AI Dialysis Symptomatology

SWI-Prolog Documentation

Francesco Pontillo (mat. 600119)
Universitá degli Studi di Bari
Dipartimento di Informatica
Via E. Orabona, 4 - 70125 Bari, Italy
francescopontillo@gmail.com

1 main.pl

main(+*Config*, +*SymptomID*) *[det]*

Start the main process with custom parameters.

Arguments

Config The name of the configuration file. It can be:

- default, for the default 'database.properties' file
- ask, in order to type in the file name
- a proper configuration file name

Symptom The symptom ID to use as positive target. It can be:

default, for the default (2)

ask, in order to type in the ID

a proper ID

main_def *[det]*

Start `main/2` with default parameters.

main *[det]*

Start `main/2` and asks for configurations.

make_doc *[det]*

Generate the documentation in both html and tex formats. Both documentations will be under the 'doc' folder.

out *[det]*

Disconnects and halts.

2 database.pl

db_param(*?Name, ?Value*) *[semidet]*

Holds database parameters in the form `db_param(Name, Value)`. Relevant names are: server, port, database, username, password.

get_config_path(-*ConfigPath*) *[det]*

Ask the user to insert the configuration file name, to be found under the 'config' folder.

Arguments

ConfigPath	The name of the database configuration file. The file has to be in the '.properties' format.

read_database_params(*-Driver, -Server, -Port, -Database, -User, -Password*) *[semidet]*

Retrieves all the database connection parameters from a config file whose name is entered by the user when requested. If the user-entered configuration file doesn't exist, fall back on 'config/database.properties'.

Arguments

Driver	The ODBC driver name.
Server	The server address.
Port	The server port.
Database	The database to connect to.
User	The username to use.
Password	The password to use.

read_database_params(*+Path, -Driver, -Server, -Port, -Database, -User, -Password*) *[semidet]*

Retrieves all the database connection parameters from a given config file.

Arguments

Path	The database configuration file name.
Driver	The ODBC driver name.
Server	The server address.
Port	The server port.
Database	The database to connect to.
User	The username to use.
Password	The password to use.

read_database_param(*+PropertiesFile, +Row*) *[det]*

Read database params from an open '.properties' file at the given index. Every read property is asserts into the Prolog in-memory database. It always iterates until the reading is over. Then, always succeeds.

Arguments

PropertiesFile	An open '.properties' file.
Row	The row to start reading from.

connect(+*ConfigPath*) *[det]*

Connect to the database by relying on the *ConfigPath* variable.

	Arguments
ConfigPath	The name of the database configuration file to be found in the 'config' folder, or: • default, use the 'config/database.properties' file • ask, prompt the user for a file name

connect(*-Driver, -Server, -Port, -Database, -User, -Password*) *[semidet]*

Try to connect to the database by using the provided prameters.

	Arguments
Driver	The ODBC driver name.
Server	The server address.
Port	The server port.
Database	The database to connect to.
User	The username to use.
Password	The password to use.

disconnect *[semidet]*

If there is an open connection, disconnects from it.

is_connected *[semidet]*

Check if there is a currently open connection.

symptom(*?ID, ?Attribute, ?Value*) *[semidet]*

Holds information for all retrieved symptoms.

	Arguments
ID	The *ID* of the symptom.
Attribute	The name of the attribute (can be '*ID*' or 'Description').
Value	The value of the cell.

clear_symptoms *[det]*

Retract all `symptoms/3`.

get_symptoms *[semidet]*

Fetch and assert all `symptom/2` from the database. Requires an open connection named dialysis_connection.

print_symptoms *[det]*

Print all the symptom facts in the 'ID:Description' format.

update_symptoms *[semidet]*

Clear (with `clear_symptoms/0`) and refresh (with `get_symptoms/0`) all symptoms.

exists_symptom(*+ID*) *[semidet]*

Check if the given symptom *ID* exists.

	Arguments
ID	*ID* of the symptom to check for.

exists_symptom(*-Count*) *[det]*

Count the number of symptoms.

