Lost API documentation

Lost Members

December 28, 2011

Contents

Chapter 1

An introduction to the lost framework

1.0.1 accuracy_report

author : Christian Theil Have

Introduction This model generates a Prolog based report of accuracy measures for a set of gene predictions compared to a golden standard annotation.

A lot of different measures are generated. An example of a report is shown below.

```
accuracy_report(genes_predicted, 2130).
accuracy_report(genes_actual, 4144).
accuracy_report(genes_correct, 1882).
accuracy_report(genes_wrong, 248).
accuracy_report(gene_stops_correct, 2003).
accuracy_report(gene_stops_wrong, 127).
accuracy_report(gene_sensitivity, 0.454150579150579).
accuracy_report(gene_specificity, 0.883568075117371).
accuracy_report(gene_specificity, 0.883568075117371).
accuracy_report(gene_stop_sensitivity, 0.483349420849421).
accuracy_report(gene_stop_specificity, 0.94037558685446).
accuracy_report(genes_wrong_ratio, 0.116431924882629).
accuracy_report(genes_missing_ratio, 0.545849420849421).
accuracy_report(nucleotide_true_positives, 1934811).
accuracy_report(nucleotide_false_positives, 75521).
accuracy_report(nucleotide_true_negatives, 619038).
accuracy_report(nucleotide_false_negatives, 2010092).
accuracy_report(nucleotide_sensitivity, 0.757605872547672).
accuracy_report(nucleotide_specificity, 0.962433568186747).
accuracy_report(nucleotide_specificity_traditional, 0.891267696480789).
accuracy_report(nucleotide_correlation_coefficient, 0.27484302373759).
accuracy_report(nucleotide_simple_matching_coefficient, 0.55046231653584).
accuracy_report(nucleotide_average_conditional_probability, 0.644903316252825).
accuracy_report(nucleotide_approximate_correlation, 0.28980663250565).
```

report(*+InputFiles, +Options, +OutputFile*)
Type signature:

```
InputFiles:
    1. text(prolog(ranges(gene)))
    2. text(prolog(ranges(gene)))
OutputFile:
    text(prolog(accuracy_report))
Options:
    - start (default value: min)
    - end (default value: max)
    - reports (default value: all)
```

Creates a accuracy report *OutputFile* based on *InputFiles*. The first input file is a 'Golden Standard' file which contains reference genes. The second input file is the predictions.

A subset of the predictions may be specified with range in specified using the options `start` and `end`. Only predictions which fully lie within this range are considered and reference predictions outside of this range are also not considered. They may both be integers or the special values `min` and `max` respectively (which indicates that all predications should be considered).

1.1 The lost framework

The lost framework is a collection of utilities and models for working with biological sequences written in PRISM/B-Prolog. The framework tries to unify all these bits and pieces and to provide a way to integrate them.

1.2 Obtaining a copy of the lost framework

The lost framework can be obtained using *git* if you have an account on the *mox* server. Assuming that you have *git* installed on your local machine, to get a copy you need to clone the central repository:

```
$ git clone ssh://your-username@mox.ruc.dk/var/git/lost.git
```

It is also possible to clone the repository directly from mox if you want to work there, rather than on your local machine. To do this, just type

```
$ git clone /var/git/lost.git
```

In the following we will use `$LOST` to refer to top-most directory of your copy of the framework.

1.3 Configuring your copy of the lost framework

After obtaining the lost framework, a little configuration is needed to get started. You will need to edit the file `$LOST/lost.pl`. In the beginning of the file there are two important facts you may need to change,

```
lost_config(prism_command,'prism').
lost_config(lost_base_directory, '/change/to/local/lost/dir/').
lost_config(platform, windows_or_unix).
```

The option `prism_command` should point to a the main PRISM executable binary. If it is in your *$PATH* then you can usually leave it unchanged.
`lost_base_directory` should be the full path of the directory (including trailing /) containing

the `lost.pl` file. Note, that even on windows platforms you should use forward slash rather than backslash in the path specification. The value of `platform` should be either `windows` or `unix`.

To get started you can examine and run `example.pl` which is located in the in the `$LOST/scripts/` directory.

1.3.1 File structure layout of the lost framework

The lost framework is divided into several parts:

- `$LOST/scripts/`: Contains scripts that run models. These scripts are just small Prolog programs. The scripts does not consult models directly, but instead use predicates that the framework provides.
- `$LOST/models/`: Contains all the biological sequence models. Each model is located in its own subdirectory of `$TLOST/models/` the name of which is also used as name of the model.
- `$LOST/lib/`: Contains shared libraries used multiple models.
- `$LOST/data/`: Contains sequences and data to be used by the models. Annotations etc. generated by model invoked by the framework stored in this directory and given the extension `.gen`
- `$LOST/tmp/`: Is used for temporary storage of files bt models.

Chapter 2

Creating lost models

2.1 Lost model conventions

Each model is located in its own subdirectory of the `$LOST/models/` directory, henceforth called `$MODELS`. So for instance, the sample model called `sample_model1` is located in `$MODELS/sample_model1/`. In the following, we will refer to the directory where a particular model resides as `$MODEL`.

Models are allowed to consult files with paths relative to the `$MODEL` directory, but should under normal circumstances only directly consult file which are located in the `$MODEL` directory or a subdirectory of it. The exception to this is the file `$LOST/lost.pl`. Consulting this file gives access to all the shared APIs (See chapter 3).

To integrate into the framework each model must provide a file called `interface.pl`, which must be located in the same directory as the model. In the file `interface.pl`, one or more interface predicates are defined. These predicates serve as entry points of using the model.

2.1.1 Model interface predicates

This section describes predicates, that when implemented in the `interface.pl` file provided by a model, allows the model provide functionalities that integrate into the general framework.

For common tasks these predicate are named according to the following convention:

- `annotate/3`: Do inference with the model and produce a "annotation" as output. The annotation take different forms depending on context.
- `train/3`: Training the model on the input data and save a switch file or similar as output.
- `generate/3`: Generate some data using the model.

It is possible to use the other predicate names than these, but it is *strongly* recommended to use these names whenever it makes sense.

The general form of these predicates is that they take a list of input files, a list of options and the name of an output file. They generally read the input files, do some processing and then write a result to the output file. The type of processing depends on the predicate. The predicate `annotate/3` is meant to be used when running the model in prediction mode, so for instance, it might produce a file containing the viterbi path for a given input sequence. The predicate `train/3` is meant to be used when training the model and the output file would usually be the learned parameters of the trained model. With the `generate/3` predicate, the model is supposed to be used in a generative fashion.

Models should declare the *type* associated to the arguments of these predicates. These declarations include

- The formats of input files, see section **??**.

- The format of the output file, see section ??.
- Options and default values for options, see section 2.1.2.
- Optionally, a set of possible values any declared option.

Interface predicates have the following form,

```
predicate_name(+InputFileNames,+Options,+OutputFilename)
```

The predicate takes the following arguments:

- `InputFileNames`: Is a list of filenames (with absolute paths), each containing an input to the model. There is no restriction on the format of the files, but the model should declare what file formats it expects. Predicates to parse a wide range of fileformats are supplied in the *io* API (see section 6.17).
- `Options`: Is a list of facts on the form, where the functor works as key and the first argument serves as the value of the option, egg. `key(Value)`. This list is used to parameterize the model in various ways. To check the value of a given option, the predicate `get_option` (see section 6.16) should be used. The options must be declared as described in section .
- `OutputFilename`: Full filename which the result should be saved to. The model is expected to save the resulting annotation to this file before the completion of `annotate`. The *io* API contains some common predicates for saving annotations (see section 6.17).

Depending the the `predicate_name` the predicates will do conceptually different things, but there is a common pattern: The model reads a processes the files given `InputFileNames`. It is then the responsibility of the predicate to save the annotation to `OutputFilename`. If the model for some reason is unable to write the annotation to `OutputFilename`, then it should `throw` an exception.

2.1.2 Declaration of options for interface predicates

Options to interface predicates should be declared in the `interface.pl` file. An option is declared by adding a fact

```
lost_option(+InterfaceGoal,+OptionName,+DefaultValue,+Description).
```

- `InterfaceGoal`: InterfaceGoal is the functor name of defined interface predicate.
- `OptionName`: An atom for the name of key. This is the functor identifying this particular option.
- `DefaultValue`: If the option is left unspecified then it will take this value.
- `Description`: A textual description of the purpose of the option. Should be in single quotes.

As example, consider an option declaration:

```
lost_option(annotate,debug,no,'Enable/disable debug prints').
```

In this declaration, we state that the goal `annotate` take the option `debug`. If this option is not a member of the `Options` list, then it will be added with the default value `no` before `annotate` is called by the framework.

For instance, assuming the model is called `my_model`, it might be called from a script with

```
get_annotation_file(my_model,['/some/path/infile'],[],FileOut),
```

which will start a new PRISM process and invoke

```
annotate(['/some/path/infile'],[debug(no)],FileOut),
```

in `$MODELS/my_model/interface.pl`. Note that the option `debug` has been inserted with the default value. A default value is only inserted if the option is left unspecified.

Declaring option values

If the values of an option can be enumerated, then the model should provide a declaration of by defining a predicate,

```
lost_option_values(+Predicate,+OptionName,-ValuesList).
```

In our example above, the option `debug` might have two possible values, `no` and `yes`, so a declaration might look like,

```
lost_option_values(annotate, debug, [yes,no]).
```

Option checking

When a model is invoked by the framework, options are checked before invoking the model. If a model is called with an option which has not been declared by the model, then a warning will be issued. Similarly, if a declared option for a predicate also has a `lost_option_values` declaration and the a predicate is called with the option set to value which is not part of the declaration, the a warning is issued.

2.1.3 Declaration of file formats for interface predicates

Input file formats

For each model interface predicate the format of input files and output files should be declared as facts in the `interface.pl` file for the model. To declare the format of the input files, a fact,

```
lost_input_format(PredicateName, [ Format1, .., FormatN ]).
```

where `PredicateName` is the name of the predicate to which the declaration belong. The second argument is a list of format identifiers as described in chapter **??**. For instance, we may have a declaration,

```
lost_input(annotate,
    [ text(prolog(sequence(dna))), text(prolog(sequence(amino_acids)))]).
```

The last entry of the list of formats may have the special star quantifier, which works similar to a kleene star: It states that the last entry is a placeholder for between zero or more entries of the specified type. For instance,

```
lost_input(annotate,
    [ text(prolog(sequence(dna))), star(text(prolog(sequence(dna))))]).
```

specifies that `annotate` takes one or more input files of the type `text(prolog(sequence(dna)))`.

Output file formats

Contrary to the input file formats, the output file format can depend on the options given to the model. That means, the model may produce files in different output formats depending on the values of the options it is called with.

The output format is thus declared with a predicate,

```
lost_output_format(PredicateName,Options, Format)
```

This predicate should specify the relation between a set of `Options` and a particular `Format` for the interface predicate `PredicateName`. `Format` should unify to one of the formats described in chapter **??**.

Checking of file formats

The framework checks that file formats are declared according to the scheme described above. If a model neglects to declare file formats or declares them wrongly, then a warning will be issued.

Chapter 3

Lost shared APIs

To use the lost APIs, the file `$LOST/lost.pl` must be consulted. To make the models independent of the absolute path of the `$LOST` directory, they should consult it with a path relative to the model path (e.g. `:- ['../../lost.pl'].`). Then, APIs, which are located in the `$LOST/lib` directory can be consulted using the goal,

```
lost_include_api(+APIName)
```

where `APIName` is the name of a Prolog file located in the `$LOST/lib/` directory except the `.pl` extension.

3.1 The interface API: `interface`

The API provides the interface to lost models following the conventions described in section 2.1.

Running a model is done using the goal `run_model`

```
run_model(Model,annotate(InputFiles,Options,OutputFile)
```

In this case we run the models `annotate` goal. This is used to retrieve the annotation generated by a specified `Model` invoked with the specified `InputFiles` and `Options`. `InputFiles` is a list of file names containing input for the model. `Options` is a list of facts on the form `key(Value)`, where `key` is the name of an option and `Value` is a concrete value. As a consequence of calling this predicate the output from the model is written to the file `OutputFile`. If the model has been previously invoked with exactly the same `InputFiles` and `Options` , then `OutputFile` will unify with the outcome of the previous invokation. In this way, the results of running a model are cached.

The generated annotation files are named according to a convention, where all annotation files will be placed in the `$LOST/data/` directory and the `Filename` is construed according to the following convention:

```
{Model}_{GoalFunctor}_{Id}.gen
```

The first time a model `OutputFile` is generated, the file `annotation.idx` will be created in `$LOST/data/`. This file serves as a database to map the filename of a generated annotation file to the model, input files and options that generated the particular annotation. This database file contains Prolog facts of the form

```
fileid(Id,Filename,Model,Goal,Options,InputFiles).
```

The annotation index is automatically maintained by `get_annotation_file/4` and should not be edited by hand.

If an annotation for a particular run of a model is not present then `get_annotation_file/4` will start a new PRISM process that invokes the `annotate` predicate provided be the model `interface.pl` file. By the contract of model conventions, the model will generate the annotation and save it to the file indicated by the provided filename.

3.1.1 Moving generated files around

Once a file has been generated, you may wish the move or rename the file. Doing this using normal system commands will fail to the update the annotation index and as result the file may be regenerated, even if it was already generated once. Instead, you should use

```
move_data_file(+OldFilename, +NewFilename)
```

This predicate makes sure that the annotation index is updated accordingly.

3.2 Input-Output API: `io`

In this module (`io.pl`), several predicates are defined to manipulate `*.seq` files :

- loading information from files that extracts from a file data information used as input of models (sequence annotation for example);
- saving information into a file;
- and maybe more.

3.2.1 Loading information from files

- `get_data_from_file(+File,+Options,-Data)`:
  This predicate aims at efficiency extract data (basically list of nucleotides) from a file with a format `text(prolog(sequence()))`. By default, we suppose that **File** is composed of `Term`:

  ```
  Term =.. [Functor,Left,Right,Data].
  ```

  Several options are defined to deals with the format constraint of `File`:

  - `data_position(Pos)` specified what is the position `Pos` in the term of data;
  - `left_position(Pos)` specified what is the position `Pos` in the term of the left value;
  - `right_position(Pos)` specified what is the position `Pos` in the term of the right value;
  - `left_position(none)` tells that no left position information is available in the term;
  - `right_position(none)` tells that no right position information is available in the term;

  By default, the predicate extracts and collects in `Data` all the data available in `File`. However, two options are available to ask for partial information

  - `range([Min,Max])` allows to extract data between `Min` and `Max`;
  - `ranges([[Min1,Max1],..,[Minn,Maxn]])` allows to extract a list of data defined the different ranges.

