Leaspy

Release 1.1.2

unknown

Dec 18, 2021

GETTING STARTED


leaspy

CHAPTER

ONE

INSTALLATION & TESTING

1.1 Dependencies

leaspy requires:

- Python (>=3.6)
- numpy (>=1.16.6)
- scipy (>=1.5.4)
- scikit-learn (>=0.21.3, <0.24)
- pandas (>=1.0.5)
- torch (>=1.2.0, <1.7)
- joblib (>=0.13.2)
- matplotlib>=3.0.3
- statsmodels (>=0.12.1)

1.2 User installation

1. (Optional) Create a dedicated *conda environment*:

```
conda create --name leaspy python=3.7
conda activate leaspy
```

2. Download *leaspy* with *pip*:

```
pip install leaspy
```

1.3 Testing

After installation, you can run the examples in *Leaspy in a nutshell* and in *the Leaspy API*. To do so, in your *leaspy environment*, you can download `ipykernel` to use *leaspy* with *jupyter*:

```
conda install -c anaconda ipykernel
python -m ipykernel install --user --name=leaspy
```

Now, you can open *jupyter lab* or *jupyter notebook* and select the *leaspy kernel*.

CHAPTER

TWO

LEASPY IN A NUTSHELL

2.1 Comprehensive example

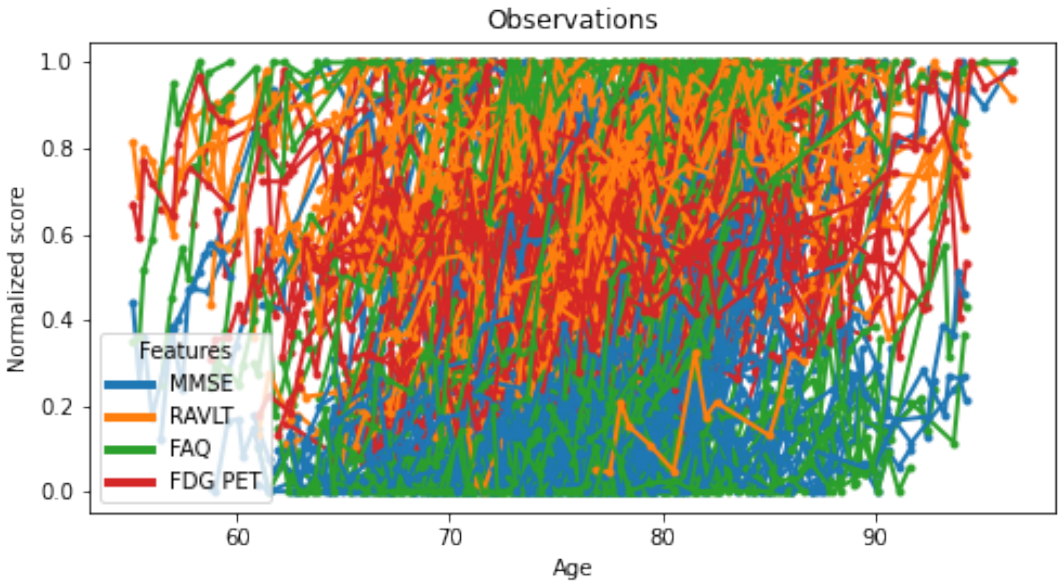
We load some synthetic data from the *leaspy.datasets* module, encapsulate them in the main *leaspy Data container*, then we plot them with the main API *Leaspy*.