Arguments

Count	The number of existing symptoms.

ask_target *[semidet]*

Print all existing symptoms and ask the user to choose the one to predict outcomes for. If the user-entered symptom is not valid, the default one is selected, by calling `fallback_default_target/0`.

fallback_default_target *[det]*

Fall back to the default target symptom, after printing a warning.

update_target(*+PositiveID*) *[det]*

Save the positive and negative symptom IDs to be used for the learning step.

positive(*?ID, ?Attribute, ?Value*) *[semidet]*

Hold all the positive examples info, will then be split: `train_positive/3`, `test_positive/3`.

Arguments

ID	The *ID* of the example.
Attribute	The *Attribute* for the cell, can be: '*ID*', 'Patient', 'Center', 'PatientSex', 'PatientRace', 'PatientAge', 'SessionID', 'SessionDate', 'KTV', 'QB', 'ProgWeightLoss', 'RealWeightLoss', 'DeltaWeight', 'ProgDuration', 'RealDuration', 'DeltaDuration', 'SAPStart', 'SAPEnd', 'SAPAverage', 'DAPStart', 'DAPEnd', 'DAPAverage', 'DeltaBloodFlow', 'DeltaUF', 'SymptomID', 'Score'
Value	The *Value* for the cell.

See also `negative/3`

negative(*?ID, ?Attribute, ?Value*) *[semidet]*

Hold all the negative examples info, will then be split: `train_negative/3`, `test_negative/3`.

Arguments

ID	The *ID* of the example.
Attribute	The *Attribute* for the cell, can be: '*ID*', 'Patient', 'Center', 'PatientSex', 'PatientRace', 'PatientAge', 'SessionID', 'SessionDate', 'KTV', 'QB', 'ProgWeightLoss', 'RealWeightLoss', 'DeltaWeight', 'ProgDuration', 'RealDuration', 'DeltaDuration', 'SAPStart', 'SAPEnd', 'SAPAverage', 'DAPStart', 'DAPEnd', 'DAPAverage', 'DeltaBloodFlow', 'DeltaUF', 'SymptomID', 'Score'
Value	The *Value* for the cell.

See also `positive/3`

clear_records(*+RecordName*) *[det]*

Clear the records from the in-memory Prolog database.

Arguments

RecordName	The name of the record, can be positive or negative.

save_records(*+Statement, +RecordName*) *[det]*

Fetch all records from a prepared *Statement*.

Arguments

Statement	The prepared statement to fetch records from.
RecordName	The name of the record to assert rows into; can be positive or negative.

get_records(*+SymptomID, +RecordName*) *[semidet]*

Get all records for a given *SymptomID* and assert them with a given *RecordName*.

Arguments

SymptomID	The ID of the target attribute of the record.
RecordName	The name of the record to assert rows into; can be positive or negative.

print_records(*RecordName*) *[det]*

Print all records with the given *RecordName*

Arguments

RecordName	The name of the record to assert rows into; can be positive or negative.

update_records(*+SymptomID*) *[semidet]*

Update both positive and negative examples by fetching them from the database. The negative ID is fixed (1), but the positive can be passed to this predicate.

Arguments

SymptomID	The ID of the symptom to be used as positive target, or: default, use the 2 ask, let the user decide

update_records *[semidet]*

Clear (`clear_records/1`) and updates (`get_records/2`) all positive and negative examples.

exists_record(*+RecordName, ?ID*) *[semidet]*

Check if the given record *ID* was asserted with the given *RecordName*.

Arguments

RecordName	The name the record was asserted with; can be positive or negative.
ID	The *ID* of the record.

count_records(*+RecordName, ?Count*) *[det]*

Count the number of existing records with a given name.

Arguments

RecordName	The name of the record to count; can be positive or negative.
Count	The number of records.

count_all_examples(*-Count*) *[det]*

Return the number of all examples in the Prolog in-memory database. @see `example/4`

Arguments

Count	The number of retrieved examples

3 categories.pl

data_type(*?Attribute, ?Type*) *[semidet]*

Define the type of the attribute data.

Arguments

Attribute	The *Attribute* name.
Type	The *Type* of the *Attribute*, can be category or number.

target_class(*?Attribute*) *[semidet]*

Holds information of the target *Attribute* to predict for.