  Note: data extraction is made without taking care about strand specification. Then, data extracted from `ranges` option corresponds always to the data specified in `File`.

- `load_annotation_from_file(++Type_Info,++Options,++File,--Annotation)`:
  Generate from `File` a sequence of `Annotation`. It is assumed that `File` is composed of terms. `Type_Info` is used to specify what format of information into file

  - `sequence` means that information is stored into a list. For example,

    ```
    data(Key_Index,1,10,[a,t,c,c,c..]).
    ```

    Note: an improvement of the predicate (option sequence) is available via the call `get_data_from_file`.

  - `db` means that information is represented by a set of range that specified specific zone (coding region for example)

    `gb(Key_Index,1,10).`

For each `Type_Info`, several options are available represented by the list `Options`. Options available for `sequence`:

  - [] (default): data list is the 2^{th} argument of the terms and these lists of data are appended;
  - data_position(Num) specified that data list is $\texttt{Num}^{th}$ argument of term;
  - `range(Min,Max)` extracts from the list of the complete annotation the sublist from position `Min` to position `Max`;
  - `all_lists`: generate a list of each data list by term. Warning: `range(Min,Max)` is not support by this option.

File Example `toto.seq`:

```
data(Key_Index,1,5,[1,2,3,4,5]).
data(Key_Index,6,10,[6,7,8,9,10]).
data(Key_Index,11,15,[11,12,13,14,15]).
```

Results of request are:

```
| ?- load_annotation_from_file(sequence,[data_position(4)],'toto.seq',R).
R = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15] ?
| ?- load_annotation_from_file(sequence,[data_position(4),range(4,10)],'toto.seq',R).
R = [4,5,6,7,8,9,10] ?
load_annotation_from_file(sequence,[data_position(4),all_lists],'toto.seq',R).
R = [[1,2,3,4,5],[6,7,8,9,10],[11,12,13,14,15]] ?
```

Options available for `db`:

  - [] (default): first and the second element of the term defined a range. A list of 0-1 values is generated, 0 when you are outside ranges and 1 you are inside at least one;
  - `in_db(Letter)` replaces the default value 1 by `Letter`;
  - `out_db(Letter)` replaces the default value 0 by `Letter`;
  - `range_position(Min,Max)` allows to specify the position argument number of a term of the minimal and maximal value of term;
  - `range(Min,Max)` extracts from the list of the complete annotation the sublist from position `Min` to position `Max`;

File Example `toto.seq`:

```
gb(3,5).
gb(7,9).
gb(8,11). % Overlap ;)
```

Results of request are:

```
| ?- load_annotation_from_file(db,[],'toto.seq',R).
R = [0,0,1,1,1,0,1,1,1,1,1] ?
| ?- load_annotation_from_file(db,[in_db(c),out_db(nc)],'toto.seq',R).
R =  [nc,nc,c,c,c,nc,c,c,c,c,c]?
```

```
| ?- load_annotation_from_file(db,[range(3,7)],R).
R = [1,1,1,0,1] ?
| ?- load_annotation_from_file(db,[in_db(c),out_db(nc),range(8,16)],'toto.seq',R).
R =  [c,c,c,nc,nc,nc,nc,nc]?
```

- `get_sequence_from_file(++File,++Options,--Sequence)`: Generate from `File` of data a sequence (read: a list) of data `Sequence`. This predicate is a more efficient implementation of `load_annotation_from_file\4` with the type option `sequence` when several ranges of data have to be computed (option `ranges`)

  We assumed that data facts of `File` are composed of facts with the format:

  ```
  Term =.. [_Data_Functor,_Key,Range_Min,Range_Max,Data]
  ```

  File Example `toto.seq`:

  ```
  data(toto,1,10,[1,2,3,4,5,6,7,8,9,10]).
  data(toto,11,20,[11,12,13,14,15,16,17,18,19,20]).
  data(toto,21,30,[21,22,23,24,25,26,27,28,29,30]).
  ```

  Result of the request is:

  ```
  | ?- get_data_from_file('toto.seq',[],R).
  R = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,
                            21,22,23,24,25,26,27,28,29,30] ?
  ```

  Different options are available to specify the format of the data term:

  – `data_position(Pos)` to specify an other position for the data;

  – `left_position(Left)` to specify an other position for the left bound of the range that specifies the position of the data into the data file. When this left bound is not available, the keyword `none` for `Pos` can be used;

  – `right_position(Right)` to specify an other position for the right bound of the range that specifies the position of the data into the data file. When this right bound is not available, the keyword `none` for `Pos` can be used.

  File Example `toto2.seq`:

  ```
  data(1,10,[1,2,3,4,5,6,7,8,9,10]).
  data(11,20,[11,12,13,14,15,16,17,18,19,20]).
  data(21,30,[21,22,23,24,25,26,27,28,29,30]).
  ```

  Result of the request is:

  ```
  | ?- get_data_from_file('toto2.seq',
                          [left_position(1),
                           right_position(2),
                           data_position(3)],R).
  R = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,
                            21,22,23,24,25,26,27,28,29,30] ?
  ```

  To ask for specific ranges of data, two options are available:

  – `range(Min,Max)` computes the sequence of data between `Min` and `Max`;

  - `ranges(List_Ranges)` computes sequences of data for a list of ranges with a format `[[Min1,Max1],[Min2,Max2],...]`. Note that the ranges should be ordered, i.e `Min1` $\leq$ `Min2` must be satisfied.

Result of the request is:

```
| ?- get_data_from_file('toto.seq',[range(5,10)],R).
R = [5,6,7,8,9,10] ?
| ?- get_data_from_file('toto.seq',[ranges([5,10],[5,8],[9,15])],R).
R = [[5,6,7,8,9,10],[5,6,7,8],[9,10,11,12,13,14,15]] ?
```

3.2.2 Saving information to files

The io API also contains some predicates for saving sequences to a file.

- `save_annotation_to_sequence_file(+KeyIndex,+ChunkSize,+Annotation,+File)`: Saves the data in the list given by `Annotation` to the file `File` in *sequence* format which is number of Prolog facts, on the form:

  ```
  data(KeyIndex,1,5, [a,t,c,c,g]).
  ```

  The first argument `KeyIndex` is used as an identifier. `ChunkSize` is the number of elements from `Annotation` to store with each fact. The second and third argument of a stored fact, corresponds to the start and end position in `Annotation`. The fourth argument of the fact is a list containing the relevant elements of the `Annotation` list. Note that if the the length of `Annotation` is not a multiple of `ChunkSize`, then the last fact stored will have a shorter range.

3.2.3 Memory mapping of sequence files

Some times it can be convenient load a sequence file into memory and then be able to extract regions of the sequence on demand. The `io` API contains a method for doing just this.

The predicate to map a file in the format `text(prolog(sequence(_)))` into memory is

```
load_sequence(+SequenceID, +Filename)
```

This will load the sequence from the file given by `Filename` and associate with the identifier `SequenceID`.

```
get_sequence_range(+SequenceID, +Min, +Max, -Data)
```

The predicate `get_sequence_range/4` can then be used to extract a part of the sequence. This is fairly efficient, but depends on the size of the data terms, $|t|$, in the file. The running time of the predicate is bounded by $O(2|t| \times n)$, where $n = \texttt{Max} - \texttt{Min}$.

3.2.4 Splitting a Prolog file into multiple files

When working with large data files, it can sometimes be practical to split a large file with Prolog facts into multiple files, each having a distinct subset of the facts in the original file. The `io` API contains the predicate `split_file/2` for doing this:

```
split_file(+InputFile,+FactsPerFile,
           +OutputFilePrefix, +OutputFileSuffix)
```

The n facts in the `InputFile` will be divided into $m = n/\texttt{FactsPerFile}$ output files.

In the same idea of split_file/*slash* 2, `split_file_fasta`/*slash*5 divide a file in the FASTA format into smallest files.

```
split_file_fasta(+InputFile,+FastaChunkPerFile,
           +OutputFilePrefix, +OutputFileSuffix,-ResultFiles)
```

The division is made given the number times that a comment line (line that starts with ¿) is met. For example, given the file

```
> First chunk
ABCDEF
> Second chunk
GHIJKL
> Third chunk
....
```

If `FastaChunkPerFile` is equals to 2, two files will be generated:

```
> First chunk
ABCDEF
> Second chunk
GHIJKL
```

and

```
> Third chunk
....
```

3.3 The Stats API: `stats`

In this module (`stats.pl`), several predicates are defined to automatically compute frequencies, probabilities of occurrences of nucleotides, codons ... This computation is based on a simple counting method given a simple input data.

Variable `Data_Type` is used to specify the type of the input data: Data type available are:

- `nucleotide` where input data are composed of letter from $\{a, c, g, t\}$;
- `codon` where a codon is represented by a list of three letters from $\{a, c, g, t\}$;
- `amino_acid` where input data are composed of letter from $\{a, c, d, e, f, g, h, i, k, l, m, n, p, q, r, s, t, v, w, y\}$;
- `length`. This type is used to compute frequencies and statistics about length of specific region of a genome.

Here is the description of main predicates of `stats.pl`

- stats(++Data_Type,++Options,++Data,++Input_Counting,–Result)
  `Data_Type` setting define on which counting procedure is initialized. User must take care then to give the right `Data`. For {`nucleotide,codon,amino_acid`} data type, `Data` must be a Prolog list with the right element, For `length` data type, `Data` must be a list of ranges represented by a list of two integer values ([Min,Max]). Two options are available:
  - `order(Num)` defines the size of the past stored to perform the counting
  - `past(List)` allows to give as input of the computation a previous past. This option is useful to make connection between two counting computations.

  `Input_Counting` is used as well to perform a counting computation on an input data divided into a set of lists. By default, counting computation is initialized when is a variable `Input_Counting`. However, the counting can restart from `Input_Counting` if this variable is unified to the result of a previous result. Finally, `Result` has the following format:

```
[(Past1,[(Elt1,Count11),(Elt2,Count12),...]),
 (Past2,[(Elt1,Count21),(Elt2,Count22),...]),
 ...]
```

For example let consider this following data:

```
data('U00096',1,5,[a,g,c,t,t]).
data('U00096',6,10,[c,c,c,c,c]).
```

Here is the result of this following request

```
| ?- stats(nucleotide,[],[a,g,c,t,t],_,R).
R = [([],[(a,1),(c,1),(g,1),(t,2)])] ?
| ?- stats(nucleotide,[order(1)],[a,g,c,t,t],_,R).
R = [([a],[(a,0),(c,0),(g,1),(t,0)]),
     ([c],[(a,0),(c,0),(g,0),(t,1)]),
     ([g],[(a,0),(c,1),(g,0),(t,0)]),
     ([t],[(a,0),(c,0),(g,0),(t,1)])]?
| ?- stats(nucleotide,[order(2)],[a,g,c,t,t],_,R1),
     stats(nucleotide,[order(2),past([t,t])],[c,c,c,c,c],R1,R).
R = [([a,a],[(a,0),(c,0),(g,0),(t,0)]),
     ([a,c],[(a,0),(c,0),(g,0),(t,1)]),
     ([a,g],[(a,0),(c,1),(g,0),(t,0)]),
     ([a,t],[(a,0),(c,0),(g,0),(t,0)]),
     ([c,a],[(a,0),(c,0),(g,0),(t,0)]),
     ([c,c],[(a,0),(c,3),(g,0),(t,1)]),
     ([c,g],[(a,0),(c,1),(g,0),(t,0)]),
     ([c,t],[(a,0),(c,0),(g,0),(t,1)]),
     ([g,a],[(a,0),(c,0),(g,0),(t,0)]),
     ([g,c],[(a,0),(c,0),(g,0),(t,1)]),
     ([g,g],[(a,0),(c,1),(g,0),(t,0)]),
     ([g,t],[(a,0),(c,0),(g,0),(t,1)]),
     ([t,a],[(a,0),(c,0),(g,0),(t,0)]),
     ([t,c],[(a,0),(c,0),(g,0),(t,1)]),
     ([t,g],[(a,0),(c,0),(g,0),(t,0)]),
     ([t,t],[(a,0),(c,1),(g,0),(t,0)]),
     ]?
```

- stats(++Data_Type,++Options,++Data,++Input_Counting,–Past,–Result) `stats/6` computes exactly the same counting. This predicate allows only to record into `Past` the last past of the computing. This predicate is usefull to compute statistics on a series of data input. For example, this following request

  ```
  | ? - stats(nucleotide,[order(2)],[a,g,c,t,t],_,Past,R1),
        stats(nucleotide,[order(2),past(Past)],[c,c,c,c,c],R1,Past2,R).
  Past = [t,t],
  Past2 = [c,c],
  R = ... ?
  ```

  can be used to obtain counting information on the previous data.

- normalize(++Data_Type,++List_Counting,–Probabilities) From the result of a counting procedure, `normalize/3` modified `List_Counting` to compute domains and probabilities distributions.

For example, here is the result of several requests:

```
| ?- stats(nucleotide,[],[a,g,c,t,t],_,R),
     normalize(nucleotide,R,S).
S = [([],([a,c,g,t],[0.2,0.2,0.2,0.4]))]
| ?- stats(nucleotide,[order(1)],[a,g,c,t,t],_,R),
     normalize(nucleotide,R,S).
S = [([a],([a,c,g,t],[0,0,1,0])),
     ([c],([a,c,g,t],[0,0,0,1])),
     ([g],([a,c,g,t],[0,1,0,0])),
     ([t],([a,c,g,t],[0,0,0,1]))] ?
```

To get a representation of the statistics where each count is represented as a single fact, the predicate

```
build_stat_facts(+Stats,-StatsFacts)
```

can be useful. For instance,

```
| ?- stats(nucleotide,[],[a,g,c,t,t],_,R), build_stat_facts(R,F).
R = [([],[(a,1),(c,1),(g,1),(t,2)])]
F = [stat([a],1),stat([c],1),stat([g],1),stat([t],2)] ?
```

3.4 The `regex` API

The `regex` API provides POSIX-like regular expressions for matching Prolog strings and atoms. A regular expression is created using

```
re_compile(+Regex, -CompiledRegex)
```

The input argument `Regex` is an atom or a list of symbols representing a particular regular expression. The output argument `CompiledRegex` is used in subsequent matching with this regular expression.

Matching with a regular expression is then done using

```
re_match(+CompiledRegex,+String,-Matches)
```

The goal `re_match/3` is true whenever the `CompiledRegex` matches the `String`. The `String` may be an atom or a list of symbols. If the goal is true, then `Matches` is a list containing the matching parts of the `String` corresponding to particular match groups (paranthesized sub-expressions) of the regular expression.

