmode(q(+,-)).
warmode(r(+-\)).
```

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).          % r's argument can but need not be bound
?- a(X), b(Y), r(Y).
?- a(X), b(Y), r(Z).
?- a(X), b(Y), r(Z), Z \== X, Z \== Y.
```

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 specifying a type after the mode of a variable.

Example 5. *These are some type specifications:*

```
warmode(info(+string, -number)).
warmode(+number < +number).
warmode(member(+-, +list)).
```

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 `warmode(p(+a,-b))` is present, a literal such as `p(1,1)` might be added if the other `warmode` declarations allow it.

4.2.3 Language Bias (advanced)

Rmode Specifications

At the lower level, the refinement operator (and consequently, the hypothesis language) is specified using the `rmode` predicate. The simplest form of an `rmode` fact is the following:

`rmode(`*conj*`).`

Such a fact specifies that a refinement step can consist of adding the conjunction of literals *conj* to the query to be refined.

If the conjunction can be added at most N times, the following form can be used:

rmode(N: *conj*).

N can be any natural number or `inf` (for infinity). Writing `inf` is equivalent to not providing a maximum (as in the simplest form of `rmode`).

Note that if several `rmode` facts allow the same conjunction to be added, the total number of times the literals can be added is the sum of all N's. You can view `rmode` facts as "bags" from which conjunctions can be taken; if one bag is empty the same conjunction might be chosen from another bag.

For the variables occurring in the conjunction, modes can be specified. The same modes are available as for warmodes. If a variable occurs several times in one conjunction, its mode should be indicated only once, at its first occurrence.

Example 6. *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).
```

Note the difference with `warmode`-specifications: variables are used instead of types, and conjunctions can be specified instead of single literals.

Generation of constants

Suppose that a literal needs to be added that contains some constant in the place of one of its arguments. One could for instance write mode declarations such as

```
rmode(p(+X, high)).
rmode(p(+X, medium)).
rmode(p(+X, low)).
```

When many constants are meaningful in such a position, specifying rmodes in this way becomes tedious. The ACE system provides several ways to automatically generate rmodes with different constants.

A simple form of constant generation, which is just an abbreviation of the above, is

```
rmode(p(+X, #[high, medium, low])).
```

In rmodes the `#`-sign always is a placeholder for a constant. When it is followed by a list, the `#` symbol will be replaced by each individual element of the list. When it stands alone, it will be replaced by any constant that appears in that place anywhere in the data. Thus, when ACE encounters the rmode

```
rmode(p(+X, #)).
```

it will check which values can occur as the second argument of the p/2 predicate in the data; the # will be replaced by each of these values.

Finally, the # symbol can be followed by a variable. In such cases, it will be replaced by any value that the mentioned variable could be instantiated to, according to the first literal where it occurs (which may but need not be the literal in which the # occurs). For instance, in

```
rmode((p(+X, Y), Y < #Y)).
```

constants are determined as follows: the variable occurring after # is Y; Y first occurs in p(X,Y), so it is checked in the data which values occur as the second argument of p/2. Assuming the values 3, 7, 12, 16 are found, then the current clause can be refined with the following four conjunctions :

```
rmode((p(+X, Y), Y < 3)).
rmode((p(+X, Y), Y < 7)).
rmode((p(+X, Y), Y < 12)).
rmode((p(+X, Y), Y < 16)).
```

Note that the stand-alone # can be seen as an abbreviation of #_ with _ an anonymous variable.

To summarize this: besides (possibly moded) variables or constants, there are three kinds of #-constructs that can be put in the place of arguments of literals: #*list*, #*var* or #.

When there is more then one #-construct in an rmode specification, all posible combinations of the constants that can be substituted into these constructs will be generated.

`begin(model(1)).`	`rmode(p(#,#)).`
`  p(a,x).`	
`end(model(1)).`	`p(a,x)`
`begin(model(2)).`	`p(a,y)`
`  p(b,y).`	`p(b,x)`
`end(model(2)).`	`p(b,y)`

Sometimes it makes sense to have all combinations, especially when comparing a variable with a #*var* construct, but most of the times it is much more efficient to try only those combinations that effectively occur in the examples. Therefor it is possible to assign each #-construct to a group. For a specific group only the combinations that occur in the data are used. To assign a construct to a group the syntax is extended as follows:

#	⟶	#*group*
#[...]	⟶	#(*group*, [...])
#*var*	⟶	#(*group*, *var*)

#-constructs without an indication for a group are always in separate groups (each #-construct a different group).

Example 7.

`begin(model(1)).`	`begin(model(2)).`
`  p(a,1).`	`  p(a,3).`
`  p(b,2).`	`  p(c,1).`
`  q(u).`	`  q(v).`
`end(model(1)).`	`end(model(2)).`

```
rmode(p(#,#)).                     | rmode(p(#1,#1)).
                                   |
p(a,1)   p(b,1)   p(c,1)           | p(a,1)   p(c,1)
p(a,2)   p(b,2)   p(c,2)           | p(a,3)
p(a,3)   p(b,3)   p(c,3)           | p(b,2)
```

```
rmode(p(#[a,c],#)).   | rmode(p(#(1,[a,c]),#1)).
                      |
p(a,1)   p(c,1)       | p(a,1)
p(a,2)   p(c,2)       | p(a,3)
p(a,3)   p(c,3)       | p(c,1)
```

```
rmode((p(#, A), q(#))).                | rmode((p(#1, A), q(#1))).
                                       |
(p(a,A), q(u))   (p(c,A), q(u))        | (p(a,A), q(u))
(p(a,A), q(v))   (p(c,A), q(v))        | (p(a,A), q(v))
(p(b,A), q(u))                         | (p(b,A), q(u))
(p(b,A), q(v))                         | (p(c,A), q(v))
```

A more complex and more general specification for automatic constant generation is of the following form:[1]

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

In this case V is a variable or a tuple $(V_1, \ldots, V_n)$ of variables that occur in *conj1* and *conj2*. Constant generation will happen as follows: first *conj1* is used to generate all substitutions for V. Then the substitutions for V are filled in in *conj2*, which is the conjunction that will be effectively added to the current clause.

The current form generalizes over the previous ones because the user can write any code for the generation of constants and have that code called during clause refinement.

For instance, the following constructs are equivalent (assuming `member` is defined as background knowledge):

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

and also the following two constructs are equivalent:

```
rmode(1: #(C: p(X,C), (p(X,Y), Y<C))).
rmode(1: (p(X,Y), Y<#Y)).
```

but the rmode

```
rmode(#(C: useful_constant(C), p(X,C))).
```

in combination with a definition of `useful_constant` in background knowledge is not equivalent to any simpler `#`-construct.

The most complex form of the `rmode`-specification is

[1]Note: due to the Prolog syntax definition, there *must* be a space between the colon and the `#` operators when both occur in one rmode.

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

where n and m are numbers or `inf` (for infinity), and V is a variable or tuple of variables that occur in 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.

Example 8. *Suppose the current query to be refined is* `?- a(X), b(Y,Z)`. *Then the following specification*

```
rmode(1: #(1*6*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=6.
?- a(X), b(Y,Z), Y=1.
?- a(X), b(Y,Z), Y=2.
...
?- a(X), b(Y,Z), Y=6.
?- a(X), b(Y,Z), Z=1.
?- a(X), b(Y,Z), Z=2.
...
?- a(X), b(Y,Z), Z=6.
```

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 different values for p's argument that occur in that example are chosen.[2] *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.*

[2] It is not specified how the examples are chosen, nor how the constants are chosen within an example.

The most complex #-construct differs from the previous one in an important aspect. For the previous construct the constants that are generated are always the same, whatever the set of examples that is currently under investigation; they are taken from the background knowledge. Here, the generated constants are taken from the examples that are covered by the clause currently being refined.

In general, the following constructs make use of background knowledge only (i.e. they will not access the examples themselves):

- #*list*
- #(*V* : *conj1*, *conj2*).

The following constructs do access the examples:

- #
- #*var*,
- #(*n***m***V* : *conj1*, *conj2*).

There is a difference in efficiency between the two groups of constructs: for those that access only background knowledge, constant generation is much more efficient. However, when using data dependend constant generation, less constants might be generated, making query evaluation more efficient (as less queries have to be evaluated). The type that is to be preferred therefore depends on the application at hand.

It is possible to convert data dependent constant generators into independent ones during a preprocessing step by using the setting `default_preprocessing(on)`. The constants that have been generated during preprocessing can be inspected by issuing the `sel` (show expanded language) command at the ACE prompt.

The most complex and powerful method for specifying constant generators in ACE is as lists of generators with options. All types of constant generation discussed above are translated into this format automatically by ACE before they are passed to the refinement operator. It is possible to to see the result of this translation with the `sel` command. The syntax is as follows

rmode(#(*generator(s)*, *conj*)).

or

rmode(N: #(*generator(s)*, *conj*)).

generator(s) can be one generator or a list of generators (i.e., $[G_1, \ldots, G_n]$).

A generator G_i is defined as *gen*:*optionlist*, where *gen* is the query that generates the constants, and *optionlist* is a list of options that influence the generation itself or that influence the final set of constants. The syntax for each option is *optionname=value.* There is one option that is obligatory, the *vars* option. This option specifies the variables for which this generator generates constants. The value of this option is always a list of one or more variables. In the example below, constants are generated for the variable `X`.

Example 9. *The rmode*

```
rmode(#(member(X,[a,b,c]):[vars=[X]], p(X))).
```

generates the refinements `p(a)`*,* `p(b)` *and* `p(c)`*.*

Except for the option vars, all the other options are optional. If an option is ommitted, it gets a default value assigned to it. Below each option is discussed in more detail.

- **models**:
  - no: the generator is executed once, and has no access to the examples (only to the background theory).
  - n (integer): the generator is executed for the n first examples in the knowledge base. The generated constants are collected from these examples (duplicates are removed).
  - inf: the same behavior as with $n =$ the number of available examples.

  Default value: no.

- **subst**:
  - n (integer): collect only the first n constant combinations for each example.

  Default value: `inf`.

- **preprocessing**:
  - off: generation is performed for every refinement step.
  - on: generation is performed once, during a preprocessing step.

  Default value: depends on the value of the setting `default_preprocessing`. The default for the setting `default_preprocessing` is off.

- **min**:
  - n (integer): only combinations of constants that occur in at least n examples are included.

  Default value: depends on the value of the setting `default_min`. The default for the setting `default_min` is 1.

Example 10. *The rmode*

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

is equivalent to:

```
rmode(1: #(p(C):[vars=[C],models=1000,subst=3],p(C))).
```

Constraints on Variables

It is possible to explicitly specify constraints that must hold for variables that occur in certain places. These constraints are akin to those imposed by using 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 11. *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 12. *The specification*

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

tells ACE that p should not be added with a first argument that already occurs as first argument of a p-literal. 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.*

The second argument of a `constraint` specification can also include the predicate `user/2`, which indicates a user defined constraint. This predicate will unify its first argument with the current query and then call the goal in its second argument, which typically includes a call to a user defined predicate that implements the constraint.

Example 13. *The specification*

```
rmode(p(+X,-Y)).
constraint(p(X,Y), user(Q, my_constraint(Q,X,Y))).
```

tells ACE that p can only be added if the background knowledge predicate `my_constraint/3` *succeeds. Before* `my_constraint/3` *is called, Q is unified with the current query thereby allowing* `my_constraint/3` *to test arbitrary properties of the query.*

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 14. *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.

The relationship between warmodes, rmodes and types is that for each warmode specification one can always generate a couple of rmode and type specifications that is equivalent. Indeed, internally warmodes are just converted into rmodes and types (and a `typed_language(yes)` declaration if needed).

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 15. *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. It defaults to 1.

Auto-lookahead

Lookahead statements can also be generated automatically by ACE by using:

```
auto_lookahead(Pred, Varlist).
```

where `Pred` is the pattern and `Varlist` are the variables that have to be bound in the lookahead statements that will be generated. ACE will generate a lookahead statement to each `rmode` that has an input variable of the same type as a variable from `Varlist`. E.g.,

```
type(relation(key, attr1, attr2)).
type(test_1(attr1)).
type(test_2(attr2, value)).

rmode(test_1(+Attr)).
rmode(test_2(+Attr, #[1,2,3])).

auto_lookahead(relation(Key, Attr1, Attr2), [Attr1, Attr2]).
```

Will generate:

```
lookahead(relation(Key, Attr1, Attr2), test_1(Attr1))
lookahead(relation(Key, Attr1, Attr2), test_2(Attr2, #[1,2,3]))
```

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 characterized 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 characterized completely by the thresholds 0.25 and 0.75.

The question is, then, how to find suitable values for these thresholds. There are two possibilities: we can either do unsupervised discretization or supervised discretization. The unsupervised discretization algorithm is based on equal-frequency discretization: it takes the discretization thresholds such that in each discretization bin there is (approximately) the same number of examples. The supervised discretization algorithm, originally developed for ICL [42], is based on Fayyad and Irani's work [29, 19] and takes the discretization thresholds such that entropy is minimised. In this manual, we do not

discuss the details of the algorithms, but focus on how the user can control the discretization process. *Note: at this moment discretization is only available in classification mode.*

To use unsupervised, equal-frequency discretization, specify:

```
discretize(equal_freq).
```

To use supervised, entropy-based discretization, specify:

```
discretize(entropy).
```

Entropy-based discretization is the default.

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 16. *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 `threshold` is available that upon backtracking consecutively returns each threshold for a variable in a predicate. The predicate follows the following format:

```
threshold(pred, varlist, const)
```

Pred and *varlist* should correspond with the *pred* and *varlist* arguments of a `to_be_discretized` fact. The *const* argument should be free when calling the predicate, and is instantiated with a threshold value.

Example 17. *For the specification of example 16, 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]).
```

where the third argument of `discretized` *shows which list of constants has been produced.*

A typical use of these results would be:

```
rmode(#(1*10*C: threshold(employee(_, _, _, Age), [Age], C),
        +Age < C)).
rmode(#(1*10*C: threshold(wage(_, Wage), [Wage], C),
        +Wage < C)).
```

The #*-construct indicates that some age or wage variable should be compared with a threshold for that variable.*

Aggregates

Aggregation is a technique to summarize data. In this context, it can be used to summarize relations that occur within examples. For more information on aggregates, please consult Van Assche and Vens's work [40]. In this manual, we concentrate on how the user can control the use of aggregates.