Arguments

Attribute	The *Attribute* name holding the target information (always ’SymptomID’).

class(*?Attribute, ?RangeList:list*) *[semidet]*

Holds range information for each attribute. Note: only attributes with at least one value will be classified

Arguments

Attribute	The *Attribute* name.
RangeList	The (ordered) list of ranges the attribute was split into.

update_categories *[det]*

Make ranges for all of the available attributes. For a category: every possible value is both the start and end of the class For a number: use ranges that span (|max-min|/10) values

make_class(*+Attribute, +Type*) *[det]*

Make ranges for a generic *Attribute* with a given *Type*.

Arguments

Attribute	The *Attribute* name.
Type	The *Type* of the *Attribute*.

get_range_span(*+Difference, -Span*) *[det]*

Calculate *Difference*/10 as the range *Span*.

Arguments

Difference	The difference between a minimum and maximum value.
Span	The resulting range cardinality.

generate_range(*+Attribute, +Min, +Max, +Span*) *[det]*

Recursively generate ranges for the given *Attribute*.

Arguments

Attribute	The *Attribute* to generate ranges for.
Min	The current minimum value of the range.
Max	The current maximum value of the range.
Span	The calculated range cardinality.

add_to_class(*+Attribute, +Bottom, +Top*) *[semidet]*

Add the input range to the ranges of the *Attribute* (`class/2`).

Arguments

Attribute	The *Attribute* to add the range to.
Bottom	The minimum value of the range.
Top	The maximum value of the range.

is_in_range(*+Value, ?Range*) *[nondet]*

Check if the given *Value* is included in the given *Range*. If *Range* is a category, Bottom equals Top, so only a check on Bottom = *Value* is performed. If *Range* is a number, *Value* must be higher or equal than Bottom and

Arguments

Value	A value, can be a number or a category.
Range	A *Range* structure, as `range(Bottom, Top)`.

4 util.pl

measure_time *[det]*

Reset the global timer.

measure_time(*-Time*) *[semidet]*

Return the time (in milliseconds) elapsed from the last call to `measure_time/0`.

Arguments

Time	The elapsed time, in milliseconds.

timer_time(*?Name, ?Time*) *[nondet]*

Holds information for existing timers.

Arguments

TimerName	The name of the timer.
TimerStart	The date (in seconds, UNIX epoch time) when the timer was started.

timer_start(*+Name*) *[semidet]*

Start a new timer with a given name. If another timer with the given name exists, an error is raised.

Arguments

Name	The name of the new timer.

timer_get(*+Name, -Elapsed***)** *[semidet]*

Get the elapsed time from a given timer.

Arguments

Name	The name of the timer.
Elapsed	The number of elapsed seconds since the timer was started.

timer_stop(*+Name, -Elapsed***)** *[semidet]*

Stop and destroy a given timer.

Arguments

Name	The name of the timer to be stopped.
Elapsed	The number of elapsed seconds since the timer was started.

timer_must_not_exist(*+Name***)** *[semidet]*

Check if a timer exists and logs an error if it does.

Arguments

Name	The name of the timer to check for.

timer_must_exist(*+Name***)** *[semidet]*

Check if a timer doesn’t exists and logs an error if it does not.

Arguments

Name	The name of the timer to check for.

format_ms(*?Time, ?Minutes, ?Seconds, ?Milliseconds***)** *[semidet]*

Convert a bunch of milliseconds into minutes, seconds an milliseconds.

Arguments

Time	The time in milliseconds.
Minutes	The number of minutes in *Time*.
Seconds	The number of seconds in *Time*.
Milliseconds	The number of remaining milliseconds in *Time*.

format_s(*?Time, ?String***)** *[semidet]*

Convert a bunch of seconds into a string representation.

Arguments

Time	The time in seconds.
Minutes	The string representation of *Time*, in the ’{M}m {S}s {MS}ms’ format.

format_ms(*?Time, ?String***)** *[semidet]*

Convert a bunch of seconds into a string representation.

Arguments

Time	The time in milliseconds.
Minutes	The string representation of *Time*, in the ’{M}m {S}s {MS}ms’ format.

println(*+Element***)** *[det]*

Print something without a separator.

Arguments

Element	The element to print.

See also `println/2`.

list_min(+*List:list, ?Min*) *[semidet]*

Get the minimum element from a numeric list.

Arguments

List	The list to look the minimum into.
Min	The minimum element of the list.

list_max(+*List:list, ?Max*) *[semidet]*

Get the maximum element from a numeric list.