Currently the regular expressions can be expressed using a significant subset of the well-known POSIX regular expression operators. The following operators are supported:

- *Concatenation:* is expressed by the concatenation of two sub expressions. For instance, the regular expression `ab` matches an `a` followed by `ab`.
- *Alternation:* is expressed with the vertical bar (`|`). For instance the expression `a|b` matches either `a` or `b`.
- *Repetition:* Repetition operators indicate the number of times a preceding regular expression may be matched. The following repetition operators are supported:
  - `*` representing the Kleene star, meaning that the preceding expression is matched zero or more times.
  - `+` the preceding expression one or more times.
  - `?` The preceding expression zero times or one time.

Note that repetition binds stronger than alternation and alternation binds stronger than concatenation. To enforce a different binding order, enclosing sub-expressions in parentheses (which binds strongest) are used.

Parenthesised sub-expressions can also be used to create *match groups*. An outer-most paranthesized sub-expression, where the paranthesis are superflouos is interpreted as a match group. When an expression is matched, then the part of the string matched by the paranthesized expression is extracted.

Additionally, ranges are supported by the regular expression syntax. A range is expressed using square brackets. For instance, `[a-z0-9]` is range expression which matches a lower case alphanumeric character or a digit.

3.5 The genecode API

This API contains a set of genetic code tables. Each table defines a relation between codons and amino acids. The main predicate of the API is

```
genecode(+Code, ?Codon, ?AminoAcid)
```

The argument code is an integer specifying the genetic code to use. Currently, only genetic code `11` (bacterial) is supported. `Codon` is a list of of three nucleotides symbols from the alphabet `[a,g,c,t]`. The argument `AminoAcid` is a symbol from the alphabet

```
[a,c,d,e,f,g,h,i,k,l,m,n,p,q,r,s,t,v,w,y,*]
```

Where all symbols except `*` represents an amino acid (the name of which starts which that letter). The special symbol `*` signifies a stop codon in the specified genetic code.

Other predicates in the genecode API include

```
genecode_start_codons(+GeneCode,-StartCodons)
```

The `genecode_start_codons/2` predicate unifies `StartCodons` with a list of all valid start codons for the given `GeneCode`.

```
genecode_stop_codons(+GeneCode,-StopCodons)
```

The `genecode_stop_codons/2` predicate unifies `StopCodons` with a list of all valid stop codons for the given `GeneCode`.

3.6 The sequence API: sequence

The `sequence` API is library containing predicates for doing common tasks with biological sequences.

3.6.1 Complementing DNA sequences

A complementary dna sequence can be achieved using the predicate

```
dna_seq_complement(+NucleotideSeq,-ComplNucleotideSeq)
```

When given the input sequence `NucleotideSeq` is a list of symbols from the alphabet `[a,g,c,t]`, then `ComplNucleotideSeq` is unified to a list of complementary symbols, e.g.

```
| ?- dna_seq_complement([a,g,a,c,t,a],X).
X = [t,c,t,g,a,t] ?
yes
```

Typically, when complementing a DNA sequence, we are actually interested in the *reverse* complement. To achieve this, use the built-in predicate `reverse/2` before `dna_seq_complement/2`.

3.6.2 Translating a DNA sequence to amino acids

The predicate,

```
dna_amino_acid(+Genecode, ?DNA_Sequence,?AminoAcids)
```

Translates a DNA sequence to a sequence of amino acids or vice versa; produces all possible DNA sequences for a given amino acids sequence. `Genecode` is an integer for the genetic code to be used in the translation. `DNA_Sequence` is a list of symbols from the alphabet `[a,g,c,t]` and length of the list is expected to be divisible by three. `AminoAcids` is a list of symbols from the alphabet,

```
[a,c,d,e,f,g,h,i,k,l,m,n,p,q,r,s,t,v,w,y,*].
```

Chapter 4

Models

4.1 Documentation for individual models

This section contains the auto-generated documentation for individual models.

4.1.1 accuracy_report

author : Christian Theil Have

Introduction This model generates a Prolog based report of accuracy measures for a set of gene predictions compared to a golden standard annotation.

A lot of different measures are generated. An example of a report is shown below.

```
accuracy_report(genes_predicted, 2130).
accuracy_report(genes_actual, 4144).
accuracy_report(genes_correct, 1882).
accuracy_report(genes_wrong, 248).
accuracy_report(gene_stops_correct, 2003).
accuracy_report(gene_stops_wrong, 127).
accuracy_report(gene_sensitivity, 0.454150579150579).
accuracy_report(gene_specificity, 0.883568075117371).
accuracy_report(gene_specificity, 0.883568075117371).
accuracy_report(gene_stop_sensitivity, 0.483349420849421).
accuracy_report(gene_stop_specificity, 0.94037558685446).
accuracy_report(genes_wrong_ratio, 0.116431924882629).
accuracy_report(genes_missing_ratio, 0.545849420849421).
accuracy_report(nucleotide_true_positives, 1934811).
accuracy_report(nucleotide_false_positives, 75521).
accuracy_report(nucleotide_true_negatives, 619038).
accuracy_report(nucleotide_false_negatives, 2010092).
accuracy_report(nucleotide_sensitivity, 0.757605872547672).
accuracy_report(nucleotide_specificity, 0.962433568186747).
accuracy_report(nucleotide_specificity_traditional, 0.891267696480789).
accuracy_report(nucleotide_correlation_coefficient, 0.27484302373759).
accuracy_report(nucleotide_simple_matching_coefficient, 0.55046231653584).
accuracy_report(nucleotide_average_conditional_probability, 0.644903316252825).
accuracy_report(nucleotide_approximate_correlation, 0.28980663250565).
```

report(*+InputFiles, +Options, +OutputFile*)
Type signature:

```
InputFiles:
    1. text(prolog(ranges(gene)))
    2. text(prolog(ranges(gene)))
OutputFile:
    text(prolog(accuracy_report))
Options:
    - start (default value: min)
    - end (default value: max)
    - reports (default value: all)
```

Creates a accuracy report *OutputFile* based on *InputFiles*. The first input file is a 'Golden Standard' file which contains reference genes. The second input file is the predictions.

A subset of the predictions may be specified with range in specified using the options `start` and `end`. Only predictions which fully lie within this range are considered and reference predictions outside of this range are also not considered. They may both be integers or the special values `min` and `max` respectively (which indicates that all predications should be considered).

4.1.2 best_prediction_per_stop_codon

author : Christian Theil Have

This is a simple model for excluding several overlapping predictions with the same stop-codon. Instead, given a set of predictions with scores (as extra field) the model selects the best scoring gene prediction for each stop codon. I.e. the final set of predictions will only contain one prediction per stop codon.

The functor of the extra field that contains the score of the prediction, needs to be specified using an option.

filter(*+InputFiles, +Options, +OutputFile*)
Type signature:

```
InputFiles:
    1. text(prolog(ranges(gene)))
OutputFile:
    text(prolog(ranges(gene)))
Options:
    - prediction_functor (default value: auto)
    - score_functor (default value: not_set)
```

For each distinct stop codon in the predictions in the the input file, write prediction which has has the highest score to the output file. The functor of predictions in the input file will be automatically inferred if option `prediction_functor=auto` and the file contains only the same type of functor. Otherwise the value of `prediction_functor` should be set. The `score_functor` option *must* be set. The predictions are expected to have an extra field indicating the score of a prediction as numeric value, .e.g. in the example below, the you would use `score_functor(score)`.

```
prediction(org,123,456,+,1,[score(0.99)])
```

4.1.3 blastgf

author
- : Christian Theil Have
- : Ole Torp Lassen

Blast based gene finder.
This gene finder predicts genes based on blast hits.
This is based on Oles (now defunct it seeems) chunk_aa_conservtion model, but considerably rewritten by Christian Theil Have. The `blastgf` model only contains the probabilistic HMM and prism model related to prediction of genes based on blast matches and more mundane tasks like preprocessing, blast xml parsing, running of blast have been omitted from this model.
FIXME: More documentation needs to be done.

annotate_single_track(*+InputFiles, +Options, +OutputFile*)
Type signature:

```
InputFiles:
    1. text(prolog(prism_parameters))
    2. text(prolog(ranges(gene)))
OutputFile:
    text(prolog(ranges(gene)))
Options:
```

InputFiles = [ParamsFile, OrfsFile] This task creates annotations for the ORFs of OrfsFile. The Orfs in OrfsFile are expected to have an extra field, `identity_seq`, a list of 0 and 1's in which ones identity positions with a blast hit. Such an extra field can be added with the `orf_blaster` model. If part of and ORF in OrfsFile is predicted as coding, then the ORF will be written to *OutputFile*. The term written to output file will have an additional extra field `blastgf(AnnotList)` where `AnnotList` is a list of 0 and 1, where one 0 means predicted as non-coding and 1 means predicted as coding.

annotate_multi_track(*+InputFiles, +Options, +OutputFile*)
Type signature:

```
InputFiles:
    1. text(prolog(prism_parameters))
    2. text(prolog(ranges(gene)))
OutputFile:
    text(prolog(ranges(gene)))
Options:
```

InputFiles = [ParamsFile, OrfsFile] This task creates annotations for the ORFs of OrfsFile. The Orfs in OrfsFile are expected to have an extra fields, `identity_seq`, a list of integers 0-8, there the number indicates the number identity positions with a blast hits to one of eight other organisms. In the future, the number of organisms may be made configurable. Such an extra field can be added with the `orf_blaster` model. If part of and ORF in OrfsFile is predicted as coding, then the ORF will be written to *OutputFile*. The term written to output file will have an additional extra field `blastgf(AnnotList)` where `AnnotList` is a list of 0 and 1, where one 0 means predicted as non-coding and 1 means predicted as coding.

parallel_annotate_multi_track(*+InputFiles, +Options, +OutputFile*)
Type signature:

```
InputFiles:
    1. text(prolog(prism_parameters))
    2. text(prolog(ranges(gene)))
OutputFile:
    text(prolog(ranges(gene)))
Options:
```

Same as `annotate_multi_track` but running with 10 parallel threads. This task is deprecated.

parallel_annotate_single_track(_+InputFiles, +Options, +OutputFile_**)**
Type signature:

```
InputFiles:
    1. text(prolog(prism_parameters))
    2. text(prolog(ranges(gene)))
OutputFile:
    text(prolog(ranges(gene)))
Options:
```

Same as `annotate_single_track` but running with 10 parallel threads. This task is deprecated.

learn_single_track(_+InputFiles, +Options, +OutputFile_**)**
Type signature:

```
InputFiles:
    1. text(prolog(ranges(gene)))
OutputFile:
    text(prolog(prism_parameters))
Options:
```

The Orfs in the input file are expected to have an extra field, `identity_seq`, a list of 0 and 1's in which ones identity positions with a blast hit. Also, the Orfs in the input file are expected to have an extra field, `ref_annot`, a list of 0 and 1's in which ones indicate a that (part of) the orf is real gene. The result of running the task is PRISM parameter file.

learn_multi_track(_+InputFiles, +Options, +OutputFile_**)**
Type signature:

```
InputFiles:
    1. text(prolog(ranges(gene)))
OutputFile:
    text(prolog(prism_parameters))
Options:
```

The Orfs in the input file are expected to have an extra field, `identity_seq`, a list of integers 0-8, there the number indicates the number identity positions with a blast hits to one of eight other organisms. Also, the Orfs in the input file are expected to have an extra field, `ref_annot`, a list of 0 and 1's in which ones indicate a that (part of) the orf is real gene. The result of running the task is PRISM parameter file.

parallel_learn_single_track(*+InputFiles, +Options, +OutputFile*)
Type signature:

```
InputFiles:
    1. text(prolog(ranges(gene)))
OutputFile:
    text(prolog(prism_parameters))
Options:
```

Same as `learn_single_track`, but running with 10 parallel threads.

parallel_learn_multi_track(*+InputFiles, +Options, +OutputFile*)
Type signature:

```
InputFiles:
    1. text(prolog(ranges(gene)))
OutputFile:
    text(prolog(prism_parameters))
Options:
```

Same as `learn_multi_track`, but running with 10 parallel threads.

4.1.4 chunk_aa_conservation.pl

4.1.5 chunk_ref_annot

author : Christian Theil Have

Augments predicted genes or orfs with information from a golden standard/reference file.

Currently, there are two principal ways of using this module. For both methods, it is the case that their InputFiles is a list

```
InputFiles = [ ReferenceFile, PutativeFile ]
```

The two methods can be summarized as follows:

1. `add_reference_track` : adding an ”extra” field with a reference annotation to a each ”gene”/orf in a file. E.g. if we have a `PutativeFile` with an entry,

   ```
   gene(seqid, 1, 10, +, 1, []).
   ```

   And a golden standard `ReferenceFile` with the entry,

   ```
   ref(seqid,4,10,+,1,[]).
   ```

   Then the updated gene record (OutputFile) will look as follows,

   ```
   gene(seqid,1,10,+,1,[ref_annot([0,0,0,0,1,1,1,1,1,1])]).
   ```

   A term with a list is added to the `extra` list (sixth argument). The list has zeroes in all non-coding positions and ones in the coding positions (of the same strand reading frame).

2. `maching_genes` : Write to the OutputFile all genes from the `ReferenceFile` which partially overlap with a gene from the `PutativeFile` (must have same Strand and Frame). This may be useful for instance when creating cross-validation sets, e.g. if you want to measure accuracy wrt to a smaller set.

add_reference_track(*+InputFiles, +Options, +OutputFile***)**
Type signature:

```
InputFiles:
    1. text(prolog(ranges(gene)))
    2. text(prolog(ranges(gene)))
OutputFile:
    text(prolog(ranges(gene)))
Options:
```

```
InputFiles = [ PutativeFile, ReferenceFile ]
```

adds and extra field to each gene in `PutativeFile` if the are a (partically) overlapping gene in `ReferenceFile` with same strand+frame. The extra field contains a list, which has zeroes in all non-coding positions and ones in the coding positions (of the same strand reading frame).

matching_genes(*+InputFiles, +Options, +OutputFile***)**
Type signature:

```
InputFiles:
    1. text(prolog(ranges(gene)))
    2. text(prolog(ranges(gene)))
OutputFile:
    text(prolog(ranges(gene)))
Options:
```

InputFiles = [File1, File2] *OutputFile* is all the gene entries in File1 which have identical the Left, Right, Frame and Strand to an of entry in File2.

4.1.6 chunk_translator

author : Ole Torp Lassen

Translates a file of frame specific nucleotide chunks into a file of equivalent fastaformatted amino-acid sequences.

translate(*+InputFiles, +Options, +OutputFile***)**
Type signature:

```
InputFiles:
    1. text(prolog(ranges(gene)))
OutputFile:
    text(fasta(ffa))
Options:
    - mode (default value: 0)
    - genecode (default value: 11)
```

```
InputFiles = [ ChunkFile ]
```

Translates each chunk in ChunkFile to get the amino acid sequence. The option `mode` determines whether to run translate entire chunk `mode=0` or the longest orf in chunk `mode=1`. The translated chunks are written *OutputFile* which in a multi-fasta format.