In ACE aggregates have the following form:

```
aggregate(F, Q, V, R),
```

where F is an aggregate function (e.g., count), V is an *aggregate variable* occurring in the *aggregate query* Q, and R is the result of applying F to the set of all answer substitutions for V that Q results in. The result R has to be compared to a (number of) value(s). The following aggregate functions are provided: *max*, *min*, *avg*, *sum*, *mode*, *count*, *count_dist*.

Example 18. *In order to predict the wage of an employee, it may be useful to include some statistics concerning the number or children he/she has (number of children, maximum age of children,...). Suppose we have the predicate child(EmployeeID, ChildID, Age). Then the following type definition and rmodes can be used:*

```
type(aggregate(aggfunction,aggquery,Aggvariable,aggresult).
rmode(5:(aggregate(count, child(+E, -C, -A), _, Res), Res > #[2,3,4])).
rmode(5:(aggregate(max, child(+E, -C, -A), -A, Res), Res > #[10,15,20])).
rmode(5: #(V: member(V,[10,15,20]),
     (aggregate(avg, child(+E, -C, -A), -A, Res), Res < V))).
```

Instead of providing the list of values, it is of course possible to use discretization:

```
discretization(bounds(3)).
to_be_discretized(child(Empl, Child, Age),[Age]).
rmode(5: #(X: threshold(child(Empl, Child, Age),[Age], X),
     (aggregate(avg,child(+E, -C, -A), -A, Res), Res <X))).
```

When including many aggregates, it may be useful to include *aggcondition* constructs (instead of writing all the separate rmodes):

```
aggcondition(aggfunctionlist, aggquery, aggvariable,operatorlist,valuelist).
```

Example 19. *For the employee example, the following type definition and* aggcondition *constructs*

```
type(aggregate(aggfunction,aggquery,aggvariable,aggresult).
aggcondition([count], child(+E, -C, -A), C, ['>','<'],[2,3,4]).
aggcondition([max,min], child(+E, -C, -A), A, ['>','<','='],[5,10,15,20]).
aggcondition([avg], child(+E, -C, -A), A, ['>','<','='],
     (X: threshold(child(Empl, Child, Age),[Age], X)).
```

are equivalent with the following list of rmodes.

```
type(aggregate(aggfunction,aggquery,Aggvariable,aggresult).
rmode(aggregate(count, child(+E, -C, -A), C, Res), Res > #[2,3,4]).
rmode(aggregate(count, child(+E, -C, -A), C, Res), Res < #[2,3,4]).
rmode(aggregate(max, child(+E, -C, -A), -A, Res), Res > #[5,10,15,20]).
rmode(aggregate(max, child(+E, -C, -A), -A,  Res), Res < #[5,10,15,20]).
rmode(aggregate(max, child(+E, -C, -A), -A, Res), Res = #[5,10,15,20]).
rmode(aggregate(min, child(+E, -C, -A), -A, Res), Res > #[5,10,15,20]).
rmode(aggregate(min, child(+E, -C, -A), -A, Res), Res < #[5,10,15,20]).
rmode(aggregate(min, child(+E, -C, -A), -A, Res), Res = #[5,10,15,20]).
rmode(#(X: threshold(child(Empl, Child, Age),[Age], X),
     (aggregate(avg,child(+E, -C, -A), -A, Res), Res < X))).
rmode(#(X: threshold(child(E,Ch,A),[A], X),
     (aggregate(avg,child(+E, -C, -A), -A, Res), Res > X))).
rmode(#(X: threshold(child(E,Ch,A),[A], X),
     (aggregate(avg,child(+E, -C, -A), -A, Res), Res = X))).
```

It is possible to learn complex aggregates (i.e. aggregates that consist of selection conditions in the aggregate query). Due to memory problems that might occur when allowing multiple conjuncts in an aggregate query, we restricted the maximum number of conjuncts to two.
A number of settings are available.

- `aggregate_refinement(`*s*`).`
  s is `yes` or `no`. This setting introduces selection conditions in the aggregate query of aggregates already present in the theory (e.g., in Tilde it adds a node with an aggregate condition which is a refinement of an aggregate condition in a node higher in the tree).
  Default: no.

- `aggregate_lookahead(`*s*`).`
  s is `yes` or `no`. This setting allows to directly introduce complex aggregates (i.e. they do not have to be refinements of an aggregate already present).
  Default: no.

- `aggregate_recursion(`*s*`).`
  s is `yes` or `no`. This setting states whether aggregates can occur inside aggregate queries.
  Default: no.

- `aggregate_refiners(`*s*`).`
  s is `yes` or `no`. This setting states whether the selection conditions used to refine aggregates can come from any rmode or from a special set of rmodes (this can control the number of refinements). If yes, a number of rmodes starting with *agg* have to be provided, e.g. *rmode(agg:male(+C))*.
  Default: no.

The search can be made more efficient by using the so-called monotone refinement operator (see [43]). Two ordered classes of aggregate functions are provided: generalized averages and generalized counts. The use of the refinement operator requires the use of *aggcondition* constructs. The syntax is as follows:

```
type(aggregate(aggfunction,aggquery,aggvariable,aggresult).
aggcondition([count_class], child(+E, -C, -A), C, ['>','<'],[[2,3,4]]).
aggcondition([avg_class], child(+E, -C, -A), A, ['>'],[[5,10,15,20]]).
```

This is the same as having

```
type(aggregate(aggfunction,aggquery,aggvariable,aggresult).
aggcondition([count,count_dist], child(+E, -C, -A), C, ['>','<'],[2,3,4]).
aggcondition([max,min,avg], child(+E, -C, -A), A, ['>'],[5,10,15,20]).
```

but will reduce induction times. Some remarks:

- These special *aggcondition* constructs can be combined with other ones.
- This efficient refinement operator can not be used with *aggregate_refinement(yes)* (but it can be used with *aggregate_lookahead(yes)* or *aggregate_lookahead(no)*).

Query sampling

Normally, when a query is refined, a whole set of possible candidate queries is generated by refining the current query. Afterwards one query from these is selected. When query sampling is performed, not all possible queries are generated, but only a random sample of them and the best one from this sample is selected. By doing so, the amount of time spend on generating these queries, and also the time spend on executing them to find the best one is largely reduced.

The user can indicate that query sampling needs to be performed, by means of

```
query_sampling_probability(prob).
```

`prob` is the fraction of the total number of queries that will be selected in the sample. It is either in the range of 0 and 1 or it is *sqrt*, then the square root of the number of queries is taken.

If only one random query needs to be selected instead of a sample, one can use the following setting:

```
single_query_generation(yes).
```

Query sampling shows to be very beneficial when the feature space becomes very large, for instance when aggregates and aggregate refinements are used.

4.2.4 General Settings

In this section we describe general settings which apply to all systems available in ACE. Settings which are specific to one system can be found in the chapter describing that system.

- `talking(t).`
  t is an integer between 0 and 4 (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.
  Default: 3.
- `classes(c).`
  c is a list of class names; it defines the classes as they are to be used by TILDE (classify setting) or WARMR. A class name is an atom.
  Default: `[pos,neg]`.

- `load(`*l*`).`
  l is either `key` or `models`. This setting tells ACE whether the data file is in key or models format.
  Default: `key`.

- (expert) `outerloop(`*s*`).`
  s is `examples` or `queries`. Specifies which implementation of TILDE or WARMR should be used. These algorithms need to evaluate a set of queries on a set of examples; to this aim two nested loops are used, one over queries and one over examples. The `outerloop` setting specifies which loop should be the outermost one. This may influence the efficiency of the induction process.
  Default: `queries`.

- (expert) `huge(`*s*`).`
  s is either `on` or `off`. This setting tells ACE whether it should use its provisions to handle large data sets. If it is `on`, the data file will never be loaded completely in main memory, only chunks of it will be loaded at a time. The `huge_chunks` package has to be loaded when this setting is `on`.
  Default: `off`.

- (expert) `autoload_packages(`*s*`).`
  s is either `on` or `off`. When `on`, packages will automatically be loaded when needed.
  Default: `off`.

- (expert) `leave_out(`*q*`).`
  q is a query. Examples for which the query succeeds will not be included in the training set.
  Default: `false`.

- `use_packs(`*p*`).`
  p is 0, 1 or 2. If the underlying Prolog engine is ilProlog, 0 indicates no packs will be used; 1 activates disjoint execution of packs; 2 activates packed execution. On other Prolog engines this setting has no effect (packs are never used).
  Default: 2.

4.2.5 Reading and Changing Settings

- `set(`*setting*`, `*value*`).`
  changes an updatable setting. Updatable settings are those settings that consist of a single fact (i.e., not `warmode`, `rmode`, `constraint` etc., which all may consist of multiple facts. For instance, `set(talking,4)` sets the talking level to 4.

- `show_settings -- ss`
  shows the current settings.

- `show_language -- sl`
  shows the current language settings.

- `load_settings -- ls`
  (re)loads the settings file.

- `set_default_settings -- sds`
  resets all settings to their default values.

4.2.6 Loading Packages

Most of the systems available in ACE are defined in separate packages. If you want to use a certain system, you have to load the corresponding package first. One exception to this rule are Tilde and Warmr. These packages are auto-loaded and always available.

- `use_package(`*Name*`)`
  load package *Name*.
- `show_packages`
  shows information about the loaded packages.
- `show_package(`*Name*`)`
  shows information about package *Name*.

Chapter 5

General predictive interface

The ACE system provides a common interface for the different prediction algorithms it incorporates. Currently available predictive systems are Tilde, ICL, and RegRules.

5.1 Building predictive models in ACE

This section lists the commands that can be used at the prompt in ACE to build predictive models, they are common for the different predictive systems that are described in the following chapters. Each of these commands has a parameter *Algo* specifying which of the available algorithms is used (*tilde*, *icl*, *regrules*).

- `induce(`*Algo*`) -- i(`*Algo*`)`
  starts induction (possibly preceded by discretization), generating a predictive model according to the provided algorithm *Algo*. Output is written to *app*`.out`. Exactly what is written to the output file, is discussed in Section 11.6. With this command, induction is always performed on the whole dataset minus the examples for which the query provided by `leave_out` succeeds.

- `nfold(`*Algo*`,`*n*`) -- nf(`*Algo*`,`*n*`)`
  `nfold(`*Algo*`,`*n*`,`*s*`) -- nf(`*Algo*`,`*n*`,`*s*`)`
  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: `uB1`, `uB2`, ..., `uB`*n*.

  The optional s parameter specified the seed used to create the random partition. If the same seed is used for different cross-validations with the same version of ACE, the same random partitions will be used.

- `leave_one_out_from_list(`*Algo*`,`*list*`) -- loofl(`*Algo*`,`*list*`)`
  assumes a predicate `testid` to be available, on which cross-validation will be based. For each constant c in *list*, the data set is 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 [18], where cross-validation is often done based on the 5 structures that appear in the data.

5.2 General settings

- `random_validation_set(`*R*`).`
  R is a number between 0 and 1. It specifies an approximate proportion of learning examples that have to be set aside to be used as a validation set. Each individual learning example is then randomly assigned to the training set (with probability $1-R$) or to the validation set (with probability R).
  Default: 0.