Arguments

List	The list to look the minimum into.
Max	The maximum element of the list.

list_most_common(+*List:list, ?Element, ?Count*) *[semidet]*

Get the most common *Element* from a *List*, returning the *Count*.

Arguments

List	The list to look into.
Element	The most common element of the list.
Count	The number of times the most common element was found in the list.

index_of(+*List:list, +Element, ?Index*) *[semidet]*

Get the position of an element in a list.

Arguments

List	The list to look into.
Element	The element to find the position for.
Index	The index of the element in the list.

list_push(+*List:list, +Element, -ResultingList:list*) *[det]*

Add an element to the end of a list.

Arguments

List	The list to append the element to.
Element	The element to be appended.
ResultingList	The resulting list.

list_unshift(+*List:list, +Element, -ResultingList:list*) *[det]*

Add an element to the top of a list.

Arguments

List	The list to unshift the element to.
Element	The element to be unshifted.
NewList	The resulting list.

list_append(+*A, +B, -ResultingList:list*) *[det]*

Concat two generic elements in one list. Both elements can be lists or atoms, indipendently.

Arguments

A	The first element (atom or list).
B	The second element (atom or list).
List	The resulting list.

log2(+*Expr, -R***)** *[det]*

Calculate the base-2 logarithm of *Expr*.

Arguments

Expr	The expression to calculate the log2 for.
R	The result of the calculation.

5 learner.pl

example(*Kind, ID, Attribute, Value***)** *[semidet]*

Accessory predicate to access any kind of example (both training or testing) by specifying the name the record was asserted with (negative or positive).

Arguments

Kind	The name the record was asserted with; can be positive or negative.
ID	The *ID* of the example.
Attribute	The *Attribute* of the example.
Value	The *Value* of the example.

split_examples(+*FoldCount,* +*TestFold***)** *[det]*

Split all examples between `train_examples/1` and `test_examples/1`.

Arguments

FoldCount	The number of total fold to split the examples between.
TestFold	The index of the testing fold at the current step.

split_examples(+*PositiveNegative,* +*FoldCount,* +*TestFold, -TrainExamples, -TestExamples***)** *[det]*

Split positive or negative examples in two different lists: training and testing, according to the fold number and the current test fold. If *FoldCount* and *TestFold* equal to 1, simulate the testing and training set as being the same.

Arguments

PositiveNegative	The classification of the example, can be positive or negative.
FoldCount	The number of total fold to split the examples between.
TestFold	The index of the testing fold at the current step.
TrainExamples	The generated training examples list of IDs.
TestExamples	The generated testing examples list of IDs.

train_examples(-*List:list***)** *[semidet]*

Return the list of current training examples.

Arguments

List	The list of training examples.

train_example(*Kind, ID, Attribute, Value***)** *[nondet]*

Access any kind of training example by specifying the name the record was asserted with (negative or positive).

Arguments

Kind	The name the record was asserted with; can be positive or negative.
ID	The *ID* of the example.
Attribute	The *Attribute* of the example.
Value	The *Value* of the example.

test_examples(*-List:list*) *[semidet]*

Return the list of current testing examples.

Arguments

List	The list of testing examples.

test_example(*Kind, ID, Attribute, Value*) *[nondet]*

Access any kind of testing example by specifying the name the record was asserted with (negative or positive).

Arguments

Kind	The name the record was asserted with; can be positive or negative.
ID	The *ID* of the example.
Attribute	The *Attribute* of the example.
Value	The *Value* of the example.

train_positive(*?ID, ?Attribute, ?Value*) *[nondet]*

Hold positive training examples information.

Arguments

ID	The *ID* of the example.
Attribute	The *Attribute* for the cell, can be: '*ID*', 'Patient', 'Center', 'PatientSex', 'PatientRace', 'PatientAge', 'SessionID', 'SessionDate', 'KTV', 'QB', 'ProgWeightLoss', 'RealWeightLoss', 'DeltaWeight', 'ProgDuration', 'RealDuration', 'DeltaDuration', 'SAPStart', 'SAPEnd', 'SAPAverage', 'DAPStart', 'DAPEnd', 'DAPAverage', 'DeltaBloodFlow', 'DeltaUF', 'SymptomID', 'Score'
Value	The *Value* for the cell.