4.1.7 cluster_gene_finder.pl

4.1.8 codon_preference

author
- : Ole Torp Lassen
- : Christian Theil Have
- : Matthieu Petit

This is a simple gene finder based on codon preference. It is a two-state Hidden Markov Model that emit the codons of a sequence from either a coding state or a non-coding state. Once a transition to a coding state has occurred, the model stays in the coding state until the end state, as can be observed from the following transitions.

```
values(trans('c'),['c','end']).
values(trans('n'),['c','n','end']).
```

The following assumes the following about input sequences: (1) The sequence length is a multiple of three (since codons are emitted) (2) The ”codingness” of the sequence follows the regular expression pattern: ”n*c*”

The gene finder was written by Ole, based on an earlier gene finder by Matthieu. Part of ”the infrastructure” later rewritten by Christian.

annotate(*+InputFiles, +Options, +OutputFile***)**
Type signature:

```
InputFiles:
    1. text(prolog(prism_parameters))
    2. text(prolog(ranges(gene)))
OutputFile:
    text(prolog(ranges(gene)))
Options:
```

```
InputFiles = [ ParamsFile, InputFile ]
```

Model is first parameterized using ParamsFile. Eaah putative gene/orf/chunk (range facts) in InputFile is annotated using the model. The gene range facts are expected to have a `sequence` extra field, which holds a list with nucleotide sequence spanned by the range. If a input orf is annotated entirely as non-coding, then it will *not* be written to the output file. Otherwise, the orf will be written to the output file with and additional extra field `codon_pref(AnnotList)`. AnnotList is a list of zeroes and ones where 0 means annotated as non-coding and 1 annotated as coding.

parallel_annotate(*+InputFiles, +Options, +OutputFile***)**
Type signature:

```
InputFiles:
    1. text(prolog(prism_parameters))
    2. text(prolog(ranges(gene)))
OutputFile:
    text(prolog(ranges(gene)))
Options:
```

Same as `annotate/3` but runs with 10 threads in parallel.

parallel_learn(*+InputFiles, +Options, +OutputFile***)**
Type signature:

```
InputFiles:
    1. text(prolog(ranges(gene)))
OutputFile:
    text(prolog(prism_parameters))
Options:
```

```
InputFiles = [ TrainingDataFile ]
```

Train the gene finder using the genes/orfs/chunks in TrainingDataFile. TrainingDataFile is expected to contain two `extra` fields: `sequence` contains the nucleotide sequence as a list and a `ref_annot` which contains a list of zeros (non-coding) and ones (coding) according to a golden standard.

4.1.9 cons_and_codon.pl

4.1.10 cons_chunk_genefinder.pl

4.1.11 consorf_genefinder.pl

4.1.12 cp_cn_voting.pl

4.1.13 dicodon_hmm.pl

4.1.14 easygene

author
- : Ole Torp Lassen
- : Matthieu Petit

This is a model for parsing the reports of the Easygene gene finder.

parse(*+InputFiles, +Options, +OutputFile***)**
Type signature:

```
InputFiles:
    1. text(easygene_report)
OutputFile:
    [test(prolog(ranges(gene)))]
Options:
```

Parse a report in the easygene format.

4.1.15 ecoparse_adph.pl

4.1.16 file.pl

4.1.17 format_converter.pl

4.1.18 gene_filter.pl

4.1.19 genebank_annotator.pl

4.1.20 genemark.pl

4.1.21 genemark_genefinder.pl

4.1.22 genemark_hmm.pl

4.1.23 genome_filter.pl

4.1.24 genome_finder.pl

4.1.25 glimmer3.pl

4.1.26 hard_to_find_genes.pl

4.1.27 hein_genefinder.pl

4.1.28 http.pl

4.1.29 iid.pl

4.1.30 length_genefinder.pl

4.1.31 logodds.pl

4.1.32 longest_prediction_per_stop_codon.pl

4.1.33 merge_chunk_matches.pl

4.1.34 merge_files

author : Christian Theil Have

Merging of files in the `text(prolog(_))` format.

merge(*+InputFiles, +Options, +OutputFile*)
Type signature:

```
InputFiles:
    1. text(prolog(_12e60)):n
OutputFile:
    [text(prolog(_12e98))]
Options:
```

Merges all *InputFiles* in the *OutputFile*.

4.1.35 mork_hmm.pl

4.1.36 orf_annotator.pl

4.1.37 orf_blaster.pl

4.1.38 orf_chopper

author : Ole Torp Lassen

Orf chopper is tool to extract open reading frames.

chop(*+InputFiles, +Options, +OutputFile*)
Type signature:

```
InputFiles:
    1. text(prolog(sequence(dna)))
OutputFile:
    text(prolog(ranges(gene)))
Options:
    - strand (default value: +)
    - frame (default value: 1)
```

Extract all open reading frames on the specified `strand` {'+','-'} in the specified reading `frame` {1,2,3}.

4.1.39 orf_finder.pl

4.1.40 orf_length.pl

4.1.41 parser_blastxml.pl

4.1.42 parser_fna.pl

4.1.43 parser_genemark.pl

4.1.44 partest.pl

4.1.45 process_blast_matches.pl

4.1.46 prodigal.pl

4.1.47 ptt

author : Ole Torp Lassen

PTT file parser

parse(*+InputFiles, +Options, +OutputFile*)
Type signature:

```
InputFiles:
    1. text(ptt)
OutputFile:
    text(prolog(ranges(gene)))
```

```
Options:
    - genome_key (default value: n/a)
```

```
InputFiles = [ PTTFile ]
```

Parses a PTT file creates a gene fact for each line in ptt file

4.1.48 range_stats.pl

4.1.49 rbs_model.pl

4.1.50 sample_model1.pl

4.1.51 sample_model2.pl

4.1.52 select_best_scoring.pl

4.1.53 stop_trimmer.pl

4.1.54 tabstats.pl

4.1.55 todo_model.pl

Pre-defined models are introduced. These models are used to build more complex models.

4.2 Parsers of Biological Data

To extract information from different Biological database, several parsers have been designed to parse report of analyses (Blast, Easygene and Genemark) and database (Genbank). These parsers generated a series of Prolog terms that can be used after that input of different probabilistic models. `script_parser.pl` collects different scripts to generate different data files.

4.2.1 Parser_fna

*.fna file of Genbank is composed of a complete genome in the FASTA format. Parser_fna permits to parse this *fna.file from Genbank and generate list of terms. These terms store the genome into a Prolog list composed of $\{a, c, g, t\}$. Two scripts are implemented:
`parser_fna(++Name_FNA_File,++Name_GBK_File,++Options)` and
`parser_fna(++Name_FNA_File,++Name_GBK_File,++Options,--OutputFile)`.
Note that *.gbk file is necessary as well. This file is used to automatically extract genome information (Genbank key and size of genome).

Output format of the generated terms are:
`data(Genebank_Key,Start,End,List_of_Data)`.

Two options are available:

- `list(Number)` can be used to divide the complete genome into several lists with a length defined by the parameter `Number`. Example with the E.Coli K12 genome:

```
>gi|48994873|gb|U00096.2| Escherichia coli .....
AGCTTTTCATTCTGACTGCAACGGGCAATATGTCTCTGTGTGG.....
.....
```

  Result by default:

```
%>gi|48994873|gb|U00096.2| Escherichia coli ....
data('U00096',1,4639675,[a,g,c,t,t,t, ...]).
```

Result when `list(280)` option is used

```
%>gi|48994873|gb|U00096.2| Escherichia .....
data('U00096',1,280,[a,g,c,t,...]).
data('U00096',281,560,[c,c,c,...]).
....
```

- `range([Min,Max])` allows to ask for the part of the genome between `Min` and `Max`.

4.2.2 Parser_ptt

A PTT file is the NCBI format for representation of a *protein table*. Typically, ptt files are composed of information about known or predicted protein coding genes in a genome. Each such gene is represented by a single line where tabulation separated fields represent various properties of the gene such as the position, length, name and protein product. The `parser_ptt` model parses PTT files and generates a Prolog term in the output file for each gene in the input (ptt) file. We use the format identifier, `text(ptt)`, for PTT files, which is also the input format of `parser_ptt`. The output format is `text(prolog(ranges(gene)))`.

As this information is not available in the PTT file, option `genome_key` should be used to give the Genbank key of the genome.

Consider the following extract of a PTT file of E.Coli K12 (GeneBank U00096):

```
Location Strand Length PID Gene Synonym Code COG Product
190..255 + 21 1786182 thrL b0001 - - thr operon leader peptide
```

The first line is just a header describing the names of the individual fields, but the second is an entry for a particular gene. The corresponding Prolog term generated by the parser for this gene is then

```
gb(U00096,190,255,'+',1,[
    gene_name(thrL),length(21),pid(1786182),
    synonym(b0001),code('-'),cog('-'),
    product('thr operon leader peptide')]
).
```

Additionally, the file `$SCRIPTS/script_parser.pl` implements two helper predicates to make it easy to use this parser:
`parser_ptt(++Name_PTT_File)` and
`parser_ptt(++Name_PTT_File,,--OutputFile)`.

4.2.3 Parser_Easygene

This parser allows to generate from the GFF format of Easygene prediction several Prolog terms with the following format: `eg(Left,Right,Dir,Frame,[])`.

Two scripts are implemented:
`parser_easygene(++Report_Name)` and
`parser_easygene(++Report_Name,--OutputFile)`.

4.2.4 Parser_Genemark

This parser allows to generate from a report of Genemark.HMM of Easygene prediction several Prolog terms with the following format: `gm(Left,Right,Dir,Frame,[])`. Genemark report of prediction coordinates is parsed.

Two scripts are implemented:
`parser_genemark(++Report_Name)` and
`parser_genemark(++Report_Name,--OutputFile)`.

4.2.5 Parser_Blast

This parser translated into Prolog terms the XML report of Tblastn. The parser generates fact for each hits detected of the XML report. Result of the parser has the following format:

```
blast(Blast_Output_Query_Def,Hit_Accession,Hsp_info)
Hsp_info:
     [Hsp_bit_score
     Hsp_score
     Hsp_evalue
     Hsp_query_from
     Hsp_query_to
     Hsp_hit_from
     Hsp_hit_to
     Hsp_pattern-from?
     Hsp_pattern-to?
     Hsp_query_frame?
     Hsp_hit_frame?
     Hsp_identity?
     Hsp_positive?
     Hsp_gaps?
     Hsp_positive?
     Hsp_align-len?
     Hsp_density?
     Hsp_qseq
     Hsp_hseq
     Hsp_midline]
```

4.3 Data generation and manipulation

In this section, several models used for data generation and manipulation are presented.

4.3.1 genebank_annotator

This models is used to produce an annotation of all the ORFs result from `orf_chopper` model (see subsection 4.3.2). This annotation is produced given the golden annotation available in the PTT file of Genebank. The model takes then as input a file composed of chunk of the genome and a parsed PTT file and generated a file with a format `prolog(sequence(ranges))`. Format of the terms that compose this file is:

```
Term=.. [genebank_annotation,Key,Left,Right,Strand,Frame,Extra_Info]
```

`Extra_Info` includes `seq_annotation(Annotation)` where annotation of the ORF is composed of 0 or 1 given if the nucleotide of ORF is a non-coding or coding given GeneBank. Note that annotation is not reversed. That means that coding ORF of the reverse strand starts with 1s.

Note: a script to use the model is available in `$LOST/scripts/consorf.pl`.

4.3.2 orf_chopper

Description to be done by Ole.

Two options are available to ask for specific length for ORFs: `minimal_length` and `maximal_length`. By default, `minimal_length` (resp. `maximal_length`) options is set to `undefined`. If a value `Min` (resp. `Max`) is given, all the ORF with a length less (resp. greater) or equal to `Min` (resp. `Max`) are removed.

Note: a script to use the model is available in `$LOST/scripts/consorf.pl`.

4.3.3 orf_annotator

This model is used to produce an annotation of ORFs computed by `orf_chopper` . Format of the output is prolog(sequence(ranges)).

```
Term=.. [orf_annotation,Key,Left,Right,Strand,Frame,Extra_Info]
```

`Extra_Info` includes `seq_annotation(Annotation)` where annotation of the ORF is represented as follows:

- a dot represents a nucleotide outside a potential coding region;
- a minus represents a nucleotide inside a potential coding region;
- $<,<,<$ represents a potential start codon;
- $>,>,>$ represents a potential stop codon.

Note: a script to use the model is available in `$LOST/scripts/consorf.pl`.

4.4 Models for measurement and statistical reports

4.4.1 accuracy_report

The model `accuracy_report` can be used the produce a report of various measures of the accuracy of particular gene predictions compared with a golden standard such a genebank.

To use the the model, you need to call `get_annotation_file/4` with the following arguments,

```
get_annotation_file(accuracy_report,
     [ReferenceFile,PredictionFile],
     [
      option(reference_functor,RefFunctor),
      option(prediction_functor,PredFunctor),
      option(start,StartPos),
      option(end,EndPos)
     ],
     OutputFile),
```

`ReferenceFile` must be the full path to a file with facts representing the "correct predictions". `PredictionFile` must be the full path to a file with facts representing the predictions. Both files must be a a `db` type format, with facts on the following form,

```
functor(To, From, Strand, ReadingFrame, Name).
```

Each such fact represent a prediction in the `PredictionFile` or correct gene in the in `ReferenceFile`. The `functor` is a any given functor, but the `ReferenceFile` and the `PredictionFile` should use different functors. The *To* argument represents the position in the genome where the prediction begins (inclusive) and the `From` argument represents the end position of the prediction `ReadingFrame` is a integer in the range $\{1,2,3\}$ `Strand` is either `+` for the forward strand or `-` for the reverse strand.

`get_annotation_file` for the `accuracy_report` model must be called with four mandatory options:

- `reference_functor`: The functor used in the `ReferenceFile`
- `prediction_functor`: The functor used in the `PredictionFile`
- `start`: An integer corresponding to the beginning of the range on which accuracy should be measured.
- `end`: An integer corresponding to the end of the range (inclusive) on which accuracy should be measured.

4.4.2 The `range_stats` model

The `range_stats` model calculates statistics for each gene or similar from a file in the format `text(prolog(ranges(gene)))`. The output file is also in the `text(prolog(ranges(gene)))` format, with an entry for each range in the original file, but with some statistics facts appended to the `ExtraList`.

The model takes two input files,

- A *ranges file* in `text(prolog(ranges(gene)))` format.
- A *data file* in `text(prolog(sequence(gene)))` format.