- `random_test_set(`*R*`).`
  R is a number between 0 and 1. It specifies an approximate proportion of learning examples that have to be set aside to be used as a test set. Each individual learning example is then randomly assigned to the training set (with probability $1-R$) or to the test set (with probability R).
  Default: 0.

- `write_predictions(`*List*`).`
  Write the predictions of the data sets specified in *List* (eg. *training*, *testing*, ...) to file named *app*`.prediction.`*x* where x=*training*, *testing*, ...
  Default: [].

5.3 General output files

All output files are written in a directory named after the predictive algorithm used, so *./tilde/* for Tilde, *./icl/* for ICL etc.

5.3.1 Detailed result files

The results of each run of a predictive algorithm are written to files. The filename is *app*`.out`, or *app*`.uB`*i* for the *i*-th random *n*-fold cross-validation, or *app*`.uL`*i* for the *i*-th left out set.

The output of a run contains the following statistics:

- CPU-times for discretization and induction itself
- Model complexity, model accuracy on training set, validation set and test set (when applicable), and model accuracy on all sets together are shown. For classification, accuracy is shown as a proportion of correctly predicted instances; for regression, accuracy is shown as “relative error” RE.

Extra output might be given depending on the specific prediction algorithm used. This is mentioned in the output section of the appropriate algorithm.

5.3.2 Summary files

When performing cross-validations, ACE not only gives detailed reports on each run, but also generates a summary of the whole cross-validation. This summary is written to a file called *app*`.summary.`*x*, where $x =$ `uL` or `uB`, depending on which kind of cross-validation is performed (this is consistent with the names of the detailed output files).

Classification

app.`summary.`*x* contains the average values of induction times, model complexities and predictive accuracies of all the individual runs.

It also 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 φ for 2×2-tables, i.e. for a table $\begin{array}{|cc|} A & B \\ C & D \end{array}$, $\varphi = \frac{AD-BC}{\sqrt{(A+B)(A+C)(B+D)(C+D)}}$.

Regression

app.`summary.`*x* contains the average induction time and model complexity of all the individual runs, as well as the global (over all runs) relative error, Pearson's correlation coefficient between real and predicted values, the mean absolute error and the root mean squared error.

Given a set of n predicted values p_i and corresponding actual values a_i (and denoting their respective means with $\bar{p}$ and $\bar{a}$), relative error is defined as follows:

$$RE = \frac{\sum (p_i - a_i)^2}{\sum (\bar{a} - a_i)^2}$$

i.e. it is the ratio of the mean squared error of the hypotheses over the mean squared error of a null hypothesis always predicting the mean of the observed values.

Pearson's correlation coefficient is defined as

$$r = \frac{\sum (p_i - \bar{p})(a_i - \bar{a})}{\sqrt{\sum (p_i - \bar{p})^2 \sum (a_i - \bar{a})^2}}$$

The mean absolute error and root mean squared error are defined as

$$MAE = \frac{\sum |p_i - a_i|}{n}$$

$$RMSE = \sqrt{\frac{\sum(p_i - a_i)^2}{n}}$$

5.4 Ensemble methods

5.4.1 Constructing ensemble models

ACE provides also some ensemble methods that can be used in combination with a specified basic algorithm. Current available ensemble methods are majority voting, bagging [7], boosting [31] and random forests [9]. To induce an ensemble model, the parameter *Algo* in the command is instantiated with the appropriate ensemble method as follows:

- $bagging(BasicAlgo, n)$: this performs n rounds of bagging with the specified basic algorithm.
- $boosting(BasicAlgo, n)$: this performs n rounds of boosting with the specified basic algorithm.
- $voting(BasicAlgo, n)$: this performs n rounds of the specified basic algorithm and performs majority voting over the predictions of the different models.

As basic algorithm *BasicAlgo* one of the prediction algorithms can be used (*tilde*, *icl*, *regrules*).

First order random forests [40] can be induced by choosing $bagging(tilde, n)$, with n the number of trees in the forest, in the command and by using the setting `query_sample_probability/1` with a certain probability as parameter. This last setting is described in the paragraph 'Query sampling' of Section 4.2.3. At each node that is to be build it only takes a sample of the total number of queries to select the best one from. This setting in combination with the bagging procedure gives random forests.

5.4.2 Settings specific for ensemble methods

Settings for all ensemble methods

- `write_ensemble_models(`*`List`*`).`
  List is a list of numbers specifying for which submodels of the entire ensemble output needs to given. For each of these numbers seperate output files are generated. They should be smaller than n, the number of trees in the ensemble. Let's say you build an ensemble of 33 trees, and *List* is [3, 11], then next to the output for 33 trees, it will also give output for the ensemble consisting of the first 3 and 11 trees. This number will be in front of the extension of the output file, eg. $app.3out$ and $app.11out$. In that case the output for 33 trees will be just *app.out*.
  Default: `[]`.

- `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. Most of these output options are only relevant for the Tilde system and are discussed in the next chapter.
  Options related to ensembles:

  - *all_models*: outputs next to statistics for the ensemble also statistics for each of the base models in the output file

Settings specific for bagging

- `out_of_bag(`*op*`).`
  op = `yes` | `no`
  When using bagging, out-of-bag error estimates [8] can be used to estimate the generalisation errors. This removes the need for a set-aside test set or cross-validation. Out-of-bag error estimation proceeds as follows: each tree is learned on a training set D_i drawn with replacement from the original training set D. For each example d in the original training set, the predictions are aggregated only over those classifiers T_i for which D_i does not contain d. This is the out-of-bag classifier. The out-of-bag error estimate is then the error rate of the out-of-bag classifier on the training set. Note that in each resampled training set, about one third of the instances are left out (actually $1/e$ in the limit). As a result, out-of-bag estimates are based on combining only about one third of the total number of classifiers in the ensemble. This means that they might overestimate the error rate, certainly when a small number of trees is used in the ensemble.
  This setting cannot be used in combination with n-fold cross-validation.
  Default: `no`.

Chapter 6

Tilde

Note: ACE 1.2.x comes with two versions of Tilde. The default is 3.0, which is the most recent version. To use version 2.2, which was included in ACE 1.1.x, set `tilde_version('2.2')`.

6.1 Growing trees with Tilde

In order to build a decision tree using the Tilde algorithm, the parameter *Algo* of the commands described in section 5.1 should be instantiated to *tilde*.

6.2 Settings specific for Tilde

Tilde has several modes of operation. We first discuss settings that are relevant for all modes, then settings that are specific for certain modes.

6.2.1 Settings used for all modes of operation

- `sampling_strategy(`s`).`
  Tilde may use only a sample of the whole data set when evaluating tests in order to decide which test should be put in a node of the tree. Once the test is put there, the whole data set will be partitioned according to that test. s controls the sampling procedure. When s is `fixed(`n`)` with n an integer greater than 0, Tilde uses a random sample of the data of at most size n. When s is `user(`p`)`, the user-defined predicate $p/2$ will be called with as first argument a list of all currently relevant examples (together with weights) and as second argument a free variable, which should be instantiated by $p/2$ to a sublist of the original list. This sublist represents the sample. When s is `none`, the whole data set will always be used.
  Default: `fixed(1000)`.

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

- `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 6.3, where the output file is discussed.
Default: `[c45,prolog]`.

- `tilde_mode(`*mode*`)`.
  Mode is a constant representing a mode. Currently three modes are built-in in TILDE: `classify`, `regression` and `cluster`.
  Default: `classify`.

6.2.2 Settings specific for classification mode

- (expert) `confidence_level(`*cl*`)`.
  cl is a number between 0 and 1, that is used in the post-pruning phase for estimating the prediction error. This number is only relevant if C4.5's post-pruning method is used.
  Default: 0.25.

- (expert) `prune_rules(`*s*`)`.
  s must be `true` or `false`. If *s* is `true`, the `load` setting is set to `models` and TILDE is in classification mode, then a rule post-pruning algorithm will be run. This algorithm is still experimental.
  Default: `false`.

- `accuracy(`*a*`)`.
  a is a number between 0 and 1. If the majority class in a node has a relative frequency of at least *a*, the node is made a leaf.
  Default: 1.

- `heuristic(`*heur*`)`.
  where *heur* = `gain` or `gainratio`
  This specifies the heuristic that is to be used. `gainratio` is C4.5's default heuristic; it usually yields better results than `gain` (see [35] for more information about these heuristics).
  Default: `gainratio`.

- `pruning(`*method*`)`.
  where *method* = `c45`, `c45_nosafepruning`, `none`, `vsb`, `bic`, `mdl`, `chi_square(Alpha)` or `randomisation(N,Alpha,Type)`
  This setting specifies the post-pruning algorithm that is to be used. `c45` stands for C4.5's post-pruning method, which performs both error-based pruning and safe pruning (error-based pruning uses an estimation of the error on unseen cases and prunes in such a way that this error is minimized [35]; safe pruning or "collapsing" prunes subtrees in which all nodes have the same majority class). `c45_nosafepruning` stands for error-based pruning without safe pruning. `vsb` stands for "validation set based" pruning: a separate validation set is used to estimate the error on unseen cases and pruning is done in such a way that it minimizes this estimate. `bic` stands for pruning based on the Bayesian Information Criterion. `mdl` stands for pruning based on the Minimum Description Length Principle. `chi_square(Alpha)`, with Alpha between 0 and 1 (e.g. 0.05), stands for pruning based on Bonferoni-corrected chi-square tests with the given alpha-level. `randomisation(N,Alpha,Type)`, with N a positive integer (e.g. 100) and Alpha between 0 and 1 (e.g. 0.05), stands for pruning based on randomisation tests with N randomisations and with the given alpha-level (the advised `Type` is `global`).
  Note that `mdl` and `bic` can only be used if the heuristic is gain. An experimental comparison of most of the above pruning methods is given by Fierens et al. [30].
  Default: `c45`

- `stopping_criterion(`*criterion*`)`.
  where *criterion* = `mincases`, `bic`, `mdl`, `chi_square(Alpha)` or `randomisation(N,Alpha,Type)`

This setting specifies the stopping criterion used for the top-down induction. `mincases` stands for using the `minimal_cases` setting (see before) as a stopping criterion. This means that refinement of a node is stopped if the node is pure or if there are no queries for that node that satisfy the mimimal cases requirement. The stopping criteria `bic`, `mdl`, `chi_square(Alpha)` and `randomisation(N,Alpha,Type)` are the counterparts of the pruning criteria discussed above. Note that this setting is to be used in conjunction with the setting `pruning`. For instance, if the user wants to use the Bayesian Information Criterion to control the size of the tree, there are two options. The first option is to use a stopping criterion by specifying `stopping_criterion(bic)` and `pruning(none)`. The second option is to use postpruning by specifying `pruning(bic)` and `stopping_criterion(mincases)`.
Default: `mincases`

- `m_estimate(`*method*`)`.
  where *method* = `none`, `laplace` or `m(M,Prior)`
  Determines how class-probabilities are computed; this affects only the output options `c45_probab`, `prolog_probab` and `roc01`. In general, the probability of a class c_i in a leaf l is computed as $\frac{N_{i,l}+m\ p_{prior}(c_i)}{N_l+m}$, with $N_{i,l}$ the number of examples of class c_i in l and $N_l = \sum_i N_{i,l}$. The parameters m and $p_{prior}(c_i)$ depend on the actual method used for m-estimation. The method `none` stands for not using m-estimation, i.e. $m = 0$. The method `laplace` stands for Laplace estimation, i.e. $m = NbClasses$ and $p_{priori}(c_i) = \frac{1}{NbClasses}$. The most general method is `m(M,Prior)`, with `M` a positive number and `Prior` a probability distribution on the classes (e.g. `[neg:0.4,pos:0.6]`). This corresponds to using $m = M$ and $p_{prior}(c_i)$ according to the prior (the classes in the prior must be ordered alphabetically).
  Default: `none`

6.2.3 Settings specific for clustering mode

- `ftest(`S`)`.
  `ftest` serves as a stopping criterion for clustering. If a set of examples is split into two subsets, the quality of the split can be measured by looking at the sum of squares of distances of each example in a set to the average of the set. The sum of squares of distances within the two subsets ($SS_1 + SS_2$) is at most as large as the sum of squares of distances SS in the whole set. To test whether a reduction in variance is significant, an F-test can be performed. S indicates the significance level that will be used for the test. S can have a value of 1.0, 0.1, 0.05 or 0.01. Lower values causes the tree building to stop earlier. 1.0 disables this stopping criterion.
  Default: 0.05.

- `heuristic(`***heur***`)`.
  heur = `eucl` | `eucl(distance)`
  "Heuristic" actually means "distance" in the clustering context. Currently only one distance measure is implemented: euclidean distance. This is a propositional distance measure. Each example is assigned a number of coordinates by the `euclid` specification (see below). The euclidean distance between these coordinates is also the dissimilarity of two examples. The clustering heuristic maximizes inter-cluster-distances and minimizes intra-cluster-distances.
  The difference between `eucl` and `eucl(distance)` is that the former heuristic minimizes intra-cluster variation whereas the latter maximizes the distance between cluster centers. The names of the heuristics, admittedly, are badly chosen.
  Default: `eucl`.