See also `negative/3`

train_negative(*?ID, ?Attribute, ?Value*) *[nondet]*

Hold negative training examples information.

Arguments

ID	The *ID* of the example.
Attribute	The *Attribute* for the cell, can be: '*ID*', 'Patient', 'Center', 'PatientSex', 'PatientRace', 'PatientAge', 'SessionID', 'SessionDate', 'KTV', 'QB', 'ProgWeightLoss', 'RealWeightLoss', 'DeltaWeight', 'ProgDuration', 'RealDuration', 'DeltaDuration', 'SAPStart', 'SAPEnd', 'SAPAverage', 'DAPStart', 'DAPEnd', 'DAPAverage', 'DeltaBloodFlow', 'DeltaUF', 'SymptomID', 'Score'
Value	The *Value* for the cell.

See also `positive/3`

test_positive(*?ID, ?Attribute, ?Value*) *[nondet]*

Hold positive testing examples information.

Arguments

ID The *ID* of the example.

Attribute The *Attribute* for the cell, can be: '*ID*', 'Patient', 'Center', 'PatientSex', 'PatientRace', 'PatientAge', 'SessionID', 'SessionDate', 'KTV', 'QB', 'ProgWeightLoss', 'RealWeightLoss', 'DeltaWeight', 'ProgDuration', 'RealDuration', 'DeltaDuration', 'SAPStart', 'SAPEnd', 'SAPAverage', 'DAPStart', 'DAPEnd', 'DAPAverage', 'DeltaBloodFlow', 'DeltaUF', 'SymptomID', 'Score'

Value The *Value* for the cell.

See also `negative/3`

test_negative(*?ID, ?Attribute, ?Value***)** *[nondet]*

Hold negative testing examples information.

Arguments

ID The *ID* of the example.

Attribute The *Attribute* for the cell, can be: '*ID*', 'Patient', 'Center', 'PatientSex', 'PatientRace', 'PatientAge', 'SessionID', 'SessionDate', 'KTV', 'QB', 'ProgWeightLoss', 'RealWeightLoss', 'DeltaWeight', 'ProgDuration', 'RealDuration', 'DeltaDuration', 'SAPStart', 'SAPEnd', 'SAPAverage', 'DAPStart', 'DAPEnd', 'DAPAverage', 'DeltaBloodFlow', 'DeltaUF', 'SymptomID', 'Score'

Value The *Value* for the cell.

See also `positive/3`

complete_set(*?CompleteSet:list***)** *[semidet]*

Get the complete set of training example IDs as a list.

Arguments

CompleteSet The list of all the training example IDs.

entropy(*+IncludedValues, -Entropy***)** *[semidet]*

Calculate the entropy of a given list of training examples (by IDs).

Arguments

IncludedValues A list of the example IDs to be included when considering the entropy calculus. The resulting examples will be intersected with the positive and negative examples. If you don't want to filter anything, pass in all of the example IDs:

```
findall(ID, (train_example(_, ID, 'ID', ID),
train_example(_, ID, 'DeltaWeight', 1)), IncludedValues).
```

Entropy The resulting entropy of the examples passed by IDs.

entropy(*-Entropy***)** *[semidet]*

Calculate the entropy of the whole set of training examples. This is a shortcut clause for:

```
findall(ID, train_example(ID,’ID’,ID), List), entropy(List, Entropy).
```

Arguments

Entropy	The resulting entropy of the whole set of training examples.

best_attribute(*+Set:list, +Attributes:list, -Attribute*) *[semidet]*

Select the best attribute from the given lists of attributes and examples, using the information gain measure.

Arguments

Set	The list of example IDs to calculate the best attribute for.
Set	The list of attributes to select the best attribute from.
Attribute	The best attribute for the given set.

best_attribute(*-Attribute*) *[semidet]*

Select the best attribute from the whole set of attributes and training examples. This is a shortcut clause for:

```
findall(ID, train_example(ID,’ID’,ID), Examples),
findall(Attribute, class(Attribute, _), Attributes),
best_attribute(Examples, Attributes, Entropy).
```

Arguments

Attribute	The best attribute for all of the exmmples and attributes.

info_gain(*+Set:list, +Attribute, -InfoGain*) *[semidet]*

Calculate the Information Gain for a set of training exampes and a given attribute.