The model can generate statistics basic nucleotide frequencies, amino acids frequencies and length. The nucleotide sequence for the range is automatically extracted from the *data file* and translated or reverse complemented as necessary. It is possible to adjust which kinds of statistics that should be generated by supplying the model with relevant options:

- `amino_acid_stats`: Whether to add statistics based on amino acid frequency. Values are `yes` (default) or `no`.
- `nucleotide_stats`: Whether to add statistics based on nucleotide frequency.Values are `yes` (default) or `no`.
- `length_stats`: Whether to add statistics based on the length of the range.Values are `yes` (default) or `no`.
- `max_nucleotide_order(N)`: Is an integer that determines the maximal order of nucleotide n-grams to calculate statistics for. Given this option the model will calculate statistics for nucleotide n-grams of size N+1 down to size 1. The default value is 1, which that single nucleotide frequency and di-nucleotide frequency will be calculated.
- `max_amino_acid_order(N)`: Is an integer that determines the maximal order of n-grams to calculate statistics for. Given this option the model will calculate statistics for amino acid n-grams of size N+1 down to size 1. The default value is 0 (e.g. only single amino acid frequencies).
- `genecode`: An integer which identifies the gene code table to use when translating to amino acid sequence. Default is 11 (bacterial).

Depending the on the options for the model a subset of the following facts are appended to the `ExtraList` of each gene in the output file:

- `nucleotide_stats(L)` where `L` is list of elements of the type `stat(Seq,Freq)` and `Seq` is list of nucleotides signifying a particular n-gram (n being the length of list) and `Freq` is the frequency of the n-gram in the gene.
- `amino_acid_stats(L)` where `L` is a list of elements of the type `stat(Seq,Freq)` and `Seq` is list of amino acid symbols signifying a particular n-gram (n being the length of list) and `Freq` is the frequency of the n-gram in the gene.
- `normalized_gene_length(F)` where `F` is a number in the range $0 \cdots 1$ representing the proportion of the length of the range relative to the longest gene in the file. For instance, if the longest gene in the file is 2000 nucleotides long and the gene in question is 1000 nucleotides long, then the normalized gene length will be $1000/2000 = 0.5$.

4.5 Prediction/Gene filter models

Common for these models is that they all take files in the `text(prolog(ranges(_)))` format and produce output in the same format. Usually, the output will be a subset of the entries from the input file.

4.5.1 longest_predication_per_stop_codon

Given an input file with gene predictions, this model selects for each stop codon the longest matching prediction. Only the longest prediction for each stop coden will be written to the output file.

For the goal `annotate/3`, the following files and options are expected:

- Input files: `text(prolog(ranges(gene)))`
- Output file: `text(prolog(ranges(gene)))`

The model only takes one option, `file_functor`, which is used to set the functor to identify the gene predictions in the input file. If the input file contains only facts with the same functor the default value `auto` can be used and the model will infer the functor automatically.

4.5.2 best_prediction_per_stop_codon

Given an input file with gene predictions this model selects, for each stop codon, the prediction with the highest score and write such predictions to the output file. The score is for each prediction is assumed to be present the `Extra` list argument of the prediction facts in the input file.

For the goal `annotate/3`, the following files and options are expected:

- Input files: `text(prolog(ranges(gene)))`
- Output file: `text(prolog(ranges(gene)))`

The model only takes two option:

- `file_functor`: Is used to set the functor to identify the gene predictions in the input file. If the input file contains only facts with the same functor the default value `auto` can be used and the model will infer the functor automatically.
- `score_functor`: Is used to specify how the score is represented in the `Extra` list. For instance, if the Extra list looks like `[...,score(0.75),..]`, then the `score_functor` option should be set to `score`. The argument of the score functor (`0.75` in the example) should be a number or at least something comparable with the standard less-than-or-equal-to Prolog operators.

4.5.3 `gene_filter`

The `gene_filter` model is a model that filters a set of genes based on various matching criteria. The input to the model is a file in `text(prolog(ranges(gene)))` format containing the list of genes to be filtered and a file in `text(prolog(sequence(dna)))` that includes the nucleotide sequence for the entire genome to which the genes belong.

The model supports filtering based on a wide range of criteria, specified using the options given to the model. The model is invoked using `get_annotation_file/4.`

The options supported by the model are:

- `match_frames`: A list of valid frames. Genes in other frames will be filtered. The list must be a subset of the *default value*: `[1,2,3]`.
- `match_strands`: A list of valid strands. Genes occuring an other strand will be filtered. The list must be a subset of the *default value*: `['+','-']`.
- `range`: A range represented by a list of two integers values . Genes with a length in this range will be keep. The *default value*: `[default,default]`, can be used to not filter on the length of a gene.
- `exact_match_extra_fields`: A list of Prolog terms terms each of which must unify with an element of the `Extra` list of the gene terms. Genes which do not are filtered.

- `exact_no_match_extra_fields`: A list of Prolog terms terms any of which *may not* unify with any element of the `Extra` list of the gene terms. Genes which do are filtered.

- `regex_match_extra_fields`: A list of regular field names in the `Extra` list and regular expressions to match those fields. For example, a regular expression matcher for the `name` field would be specified as `name('^y.*$')`, here matching genes with a name starting with the letter y. The syntax of the regular expressions are as specified in section 6.24. Gene tems that do not match each of the specified regular expressions are filtered.

- `regex_no_match_extra_fields`: A list of field names in the extra and regular expressions to match those fields. If a gene is matched by any of these regular expression, it will be filtered.

- `match_protein_coding`: Must be either `yes` or `no`. Default is `no`. If set to `yes`, this will filter gene terms which do not have a valid start and stop codon. The codon table to be used is specified using the the `genecode` option.

- `genecode`: An integer specifying what genecode to use. Default is `11`, which is the bacterial gene code. See 6.13 for a list of valid gene code tables.

- `invert_results`: Must be either `yes` or `no`. Default is `no`. If set to `yes`, then only the otherwise non-filtered results are filtered.

4.6 Implementation of well-known HMM architecture

4.6.1 genemark_genefinder

This models implements the architecture of HMM described in [SMCB99]. Thanks to PRISM, two parameters can be set to specify the number of codons with a specific frequency after the start codon and the number codons with a specific frequency after the start codon. Note that these parameters encoded in the PRISM program. However, it is possible to define as an option of the model (Not done).

4.6.2 hein_genefinder

This models implements the architecture of HMM described in [MJ06]. This model has been used to detect overlap genes in viral genome.

4.7 Model that integrate other programs

4.7.1 Genemark

To be documented...

4.7.2 Glimmer3

To be documented...

4.8 Prediction of Stop codons (Sren Mrk models)

In this section, we describe several models used for the prediction of stop codons. Some scripts available via $LOST/scripts/script_soer.pl can be used to play with this models

4.8.1 ecoparse_adph

This model is a possible model for a coding region. The model is HMM for a coding region that models essentially the Ecoparse Model with a ADPH length modeling.

Annotation process for this models allows associating a set of data with the Viterbi probability and the probability to observe a sequence of data. Output format of the file is `prolog(range((genes)))`. Terms of this file have the following format

```
 Term =.. [annotation_ecoparse_adph,Key,Left,Right,Strand,Frame,
                  [probability(Proba),viterbi_probability(VProba)]]
```

Learning process for this models allows to set the parameter of the models given a specification of data (`prolog(range((genes)))`) and a raw genome (`prolog(sequence())`). See `interface.pl` to see options available.

4.8.2 logodds

Given the computation of Viterbi for a Null model and a model, this simple model permits to collapse the result of the two predictions and compute the logodds probability for each prediction. Output format of the file is `prolog(range((genes)))`. Terms of this file have the following format

```
Term =.. [Functor_HMM,Key,Left,Right,Strand,Frame,[logodds(LogOdds)|Extra_Term_HMM]],
```

4.8.3 iid

This model is used as null model for the computation of logodds. It is a simple Markov chain that emits a nucleotide.

Annotation predicate for this models allows associating a set of data with the Viterbi probability. Output format of the file is `prolog(range((genes)))`. Terms of this file have the following format

```
Term =.. [viterbi_iid,Key,Left,Right,Strand,Frame,
                         [viterbi_probability(Proba)]].
```

4.9 Various models

4.9.1 hard_to_find_genes

The model `hard_to_find_genes` ranks a given set of genes according to how many genefinders that find the gene.

The input to the model is a list of files, of which the first element is taken to the *golden standard* - a file containing a set of *true* genes. The rest of the n files in the input file list are files that contain gene predictions of various genefinders (one genefinder per input file). All the files are expected to be in the `text(prolog(ranges(gene)))` format.

For each gene in the *golden standard* file, the model checks which of the gene finders predicts this gene. It writes each of these genes to the output file, but with facts added to the `Extra` list. The added facts are

- `found_by_genefinders(F)`: Where `F` is a list of n entries containing
  - `1` if the nth gene finder predicts the gene.
  - `0` if the nth gene finder did not predict the gene.
- `gene_finding_difficulty_score(S)`: Where `S` is a decimal number between zero and one representing the inverse of the proportion of genefinders that predicts this gene. E.g. if it is zero, no gene finders predicts the gene, and if it is one, then all n gene finders predicts the gene.

The model take a number of options:

- `start`: Is a positive integer indicating the start of the range in which to consider genes. Only genes starting/ending after this position are included in the output file. The default value `min` is taken to mean the start of the file, e.g. position 1.
- `end`: Is a positive integer indicating the end of the range in which to consider genes. Only genes with a right border below this value included in the output file. The default value `max` is taken to mean the maximal position of any gene in the *golden standard* file.
- `gene_match_criteria`: Is used to determine what is required for a gene prediction to *match* a gene. If the value is set to `start_and_stop` (default), then both start codon and stop codon needs to match. If the value is set to `stop` then only the stop codon needs to match.

4.10 TODO Model

4.10.1 soer

Copy paste of *.psm files + usefull files: default.pl and a data file.

4.10.2 petit

In this repertory, I put two models not included yet in the framework. My Easygene implementation, and length genefinder model. I did not have time to finish the second model. The idea of this model is to have a coding region model that only looks for gene with a specific length represented by a range. Beside, I expect that the codon frequencies differ if we look for gene with a length 200-400 or 2000-2200 for example.

Chapter 5

File formats

The chapter documents the different types of file formats used in the lost framework. The naming of file formats follow a simple scheme using nested Prolog functors where the outermost functor is the most general and the innermost is the most specific description of the file format. For instance, consider the naming for a DNA sequence expressed as a Prolog file,

```
text(prolog(sequence(dna)))
```

The outermost functor `text` specifies that the file is a text file, and the next functor `prolog` says that the text file contains Prolog code. The next one, `sequence`, specifies the type of data (sequence data) we expect to be expressed in the prolog facts finally, the innermost functor `dna` specifies the type of sequence that we are dealing with.

5.1 text(prolog(_))

`text(prolog(_))` formats contain Prolog facts. The functor and arity of those facts cannot be determined by knowing that it is `text(prolog(_))`, but needs specification using further embbed functors.

5.1.1 text(prolog(sequence(_)))

This format should be specified for sequence data expressed as prolog facts. For instance, a file of this format may contain facts like the ones below, which specifies the alphabet,

```
data(alphabet,1,10, [a,b,c,d,e,f,g,h,i,j]).
data(alphabet,11,20],[k,l,m,n,o,p,q,r,s,t]).
data(alphabet,21,27,[u,v,w,x,y,z]).
```

The form of these facts are,

```
functor(Identifier,To,From,SequenceElementList).
```

The format does not dictate the `functor` of the facts (e.g. `data`), nor the size of the data list in the fourth argument. However, the range expressed by `To` and `From` should correspond to the numer of elements in `SequenceElementList`.

`text(prolog(sequence(dna)))`

Like the above format but restricts the alphabet of the data elements in the `SequenceElementList` to the set {`a`,`g`,`c`,`t`}.

`text(prolog(sequence(rna)))`

Like the above format but restricts the alphabet of the data elements in the `SequenceElementList` to the set {`a,g,c,u`}.

`text(prolog(sequence(amino_acids)))`

Like the above format but restricts the alphabet of the data elements in the `SequenceElementList` to the set {`a,c,d,e,f,g,h,i,k,l,m,n,p,q,r,s,t,v,w,y`}.

5.1.2 `text(prolog(ranges(_)))`

The format consists of Prolog facts, each of which contain an annotation for a particular range of some sequence

```
F =.. [ Functor, SequenceId, LeftEnd, RightEnd | _ ]
```

The functor may vary depending on the type of annotation but is expected to be same within a file. Similarly, `SequenceId` is an atom which serves as sequence identifier. The next two arguments, `LeftEnd` and `RightEnd` are positive integers which specifies the range relative to the sequence. Both are inclusive. It must be the case that `LeftEnd` $\leq$ `RightEnd`.

`text(prolog(ranges(gene)))`

This is a compatible subtype `text(prolog(ranges(_)))`. The format consists of Prolog facts, each of which contain an gene annotation for a particular range of some DNA sequence (all facts refer to the same sequence).

The facts are on the form,

```
functor(SequenceId,LeftEnd,RightEnd,Strand,Frame,ExtraList).
```

The functor may vary depending on the type of annotation but is expected to be same within a file. The first argument `SequenceId` is a sequence identifer. Is is recommended that the Genbank accession id is used where possible. The next two arguments, `LeftEnd` and `RightEnd` are positive integers which specifies the range of the annotation relative to the DNA sequence. Both are inclusive. It must be the case that `LeftEnd` $\leq$ `RightEnd`. The `Strand` argument is `'+'` for the forward strand or`'-'` for the reverse strand. The `Frame` argument is one of {`1,2,3`}. The final argument, `ExtraList` is a list possibly containing extra information. Each element in the `ExtraList` is on the form `F =.. [ Key, Value ]` where `Key` must be unique in the list. There are no restrictions on `Value`.

5.1.3 `text(prolog(prism_switches))`

This is the format used by PRISM to save and load parameter files.

5.1.4 `text(ptt)`

TODO ptt files

5.1.5 `text(easygene_report)`

TODO ptt files

5.1.6 `text(fasta(SequenceType)`

A sequence in FASTA format begins with a single-line description, followed by lines of sequence data. The description line is distinguished from the sequence data by a greater-than ("¿") symbol in the first column. The word following the "¿" symbol is the identifier of the sequence, and the rest of the line is the description (both are optional). There should be no space between the "¿" and the first letter of the identifier. It is recommended that all lines of text be shorter than 80 characters. The sequence ends if another line starting with a "¿" appears; this indicates the start of another sequence.