- `euclid(`Q`, `X`)`.
  X should be a variables occurring in Q. This setting indicates that X is one of the coordinates of an example to be used for computation of the euclidean distance, where X is to be computed by executing the query Q for the example.

- `assign_class(`L`, `C`) :- ...`
  The predicate *assign_class* defines how a class can be assigned to a cluster of examples. The predicate will be called by TILDE with as first argument L a list of examples and as second argument C a free variable. The user should define the predicate so that C is instantiated to a class constant. In the definition of this predicate, the user can make use of the predicate *query_model(Model, Query)* which allows to extract relevant information from the models.

6.2.4 Settings specific for regression mode

- `ftest(`S`).`
  `ftest` serves as a stopping criterion. If a set of examples is split into two subsets, the quality of the split can be measured by looking at the sum of squares of distances of each example in a set to the average of the set. The sum of squares of distances within the two subsets ($SS_1 + SS_2$) is at most as large as the sum of squares of distances SS in the whole set. To test whether a reduction in variance is significant, an F-test can be performed. S indicates the significance level that will be used for the test. S can have a value of 1.0, 0.1, 0.05 or 0.01. Lower values causes the tree building to stop earlier. 1.0 disables this stopping criterion.
  Default: 0.05.

- `pruning(`*`method`*`).`
  where *method* = `vsb` or `none`
  `vsb` stands for "validation set based" pruning: a separate validation set is used to estimate the error on unseen cases; pruning is done in such a way that it minimizes this estimate.
  Note that in classification mode several other options are available next to `vsb`, but when used in regression mode these options all correspond to no pruning.

- `multiscore(`L`).`
  L is a (possibly empty) list, or `off`. The multiscore attribute is useful when multivariate regression is performed (i.e. predicting multiple variables). When `multiscore` is not `off`, not only the global relative error (over all predicted variables) will be reported, but also the relative error of each individual variable (see also section "Output files"). Some of these variables may be auxiliary variables, numeric variables that are actually derived from nominal variables (e.g. a dichotomous variable could be represented by 0 or 1). For such variables one might want to see the accuracy of the prediction rather than its relative error. This can be performed in the following manner.

  L, except when `off`, is a list of couples (X, Y). X is typically of the form `predicted(`*`list`*`)`, with *list* a list of the variables that are predicted by the regression tree[1]. Y is a query through which TILDE can decide for a specific example whether some prediction is correct or not. An accuracy (correct predictions / total predictions) will be computed on the basis of this, and this accuracy will be reported.
  Default: `off`.

Example 20. *Suppose we want to perform regression on three variables* length*,* weight *and* sex*.* Sex *can have two symbolic values (*male *or* female*) but has been represented as 0 or 1 because the regression algorithm wants to see numbers, not symbolic values.*

If the settings file contains the setting

```
multiscore([]).
```

then the output will contain relative errors for the prediction of all three variables weight*,* length *and* sex*. For instance, one output line could be:*

[1]The predicate *predicted* is predefined in TILDE.

```
pruned_testing_relist: [0.5 = 4.6/9.2, 0.4 = 4.2/10.5, 0.25=0.1/0.4]
```

which means that the predictions for the test set, using the pruned tree, had relative error of 0.5, 0.4 and 0.25 for length, weight *and* sex *respectively.*

As a relative error of 0.25 may be rather hard to interpret, the user may want to use the following multiscore list:

```
multiscore([ (predicted([L,W,S]), (S>0.5 -> male; female)) ]).
```

Note that this is a list with one element (a couple). The meaning of the couple is the following: if TILDE *predicts a value for S (i.e. the* sex *variable), it should consider the prediction correct if the query*

```
(S>0.5 -> male; female)
```

succeeds, and incorrect otherwise. The query is a Prolog if-then-else construct: if $S > 0.5$ *then* male *is tested, otherwise* female. *(It is assumed here that the sex of the persons is indeed indicated in the examples by a fact* male *or* female.*) In other words: the prediction is correct if* $S > 0.5$ *and the person is male, or if* $S \leq 0.5$ *and the person is female.*

Using this information, TILDE *can compute how many examples of the test set were predicted correctly.*

```
pruned_testing_relist: [0.5 = 4.6/9.2, 0.4 = 4.2/10.5, 0.25=0.6125/0.25]
pruned_testing_multiscore: [0.857 = 12/14]
```

This makes clear that for 14 test cases, the sex was predicted correctly 12 times, which yields an accuracy of 0.857.

6.2.5 Settings specific for model tree mode

TILDE can also be used to produce model trees, i.e., regression trees where the leaves can contain linear models instead of constants [44]. A different heuristic function is used: splits are chosen to minimize residual (i.e., after fitting a linear regression model) standard deviation. Model trees should only be used when the multiscore setting is off. The setting that produces model trees is `tilde_test_eval_model(slr)`. (`slr` = simple linear regression, the default is `std`)

The resulting model tree may contain, next to regular splitting nodes, regression nodes. These do not split the data (and therefore have an artificial right branch "no: [0.0] 0 0.0"), but only serve to introduce a numeric attribute in the linear models in the leaves of the subtree. These regression nodes are always in the form of a min or max aggregate, such that a unique numeric value is obtained in case the attribute is multi-valued.

The model stored in a leaf is either a numeric constant, or a linear regression model. The latter looks as follows: $[coeff * attr + coeff * attr + ... + intercept]$. The attributes are the numeric attributes (split and regression attributes) occurring on the path from the root down to the leaf. They look as follows:

```
attr([concatenation of queries on the path higher in the tree],[attribute query],
[common variables between these two], actual variable denoting the attribute,
min/max in case of multivaluedness, default value in case of emptyness).
```

For example, an attribute might look as follows

```
attr([atom(AtID1,carbon,Charge1),bond(AtID1,AtID2,BondType)],
[atom(AtID2,Type2,Charge2)],[AtID2],Charge2,max,0.7).
```

De corresponding value is obtained by first looking for all atoms $AtID2$ that share a bond with a carbon atom. For each unique $AtID2$ instantiation the value for $Charge2$ is taken. If this results in multiple values, the maximum is taken; if it results in no values, the default value of 0.7 is taken.

6.3 Specific output for Tilde

For Tilde extra information can be written in the standard output files 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 safely-pruned 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 safely-pruned version of the original tree with examples shown
- `c45_probab`
  only available in classification mode. Same as `c45` except that now the probabilities of the classes are shown in the leaves.
- `prolog`
  writes the Prolog program corresponding to the pruned tree
- `prolog_probab`
  only available in classification mode. Same as `prolog` except that now the probabilities of the classes are shown in the leaves.
- `ldt_term`
  writes the tree as a prolog-readable term, using the following grammar: A Tree is written as ldt(Node), a Node is either written as inode(Test,Left,Right) where Left and Right are Trees and Test is written as test(Conjunction), or a Node is written as leaf(LeafModel).
- `roc01`
  writes the ROC-curve for the predictions made and computes the area under the ROC-curve (AUC). In case the number of classes exceeds two, the 'expected AUC' is computed, this is the

weighted average of the AUC's obtained by using in turn each of the classes as positive and all others as negative. When used in regression mode, examples are considered positive if their target is greater than 0.5.

- `likelihood`
  computes the "conditional log-likelihood" of the examples (i.e. log_2 of the probability of the classes of the examples given the rest of the examples). Only available in classification mode.
- (expert) `lp`
  writes a logic program corresponding to the pruned tree. Only works when the `load` setting is set to `models`.
- (expert) `elaborate`
  writes a lot of information to the output file that is mainly useful for debugging purposes.

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

Some other output files are generated by TILDE 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.
- 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.
- The file `pruned_ct` contains a couple (*actual value*, *predicted value*) for each example in the dataset. These data are used during crossvalidation experiments for computing the contingency table (classification) or correlation coefficient (regression and clustering), but can also be of interest to the user for performing other tests (e.g., McNemar's).

Chapter 7

The ICL system

Inductive Classification Logic (ICL in short) is a ILP learning system that learns first order logic formulae from examples which belong to two or more classes. The learned theory can be used to classify unseen examples. Examples are viewed as (Herbrand) interpretations. These are assumed be specified completely (we also say that we learn from closed examples). So ICL performs discriminating induction from closed examples.

7.1 Building rule sets with ICL

In order to run ICL one extra file is needed next to the common input files:

- L file: *app*.l is the language file. ICL does not support the *rmodes*. Defining a DLAB bias is the only way to provide a language bias to this system. Section 7.3 provides an example of how to define this language bias.

In order to build a rule set using the ICL algorithm, the setting file must countain

```
use_package(icl).
```

The parameter *Algo* of the commands described in section 5.1 should be instantiated to *icl* as follows:

```
> induce(icl)
```

7.2 Settings specific for ICL

- `icl_multi(`*R*`)`, where R must be on or off (default: `off`).
- `maxhead(`*N*`)`, where N must be a positive number (default: `10`). The maxhead setting specifies the maximum number of literals in head of clause (not for DNF).
- `maxbody(`*N*`)`, where N must be a positive number (default: `10`). The maxhead setting specifies the maximum number of literals in body of clause.

- `simplify(R)`, where R must be on or off (default: `on`). The simplify setting can be used to enable rule simplification. ICL uses the smartcall transformation [37] to simplify rules. Set this setting to off if the rules are expected to be very relational. Set it to on for attribute-value applications and for applications with several non-linked parts.
- `multi_prune(R)`, where R must be on or off (default: `on`). Multi_prune is used to prune rules for seperate classes when merging into a multi-class theory. Set it to `off` for no pruning of learned rules per class and to `on` to remove rules for a class, which predict another class.
- `multi_test(R)`, where R must be either cn2, bayes or bayes_old (default: `bayes`). Specify how a multi-theory should be tested. Set to `cn2` for using the same procedure as in cn2 (adding absolute values) and to `bayes` for applying naive bayes for classification.
- `language_init(R)`, where R must be either local or global (default: `global`). This setting specifies when to to initialize the language. Set to `global` for initialisation at beginning of the ICL search. Set to `local` for initialisation at each covering step. This setting can be used for dlab_queries with discretization: global/local discretization.
- `icl_heuristic(X)`, where X must be either laplace, m_estimate or m_estimate(M) (default: `m_estimate`).
- `significance_level(R)`, where R must be either 0, 0.995, 0.99, 0.98, 0.95, 0.90, 0.80 (default: `0.90`). This setting specifies the onfidence level (as percentage) for the significance test.
- `min_coverage(N)`, where N must be a strict positive number (default: `1`).
- `min_accuracy(R)`, where R must be a positive real number ≤ 1 (default: `0.0`). This setting specifies the minimal accuracy for each individual rule.
- `beam_size(N)`, where N must be a strict positive number (default: `5`). This setting specifies the maximum number of rules to be kept in the beam.
- `cn2_mode(R)`, where R must be on or off (default: `off`). ICL tries to imitate cn2 as good as possible if this setting is switched on (only changes things which cannot be changed with the settings) heuristic(laplace), significance_level(0.0)).
- `icl_heuristic_wra(R)`, where R must be yes - for weighted relative accuracy - or no (default: `no`). Specifies if weighted relative accuracy should be used as heuristic (by Peter Flach).
- `beam_max_internal(V)`, where V must be an integer, >0 (default: `1`). In a ICL beam search, the beam is a list of lists. The first list is maximum beam_size long, each internal list is maximum beam_max_internal long.

7.3 An Example Application

This example has been used in a tutorial "Three companions for first order data mining" [12]. Imagine that you have just been hired by a professional seminar organizer (PSO) in order to discover new knowledge about the activities of the PSO that is to be used for commercial purposes. PSO also informs you that they have a database about past activities (Figure 7.1).

It contains information about participants in a recent Seminar on Data Mining. Note that information about each person is contained in multiple tables of the database. To obtain a set of examples for ICL, we partition the the database into examples. Global information (like course table) is put in the background.

We would like to find out what type of people attend the parties at our seminar (can be useful in order to set the price of the party as well as to decide upon the activities at parties).

PARTICIPANT Table

NAME	JOB	COMPANY	PARTY	R_NUMBER
adams	researcher	scuf	no	23
blake	president	jvt	yes	5
king	manager	ucro	no	78
miller	manager	jvt	yes	14
scott	researcher	scuf	yes	94
turner	researcher	ucro	no	81

SUBSCRIPTION Table

NAME	COURSE
adams	erm
adams	so2
adams	srw
blake	cso
blake	erm
king	cso
king	erm
king	so2
king	srw
miller	so2
scott	erm
scott	srw
turner	so2
turner	srw

COMPANY Table

COMPANY	TYPE
jvt	commercial
scuf	university
ucro	university

COURSE Table

COURSE	LENGTH	TYPE
cso	2	introductory
erm	3	introductory
so2	4	introductory
srw	3	advanced

Figure 7.1: The PSO database.