Arguments

Set	A list of IDs to be included when considering the info gain calculus.
Attribute	The attribute to calculate the information gain for.
InfoGain	The calculated information gain.

info_gain(*+Attribute, -InfoGain*) *[det]*

Calculate the Information Gain for all training examples and one attribute. This is a shortcut clause for:

```
findall(ID, train_example(ID,’ID’,ID), Examples),
info_gain(CompleteSet, Attribute, InfoGain).
```

Arguments

Attribute	The attribute to calculate the information gain for.
InfoGain	The calculated information gain.

partial_info_gain(*+Set:list, +Attribute, +Range, -PartialInfoGain*) *[semidet]*

Calculate a partial value used to compute the info gain for a given attribute.

Arguments

Set	The list of example IDs to calculate the value for.
Attribute	The attribute name to calculate the value for.
Range	The specific `range(Bottom, Top)` for the given *Attribute*.
PartialInfoGain	The partial value to be used to compute the whole *Attribute* Info Gain.

clean_set(*+Set:list, +Attribute, -CleanSet*) *[det]*

Clean the given list of example IDs (doesn’t matter if training or testing) from ’$null$’ values.

Arguments

Set	The set of example IDs to clean.
Attribute	The attribute whose $null$ value must be deleted.
CleanSet	The *CleanSet*, a list whose *Attribute* does not have ’$null$” values.

partition_examples(*+InExamples:list, +Attribute, +Range, -OutExamples*) *[det]*

Partition a list of example IDs by analyzing an attribute in a given range.

Arguments

InExamples	List of example IDs to analyze and filter.
Attribute	The attribute to filter on.
Range	The range to filter with.
OutExamples	List of example IDs to return.

node(*?NodeName, ?ParentNode, ?SplitAttribute, ?SplitRange*) *[semidet]*

Holds the node name information, the parent node, and the splitting attribute and range.

node_label(*?Node, ?Label*) *[semidet]*

Holds the node name information, the parent node, and the splitting attribute and range.

learn_please *[det]*

Start the learning process:

1. partition the `positive/3` data set in 10 folds
2. partition the `negative/3` data set in 10 folds
3. for every generated fold: a. start the learning phase b. start the testing phase

test_step(*?Step, ?List*) *[nondet]*

Holds information about a particular step run.

Arguments

Step The step the error rate was calculated at.

List *List* containing the following:

`n(AllNegatives)`

`p(AllPositives)`

`rules(GeneratedRules)`

`tn(TrueNegatives)`

`fn(FalseNegatives)`

`tp(TruePositives)`

`fp(FalsePositives)`

`true_pos_rate(TruePosRate)`

`true_neg_rate(TrueNegRate)`

`false_pos_rate(FalsePosRate)`

`false_neg_rate(FalseNegRate)`

`precision(Precision)`

`recall(Recall)`

See also `test/1`.

test_final(*?List*) *[nondet]*

Holds information about the final testing process.

Arguments

List *List* containing the following:

`n(AllNegatives)`

`p(AllPositives)`

`rules(GeneratedRules)`

`tn(TrueNegatives)`

`fn(FalseNegatives)`

`tp(TruePositives)`

`fp(FalsePositives)`

`true_pos_rate(TruePosRate)`

`true_neg_rate(TrueNegRate)`

`false_pos_rate(FalsePosRate)`

`false_neg_rate(FalseNegRate)`

`precision(Precision)`

`recall(Recall)`

See also

- `test_step/2`.
- `test/0`.

is_positive(+*ID, ?LearningStep*) *[nondet]*

Check if a given `example/4` *ID* is positive according to an optionally provided *LearningStep*.

Arguments

ID	The *ID* of the `example/3` to check for.
LearningStep	The step number of the learning process.

See also `check_condition_list/3`

check_positive(+*ID, ?LearningStep*) *[semidet]*

Semi-deterministic version of `is_positive/2`. If there is at least one `is_positive/2` that satisfies the current *ID*, succeed; otherwise, fail.

Arguments

ID	The *ID* of the `example/3` to check for.
LearningStep	The step number of the learning process.

See also `is_positive/2`

check_final_positive(+*ID*) *[semidet]*

Final one-time check for the complete and purged set of rules. If there is at least one `final_positive/2` that satisfies the current *ID*, succeed; otherwise, fail.

