Lost API documentation

Lost Members

March 18, 2010

Contents

Chapter 1

Setting up the lost framework

1.1 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
```

1.2 Configuring your copy of the lost framework

After obtaining the lost framework, for instance by checking it out from git, a little configuration is needed to get started.

The file `lost.pl` in the top-most directory of the copy of the framework. 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.

Chapter 2

Creating lost models

2.1 Lost model conventions

Each model is located in its own subdirectory of the of the {`lost`}`/models/` directory, henceforth called `$MODELS`. So for instance, the sample model called `sample_model1` is located in {`lost`}`/models/sample_model1/`. We will refer to directory as `$MODEL`.

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. `interface.pl` can then implement various predefined predicates which serves as an entry point of using the model.

The supported interface predicates which a model may provide are:

- `lost_best_annotation/3`.

- `lost_learn/3`

By convention models are expected to store switch probabilities the directory`$MODEL/parameters/`. Switch parameter files should be given the extension `.prb`.

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.

2.1.1 Model interface predicates

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

lost_best_annotation(+InputFileNames,+Options,+OutputFilename) The framework calls this predicate to obtain a "best annotation" from the model. The model is free to provide this annotation in any way it sees fit. It is the models responsibility to save the annotation to `OutputFilename`, before the completion of `lost_best_annotation`.

`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 and the model is expected to be able to parse then. Predicates to parse a wide range of fileformats are supplied in the *io* API (see section 3.2).

`Options`: Is a list of facts on the form, `option(Key,Value)`. This list is use to paramterize the model in various ways. For convinience, option values can be checked an extracted using the the predicates `lost_option` and `lost_required_option`, (see section ??). Some options may be quite common and it is suggested to use the same `Keys` for such option. An incomplete list of these common option keys are,

- `parameter_file`: Indicates that the model should use the switch probability associated with the `Value`.

`OutputFilename`: Full filename which the resulting "best annotation" should be saved to. The model is expected to save the resulting annotation to this file before the completion of `lost_best_annot`. The *io* API contains some common predicates for saving annotations (see section 3.2).

lost_learn(+InputFileNames,+Options,+OutputFilename) This predicates is used for training models. The model is expected to save the result of the training session (e.g. a switch parameter file or similar) to `OutputFilename`.

`InputFileNames`: Is a list of filenames (with absolute paths), each containing an input to the model. These are used for providing the traning data. There is no restriction on the format of the files and the model is expected to be able to parse then. Predicates to parse a wide range of fileformats are supplied in the *io* API (see section 3.2).

`Options`: Is a list of facts on the form, `option(Key,Value)`. This list is use to paramterize the model in various ways. For convinience, option values can be checked an extracted using the the predicates `lost_option` and `lost_required_option`, (see section ??).

`OutputFilename`: Full filename which the resulting switch parameters or similar should be saved to. The model is expected to save the result to this file before the completion of `lost_learn`. The *io* API contains some common predicates for saving annotations (see section 3.2).

Chapter 3

Lost shared APIs

To use the lost APIs, the file `$LOST/lost.pl` located in the top-most {`lost`} directory must be consulted. Then, APIs, which are located in the `$LOST/shared` directory can be consulted using the goal,

```
lost_include_api(+APIName)
```

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

3.1 interface.pl

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

get_annotation_file(Model, Inputs, Options, Filename)

This API provides `get_annotation_file/4` which is used to retrieve the best annotation generated by a specified model with specified parameters and input sequences. If no such file currently exists, then the model will be run (e.g. the `lost_best_annotation/3` provided by the model will be called).

The generated annotation files are named according to a convention. All annotation files will be placed in the {`lost`}`/sequences/` directory. The `Filename` is construed according to the following convention:

```
{Modelname}_annot_{Id}.seq
```

The first time an annotation is generated the file `annotation.idx` will be created in this directory. This file serves as a database to map filenames of the generated annotation files to the (models ,inputs,probability parameters) that generated the particular annotations. This database file contains Prolog facts on the form,

```
fileid(Id,Filename,Model,SwitchParameters,InputFiles).
```

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

If 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 `lost_best_annotation` 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.

3.2 Input-Output API

In this module (`io.pl`), severals 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

- `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..]).`

  - `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]?
```

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.

3.3 Stats API

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:

- `nucleotid` 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 $\{a, c, g, t\}$;
- `animo_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 {`nucleotid,codon,animo_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 connexion 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(nucleotid,[],[a,g,c,t,t],_,R).
R = [([],[(a,1),(c,1),(g,1),(t,2)])] ?
| ?- stats(nucleotid,[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(nucleotid,[order(2)],[a,g,c,t,t],_,R1),
     stats(nucleotid,[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(nucleotid,[order(2)],[a,g,c,t,t],_,Past,R1),
       stats(nucleotid,[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(nucleotid,[],[a,g,c,t,t],_,R),
     normalize(nucleotid,R,S).
S = [([],([a,c,g,t],[0.2,0.2,0.2,0.4]))]
| ?- stats(nucleotid,[order(1)],[a,g,c,t,t],_,R),
     normalize(nucleotid,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]))] ?
```

Chapter 4

Models

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

4.1 Parsers of Biological Data

To extract information from different Biological database, several parsers have been designed to parse report of analyses available on different web-servers (Easygene, 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.

4.1.1 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)`. Option `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,...]).
....
```

4.1.2 Parser_ptt

*.ptt file of Genbank is composed of information about known or predicted genes in a genome. Parser_ptt parses this *.pt file from Genbank and generate list of terms. Two scripts are implemented:
`parser_ptt(++Name_PTT_File)` and
`parser_fna(++Name_PTT_File,,--OutputFile)`
Output format of the generated terms are:
`gb(Genebank_Key,Start,End)`.

4.1.3 Parser_Easygene

4.1.4 Parser_Genemark

4.2 Models for measurement and statistical reports

4.2.1 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.

Chapter 5

Notes and stuff

5.1 Feature wish list

- A `data` directory instead of a `sequences` directory.
- Division of models into models (probabilistic models) and nodes (which are just data processing).
- kind of `make.pl` file that manages the consulting of all the useful file (lost, /shared/*.*, /scripts/*.*). Just let to the user to do consulting of model file. If it is possible, have a very simple interaction with the user to set the lost location, set platform, or list models or data available.

Index