7.3.1 The background knowledge (BG file)

This file defines all the additional information on the domain (definition of new predicates) that can be used in the language (.l) file to construct the rules.

```
job(_J):-
    participant(_J, _, _, _).
company(_C):-
    participant(_, _C, _, _).
party(_P):-
    participant(_, _, _P, _).

company_type(_T):-
    company(_C),
    company(_C, _T).

course_len(_C, _L):-
    course(_C, _L, _).
```

```
course_type(_C, _T):-
    course(_C, _, _T).

company(jvt,commercial).
company(scuf,university).
company(ucro,university).
course(cso,2,introductory).
course(erm,3,introductory).
course(so2,4,introductory).
course(srw,3,advanced).
```

7.3.2 The example data (KB file)

We only show part of the example data. All input files for this application are available from the ACE web-site. In ICL, the knowledge base should be used in the *models* format (see Section ??).

```
begin(model(adams)).
participant(researcher,scuf,no,23).
subscription(erm).
subscription(so2).
subscription(srw).
end(model(adams)).

begin(model(blake)).
participant(president,jvt,yes,5).
subscription(cso).
subscription(erm).
end(model(blake)).
....
```

7.3.3 The language file

The language file (L file) contains these DLAB specifications:

```
dlab_template('
0-2: [   job(c_job),
                company_type(c_company_type),
                subscription(_S),
                course_len(_S,c_course_len),
                course_type(_S,c_course_type)
        ]
<--
        0-len: [ job(c_job),
                company_type(c_company_type),
                subscription(_S),
                course_len(_S,c_course_len),
                course_type(_S,c_course_type)
        ]
').

dlab_variable(c_job, 1-1, [researcher,manager,president]).
dlab_variable(c_company_type, 1-1, [commercial,university]).
dlab_variable(c_course_len, 1-1, [2,3,4]).
dlab_variable(c_course_type, 1-1, [introductory,advanced]).
```

In general, a DLAB bias specifies the space of valid conjunctions of literals (if you choose the DNF setting) or disjunctions of literals (if you choose the CNF setting) that ICL searches for. A DLAB bias is composed of a *dlab_template* and of the specification of some *dlab_variable*. Each *dlab_template* is written as $HeadTemplate \leftarrow BodyTemplate$ where $HeadTemplate$ and $BodyTemplate$ are DLAB terms. A DLAB term specifies which literals and arguments should be used in the hypotheses. A term is either an atom or a formula of the form $Min - Max : L$, where L is a list of DLAB terms and where Min and Max are two integers such that $0 \leq Min \leq Max \leq size(L)$. The system will then choose recursively every subsets of L with size between Min and Max.

In the above example, the left part of the template will consist in any subset of size $0, 1$ or 2 of the set `[job(c_job),company_type(c_company_type),subscription(_S), course_len(_S,c_course_len),course_type(_S,c_course_type)]` and the right part, of any subset of size $0, 1, 2, 3, 4$ or 5 (there are 5 elements in the set and *len* is a keyword to specify the maximum length) of the same set. To choose exactly one element in the set L, you must use the setting $1-1 : L$.

A *dlab_variable* allows to define shortcuts for frequently occurring parts (for example, `1-1:[researcher,manager,president]`).

For more information about this bias, the user should refer to [13, 34].

7.3.4 The settings file

The setting file should contain all the settings required in Section in addition to any other ICL-specific optional settings the user might be interested in.

```
use_package(icl).
```

```
predict([party_no, party_yes]).
...
```

7.3.5 The output file

In the setting file, the user has specified to learn rules for *all* target attributes, so the system will then induce a set of rules for *party_no* as well as for *party_yes*.

```
ace-registration> induce(icl)
```

ICL gives as output (in the *app.out* file) two rules for the default DNF setting:

```
rule(party_yes, [[2,0],[1,3]], company_type(commercial)),
rule(party_no, [[0,3],[3,0]], subscription(A),course_len(A,4),\+company_type(commercial))
```

$[[2,0],[1,3]]$ gives in the first list, the number of examples of each class covered by this rule and in the second one, the number of examples of each class not covered by this rule.

Chapter 8

The WARMR System

8.1 The WARMR Algorithm

WARMR is a general purpose Inductive Logic Programming (ILP) data mining algorithm [16, 17]. It can discover knowledge in structured data, where patterns reflect one-to-many and many-to-many relationships in the data. The patterns discovered by WARMR are expressed as Prolog queries. Given a language bias $\mathcal{L}$ (Section 4.2.2), a set of examples E, and a minimum support threshold $minfreq \in \mathbb{R}^+$, WARMR finds all queries in $\mathcal{L}$ that cover at least $minfreq \cdot |E|$ examples.

WARMR uses the efficient level-wise search known from the Apriori algorithm [1]. This allows it to be used on very large databases. Figure 8.1 contains a high-level description of the WARMR algorithm. Each iteration of the main loop corresponds to a certain level k of the level-wise search through the pattern space (Figure 8.2). For each level, there are two phases: a candidate generation phase and a candidate evaluation phase. For level $k = 0$, the set of candidate patterns C_k is initialized to contain only the query $\texttt{root}(X_1, \ldots, X_{m1})$ (with `root` the root predicate of the language bias). During the candidate evaluation phase, the frequency of each candidate query is computed by counting the number of covered examples in E. Queries that are found frequent are added to the set of frequent queries F_k, and infrequent queries are added to the infrequent query set I. In the second phase, the candidates for the next level are generated. Each refinement of a frequent query is a new candidate unless certain constraints are violated (see further). After all candidates have been generated, the main loop enters the next level. The algorithm terminates if the candidate set is empty and the return value is the set of frequent queries for each level.

Three constraints (lines 9-11) in the query generation phase ensure that each query is generated only once and that queries that can be proved to be infrequent are not generated. The first two tests check if the new candidate q' is equivalent to its parent query q or to one of the preceding candidates. The last test checks if the candidate is a specialization of an infrequent query; this implies that the new candidate must also be infrequent.

The level-wise approach has two crucial useful properties. First, the database is scanned at most k times, where k is the maximum level (size) of a frequent pattern; all candidates of a level are tested in single database pass. This is an important factor when mining large databases. Second, the time complexity is linear int the number of examples.

More details about the WARMR algorithm can be found in [15].

algorithm WARMR(E, *minfreq*)
1 $k := 0$; $C_0 := \{$ `root`$(X_1, \ldots, X_{m1})$ $\}$; $I := \emptyset$
2 **while** $C_k \neq \emptyset$
3 $S := 0$
4 **for each** $q \in C_k$, $e \in E$
5 **if** covers(q, e) **then** increment $S[q]$
6 $F_k := \{q \in C_k \mid S[q] \geq minfreq \cdot |E|\}$
7 $I := I \cup (C_k - F_k)$; $C_{k+1} = \emptyset$
8 **for each** $q \in F_k$, $q' \in \rho(q)$
9 **if not** ($q \sim q'$
10 **or** $\exists\, q'' \in C_{k+1},\ q'' \sim q'$
11 **or** $\exists\, q'' \in I,\ q'' \succeq q'$)
12 **then** $C_{k+1} := C_{k+1} \cup \{q'\}$
13 $k := k + 1$
14 **return** $\bigcup F_k$

Figure 8.1: The WARMR algorithm returns all queries in the search space that cover at least a fraction *minfreq* of the given examples E. (The refinement operator $\rho(q)$ returns the set of refinements of the given query q that are in the language bias $\mathcal{L}$.)

8.2 An Example Application

Consider again the Machines example from Chapter 3 and suppose that we have the following settings file:

```
warmode_key(machine(-machine)).

minfreq(0.2).

typed_language(yes).
type(worn(machine,comp)).

rmode(worn(+M,#)).
```

The first setting `warmode_key(machine(-machine))` tells WARMR that it has to count machines. As a result, each query will start with the root predicate `machine(M)`. The next setting `minfreq(0.2)` sets the minimum frequency threshold. In this case only queries covering at least 20% of the examples will be returned.

Suppose that the constants generated for `worn/2` are `chain`, `engine`, `control_unit`, `gear`, and `wheel`, then the search tree traversed by WARMR is shown in Figure 8.2.

To start the WARMR algorithm, we use the following command.

1. `warmr -- w`
   starts the discovery process.

The frequent queries are written to the file `mach.freq_queries.out` in the `warmr3` folder. We will discuss the content of this file in the following section. The file `mach.freq_packs.out` essentially contains the same information, but in a different format: in this file, the queries are represented in a trie-structure known as a query-pack [6].

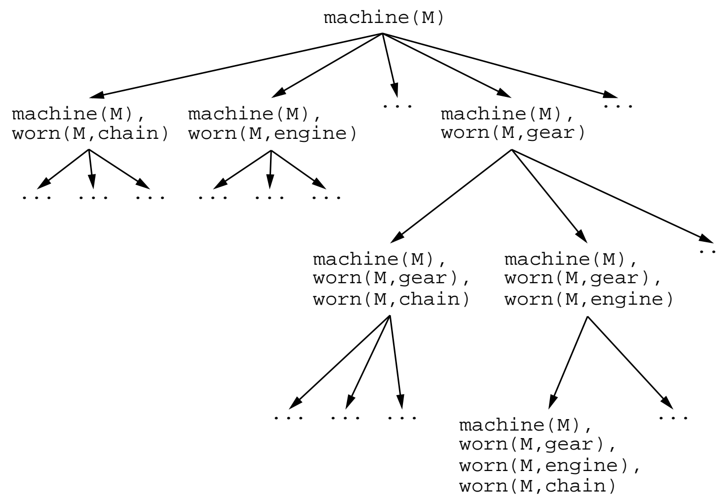

Figure 8.2: The search-tree for the Machines (Chapter 3) data set. (Assuming constant generation is used for the second argument of `worn/2`.)

8.2.1 The Frequent Queries

Warmr writes the frequent queries to the file `app.freq_queries.out`, with `app` the name of the application. This file can contain the following facts.

- `sample_size(S)`: `S` is the total number of examples $|E|$.
- `min_rel_freq(RF)`: `RF` is the minimal frequency (i.e., the `minfreq` setting)
- `min_abs_freq(AF)`: `AF` is $minfreq \cdot |E|$.
- `level(L)`: indicates that Warmr enters level `L`.
- `c_counter(L,C)`: indicates that `C` candidates were generated at level `L`.
- `f_counter(L,F)`: indicates that `F` frequent queries were found at level `L`.
- `freq(L,I,Q,F)`: query `Q` is the `I`th frequent query found at level `L` (its frequency is `F`).

Consider again the Machines example. The frequent queries for this example are the following.

```
sample_size(15.0).
min_rel_freq(0.2).
min_abs_freq(3.0).

level(1).
freq(1,1,[machine(A)],1.0).
c_counter(1,1).
f_counter(1,1).

level(2).
freq(2,1,[machine(A),worn(A,chain)],0.333).
freq(2,2,[machine(A),worn(A,engine)],0.333).
freq(2,3,[machine(A),worn(A,gear)],0.4).
```

```
freq(2,4,[machine(A),worn(A,wheel)],0.4).
c_counter(2,5).
f_counter(2,4).

level(3).
freq(3,1,[machine(A),worn(A,chain),worn(A,gear)],0.2).
freq(3,2,[machine(A),worn(A,chain),worn(A,wheel)],0.2).
freq(3,3,[machine(A),worn(A,gear),worn(A,wheel)],0.2).
c_counter(3,6).
f_counter(3,3).

level(4).
c_counter(4,1).
f_counter(4,0).
```

In this case, the data set contains 15 examples and the minimum frequency threshold was set to 20%, meaning that a query is frequent if it covers at least 3 examples. At the second level, 4 frequent queries were found (the query `machine(A),worn(A,control_unit)` covers only 2 examples and is not frequent). At the third level, 3 frequent queries were found and at the fourth level, no frequent queries were found.

8.3 Generating Rules

WARMR can generate two types of rules: query extensions and Horn clauses. We first discuss query extensions and after that Horn clauses.

8.3.1 Generating Query Extensions

A query extension is defined as follows.

Definition 8.1 (Query extension). *A query extension is an implication of the form* $\forall X_1, \ldots, X_m : \exists(b_1 \wedge \ldots \wedge b_n) \rightarrow \exists(b_1 \wedge \ldots \wedge b_n \wedge h)$*, with* $X_1, \ldots, X_m$ *the key variables introduced by the root predicate,* h *the head of the query extension, and* $b_1 \wedge \ldots \wedge b_n$ *the body of the query extension. A query-extension can also be written as* $b_1 \wedge \ldots \wedge b_n \rightsquigarrow h$*.*

For example,

`machine(M),worn(M,chain)` ⇝ `worn(M,gear)`

is the query extension (`M` is the key variable)

∀ `M`: (`machine(M)` ∧ `worn(M,chain)` → `machine(M)` ∧ `worn(M,chain)` ∧ `worn(M,gear)`)

and should be read as: for all machines `A` it holds that if the machine has a worn chain, it will also have a worn gear. Note that this query extension is very similar to a propositional association rule.