Arguments

ID	The *ID* of the `example/3` to check for.

See also `is_positive/2`

final_positive(*?ID, ?StepList*) *[nondet]*

Check if a given `example/4` *ID* is positive. If the example is positive, *StepList* will be instantiated with a list of steps that generated the rule that classifies the example as positive.

Arguments

ID	The *ID* of the `example/3` to check for.
StepList	A list of step indexes where the satisfactory rule was asserted in.

See also `is_positive/2`, `check_condition_list/3`

purge_rules *[det]*

From the list of rules generated at each step of the learning process, assert in the Prolog memory a set of the same rules, but without any duplicate. A duplicate rule is considered a `is_positive/2` rule with the same body. Rules will be asserted as `final_positive/2`, where the first argument is the ID of the example to check for and the second argument is the list of steps where the rule was found (duplicated).

test(+*Step*) *[det]*

Test all `test_positive/3` and `test_negative/3` and calculates several useful information. The calculated information is asserted as `test_step/2`. In the end, it prints a report of the run.

	Arguments
Step	The learning step, defines the particular rules to test against.

See also `test_step/2`

test *[det]*

Test all `positive/3` and `negative/3` and calculates several useful information. The calculated information is asserted as `test_final/1`.

See also `test_final/1`

learn(+*Step*) *[det]*

Start the learning process using given sets of training and testing examples by adding `node/4` and `node_label/2` clauses to the database.

	Arguments
TotalFolds	The number of folds to split the examples into.
Step	The current learning step.

c45(+*Node*, +*Examples:list*, +*Attributes:list*) *[det]*

Apply the C4.5 algorithm to a given node, that will be split according to the training examples passed and the available attributes still left.

	Arguments
Node	The node to build.
Examples	The training examples.
Attributes	The list of attributes to be tested.

print_le_tree *[semidet]*

Print the learnt tree, if there is a `node('root', root, Root, root).`

print_le_branch(+*Node*, +*NestLevel*) *[semidet]*

Print a given node with a nest level that decides how much left space the node representation must have.

A node will be printed with:

- a check character, if the node is terminal and classifies positive examples
- a uncheck character, if the node is terminal and doesn't classify positive examples
- a down arrow, if the node is not terminal

	Arguments
Node	The node to print.
NestLevel	The nesting level to be used.

gen_all_the_rulez(+*Step*) *[semidet]*

For each `node_label/2` with a positive outcome, generate the corresponding rule by going up from the leaf to the tree root node. The rule will be `is_positive/2`.

	Arguments
Step	The current learning step.

gen_rule(+*Node*, +*Step*) *[semidet]*

Generate the rule for the corresponding input *Node*.

Arguments

Node	The leaf *Node* that holds the rule information.
Step	The current learning step.

condition(*?Attribute*, *?Range*) *[semidet]*

Holds information about a condition of success for an *Attribute* in a given *Range*.

Arguments

Attribute	The attribute to test the condition onto.
Range	The range to be used for the test.

get_rule_list(+*Node*, +*PrevList:list*, -*List:list*) *[semidet]*

Builds a list of `condition(Attribute, Range)` given a node, concatenating the conditions to the given *PrevList* (can be empty).

Arguments

Node	The *Node* to build the condition list for.
PrevList	The temporary list for recursion.
List	The list of conditions to return.

ensure_not_null_conditions(+*ID*, +*List:list*) *[semidet]*

Succeed if:

- the *List* is empty
- there is at least one condition Attribute that `example/4` with the given *ID* has not null

Otherwise, it fails.

Arguments

ID	The *ID* of the `example/4` to check.
List	The list of `condition/2` to loop through.

check_condition_list(+*ID*, +*List:list*) *[semidet]*

Check if a given example with an *ID* matches all the conditions in the input list.

Arguments

ID	The *ID* of the example.
List	The *List* of `condition/2`.

get_conditions_from_list(+*Conditions:list*, -*Condition*) *[det]*

Generate a set of Prolog conjunctives from a list of `condition/2`.

Arguments

List	The list of `condition/2`.
Set	The set of Prolog conjunctives.

print_report(+*Elapsed*) *[det]*

Print a detailed report of the learning algorithm and of the executed tests.