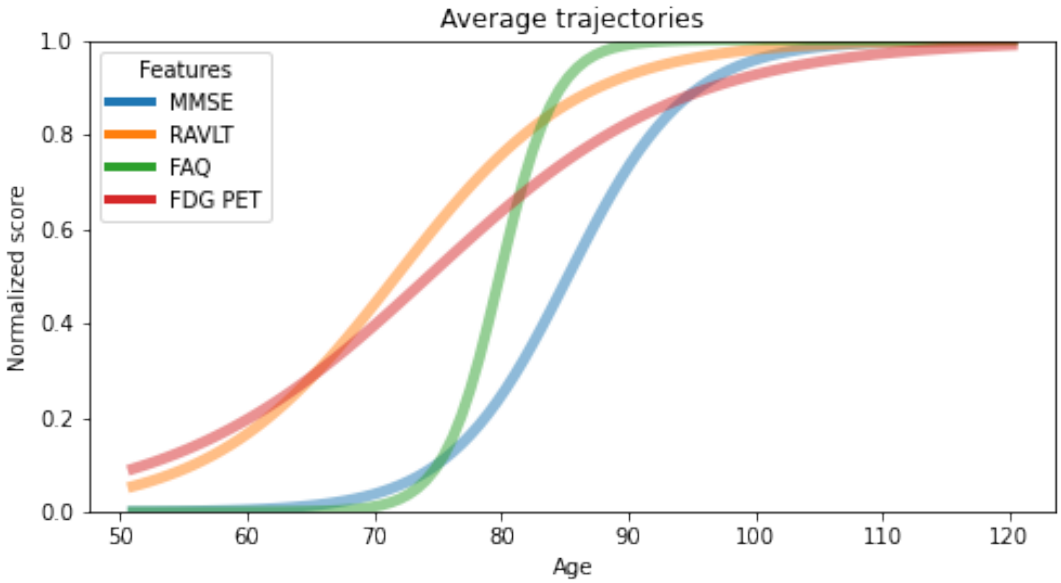
```
>>> from leaspy import AlgorithmSettings, Data, Leaspy
>>> from leaspy.datasets import Loader
>>> alzheimer_df = Loader.load_dataset('alzheimer-multivariate')
>>> print(alzheimer_df.columns)
Index(['E-Cog Subject', 'E-Cog Study-partner', 'MMSE', 'RAVLT', 'FAQ',
   'FDG PET', 'Hippocampus volume ratio'],
  dtype='object')
>>> alzheimer_df = alzheimer_df[['MMSE', 'RAVLT', 'FAQ', 'FDG PET']]
>>> print(alzheimer_df.head())
                     MMSE     RAVLT       FAQ   FDG PET
ID     TIME
GS-001 73.973183  0.111998  0.510524  0.178827  0.454605
       74.573181  0.029991  0.749223  0.181327  0.450064
       75.173180  0.121922  0.779680  0.026179  0.662006
       75.773186  0.092102  0.649391  0.156153  0.585949
       75.973183  0.203874  0.612311  0.320484  0.634809
>>> data = Data.from_dataframe(alzheimer_df)
>>> leaspy_logistic = Leaspy('logistic')
>>> ax = leaspy_logistic.plotting.patient_observations(data)
```

Not so engaging, right? With *leaspy*, we can derive the group average trajectory of this population. We use the *Leaspy.fit* method by providing it the settings for the MCMC-SAEM algorithm. Then, we plot the group average trajectory:

```
>>> model_settings = AlgorithmSettings('mcmc_saem', seed=0, progress_bar=True)
>>> leaspy_logistic.fit(data, model_settings)
==> Setting seed to 0
|##################################################|   10000/10000 iterations
The standard deviation of the noise at the end of the calibration is:
0.0718
Calibration took: 5min 55s
>>> ax2 = leaspy_logistic.plotting.average_trajectory()
```

We can also derive the individual trajectory of each subject. To do this, we use the *Leaspy.personalize* method, again by providing the proper settings. Then we plot both, the first subjects observations and trajectories:


Observations
Normalized score
Age
Features
MMSE
RAVLT
FAQ
FDG PET
0.0
0.2
0.4
0.6
0.8
1.0
60
70
80
90


Average trajectories
Normalized score
Age
Features
MMSE
RAVLT
FAQ
FDG PET
0.0
0.2
0.4
0.6
0.8
1.0
50
60
70
80
90
100
110
120

```
>>> personalize_settings = AlgorithmSettings('scipy_minimize', progress_
↪bar=True, \
use_jacobian=True, seed=0)
>>> individual_parameters = leaspy_logistic.personalize(data, personalize_
↪settings)
 ==> Setting seed to 0
|################################################|   200/200 subjects
The standard deviation of the noise at the end of the personalization is:
0.0686
Personalization scipy_minimize took: 11s
>>> ax = leaspy_logistic.plotting.patient_trajectories(data, individual_
↪parameters, 'GS-001')
>>> leaspy_logistic.plotting.patient_observations(data, 'GS-001', ax=ax)
>>> import matplotlib.pyplot as plt
>>> plt.legend(loc='upper left', title='Features', bbox_to_anchor=(1.05, 1))
>>> plt.title('Subject GS-001 - observations & trajectories')
```

2.2 Using my own data

2.2.1 Data format

Leaspy use its own data container. To use it properly, you need to provide a *.csv* file or a *pandas.DataFrame* in the right format. Let's have a look on the data used in the previous example:

```
>>> print(alzheimer_df.head())
                     MMSE     RAVLT       FAQ   FDG PET
ID     TIME
GS-001 73.973183  0.111998  0.510524  0.178827  0.454605
       74.573181  0.029991  0.749223  0.181327  0.450064
```

(continues on next page)

(continued from previous page)

```
       75.173180  0.121922  0.779680  0.026179  0.662006
       75.773186  0.092102  0.649391  0.156153  0.585949
       75.973183  0.203874  0.612311  0.320484  0.634809
```

You **MUST** have *ID* and *TIME*, either in index or in the columns. The other columns must be the observation variables, or *features*. In this fashion, you have one column per *feature* and one line per *visit*.