It becomes more difficult if the body of the query extension introduces non-key variables. Consider for example a Bongard problem [14]. In a Bongard problem, each example is a picture containing geometrical objects, such as triangles, squares and circles.

In this example,

`bongard(B),triangle(B,T)` $\leadsto$ `orientation(B,T,up)`

is the query extension

∀ B: (∃ T: `bongard(B)` ∧ `triangle(B,T)` →
∃ T: `bongard(B)` ∧ `triangle(B,T)` ∧ `orientation(B,T,up)`)

and should be read as: for all Bongard pictures B it holds that if the picture contains a triangle, that it will also contain a triangle pointing upwards. Because the variable `T` is existentially quantified, it is sufficient that there is one triangle in the picture that points upwards, i.e., the query extension does not say that all triangles in the picture should point upwards.

Similar to association rules, the interpretation of the query extension can be changed by adding a confidence and support parameter, which are defined as follows.

$$\text{support}(b_1 \wedge \ldots \wedge b_n \leadsto h) = \text{freq}(\exists(b_1 \wedge \ldots \wedge b_n \wedge h)) \tag{8.1}$$

$$\text{confidence}(b_1 \wedge \ldots \wedge b_n \leadsto h) = \frac{\text{freq}(\exists(b_1 \wedge \ldots \wedge b_n \wedge h))}{\text{freq}(\exists(b_1 \wedge \ldots \wedge b_n))} \tag{8.2}$$

Consider again `machine(M),worn(M,chain)` $\leadsto$ `worn(M,gear)`. Because

`freq(machine(M),worn(M,chain))` = 0.333

and

`freq(machine(M),worn(M,chain),worn(M,gear))` = 0.2,

this query extension has a support of 0.2 and a confidence of 0.2/0.333 = 0.6.

To generate query extensions with Warmr, add the following settings to the settings file:

```
warmr_assoc([warmr_rules,
             warmr_rules_min_confidence(0.4),
             warmr_rules_min_support(0.2)]).

warmr_assoc_output_options([assoc_pred]).
```

With in this case 0.4 as minimum confidence threshold and 0.2 as minimum support threshold for the generated query extensions. The output is written to the file `mach.rules_assoc_pred.out` in the folder `warmr3`. In this example, it will contain the following rules:

```
rules(([worn(A,gear)]:-[machine(A),worn(A,chain)]),
      bodyfreq(0.333),sup(0.2),conf(0.6),lift(none)).

rules(([worn(A,wheel)]:-[machine(A),worn(A,chain)]),
      bodyfreq(0.333),sup(0.2),conf(0.6),lift(none)).

rules(([worn(A,wheel)]:-[machine(A),worn(A,gear)]),
      bodyfreq(0.4),sup(0.2),conf(0.5),lift(none)).
```

8.3.2 Generating Horn Clauses

Warmr can also generate Horn clauses. In this case, all variables in the head are universially quantified. Consider again the Bongard example.

`bongard(B),triangle(B,T)` $\rightarrow$ `orientation(B,T,up)`

In this case, also `T` is universially quantified, meaning that for all pictures `B` and all objects `T` that point upwards, it holds that `T` is a triangle.

To generate Horn clauses with Warmr, add the following settings to the settings file:

```
warmr_assoc([horn_clauses,
             assoc_min_confidence(0.4),
             assoc_min_support(0.2)]).

warmr_valid_func(assoc_std).
warmr_assoc_output_options([assoc_pred]).
```

This results in the following Horn clauses:

```
assoc(([worn(A,gear)]:-[machine(A)]),
      bodyfreq(1.0),sup(0.4),conf(0.4),lift(none)).
assoc(([worn(A,wheel)]:-[machine(A)]),
      bodyfreq(1.0),sup(0.4),conf(0.4),lift(none)).
assoc(([worn(A,engine)]:-[machine(A),worn(A,chain)]),
      bodyfreq(0.33333),sup(0.13333),conf(0.4),lift(none)).
assoc(([worn(A,chain)]:-[machine(A),worn(A,gear)]),
      bodyfreq(0.4),sup(0.2),conf(0.5),lift(none)).
assoc(([worn(A,wheel)]:-[machine(A),worn(A,gear)]),
      bodyfreq(0.4),sup(0.2),conf(0.5),lift(none)).
assoc(([worn(A,chain)]:-[machine(A),worn(A,wheel)]),
      bodyfreq(0.4),sup(0.2),conf(0.5),lift(none)).
assoc(([worn(A,chain)]:-[machine(A),worn(A,gear),worn(A,wheel)]),
      bodyfreq(0.2),sup(0.13333),conf(0.66667),lift(none)).
```

8.4 Commands

This section presents an overview of all commands available in a Warmr session.

- `warmr` – `w`
  Start the Warmr discovery process
- `generate_arff` – `gen_arff`
  Generate a Weka .arff file, which indicates for every example if the patterns in FreqFile (as feature) hold or not
- `get_warmr_frequent(ResultVar)` – `gwf(ResultVar)`
  starts the Warmr discovery process and returns the resulting queries in ResultVar

8.5 Settings

This section describes the settings supported by Warmr.

- `minfreq(R)`, where R must be a positive real number ≤ 1 (default: `0.01`). This setting specifies the minimum frequency threshold.
- `warmr_maxdepth(T)`, where T should be an integer (default: `100`). This setting specifies the maximum search depth for the level-wise search carried out by Warmr.
- `warmr_batch_size(T)`, where T should be an integer (default: `50000`). If the number of candidate patterns is too large to fit in memory, then candidate queries can be generated and tested in batches. This setting specifies the size of a query batch.
- `warmr_infreq_spec_level(T)`, where T should be an integer (default: `inf`). Warmr by default keeps a list of infrequent queries. However, testing each query against all infrequent queries can be expensive time-wise and storing all infrequent queries can also consume too much memory. Using this setting, one can make Warmr only store infrequent queries up to a certain level.
- `warmr_output_options(T)`, where T should be a list of options: freq_queries, freq_packs, ... (default: `[freq_queries, freq_packs, show_query_index]`). Warmr writes its output to the folder warmr3. Using this setting, one can control which output files are generated by Warmr.
- `warmr_valid_func(T)`, where T name of the set of tests done by the validation function (default: `std_meta`). Warmr has several implementations of the validation function, i.e., the function that performs the three tests on each candidate (lines 9-11 of the pseudo algorithm). The default, std_meta, should be ok in most cases. See "J. Ramon, and J. Struyf, Efficient theta-subsumption of sets of patterns, Annual Machine Learning Conference of Belgium and the Netherlands (Benelearn 2004), pp. 95-102, 2004".
- `warmr_assoc(T)`, where T must be off or a list of options (default: `off`). This setting should be used to turn association rule generation on. The argument of this setting is a list of options.
  - Include warmr_rules to generate query extensions, in combination with:
  - warmr_rules_min_confidence(0.4)
  - warmr_rules_min_support(0.2)
  - Include horn_clauses to generate Horn Clauses, in combination with:
  - assoc_min_confidence(0.4)
  - assoc_min_support(0.2)
- `warmr_assoc_output_options(T)`, where T should be a list of options: queries, packs, ... (default: `[queries, show_query_index]`). Setting used to generate query-extensions and Horn Clauses. See discussion in Section 8.3.2 and Section 8.3.1.

Chapter 9

The RRL system

The RRL system implemented in ACE is a Q-learning system that uses relational function approximation for both the Q-function and the policy. It explores a user defined world through stochastic (Boltzmann-driven) action selection and an SMDP-version of the Bellman equations to generate learning examples for the Q-function approximation. A high-level overview of the algorithm is shown in Figure 9.1, more information can be found in e.g. [20].

Several relational function approximators are included in the ACE system, such as (among others) an incremental tree-builder and instance based regression. See Section 10 for more information on the available systems and how to use them.

algorithm RRL
1 initialize the Q-function hypothesis $\hat{Q}_0$
2 $e \leftarrow 0$
3 **repeat** {for each episode}
4    $Examples \leftarrow \emptyset$
5   generate a starting state s_0
6    $i \leftarrow 0$
7   **repeat** {for each step of episode}
8   choose a_i for s_i using a policy derived from the current hypothesis $\hat{Q}_e$
9   take action a_i, observe r_i and s_{i+1}
10       $i \leftarrow i + 1$
11   **until** s_i is terminal
12   **for** $j = i - 1$ to 0 **do**
13   generate example $x = (s_j, a_j, \hat{q}_j)$ where $\hat{q}_j \leftarrow r_j + \gamma max_a \hat{Q}_e(s_{j+1}, a)$
14       $Examples \leftarrow Examples \cup \{x\}$
15   **end for**
16   Update $\hat{Q}_e$ using *Examples* and a relational regression
         algorithm to produce $\hat{Q}_{e+1}$
17    $e \leftarrow e + 1$
18 **until** no more episodes

Figure 9.1: The Relational Reinforcement Learning Algorithm

9.1 RRL input files

The input files for RRL are the same as used for other applications in the ACE system. The .kb-file is however not used (and therefore not required) as the RRL system generates its own learning examples through interaction with its environment.

The RRL system does require the user to supply an environment, which can be defined in the background file, i.e., the .bg-file. (It is however common practice, cfr. the examples included with the ACE system, to describe the environment in a .env file and load this file from the .bg file with the `:-['blocks.env'].` statement.

9.1.1 Defining the environment

RRL interacts with its environments through a number of PROLOG predicates that have to be implemented by the user:

`rrl_env_init/0` Allows the user to initialize the environment (e.g. create data-structures) if needed. This predicate will be called at the start of an experiment and between each of the experiments when using the mrrl/1 command, see Section 9.2).

`rrl_env_cleanup/0` Complimentary with the previous predicate, this predicate is called at the end of each experiment.

`rrl_env_worldType/2` RRL allows the user to vary the environment in which the agent is trained during one experiment. This allows the user to create a policy which works on a set of related environments. This predicate is called before each learning or testing episode is called to allow the definition of a world-type. This world-type is then used as the first argument in all the predicates below. The first (input) argument of `rrl_env_worldType/2` is either the number of the current episode for normal learning episodes, the term '`test`' for testing episodes or the term '`teacher`' for guided episodes (See Section 9.1.2), the second (output) argument is the user-defined world-type. (Defining this predicate is not required. If this predicate fails, the first argument in the predicates below will be a free variable.)

`rrl_env_beginstate/2` This predicate is called at the start of each episode. The first (input) argument is the world-type the second argument should define a/the starting state for the given world-type.

`rrl_env_possibleActions/2` This predicate is called for each encountered state. The first argument is again the world-type, the second is the state under consideration. The predicate should return a list of actions that can be applied in that state as the third argument.

`rrl_env_apply/5` This predicate is called when RRL has decided to take an action. The first argument is the world-type as usual, the second argument is the state from which the action will be executed and the third is the chosen action. The two output arguments are the resulting state as the fourth argument and the resulting reward as the fifth.

`rrl_env_stopcondition/2` This predicate is called for each encountered state and should succeed when the end of an episode is reached. The first argument is the world-type, the second is the state under consideration. For learning episodes, this is when the encountered state-action pairs are translated into learning episodes, for testing episodes this is when the episode is evaluated.

`rrl_env_absorbingstate/2` This predicate is called for all states for which `rrl_env_stopcondition/2` succeeds. States for which this predicate succeeds get a state-value of 0 (and thus a maximum Q-value of 0) during the generation of learning examples.

`rrl_env_equal/2` This predicate should succeed when the two states supplied as input arguments can be considered equal in the environment. This can be useful when semantically equal states are not always guaranteed to be syntactically equal as well.

9.1.2 Using guidance

The RRL system also supports the use of guidance. More specifically, a user-defined policy can be used to create traces of execution of a "reasonable policy" that helps the learning RRL agent to explore the environment and receive positive feedback. More details of this integration can be found in [27]. This user-defined policy can be implemented through the following predicate:

`rrl_env_getActionFromTeacher/2` This predicate is called in every step where RRL uses guidance. The second argument should output the action that will be taken by the "guidance policy" in the state provided in the first argument.

9.2 Commands

This section presents an overview of all commands available in a RRL session.

- `rrl` – `rrl`
  Starts the relational reinforcement learning process
- `mrrl(N)` – `mrrl(N)`
  Starts N sequential relational reinforcement learning processes

9.3 Settings

This section describes the settings supported by RRL.