The format identifier `text(fasta(_))` is used to refer to any type of file in FASTA format, but we distinguish between different subtypes with `SequenceType`. A ground value for `SequenceType` may be one of the following:

- `fasta`: Specifies generic fasta format. This is the same as specifying `text(fasta(_))`.
- `fna`: fasta nucleic acid.
- `ffn`: FASTA nucleotide coding regions. Contains coding regions for a genome.
- `ffa`: fasta amino acid.

Chapter 6

Library modules

This chapter describes the libraries that are available to components in the framework.
A library module is loaded using the command

```
:- lost_include_api(lists).
```

Which will include the lists module. Note that there is no namespace protection. The above call merely consults the `$LOST_BASE_DIR`/lib/lists.pl.

In the following sections the available libraries are described in detail. We use the SWI-Prolog pldoc system [WA07] to generate documentation for the libraries. To be able to produce file-level comments, all the libraries are treated as Prolog modules. To achieve this we added a dummy `:- module(Name, Exports)` statement to the library. Since libraries are not real modules, e.g. not present in PRISM/B-Prolog, there are no encapsulation and every predicate in the file is exported. Hence, in the documentation below, if a predicate is marked as *private* it is still "publicly" exported and can be used without further ado.

6.1 accuracy.pl – accuracy measures for gene predictions

author : Christian Theil Have

This is a tool to calculate the accuracy of gene finder predictions against a reference annotation. It expects to find facts representing the predictions and the reference annation. These facts are expected to be on the form functor(From, To, Strand, ReadingFrame, Name).

sensivity(*+TP, +TN, -SN*) *[private]*

Computation of the sensity

```
SN is TP/(TP+FN) or undefined
```

specificity(*+TP, +FP, -SP*) *[private]*

Computation of the specificity

```
SP is TP/(TP+FN) or undefined
```

specificity_traditional(*+TN, +FP, -SP*) *[private]*

Computation of the tradional specificity

```
SP is TN / (TN + FP)
```

correlation_coefficient(*+TP, +FP, +TN, +FN, -CC*) *[private]*
Computation of correlation coefficient

Note: 0.0 is add to make numbers floats instead of ints. Ints are not allowed to be larger than 268435455 in bprolog which would otherwise cause this predicate to overflow for realistic data sizes

simple_matching_coefficient(*+TP, +FP, +TN, +FN, -SMC*) *[private]*
Computation of the simple matching coefficient

```
SMC is (TP + TN) / (TP + FN + FP + TN)
```

average_conditional_probability(*+TP, +FP, +TN, +FN, -ACP*) *[private]*
Computation of the average condtional probability

approximate_correlation(*+TP, +FP, +TN, +FN, -AC*) *[private]*
Computation of the average correlation

gene_difficulty_score(*+NumGeneFinders, +NumFoundGene, -DifficultyScore*) *[private]*
DifficultyScore is computed given the following formula

```
DifficultyScore is 1 - ( NumFoundGene / NumGeneFinders)
```

report_nstats(*+ReferenceAnnotFunctor, +PredictionAnnotFunctor, +Start, +End*) *[private]*
This predicate computes and prints in the standart output: True Positive, True Negative, False Positive, False Negative If I am right, *Start* and *End* specified a sub-part of the genome where this computation is performed

nucleotide_level_accuracy_counts(*+Begin, +End, +RefAnnot, +PredAnnot, -TP, -FP, -TN, -FN*) *[private]*
Compute some statistics at the nucleotide level.

strand(*+Strand*) *[private]*
Strand in {+,-}

reading_frame(*+Frame*) *[private]*
Frame in {1,2,3}

6.2 annotation_index.pl – annotation index

author : Christian Theil Have

The annotation index is used to keep track of files generated by running tasks in models. It is primarily used internally, and is not expected te be used directly from models. In fact, updating the index from a model (which runs a separate process), may damage the index.

The index file, $LOST_BASE_DIR/annotation.idx contains a list of facts on the form

```
fileid(FileId,Filename,Model,Goal,InputFiles,Options)
```

where InputFiles and Options are a lists.

The quadruple (Model,Goal,InputFiles,Options) uniquely identifies Filename and the relation is deterministic in this sense.

lost_file_index_get_filename(*+IndexFile, +Model, +Goal, +InputFiles, +Options, -Filename*) *[det]*
Retrieve the filename matching (*Model*,*Options*,*InputFiles*) from the file index If no such filename exists in the index, then a new unique filename is created and unified to *Filename*.

lost_file_index_next_available_index(*+Terms, -NextAvailableIndex***)** *[private]*
Given *Terms*, unify *NextAvailableIndex* with a unique index not occuring as index in terms.

lost_file_index_get_file_timestamp(*+IndexFile, +Filename, -Timestamp***)** *[det,private]*
Timestamp is a term

```
time(Year,Mon,Day,Hour,Min,Sec)
```

which symbolize the time at which the file *Filename* was generated.

lost_file_index_update_file_timestamp(*+IndexFile, +Filename***)** *[private]*
Update the timestamp associated with *Filename* to a current timestamp. This should be used if the file is (re) generated.

lost_file_index_filename_member(*+IndexFile, ?Filename***)** *[private]*
True if *Filename* occurs in the annotation index identified by *IndexFile*.

lost_file_index_inputfiles(*+IndexFile, ?Filename, ?InputFiles***)** *[private]*
True if *Filename* occurs together with *InputFiles* in the annotation index identified by *IndexFile*.

lost_file_index_move_file(*+IndexFile, +OldFilename, +NewFilename***)** *[det,private]*
Allows renaming a file *OldFilename* occuring the annotation index (*IndexFile*) to a new name, *NewFilename* all references to the *OldFilename* in the *IndexFile* will be replaced by *NewFilename*.

6.3 arithmetic.pl – arithmetic

author : Christian Theil Have

Various arithmetic predicates.

abs(*+Number, -AbsoluteNumber***)** *[det]*
AbsoluteNumber is the absolute value of the integer *Number*.

min(*+Number1, +Number2, Smallest***)** *[det]*
Smallest is the smallest of *Number1* and *Number2*

max(*+Number1, +Number2, Largest***)** *[det]*
Largest is the largest of *Number1* and *Number2*

list_min(*+List, -SmallestElement***)**
SmallestElement is the smallest element in the list.

list_max(*+List, -LargestElement***)**
LargestElement is the largest element in the list.

6.4 autoAnnotations.pl – autoAnnotations

author : Henning Christiansen

This file defines a preprocessor for PRISM programs that include annotations, which are redundant arguments intended to present abstract descriptions from the data being modelled.

Such annotations can be extracted from the proof trees generated by PRISM's viterbi predicates, but this is quite tedious to program

Such annotations in a model tend to make PRISM's viterbi calculations run very slow (in many cases, prohibitively slow). On the other hand, annotations are

1. essential for doing supervised learning
2. convenient when presenting predictions (most probably analyses) for the user. The autoAnnotations snystem takes a PRISM program that includes annotations; the user must indicate which arguments that are annotations by a special syntax illustrated in the supplied sample file. It can produce automatically, the following programs
3. a version of the PRISM program without annotations

   (a)

   useful for viterbi analyses
4. an executable version of the PRISM program with annotations
   - useful for initial testing of the model, for sampling and for supervised learning Probabilities found by learning with program (2) can be used with program (1)
5. a translator from the proof trees produced by program (1) under viterbi prediction The main advantage of using autoAnnotations is that you need only maintain a single program containing the 'logic' of your model.

   Version 2.1 for PRISM 1.12 and higher. January 2009 Changes:
   - PRISM has a viterbit that produces a proper tree (so the code gets cleaner).
   - PRISMs target declaration is obsolete - which is fine as previous version of autoAnnotations did not treat it correctly.

   Written by Henning Christiansen, henning@ruc.dk, (c) 2008 Beta version, December 2008; testing still very limited Beta was tested extensively by students - no bugs were found

   Version 2 is developed Jan 2009 - based on new facilities of PRISM 1.12

   Bug fixed 30 mar 2009 `->` v 2.1

- nb: fix is based on undocumented prism predicate '$is_prob_pred'(Pname,Parity)
- Fixed to use $pd_is_prob_pred instead of deprecated $is_prob_pred (works with PRISM ver. 2.x) STILL NEEDS TO BE TRIMMED FOR OPTIMAL STORAGE UTILIZATION

prismAnnot(*+File***)** *[det]*

Should be called to load a PRISM file. This will transform the progrram and subsequent calls to `viterbiAnnot/2` will work efficiently using the transformed program. Note that this some intermediate Prolog files files.

prismAnnot(*+File, +Mode***)** *[det]*

Mode is `direct` or `separate` For learning you need to work with the 'direct' program, and then put probabilities into a file - and then load them in when you have compiler the separate program. Using both `prismAnnot(`*File*`,separate)` and `prismAnnot(`*File*`,direct)` at the same time is not possible; only the most recent one counts.

viterbiAnnot(*+Call, -P***)** *[det]*

viterbiAnnot is analogous to PRISMs `viterbig/2` Use PRISM's tools for executing the direct version of the program

6.5 chmm.pl – Constrained Hidden Markov Models

author : Christian Theil Have

This module is an implementation of Constrained Hidden Markov Models in PRISM. The implementation includes a few well-known global constraints which may be used with the model. The implementation is described in detail in ICLP 2010 paper:

Henning Christiansen, Christian Theil Have, Ole Torp Lassen and Matthieu Petit "Inference with Constrained Hidden Markov Models in PRISM".

Abstract: A Hidden Markov Model (HMM) is a common statistical model which is widely used for analysis of biological sequence data and other sequential phenomena. In the present paper we show how HMMs can be extended with side-constraints and present constraint solving techniques for efficient inference. Defining HMMs with side-constraints in Constraint Logic Programming have advantages in terms of more compact expression and pruning opportunities during inference. We present a PRISM-based framework for extending HMMs with side-constraints and show how well-known constraints such as cardinality and all_different are integrated. We experimentally validate our approach on the biologically motivated problem of global pairwise alignment.

init_store
Initialization of constraints added on the model

forward_store(*?S***)**
The predicate get the current store or remove it

get_store(*S***)**
Get the current store

init_constraint_stores(*+Constraints, -Store***)** *[private]*
Recursive predicate that gets every initial *Store* associated with each constraint of *Constraints*

6.6 dnaseq.pl – working with DNA sequences

author : Christian Theil Have
This library contains predicates for various basic operations on DNA sequences

dna_seq_complement(*+NucleotideSequence, -ReverseComplementedNucleotideSequence***)**
Complement the DNA sequence given in the list *NucleotideSequence* yield its complement as the list *ReverseComplementedNucleotideSequence*

dna_complement(*?Base, ?Complement***)**
Relation that maps bases to their complements

dna_translate(*+Genecode, ?DNASequence, ?AminoAcids***)**
Translates a DNA sequence to a sequence of amino acids or vice versa: Produce all possible DNA sequences for a given amino acids sequence

6.7 dynamic_consult.pl – dynamic consulting and unconsulting of Prolog files

author : Christian Theil Have

This library allows you to consult a file, perform some queries on the contents of consulted file, and later ”unconsult” the file again.

dynamic_consult(*+File*)
Consult the prolog file *File*. All terms in the file will be asserted.

dynamic_unconsult(*+File*)
Undo consult of *File*. All asserted terms associated with *File* are retracted.

6.8 entropy.pl – Tools for calculating entropy of distributions.

author : Ole Torp Lassen

NOTE TO THE USER: `entropy/2` works for

- single random variables that have been declared using `values/2` and have been instatniated, i.e.:

```
values(t,[1,2]).
msw(t,V),
entropy(t,1.0.)
```

- lists of random variables that have been declared using `values/2` and have been instatniated, i.e.:

```
nvalues(t,[1,2]).values(u,[1,2]).
msw(t,T),msw(u,U),
entropy([t,u],2.0).
```

- parameterfiles that been saved with `save_sw/1`

```
entropy(Filename,E).
```

`entropy/1` writes to output.

6.9 errorcheck.pl – convenient handling of errors/exceptions.

This library contains predicates to easy exception handling.
author: Christian Theil Have

check_or_fail(*Goal, Error*)
call *Goal* and throw and exception with error if *Goal* fails. Also, never backtrack beyond this point.

check_or_warn(*Goal, Error*)
call *Goal* and warn with error if *Goal* fails. Also, never backtrack beyond this point.

6.10 fasta.pl – working with the FASTA file-format

author
- : Christian Theil Have
- : Matthieu Petit
- : Ole Torp Lassen

This file includes some utilities for working with the FASTA format

fasta_load_sequence(*+InputFile, +SequenceIdentifier, -FastaHeaderLine, -Sequence* **)**
From a FASTA file, this predicate transforms the header and data lines in in a Prolog format

fasta_save_sequence(*+OutputFile, +SequenceData, +Header* **)**
Given a header and a sequence of data, this predicate writes in Outputfile *SequenceData* and *Header* in the fasta format.

split_file_fasta(*+Filename, +ChunkSize, +OutputFilePrefix, +OutputFileSuffix, -ResultFiles* **)**
Split a FASTA composed of several header (> ...) into multiple files. We consider that a chunk has been seen each time that the symbol > appears at the beginning of a line

6.11 file_management.pl – file management

author : Christian Theil Have

This library contains utility predicates for file management.

is_generated_file(*+File* **)**
True iff *File* is a file generated by one of the model of the framework

rm_gen
This predicate deletes *.gen files in $LOST_BASE_DIR/data repository

rm_tmp
This predicate deletes *.gen files in $LOST_BASE_DIR/tmp repository

move_data_file(*+OldFilename, -NewFilename* **)**
Move a generated data file somewhere else and update the annotation index

6.12 filepath.pl – filepath

author Christian Theil Have
Directory and file management in the LoSt framework.

lost_testcase_directory(*-TestCaseDir* **)** *[det,private]*
TestCaseDir is the absolute path to the directory containing test cases:

```
$LOST_BASE_DIR/test/
```

lost_models_directory(*-TmpDir* **)** *[det,private]*
TmpDir is the absolute path to the directory containing temporary files:

```
$LOST_BASE_DIR
```

lost_utilities_directory(*-Dir*) *[det,private]*
Dir is the absolutate path to the directory containing utility scripts

```
$LOST_BASE_DIR/utilities/
```

lost_models_directory(*-ModelsDir*) *[det,private]*
ModelsDir is absolutate path to the directory containing models:

```
$LOST_BASE_DIR/models/
```

lost_data_directory(*-Dir*) *[det,private]*
Dir is the absolutate path to the directory containing data files

```
$LOST_BASE_DIR/data/
```

lost_script_directory(*-Dir*) *[det,private]*
Dir is the absolutate path to the directory containing LoSt scripts

```
$LOST_BASE_DIR/scripts/
```

lost_model_directory(*+Model, -ModelDir*) *[det,private]*
ModelDir is the absolute path to the directory containing the the model with name *Model*

lost_model_interface_file(*+Model, -ModelFile*) *[det,private]*
ModelFile is the absolute path the '`interface.pl`' file for the model with name *Model*

lost_data_index_file(*-IndexFile*) *[private]*
IndexFile is absolute path to the database containing the an index of Lost script result files

lost_tmp_file(*+Prefix, -TmpFile*) *[det,private]*
Creates a temporary file in the $LOST_TMP_DIR directory with a unique name starting with *Prefix*

6.13 genecode.pl – Table of genetic codes.