Arguments

Elapsed	The number of seconds the algorithm has taken to complete

save_log(+*Elapsed*) *[det]*

Save a log .csv file: 'runs/log_{ID}.csv'. The 'runs' folder must exist.

The file will list, for each run:

the symptom ID (always the same for each file)

the run index

the number of generated rules

the true positive rate

the false positive rate

the precision

the recall

the F-Measure

The last line contains:

- the time of total execution (in seconds)
- the number of positive examples (including both training and testing examples)
- the final number of generated rules
- the final true positive rate
- the final false positive rate
- the final precision
- the final recall
- the final F-Measure

Arguments	
Elapsed	Total time of execution, in seconds.

save_rules *[det]*

Save all generated rules in the prolog file: 'runs/rules_{ID}.pl'. The 'runs' folder must exist.

6 log.pl

log_level(*?Priority, ?Name, ?StringRepr, ?OptList:list*) *[nondet]*

Holds information for every logging level.

Arguments	
Priority	The priority value, the higher the better.
Name	The logging level name.
StringRepr	The string representation of the log level, to be used for printing.
OptList	The options for color printing the level.

log_level(+*Level*) *[semidet]*

Set the log level, any level lesser than this will not be printed. This is a setter that checks that the input parameter is instantiated and that the specified level exists.

	Arguments
Level	The level to set the logging information to.

log_level(-*Level*) *[semidet]*

Return the current log level, checking that the input parameter is an unbound variable that can be instantiated.

	Arguments
Level	The current minumum log level.

can_log(*?Level*) *[nondet]*

Checks if a specified level can be currently printed. Succeed if the input log has a higher or equal priority than the current minimum level. Fails otherwise.

	Arguments
Level	The level to be checked.

log_at(+*Level*, +*Tag*, +*Log*) *[det]*

Log a message at a certain level with a given tag.

	Arguments
Level	The level to log the message with, choose between verbose, debug, info, warn, error, assert.
Tag	The tag to use as the log header.
Log	The actual log message.

log_at(+*Level*, +*Log*) *[det]*

Log a message at a certain level without a tag. Shortcut method to avoid the logging of the tag.

See also `log_at/3`.

log_tag(+*Level*, +*Tag*, +*Max*) *[det]*

Log a tag with a certain level (used for the color) and with a character limit, beyond which ellipsis are added.

	Arguments
Level	The level to log the tag with, choose between verbose, debug, info, warn, error, assert.
Tag	The tag to print.
Max	The maximum tag length.

log_v(+*Tag*, +*Log*) *[det]*

Log a tag and a message at verbose level.

	Arguments
Tag	The tag to be used in the logging entry.
Log	The log message to be printed.

log_d(+*Tag*, +*Log*) *[det]*

Log a tag and a message at debug level.

Arguments

Tag The tag to be used in the logging entry.
Log The log message to be printed.

log_i(+*Tag*, +*Log*) *[det]*

Log a tag and a message at info level.

Arguments

Tag The tag to be used in the logging entry.
Log The log message to be printed.

log_w(+*Tag*, +*Log*) *[det]*

Log a tag and a message at warn level.

Arguments

Tag The tag to be used in the logging entry.
Log The log message to be printed.

log_e(+*Tag*, +*Log*) *[det]*

Log a tag and a message at error level.

Arguments

Tag The tag to be used in the logging entry.
Log The log message to be printed.

log_wtf(+*Tag*, +*Log*) *[det]*

Log a tag and a message at assert level.

Arguments

Tag The tag to be used in the logging entry.
Log The log message to be printed.

log_v(+*Log*) *[det]*

Log a message at verbose level.

Arguments

Log The log message to be printed.

log_d(+*Log*) *[det]*

Log a message at debug level.

Arguments

Log The log message to be printed.

log_i(+*Log*) *[det]*

Log a message at info level.

Arguments

Log The log message to be printed.

log_w(+*Log*) *[det]*

Log a message at warn level.

Arguments

Log The log message to be printed.

log_e(+*Log*) *[det]*

Log a message at error level.

	Arguments
Log	The log message to be printed.

log_wtf(+*Log*) *[det]*

Log a message at assert level.

	Arguments
Log	The log message to be printed.

test_log *[det]*

Test the logging system by printing some messages.