- `rrl_Qlearner(`*I*`)`, where I must be tg or rib3 or csrib or earib or ribc or nlp or kbr or tni or irc or tgconv or gtb (default: `tg`). This setting specifies which incremental learner should be used to estimate the Q-function
- `rrl_numberOfEpisodes(`*I*`)`, where I must be an integer (default: `1000`). This setting specifies the total number of episodes
- `rrl_testFrequency(`*I*`)`, where I must be off or a positive integer (default: `100`). This setting specifies the frequency with which the learned policy should be tested. The results will be logged in a .results file This setting can be set to off if no testing episodes should be performed
- `rrl_numberOfTests(`*I*`)`, where I must be an integer (default: `100`). This setting specifies the number of episodes used when testing the current policy.
- `rrl_saveQFunctions(`*I*`)`, where I must be off or a positive integer (default: `50`). This setting indicates if intermediate trees should be saved or not.
- `rrl_gamma(`*I*`)`, where I must be between 0.0 and 1.0 (default: `0.9`). This setting specifies the reward discound factor ($0.0 \leq \gamma \leq 1.0$)

- `rrl_alpha(I)`, where I must be between 0.0 and 1.0 (default: `0.1`). This setting specifies the learning rate used by the system ($gamma \leq 1.0$)
- `rrl_startTemperature(I)`, where I must be a number (default: `5.0`). This setting specifies the start temperature of the Boltzman policy
- `rrl_minTemperature(I)`, where I must be a number (default: `1.0`). This setting specifies the start temperature of the Boltzman policy This setting specifies the minimal temperature of the Boltzman policy
- `rrl_temperatureDecay(I)`, where I must be between 0.0 and 1.0 (default: `0.95`). This setting specifies the temperature evoluation (decay) of the Boltzman policy
- `rrl_teacherFrequency(I)`, where I must be off or a positive integer (default: `off`). This setting specifies if guidance should be used. If it is set to off, no guidance will be used otherwise the system will generate episodes supplied by the teacher with the frequency specified with this setting
- `rrl_numberOfQuestions(I)`, where I must be an integer (default: `5`). This setting specifies the number of traces supplied by the teacher during every guidance episode
- `rrl_askQuestions(I)`, where I must be yes or no (default: `no`). This setting specifies if active guidance should be used. This setting is only relevant if rrl_teacherFrequency is not set to 'off'.
- `rrl_freezePolicy(I)`, where I must be yes or no (default: `yes`). If this setting is set to yes, a greedy policy is used during testing, if set to no the exploration policy will be used.

Chapter 10

Incremental learning systems

Even though we describe incremental learning systems in the context of RRL, they can also be used as separate components that can be used when developing other systems (active learning, stream learning, ...) that can benefit from incremental learning.

An incremental learning system has an operational learned theory at any time and can update its theory by processing one example or a small batch of examples. After processing examples, the learning system never needs them again. This can save a substantial amount of memory compared to batch learners that need all examples until they are finished.

The disadvantage is of course that learning efficiency may be lower, as not all information may be exploited optimally.

10.1 The TG algorithm

10.1.1 Introduction

The TGsystem is an incremental relational decision tree learner and can be seen as a first order extension of the G algorithm [10]. Figure 10.1 shows a high level description of the regression algorithm. It has first been described in [24].

Algorithm TG

```
1  initialize by creating a tree with a single leaf with empty statistics
2  for each learning example that becomes available do
3     sort the example down the tree using the tests of the internal nodes until it reaches a leaf
4     update the statistics in the leaf according to the new example
5     if the statistics in the leaf indicate that a new split is needed then
6        generate an internal node using the indicated test
7        grow 2 new leafs with empty statistics
8     end if
9  end for
```

Figure 10.1: The TG-algorithm

10.1.2 Language bias

The construction of new tests is done through a refinement operator similar to the one used by the TILDE system. The TG system should be provided an rmode language bias that specifies the predicates that can be used in every node together with their possible variable bindings.

Such a language bias can be defined as documented in Section 4.2.3. Note however that the current implementation of the TG system is based on an older version of the refinement operator and that not all extensions given in Section 4.2.3, such as aggregates, sampling, etc. will be readily available. For expert users: The construction of candidate tests still happens in the "findall-pack" way and not yet in the more recent "rmode_refop class" way.

TG uses the term setting to store examples. This means examples are never asserted in the form of a logic program, but every example is just a prolog term. Implicit knowledge can be defined in the background `.bg` file.

The root of the language should contain as the first literal a `model_info/1` literal, which will bind its argument to the example variable. The example has the following format: It is a `tgmodel/3` functor, where only the second argument is to be used by the language (as the first and third argument contain Q-value information and these will not be instantiated correctly at the time of prediction). The second argument of this `tgmodel/3` functor is a `saPair/2` functor containing the state as its first argument and the action as its second argument.

An example language bias specification for the blocks world learning $on(X, Y)$

Assume a state in the blocks world is represented by a `blocks/4` functor (`blocks(Clear,On,Goal,Steps)`) with a list of all clear blocks as the first argument, the list of all *on* pairs in the second argument. The third argument contains the learning goal and the last argument the number of remaining steps allowed by the agent in this episode.

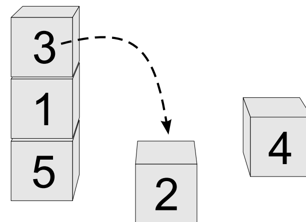

Figure 10.2: An example state of a blocks world with 5 blocks.

The blocksworld state depicted in Fig. 10.2 would be represented by the following functor assuming the current goal of the agent is learning to stack the block with id 1 on the block with id 2:

```
blocks([clear(3),clear(2),clear(4)],[on(3,1),on(1,5),on(5,fl),on(2,fl),on(4,fl)],on(1,2),2)
```

The following can be the root definition in the `.s` file.

```
type(model_info(modelinfo)).
type(get_model(modelinfo,state,block,block)).
type(goal_on(state,block,block)).

rmode_key(model_info(MI)).
```

```
root( (model_info(MI),
       get_model(MI,State,X,Y),
       goal_on(State, A,B)) ).
```

In the background (`.bg` file), one should then have:

```
get_model(
         tgmodel(_,saPair(State,move(X,Y)),_),
         State,X,Y).
goal_on(blocks(_,_,on(A,B),_),A,B).
```

A few possible candidate test can be defined as follows in the `.bg` file:

```
rmode(10: clear(+State,+-X)).
rmode(10: on(+State,+-X,+-Y)).
rmode(10: on(+State,+-X, floor)).
```

with these predicates defined in the background file:

```
%clear(state,block)
clear(blocks(Clear,_,_,_),Block):-
    member(clear(Block),Clear).

%on(state,block,block)
on(blocks(_,On,_,_),X,Y):-
    member(on(X,Y),On).
```

10.1.3 Settings

The following settings can be used for TG.

- `tg_min_sample_size(`*I*`)`, where I must be a positive integer (default: `100`). This setting indicates the minimal examples a leaf should contain before it is split. This number should ideally be sufficiently large so that the chance that a leaf is split based on a bad test is low, but at the same time the value should not be too large as this may slow down learning.
- `tg_leafdrift(`*I*`)`, where I must be a number (default: `100`). When a leaf is created, the system starts to predict for that leaf the average Q-value of the examples that would have been sorted in that leaf since the construction of its parent. The 'weight' of this initial value is set by this setting. So the higher this value, the slower the value of the leaf will drift when new examples are added.
- `tg_prop_opt(`*I*`)`, where I must be on or off (default: `on`). If this setting is set to on, TG will do the 'propositional' optimisation, i.e. if a test does not introduce new variables, the pack for collecting statistics in the left child node will be the same as the one in the right child node.
- `tg_mode(`*I*`)`, where I must be regression (default: `regression`). This setting is always set to 'regression' at the moment.
- `tg_conv(`*T*`)`, where T must be parameter list or off (default: `off`). Since the implementation of the TG-Conv system, this setting is getting more and more obsolete. Set this setting to `off` if you're not an expert.

10.1.4 Known issues

The TG system does not check the size of its theory. The memory consumed is at least linear in the size of the theory. If you use TG as a learner for RRL in an experiment with an infinite amount of episodes on a sufficiently complex world, there is a good chance that you will eventually run out of memory.

Also, the TG system does not revise a test once it is chosen. Use the TGR (TG-Conv) system[25] in Section 10.2 if you want theory revision features.

10.2 The TG-Conv algorithm

The TG-Conv system implements three different incremental decision three learners, but not all of these are at the moment stable enough to be safely used if you are not an expert.

A short overview of the three different algorithms:

- TG-Conv: a decision tree learner that has formal convergence garantuees to the optimal policy when used to approximate the Q-function. More details are available in [36]
- a new implementation of the TGalgorithm that uses the newer refinement operator
- TGR: an extension of the TGalgorithm that can revise tests in the decision tree, see [25] for more details.

10.3 The KBR algorithm

10.3.1 Introduction

The KBR algorithm uses Gaussian processes as an approximation of the Q-function. In order to employ Gaussian processes in a relational setting, graph kernels are used as a covariance function between state-action pairs. More details can be found in [26].

In order to use this algorithm, a kernel needs to be defined between state-action pairs of the environment at hand.

10.3.2 Settings

- `kbr_kernel(`*K*`)`, where K is of the form `(X,Y,D,KernCall)` with `KernCall/3` a predicate that will be called to calculate the kernel D(third argument) between two state-action pairs X and Y (first 2 arguments). (default: `(X,Y,D,zero_kernel(X,Y,D))`).

10.4 The RIB algorithm

10.4.1 Introduction

The RIB3 algorithm uses an instance based approach to approximate the Q-values of unseen state-action pairs [23]. It requires the user to define a distance function between state-action pairs.

10.4.2 Settings

- `rib_distance(R)`, where R must be an oo distance (default: `(X,Y,D,delta_distance(X,Y,D))`).
- `rib_maxdiff(X)`, where X must be a float (default: `-1.0`).
- `rib_maxlndiff(X)`, where X must be a float (default: `-1.0`).
- `rib_max_theory_size(X)`, where X must be an integer (default: `10000`).
- `rib_safe_lowerbound(X)`, where X must be yes or no (default: `yes`).
- `rib_const_fg(X)`, where X must be off or float (default: `off`).
- `rib_const_fl(X)`, where X must be off or float (default: `off`).
- `rib_limit_inflow(X)`, where X must be off, fg or fl (default: `off`).
- `rib_size_reduction_mode(X)`, where X must be off, error_proximity or error_contribution (default: `off`).
- `rib_reduction_size(X)`, where X must be an integer (default: `0`).

10.5 The TNI algorithm

The TNI (or TRENDI) algorithm uses a combination of the TG and RIB3 algorithms. At a high level, it uses a tg like algorithm to divide the example space into regions and uses instance based predictions in each of the sub-spaces. The algorithm builds a first order decision tree incrementally and stores a copy of the rib algorithm at each of its leafs. More details can be found in [22].

10.6 The IRC algorithm

The IRC algorithm does incremental rote learning for clustering. By setting the `irc_class` setting to 'simple', it will act as purely table based Q-learning.

Chapter 11

Utility Packages

This chapter describes some utility packages.