This library contains various genetic code tables, i.e. what codons are start codons and stop codons. See, `http://www.ncbi.nlm.nih.gov/Taxonomy/Utils/wprintgc.cgi` for further information.

genecode(*+TableNum, ?Codon, ?AminoAcid*)
This predicate defines the relationship between amino acids and codons with respect to a genetic code identifier, *TableNum*. *Codon* is a list of of three nucleotides symbols from the alphabet [a,g,c,t]. The argument *AminoAcid* is a symbol from the alphabet:

```
[a,c,d,e,f,g,h,i,k,l,m,n,p,q,r,s,t,v,w,y,*]
```

where all symbols except `*` represents an amino acid (the name of which starts which that letter). The special symbol `*` signifies a stop codon in the specified genetic code.

genecode_start_codon(*+TableNum, -StartCodon***)**
This predicate specifies *StartCodon* given a *TableNum*

genecode_start_codons(*+TableNum, -StartCodons***)**
StartCodons is a list of all start codons given a genetic code *TableNum*.

genecode_stop_codons(*+TableNum, -StopCodons***)**
This predicate computes the list of stop codon given a *TableNum*

6.14 genedb.pl – working with text(prolog(ranges(_))) files

author : Christian Theil Have

This library contains utility predicates for working with the `text(prolog(ranges(_)))` format. This format, which is descibed in detail in the manuals section on file formats contain facts,

```
gene(SequenceId,Left,Right,Strand,Frame,Extra)
```

Note that the functor 'gene' may vary. `Left` and `Right` are integers indicating the left and right position of the gene in the genome. They are both inclusive. `Strand`, which is either '+' or '-' indicates whether the gene is located on the direct or reverse strand. `Frame` is an integer in [1,2,3,4,5,6] which indicates the reading frame of the gene. `Extra` is a list of key/value pairs, e.g.

```
[product('hypothetical protein'), cid(12351)]
```

The format does not dictate which key/value pairs are required, so this may vary depending on the application and how the database was generated.

gene_sequence_id(*+GeneRecord, -SequenceId***)** *[det]*
Extract the *SequenceId* field from a *GeneRecord*

gene_left(*+GeneRecord, -Left***)** *[det]*
Extract the *Left* position of *GeneRecord*

gene_right(*+GeneRecord, -Right***)** *[det]*
Extract the *Right* position of *GeneRecord*

gene_strand(*+GeneRecord, -Strand***)** *[det]*
Extract the *Strand* (+,-) from a *GeneRecord*

gene_frame(*+GeneRecord, -Frame***)** *[det]*
Extract the *Frame* from a *GeneRecord*

gene_extra_field(*+GeneRecord, +Key, -Value***)**
Extract the *Value* of the extra field which has *Key* as its functor.

genedb_distinct_predictions(*+GeneDBFunctor, GeneEnds, PredictionsForEnd***)**
Find all distinct predictions in a *consulted* genedb: *GeneEnds* is a list of stop codon positions for the predictions and *PredictionsForEnd* is a list of lists. The list has the same length as *GeneEnds* and each element corresponds to to an element in *GeneEnds*. The elements of the list are lists of predictions in list predictions for the parstop codon

genedb_distinct_stop_codons(*+GeneDBFunctor, -DistinctStops***)** *[det]*
DistinctStops is a list (a set) of all stop codons of predictions in the database.

genedb_predictions_for_stop_codon(*+GeneDBFunctor, +StopMatchPattern, -Predictions*)
Finds all predictions that matches *StopMatchPattern*. *StopMatchPattern* is a list:

```
[+StopCodonEnd,?Strand,?Frame]
```

StopCodonEnd is position of the last nucleotide in the stop codon (i.e. right-most on direct strand and left-most on reverse strand). Strand is '+' or '-' and frame is one of [1,2,3,4,5,6].

6.15 help.pl – Model information and help

author : Matthieu Petit
This library extracts information about models such as,

- what are the valid task of the model
- what options does a particular task predicate take/require
- what is the format of the file produced by the task
- what input files and their formats does each task assume

bug : (cth) I think this help library has become slightly outdated by recent developments in the framework. It should be updated!

help_model(*+Model*) *[private]*
Help predicate for defined model of the framework. If *Model* iss not defined, help_model print out a list of defined models

help_information(*+Term_Input, +Term_Output, +Term_Option, -Predicates*) *[private]*
Compute a list of informations for each defined predicate of the models

print_predicates_help(*+Predicates*) *[private]*
Predicate that writes information of the defined predicates on the standart output

print_available_model *[private]*
This predicate prints on the default output a list of available models

6.16 interface.pl – interface

author: Christian Theil Have
Module that defines main predicates of the framework.

run_lost_model(*+Model, +Goal*) *[private]*
Can be called as,

```
run_model(Model, Goal)
```

Model: The name of the model to be run

Goal must be of the from, *Goal* `=..` [F, InputFiles, Options, OutputFile] InputFiles: A list of filenames given as input to the model. Options: A list of options to the model.

verify_models_options_declared(*+Model, +Goal, +Options*) *[private]*
Fail if a given option is undeclared.

expand_model_options(*+Model, +Goal, +Options, -ExpandedOptions***)** *[private]*
: Expand the set of given options to contain options with default values as declared by the model

launch_prism_process(*+PrismPrologFile, +Goal***)** *[private]*
: Launch a prism process that first consults the file named by *PrismPrologFile* and then execute the goal *Goal*.

load_script *[private]*

list_lost_models_to_file(*File***)** *[private]*
: Write in *File* the list of lost models This is used by the CLC gui

list_lost_models(*-Models***)** *[private]*
: Compute the list of models available

lost_model_input_formats_to_file(*+Model, +Goal, +OutputFile***)** *[private]*
: Write the input formats of a *Model* called with *Goal* to the file *OutputFile* Used by the CLC gui

6.17 io.pl

load_annotation_from_file(*+Type_Info, +Options, +File, -Annotation***)**
: Description: given some *Options*, generate an *Annotation* from a set of terms contained into *File*

Type_Info: sequence Terms in *File* are composed of List of data. *Annotation* is a list *Options* available: *Options* = [data_position(Position), all_lists,range(Min,Max), range(Range) consult] all_lists does not support a range option

Type: db Terms in *File* are composed of Range that delimites a specific region. *Options* available: *Options* = [in_db(Letter),not_in_db(Letter), range_position(Position),range(Min,Max)] Note: assumption is done on the format of the database. Information is extracted from a list of terms that have a parameter Range to describe a specific region of the genome. By defaut, annotation of the specific region is 1 and 0 when the region is not specific.

concat_files(*+SmallFiles_List, +BiggerFile_Name***)**
: concatenates the contents of a list of files into one

6.18 lists.pl – working with lists.

author : Christian Theil Have

List manipulation predicates.

atom_concat_list(*+List, -Atom***)** *[det]*
: Concatenates all atoms in *List* in the order they appear to form a concatenation, *Atom*

inlists_nth0(*+ListOfLists, +N, -NthElemFromEachList***)**
: Extract the n'th element (0 based indexing) from a list of lists

flexible_append(*+L1, +L2, -L3*) *[det]*
Append variant which permit atom elements as first/second argument

flatten_once(*+ListIn, -ListOut*) *[det]*
Merge list of lists into one long list, e.g.

```
flatten_once([[a,b],[c,d],[e,f]],E) => E = [a, b, c, d, e, f].
```

Note that unlike the more generic flatten predicate, this only flattens out one nesting level

map(*+Goal, +InList, +OutList*)
applies the deterministic *Goal* (which must be arity two) to each element of *InList* producing *OutList* E.g. if A is member of *InList*, then for *Goal* `=..` [Functor, A, B], call *Goal* and B is a member of *OutList*

map_with_arglist(*+GoalPattern, +InList, -OutList*) *[det]*
More flexible version of `map//3`. *GoalPattern* is a special Prolog goal with an argument list, where one of the arguments can be `input` and one of the arguments is named `output`. The goal will be called for each element in *InList* as replacing each argument named `input` and `output` being a variable corresponding an element from *OutList*. As example consider,

```
map_with_arglist(atom_concat(input,input,output),[a,b],[aa,bb]).
```

replace(*+Symbol, +Replacement, +Inlist, -Outlist*)
Outlist is a replicate of *Inlist* which has all instances of *Symbol* replaced with *Replacement*

match_tail(*+InputList, -HeadOfInputList, +TailOfInputList*)
true if *InputList* ends with *TailOfInputList*

not_member(*+Elt, +List*)
true is *Elt* is not a member of *List* Note: well-behaved for *Elt* and *List* ground

intersperse(*?Separator, UnseparatedList, SeparatedList*)
Intersperse a list with a particular separator e.g.

```
intersperse(',', ['a','b','c'], ['a',',','b,',','c'])
```

take(*+N, +ListIn, -ListOut*)
true if *ListOut* is the first *N* elements of *ListIn*

split_list(*+N, +List, -FirstPart, -LastPart*) *[det]*
FirstPart is the first *N* elements of *List* *LastPart* is the remaining

zip(*+List1, +List2, ZippedList*)
Combines two lists into one

6.19 misc_utils.pl – predicates that doesn't fit anywhere else

author : Christian Theil Have

Various useful utility predicates that doesn't fit anywhere else.

terms_has_rule_with_head(*+Terms, +Functor, +Arity*) *[private]*
True if the list *Terms* has a rule with a given *Functor* and *Arity*

atom_integer(*?Atom, ?Integer*) *[private]*
Converts between atom representing an integer number to an integer usuable in arithmetic operationes and vice versa

6.20 path.pl – filepath

author Christian Theil Have
Directory and file management in the LoSt framework.

lost_testcase_directory(*-TestCaseDir*) *[det,private]*
TestCaseDir is the absolute path to the directory containing test cases:

```
$LOST_BASE_DIR/test/
```

lost_models_directory(*-TmpDir*) *[det,private]*
TmpDir is the absolute path to the directory containing temporary files:

```
$LOST_BASE_DIR
```

lost_utilities_directory(*-Dir*) *[det,private]*
Dir is the absolutate path to the directory containing utility scripts

```
$LOST_BASE_DIR/utilities/
```

lost_models_directory(*-ModelsDir*) *[det,private]*
ModelsDir is absolutate path to the directory containing models:

```
$LOST_BASE_DIR/models/
```

lost_data_directory(*-Dir*) *[det,private]*
Dir is the absolutate path to the directory containing data files

```
$LOST_BASE_DIR/data/
```

lost_script_directory(*-Dir*) *[det,private]*
Dir is the absolutate path to the directory containing LoSt scripts

```
$LOST_BASE_DIR/scripts/
```

lost_model_directory(*+Model, -ModelDir*) *[det,private]*
ModelDir is the absolute path to the directory containing the the model with name *Model*

lost_model_interface_file(*+Model, -ModelFile*) *[det,private]*
ModelFile is the absolute path the '`interface.pl`' file for the model with name *Model*

lost_data_index_file(*-IndexFile*) *[private]*
IndexFile is absolute path to the database containing the an index of Lost script result files

lost_tmp_file(*+Prefix, -TmpFile***)** *[det,private]*
Creates a temporary file in the $LOST_TMP_DIR directory with a unique name starting with *Prefix*.

dirname(*+Filename, -DirPart***)** *[private]*
Separate out the directory part (*DirPart*) of a filename.

6.21 prism_parallel.pl

6.22 prologdb.pl – accessing Prolog source files.

author Christian Theil Have

Predicates for working with Prolog source files, i.e. in the format `text(prolog(_any))`.

file_functor(*+File, ?Functor***)**
True if all terms in *File* have the functor *Functor*.

file_functors(*+File, ?Functors***)** *[det]*
Functors is a set of the functors in *File*

terms_from_file(*+File, -Terms***)**
Reads all *Terms* from named file *File*

collect_stream_terms(*+Stream, -Terms***)** *[private]*
Terms is a list of all *Terms* read from *Stream* untill end_of_file

split_prolog_file(*+File, +ChunkSize, +OutputFilePrefix, +OutputFileSuffix***)**
Split a file of Prolog terms into multiple files, such that each file contains a (disjunct) fraction of the Prolog terms in *File*. Each of the resulting files are valid Prolog files.

- *File* is the file to split up.
- *ChunkSize* is the maximal number of terms in resulting fragment files.
- *OutputFilePrefix* is a prefix given to resulting fragment files
- *OutputFileSuffix* is a suffix given to resulting fragment files

merge_prolog_files(*+SeparateFiles, +MergedFile***)** *[det]*
Merges terms from *SeparateFiles* into the file *MergedFile*

6.23 ranges.pl – working with ranges

author Christian Theil Have

Predicates for working ranges. A range is usually specified by two integer denoting the beginning and end of the range (both inclusive).

overlaps(*+Start1, +End1, +Start2, +End2***)**
Determine if two ranges overlap

6.24 regex.pl

6.25 scheduler.pl – Task scheduler

author Christian Theil Have

The Task scheduler is a mechanism for running PRISM processes with dependencies in parallel. It is used behing the scenes by the script language. It uses the scheduler tree structure to repreent tasks and their dependencies and calls an external shell script ($LOST_BASE_DIR/utilities/procrun.sh) to launch new processes. Communicates with the shell script via two files:

It writes tasks to be run to `pending_tasks_file.txt` and reads signals about completion of tasks from the file `completed_tasks_file.txt`.

scheduler_init
Open communication input and output files of the scheduler and starts the shell script based PRISM process runner.

scheduler_shutdown
Close files associated with the scheduler

scheduler_loop(*+SchedulerTree, +RunQueue***)**
This is the main loop that runs the scheduler

schedule_task(*+TaskId, +Model, +Goal***)** *[private]*
Start a new tas

scheduler_wait(*+SchedulerTree, +Queue, -UpdatedSchedulerTree, -UpdatedQueue***)** *[private]*
This will block until a task has completed When the task has completed, it will be removed from the queue and the scheduler tree resulting in the updated ouput variables.

scheduler_wait_for(*-TaskId***)** *[private]*
Read the next task id from the completed tasks file and unify *TaskId* to this id If there are no (next) completed task yet, then it will wait for 500 msecs and loop

scheduler_wait_for_shutdown(*+Count***)** *[private]*
Gracefully wait for scheduler shutdown. Wait for hangup signal from process runner for a number of seconds before terminating.

scheduler_update_index(*+Model, +Goal***)** *[private]*
This will update the state and result file of the task associated *Model* and *Goal* to symbolize the the task completed.