11.1 The Hypothesis Space Package

The Hypothesis Space package may be useful to find possible problems in the language definition of your application. It has commands for displaying all queries in the language up to a given level.

```
ace-mach> use_package(hypspc)
ace-mach> show_hypothesis_space(2)
** hypothesis space **

worn(A)
worn(A),replaceable(A)
worn(A),not_replaceable(A)
worn(A),worn(B)
```

11.2 The Query Package

The Query package is useful for checking a certain query. It computes statistics for the sets of examples for which the query succeeds and fails.

```
ace-mach> use_package(query)
ace-mach> query(worn(A))
Succeeds for 12 examples, fails for 3 examples.
S1: [fix,sendback,ok] = [6.0,6.0,0.0] entropy: 1.0 (query succeeds).
S2: [fix,sendback,ok] = [0.0,0.0,3.0] entropy: 0.0 (query fails).
Heuristic: 1.0.
Execution time 0.0s.
```

The global distribution of the examples in your application can be obtained with:

```
ace-mach> query(true)
```

```
Succeeds for 15 examples, fails for 0 examples.
S1: [fix,sendback,ok] = [6.0,6.0,3.0] entropy: 1.52192809488736 (query succeeds).
S2: [fix,sendback,ok] = [0.0,0.0,0.0] entropy: 0.0 (query fails).
...
```

11.3 The Prolog Prompt

The Prolog prompt can be used to run Prolog queries. One obtains the Prolog prompt after entering the "prompt" command at the ACE prompt.

```
ace-mach> prompt
Type 'ace.' to restart interactive session, 'halt.' to quit.

ACE ?-
```

At the Prolog prompt one can enter any valid Prolog query. All predicates defined in the background knowledge can be used. One can also use builtin predicates form ACE such as `member/2` and `append/3` and meta-predicates such as `findall/3` (See Appendix **??** for a complete list). In the keys setting, one can also use predicates from the knowledge base. In the models setting this is more difficult.

```
ACE ?- findall(X, replaceable(X), L).
L = [gear,wheel,chain]
```

One can also use module qualifications at the Prolog prompt. This is useful for testing purposes if you are extending ACE with a new package.

```
ACE ?- module_name:pred_name(X,Y,Z).
```

It is also possible to let the system enter a number of Prolog queries at the Prolog prompt automatically after startup. This can be accomplished by adding the queries with `execute/1` in the settings file.

```
execute(prompt).
execute(findall(X, replaceable(X), L)).
execute(ace).
```

11.4 Destructive Arrays and Matrices

When defining background knowledge, it may be interesting to have access to destructive arrays (and matrices) that can store integers or floating point numbers. ACE provides dedicated objects for this purpose. One can construct an array as follows.

```
new_c_object(int_array, Array)
new_c_object(float_array, Array)
new_c_object(int_matrix, Matrix)
new_c_object(float_matrix, Matrix)
```

An array can be accessed with the following predicates. We only list the predicate names for the case of an `int_array`. Similar predicates exist for `float_array`'s. Arrays expand automatically when using the `set` and `add` predicates. The `expand_to` predicate can be used to allocate an array of a certain size that is filled with zero's.

```
int_array_pushback(Array, Value).
int_array_add_value(Array, Index, Value).
int_array_set(Array, Pos, Value).
int_array_get(Array, Pos, Value).
int_array_get_size(Array, Size).
int_array_expand_to(Array, Size).
```

The matrices can be accessed with the following predicates. Unlike the arrays, matrices do not grow automatically. One should specify the size with the `set_size` predicates.

```
int_matrix_set_size(Matrix, Rows, Cols, Fill).
int_matrix_resize(Matrix, Rows, Cols).
int_matrix_get_size(Matrix, Size).
int_matrix_get_cols(Matrix, Size).
int_matrix_get_rows(Matrix, Size).
int_matrix_put(Matrix, Row, Col, Value).
int_matrix_get(Matrix, Row, Col, Value).
int_matrix_clear(Matrix).
```

An array or matrix can also be printed to screen. If an array or matrix is no longer used, it should be deleted, so that the allocated memory is freed.

```
print_c_object(ArrayOrMatrix).
delete_c_object(ArrayOrMatrix).
```

11.5 The Linear Regression Package

The linear regression package provides support for performing a linear regression. In order to be able to use the functions defined by this package, on should put `load_package(linreg)` in the ACE settings file. A linear regression object can be constructed and accessed using the following functions.

```
new_c_object(linreg, LinReg).
linreg_clear(LinReg).
linreg_update(LinReg, XValue, YValue).
linreg_covariance(LinReg, Covariance).
linreg_variance(LinReg, XOrY, Variance).
linreg_mean(LinReg, XOrY, Mean).
linreg_intercept(LinReg, Intercept).
linreg_slope(LinReg, Slope).
linreg_count(LinReg, Count).
linreg_r(LinReg, R).
print_c_object(LinReg).
delete_c_object(LinReg).
```

11.6 Loading .ARFF Files

ACE has support for loading and accessing .arff files (the Weka file format). The relevant predicates are defined in the DBMS package, so, one should put `load_package(dbms)` in the ACE settings file in order to be able to use this feature.

```
dbms_load_arff('test.arff', ID).
```

The number of rows and the database schema can be obtained as follows.

```
dbms_get_rows(ID, NbRows).
dbms_get_schema(ID, Schema).
```

The returned schema contains for each attribute its predicate name and type. The values of the attributes can be retrieved using these predicate names. Suppose that the `RELATION` tag of the .arff file defines the name of the database as "Pen Digits" and the name of the first attribute is "A0", then the value of this attribute for a given row can be retrieved as follows.

```
pen_digits_a0(Row, Value).
```

The following predicate provides more low-level access to the data.

```
dbms_get(ID, Row, Col, Value)
```

Bibliography

[1] R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, and A.I. Verkamo. Fast discovery of association rules. In U. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, editors, *Advances in Knowledge Discovery and Data Mining*, pages 307–328. The MIT Press, 1996.

[2] H. Blockeel. *Top-Down Induction of First Order Logical Decision Trees.* PhD thesis, Department of Computer Science, Katholieke Universiteit Leuven, 1998. `http://www.cs.kuleuven.ac.be/~ml/PS/blockeel98:phd.ps.gz`.

[3] H. Blockeel and L. De Raedt. Lookahead and discretization in ILP. In *Proceedings of the Seventh International Workshop on Inductive Logic Programming*, volume 1297 of *Lecture Notes in Artificial Intelligence*, pages 77–85. Springer-Verlag, 1997.

[4] H. Blockeel and L. De Raedt. Top-down induction of first order logical decision trees. *Artificial Intelligence*, 101(1-2):285–297, June 1998.

[5] H. Blockeel, L. De Raedt, and J. Ramon. Top-down induction of clustering trees. In *Proceedings of the 15th International Conference on Machine Learning*, pages 55–63, 1998.

[6] H. Blockeel, L. Dehaspe, B. Demoen, G. Janssens, J. Ramon, and H. Vandecasteele. Improving the efficiency of inductive logic programming through the use of query packs. *Journal of Artificial Intelligence Research*, 16:135–166, 2002.

[7] L. Breiman. Bagging predictors. *Machine Learning*, 24(2):123–140, 1996.

[8] L. Breiman. Out-of-bag estimation.
ftp.stat.berkeley.edu/pub/users/breiman/OOBestimation.ps, 1996.

[9] L. Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.

[10] David Chapman and Leslie P. Kaelbling. Input generalization in delayed reinforcement learning: An algorithm and performance comparisions. In *Proceedings of the 12th International Joint Conference on Artificial Intelligence*, pages 726–731, 1991.

[11] P. Clark and T. Niblett. The CN2 algorithm. *Machine Learning*, 3(4):261–284, 1989.

[12] L. De Raedt, H. Blockeel, L. Dehaspe, and W. Van Laer. Three companions for data mining in first order logic. In S. Džeroski and N. Lavrač, editors, *Relational Data Mining*, pages 105–139. Springer-Verlag, 2001.

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

[14] L. De Raedt and W. Van Laer. Inductive constraint logic. In Klaus P. Jantke, Takeshi Shinohara, and Thomas Zeugmann, editors, *Proceedings of the Sixth International Workshop on Algorithmic Learning Theory*, volume 997 of *Lecture Notes in Artificial Intelligence*, pages 80–94. Springer-Verlag, 1995.

[15] L. Dehaspe. *Frequent Pattern Discovery in First-Order Logic.* PhD thesis, Department of Computer Science, Katholieke Universiteit Leuven, 1998.
`http://www.cs.kuleuven.ac.be/~ldh/`.

[16] L. Dehaspe and H. Toivonen. Discovery of frequent datalog patterns. *Data Mining and Knowledge Discovery*, 3(1):7–36, 1999.

[17] L. Dehaspe and H. Toivonen. Discovery of relational association rules. In S. Džeroski and N. Lavrač, editors, *Relational Data Mining*, pages 189–212. Springer-Verlag, 2001.

[18] 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.

[19] J. Dougherty, R. Kohavi, and M. Sahami. Supervised and unsupervised discretization of continuous features. In A. Prieditis and S. Russell, editors, *Proceedings of the Twelfth International Conference on Machine Learning*, pages 194–202. Morgan Kaufmann, 1995.

[20] K. Driessens. *Relational Reinforcement Learning.* PhD thesis, Department of Computer Science, Katholieke Universiteit Leuven, 2004.

[21] K. Driessens and S. Džeroski. Integrating experimentation and guidance in relational reinforcement learning. In C. Sammut and A. Hoffmann, editors, *Proceedings of the Nineteenth International Conference on Machine Learning*, pages 115–122. Morgan Kaufmann Publishers, Inc, 2002.

[22] K. Driessens and S. Džeroski. Combining model-based and instance-based learning for first order regression. In *Proceedings of the 22nd International Conference on Machine Learning*, pages 193–200, 2005.

[23] K. Driessens and J. Ramon. Relational instance based regression for relational reinforcement learning. In *Proceedings of the 20th International Conference on Machine Learning*, pages 123–130. Morgan Kaufmann, 2003.

[24] K. Driessens, J. Ramon, and H. Blockeel. Speeding up relational reinforcement learning through the use of an incremental first order decision tree learner. In L. De Raedt and P. Flach, editors, *Proceedings of the 12th European Conference on Machine Learning*, volume 2167 of *Lecture Notes in Artificial Intelligence*, pages 97–108. Springer-Verlag, 2001.

[25] K. Driessens, J. Ramon, and T. Croonenborghs. Transfer learning for reinforcement learning through goal and policy parametrization. In *ICML Workshop on Structural Knowledge Transfer for Machine Learning (Online Proceedings)*, 2006.

[26] K. Driessens, J. Ramon, and T. Gärtner. Graph kernels and Gaussian processes for relational reinforcement learning. *Machine Learning*, 64(1–3):91–119, 2006.

[27] Kurt Driessens and Saso Dzeroski. Integrating guidance into relational reinforcement learning. *Machine Learning*, 57(3):271–304, December 2004. URL = http://www.cs.kuleuven.ac.be/cgi-bin-dtai/publ_info.pl?id=40988.

[28] S. Džeroski, L. De Raedt, and K. Driessens. Relational reinforcement learning. *Machine Learning*, 43:7–52, 2001.

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

[30] Daan Fierens, Jan Ramon, Hendrik Blockeel, and Maurice Bruynooghe. A comparison of approaches for learning probability trees. In *Proceedings of 16th European Conference on Machine Learning*, volume 3720 of *Lecture Notes in Artificial Intelligence*, pages 556–563, 2005.

[31] Y. Freund and R. E. Schapire. Experiments with a new boosting algorithm. In L. Saitta, editor, *Proceedings of the Thirteenth International Conference on Machine Learning*, pages 148–156. Morgan Kaufmann, 1996.

[32] T. Gärtner, Kurt Driessens, and Jan Ramon. Graph kernels and gaussian processes for relational reinforcement learning. In *International Conference on Inductive Logic Programming*, 2003.

[33] L. Kaelbling, M. Littman, and A. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.

[34] C. Nédellec, H. Adé, F. Bergadano, and B. Tausend. Declarative bias in ILP. In L. De Raedt, editor, *Advances in Inductive Logic Programming*, volume 32 of *Frontiers in Artificial Intelligence and Applications*, pages 82–103. IOS Press, 1996.

[35] J.R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann series in Machine Learning. Morgan Kaufmann, 1993.

[36] Jan Ramon. On the convergence of reinforcement learning using a decision tree learner. In *Proceedings of ICML-2005 Workshop on Rich Representation for Reinforcement Learning, Bonn, Germany*, 2005. URL = http://www.cs.kuleuven.ac.be/cgi-bin-dtai/publ_info.pl?id=41743.

[37] V. Santos Costa, A. Srinivasan, R. Camacho, H. Blockeel, B. Demoen, G. Janssens, J. Struyf, H. Vandecasteele, and W. Van Laer. Query transformations for improving the efficiency of ILP systems. *JMLR*, 4(August):465–491, 2003.

[38] J. Struyf and H. Blockeel. Query optimization in inductive logic programming by reordering literals. In *Proceedings of the 13th International Conference on Inductive Logic Programming*, volume 2835 of *Lecture Notes in Artificial Intelligence*, pages 329–346. Springer-Verlag, 2003.

[39] R. Sutton and A. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, Cambridge, MA, 1998.

[40] Anneleen Van Assche, Celine Vens, Hendrik Blockeel, and Sašo Džeroski. First order random forests: Learning relational classifiers with complex aggregates. *Machine Learning*, 64(1-3):149–182, 2006.

[41] W. Van Laer. *From Propositional to First Order Logic in Machine Learning and Data Mining. Induction of First Order Rules with ICL.* PhD thesis, Department of Computer Science, K.U.Leuven, 2002.

[42] 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*, pages 53–60, 1996.

[43] Celine Vens, Jan Ramon, and Hendrik Blockeel. Refining aggregate conditions in relational learning. In J. Fürnkranz, T. Scheffer, and M. Spiliopoulou, editors, *Principles of Data Mining and Knowledge Discovery, Proceedings of the 10th European Conference*, volume 4213 of *Lecture Notes in Artificial Intelligence*, pages 383–394. Springer, September 2006.

[44] Celine Vens, Jan Ramon, and Hendrik Blockeel. ReMauve, a relational model tree learner. In S. Muggleton, R. Otero, and A. Tamaddoni-Nezhad, editors, *Proceedings of the 17th International Conference on Inductive Logic Programming*, volume 4455 of *Lecture Notes in Computer Science*, pages 424–438. Springer, 2007.