6.26 scheduler_tree.pl – Scheduler tree data structure

author : Christian Theil Have

The scheduler tree is a data tree data structure to represent call graphs of scripts written for the lost framework.

It is used by the process scheduler to infer which tasks can be run in parallel.

A Task may be in several states

- ready (in the tree, but not yet running)
- Running (but not yet completed)
- Completed (in which case it is removed from the tree)

scheduler_tree_create(-*EmptyTree***)** *[private]*
Creates a new scheduler tree which is initially empty

scheduler_tree_add(+*Model*, +*Goal*, +*ParentId*, +*Tree*, -*UpdatedTree*, -*NextId***)** *[private]*
UpdatedTree is *Tree* with a node inserted as a child to *ParentId*. *NextId* is the Id of the newly inserted node. If there are allready an existing node for the same *Model* and Gaol, but possibly a different parent, then the new node is given the same task id

scheduler_tree_remove(+*TaskId*, +*InitialTree*, -*UpdatedTree***)** *[private]*
UpdatedTree is the initial tree without the subtrees rooted at the nodes with *TaskId*

scheduler_tree_replace(+*Node*, +*Tree*, *UpdatedTree***)** *[private]*
Node replaces the subtree of the root node has the same task id as node

scheduler_tree_replace_by_taskid(+*TaskId*, +*Node*, +*Tree*, *UpdatedTree***)** *[private]*
Node replaces the subtree of the root node has the same task id as node

scheduler_tree_reduce(+*Tree*, -*ReducedTree***)** *[private]*
This reduces a scheduler tree by compacting nodes which are structurally the same (i.e. only differing in the task id) Such task will be given the same task id

scheduler_tree_lookup(+*TaskId*, +*Tree*, -*Node***)** *[private]*
Node is the subtree which rooted at the node identified by *TaskId*

scheduler_tree_change_taskid(+*TaskId*, +*TaskIdUpdated*, +*TreeIn*, +*TreeOut***)** *[private]*
scheduler_tree_change_taskid(*TaskId*,*TaskIdUpdated*,*TreeIn*,*TreeOut*) :- scheduler_tree_lookup(*TaskId*,*TreeIn*,[
scheduler_tree_replace(*TaskId*,[node(*TaskId*,State,Model,Goal,Children)],*TreeIn*,*TreeOut*).

scheduler_tree_ready_task(+*Tree*, -*TaskId***)** *[private]*
Find a process in the tree which is ready to run. This process must be a node which a) is a leaf node and b) has the state 'ready'.

scheduler_tree_set_running(+*TaskId*, +*Tree*, -*UpdatedTree***)** *[private]*
True if, a) *TaskId* points to leaf node and b) that node has state 'ready'

scheduler_tree_set_completed(+*TaskId*, +*InitialTree*, -*UpdatedTree***)** *[private]*
1) *TaskId* `-->` leaf node 2) Node as state 'running'

scheduler_tree_empty(+*Tree***)** *[private]*
True if *Tree* is empty

scheduler_tree_print(+*Tree***)** *[private]*
Pretty-prints a scheduler tree

6.27 script.pl – The LoSt scripting language

This is the library for running scripts in the LoSt script language.

The basic way of running a script is to consult the script from PRISM and the call `run/1` to invoke the goal of interest in the script.

view(+*Goal***)** *[private]*
Launch external viewer to view the file corresponding to script goal: *Goal*. Currently, just launches vim. If the goal has not yet been run and the result is not yet available on file, then the goal will be run before launching the file viewer.

run(+*Goal***)**
Will run the script goal *Goal* with the sequential semantics (one task/process at a time).

run_parallel(_+Goal_**)**
Runs the _Goal_ using the parallel semantics.

6.28 script_par.pl

6.29 sequencedb.pl – sequencedb

author : Matthieu Petit and Christian Theil Have

This library contains predicates for working with files of the `prolog(sequence(_any))`.
The typical `prolog(sequence(_))` file contains facts,

```
data(somekey, 1, 4, [a,b,c,d]).
data(somekey, 5, 8, [e,f,g,h]).
```

The size of list may vary and is often tweaked for the efficiency of applications, e.g. in the extreme case the file will only contain one fact where the fourth argument is a long list. This, may however be quite inefficient to access.

Often, these files may even be prohibitively large to consult and predicates for accessing the files without consulting are necessary.

The predicate `seqdb_data_from_file` address this problem by accessing the file in a sequential fashion and extracting data from one fact at a time.

If many repeated queries are going to be made to the file, then repeated sequential access may be to slow. If the file is not to big to fit in memory then loaded into memory and accessed via the predicate, `seqdb_memory_map_file`. Then, the predicate `get_sequence_range/4` can then be used to extract a part of the sequence. This is fairly efficient, but depends on the size of the data terms, `|t|`, in the file. The running time of the predicate is bounded by O(2`|t|` n), where n = Max Min.

sequence_from_file(_+File, +Options, -Data_**)**
This predicate aims at efficiently extracting data (for instance, a list of nucleotides) from a file with a format `text(prolog(sequence(_any)))`. Given of file composed of prolog facts, this predicate generates a list, _Data_. By default, data predicate is in the form but the predicate is able to handle slightly different formats as well.

```
Functor(Key,LeftPosition,RightPosition,Data,...)
```

Type of _Data_ is a list

Options: - data_position(Pos): Where Pos is an integer that specifies the argument that contains the list with of data, i.e. it is 4, meaning the fourth argument.

- left_position(Left): Left is and integer that specifies the argument of which holds Left-position. If left unspecified, it has default value 2. If the term has no argument to indicate left position, then `left_position(none)` should be used.
- right_position(Right): Right is and integer that specifies the argument of which holds Rightposition. If left unspecified, it has default value 3. If the term has no argument to indicate right position, then `right_position(none)` should be used.

By default, the predicate extracts and collects in _Data_ all the data available in _File_. However, two options are available to ask for partial information:

- range(Min,Max): extract a range of data
- ranges(List_Ranges): extract a list of data given a list of Range

Note that data extraction is made without taking care about the strand (for a DNA sequence).

author : Matthieu Petit

sequence_to_file(*+KeyIndex, +ChunkSize, +Sequence, +File*)
Stores List to *File* in the `text(prolog(sequence(_)))` format, e.g. Prolog facts on the form:

```
data(KeyIndex,1,5, [a,t,c,c,g]).
```

The first argument *KeyIndex* is used as an identifier. *ChunkSize* is the number of elements from *Sequence* to store with each fact, e.g. it is 5 in the example above. The second and third argument of a stored fact, corresponds to the start and end position in *Sequence* (indexing starts at position one). The fourth argument of the fact is a list containing the relevant elements of the *Sequence* list. Note that if the the length of *Sequence* is not a multiple of *ChunkSize*, then the last fact stored will have a shorter range.

6.30 stats.pl – Statistics computation module

NOM ET VERSION :
`stats.pl` – Version 0.0

GOAL : Computation of several statistics

HISTORIQUE : M.P 10/03/2010

DESCRIPTION :

normalize(*+Type, +Counting, -Probabilities*) *[private]*
Predicate used to normalize the result of a counting Probabilies is a list composed of element with the format

```
(Past,(Domain,Distribution))
```

build_stat_facts(*+Stats, -StatsFacts*) *[private]*
builds a list of facts, one for each statistic

```
build_stat_facts(
      [([a],[(a,0),(c,1),(g,0),(t,0)]),([c],[(a,0),(c,0),(g,0),(t,1)]),([g],[(a,0),(c,0),(g,1),
      [stat([a,a],0),stat([a,c],1),stat([a,g],0),stat([a,t],0),stat([c,a],0),stat([c,c],0),stat
       stat([c,t],1),stat([g,a],0),stat([g,c],0),stat([g,g],1),stat([g,t],2),stat([t,a],1),stat
      )
```

6.31 unittest.pl – unittest

author : Christian Theil Have

This is a library for setting up unit tests for the Lost framework.

bug : unfinished.

run_tests *[private]*

Runs all test cases specified in files in the testcase directory

report_test_results_text(*-Results*) *[private]*

Reports the results of running a test-suite as simple text with one line per testcase result

testcase_files(*-TestCaseFiles*) *[private]*

Produce a list of files containing testcases

testxase_files_rec(*+Directories, -TestCaseFiles*) *[private]*

recursively scan *Directories* for files containing testcases

run_testcases_in_file(*+Filename, -TestCaseResults*) *[private]*

Runs all testcases in a particular testcase file.

6.32 utils_parser_report.pl – Report parsing utilities

author : Ole Torp Lassen and Matthieu Petit

Various utilities for parsing text files.

parser_line(*+List_Codes, -List_Tokens*) *[private]*

Translate a list of ASCII code into a list of tokens. Parsing is based on the

grammar: Tokens ::= Token | Token Ignore_Characters Tokens

Default: Ignore Caracters = [9,32] (space and tab)

is_number(*+List_Codes*) *[private]*

predicate is true iff *List_Codes* can be parsed by the following grammar

Numbers :== Number | Number . Rest Numbers | Number Numbers

Rest Numbers :== Number | Number Rest Numbers

Number :== 0|1|2|3|4|5|6|7|8|9 (Ascci 48..57)

bug: Fail if number is bigger than 268435455

readline(*+Stream, -CodeList*) *[private]*

Reads in a line or a tab or a token with no spaces of data from specified input-stream and represents it as a list of character codes..

read_token(*+Stream, -N_Spaces, -CodeList*) *[private]*

Reads next nonspace-initiated token, terminated by a some whitespace counts number of spaces encounterd before first nonspace

6.33 viterbi_learn.pl – Fast supervised training

author : Christian Theil Have

This library uses PRISM viterbi algorithm in order to perform supervised learning. It is a substitute for PRISMs builtin learn predicate, but it is significantly faster for supervised learning.

It works by running the viterbi predicate for a list of observation goals and records and counts the msw's that is used in the for the viterbi derivation. Counts are then normalized to produce parameters.

To figure out what msw declarations exist in the PRISM program, it need to the PRISM source file to be consulted in the usual way for Prolog programs (in addition to loading the file with prism(File)).

viterbi_learn_file(*+File***)** *[private]*
Learns the parameters of a PRISM program from the observed goals supplied in *File*

viterbi_learn_file_count_only(*+File***)** *[private]*
Does empirical frequency counting of the MSWs involved in the viterbi path of the observed goals in *File*

viterbi_learn_stream(*+Stream***)** *[private]*
Does empirical frequency counting of the MSWs involved in the viterbi path of the observed goals in *Stream*

viterbi_learn(*+ListOfGoals***)** *[private]*
Does empirical frequency counting of the MSWs involved in the viterbi path of the observed goals in *ListOfGoals*

viterbi_learn_term(*+G***)** *[private]*
Does empirical frequency counting of the MSWs involved in the viterbi path of the observed goal *G*

count_msw(*+MSW***)** *[private]*
increment the counter for *MSW*

clear_counters *[private]*
reset all MSW counts to zero

merge_counts_files(*+FileList***)** *[private]*
Takes a *FileList* and reads MSW counts from each of the files in the list. The counts read are added to the allready existing counts, effectively summing the counts of all the files.

load_counts_file(*+CountsFile***)** *[private]*

add_pseudo_counts *[private]*
Add a small pseudo count for each possible mws outcome

set_switches_from_counts *[private]*
Set the probabilities of all switches using recorded counts

set_switches_from_couns(*+Switch***)** *[private]*
Set the probability for outcome from *Switch* to the normalized value of the observed counts for the switch

count_list_from_outcome_list(*+Switch, +OutcomesList, -CountList***)** *[private]*
produce a list of counts for outcome of *Switch*

compute_frequencies_from_counts(*+Total, +CountList, +FreqList***)** *[private]*
Compute the relative frequency of each outcome

6.34 xml.pl – xml parsing

`xml.pl` : Contains xml_parse/[2,3] a bi-directional XML parser written in Prolog.

Copyright (C) 2001, 2002 Binding Time Limited

Current Release: 1.5

TERMS AND CONDITIONS:

This program is offered free of charge, as unsupported source code. You may use it, copy it, distribute it, modify it or sell it without restriction.

We hope that it will be useful to you, but it is provided "as is" without any warranty express or implied, including but not limited to the warranty of non-infringement and the implied warranties of merchantability and fitness for a particular purpose.

Binding Time Limited will not be liable for any damages suffered by you as a result of using the Program. In no event will Binding Time Limited be liable for any special, indirect or consequential damages or lost profits even if Binding Time Limited has been advised of the possibility of their occurrence. Binding Time Limited will not be liable for any third party claims against you.

History: $Log: `xml.pl`,v $ Revision 1.2 2002-05-25 23:17:54+01 john Set Current Release to 1.5.

Revision 1.1 2002-01-31 21:04:45+00 john Updated Copyright statements.

Revision 1.0 2001-10-17 20:46:24+01 john Initial revision

xml2pl(_+Input, -Output_**)** _[det]_

Create and Prolog representation, _Output_, of the XML file with name _Input_ added by Neng-Fa Zhou

pl2xml(_+Document, -Codes_**)**

Convert Prolog representation to XML document

Chapter 7

Notes and stuff

7.1 Feature wish list

- Division of models into models (probabilistic models) and nodes (which are just data processing).
-
- Document gene database file format and make sure that all models adhere to this format.
- Models should `lost_tmp_directory` for intermediary files.
- More flexibility in the definition of lost_option. Sometimes, it could be useful to not have a default value. What happens if there is dependencies between lost options (added by Matthieu).
- Suggestion/Flying idea for unitary testing of a model: not clear in my mind but it could be a predicate in interface that launches a predefined unit tests of the model with small input. Verdict can be done by regression testing: comparison of the generated file and the previous generated file. Maybe, had a repertory test in the model repertory. (Matt)
- Document all of undocumented models. (Matthieu, Christian)

Some things that we have discussed, but decided not to do

- Do something to avoid multiple declaration of lost_include_api everywhere... (Matthieu).

Bibliography

[MJ06] S. McCauley and Hein J. Using hidden markov models and observed evolution to annotate viral genomes. *Bioinformatics*, 22(11):1308–1316, 2006.

[SMCB99] A.M Shmatkov, A.A. Meikyan, F.L. Chernousko, and M. Borodovsky. Finding prokaryotic genes by the 'frame-by-fram' algorithm: targeting gene starts and overlapping genes. *Bioinformatics*, 11:874–886, 1999.

[WA07] Jan Wielemaker and Anjo Anjewierden. PlDoc: Wiki style literate programming for prolog, November 05 2007. Comment: Paper presented at the 17th Workshop on Logic-based Methods in Programming Environments (WLPE2007).