2.2.2 Data scale & constraints

Leaspy use *linear* and *logisitic* models. The features **MUST** be increasing with time. For the *logisitic* model, you need to rescale your data between 0 and 1.

2.2.3 Missing data

Leaspy automatically handle missing data. However, they **MUST** be encoded as `numpy.nan` in your *pandas.DataFrame*.

2.3 Going further

You can check the *User guide* and the *full API documentation*.

CHAPTER

THREE

API DOCUMENTATION

Full API documentation of the *Leaspy* Python package.

3.1 `leaspy.api`: Main API

The main class, from which you can instantiate and calibrate a model, personalize it to a given set a subjects, estimate trajectories and simulate synthetic data.

Leaspy(model_name, **kwargs)	Main API used to fit models, run algorithms and simulations.

3.1.1 `leaspy.api`.Leaspy

class leaspy.api.**Leaspy**(*model_name*, ***kwargs*)

Bases: object

Main API used to fit models, run algorithms and simulations. This is the main class of the Leaspy package.

Parameters

model_name: str The name of the model that will be used for the computations. The available models are:

- `'logistic'` - suppose that every modality follow a logistic curve across time. This model performs a dimensionality reduction of the modalities.
- `'logistic_parallel'` - idem & suppose also that every modality have the same slope at inflexion point
- `'linear'` - suppose that every modality follow a linear curve across time. This model performs a dimensionality reduction of the modalities.
- `'univariate_logisitic'` - a 'logistic' model for a single modality => do not perform a dimensionality reduction.
- `'univariate_linear'` - idem with a 'linear' model.
- `'constant'` - benchmark model for constant predictions.
- `'lme'` - benchmark model for classical linear mixed-effects model.

****kwargs:** Keyword arguments directly passed to the model for its initialization (through *ModelFactory.model()*). Refer to the corresponding model to know possible arguments.

source_dimension: int, optional *For multivariate models only*. Set the spatial variability degree of freedom. This number MUST BE lower than the number of features. By default, this number is equal to square root of the number of features.

See also:

`leaspy.models`

Attributes

model [*AbstractModel*] Model used for computations, is an instance of *AbstractModel*.

type [str (read-only)] Name of the model - will be one of the names listed above.

plotting [`Plotting`] Main class for visualization.

Methods

calibrate(data, algorithm_settings)	Duplicates of the *fit()* method.
check_if_initialized()	Check if model is initialized.
estimate(timepoints, individual_parameters, *)	Return the model values for individuals characterized by their individual parameters z_i at time-points $(t_{i,j})_j$
fit(data, algorithm_settings)	Estimate the model's parameters θ for a given dataset and a given algorithm.
load(path_to_model_settings)	Instantiate a Leaspy object from json model parameter file or the corresponding dictionary
personalize(data, settings[, return_noise])	From a model, estimate individual parameters for each *ID* of a given dataset.
save(path, **kwargs)	Save Leaspy object as json model parameter file.
simulate(individual_parameters, data, settings)	Generate longitudinal synthetic patients data from a given model, a given collection of individual parameters and some given settings.

__init__(*model_name*, ***kwargs*)
Instantiate a Leaspy class object.

__weakref__
list of weak references to the object (if defined)

calibrate(*data*, *algorithm_settings*)
Duplicates of the *fit()* method.

check_if_initialized()
Check if model is initialized.

Raises

ValueError Raise an error if the model has not been initialized.

estimate(*timepoints*, *individual_parameters*, *, *to_dataframe=None*)
Return the model values for individuals characterized by their individual parameters z_i at time-points $(t_{i,j})_j$

Parameters

timepoints [dictionary {string/int: array_like[numeric]} or `pandas.MultiIndex`] Contains, for each individual, the time-points to estimate. It can be a unique time-point or a list of time-points.

individual_parameters [*IndividualParameters*] Corresponds to the individual parameters of individuals.

to_dataframe [bool or None (default)] Whether to output a dataframe of estimations? If None: default is to be True if and only if timepoints is a *pandas.MultiIndex*

Returns

individual_trajectory [dict or `pandas.DataFrame` (depending on *to_dataframe* flag)] Key: patient indices. Value: `numpy.ndarray` of the estimated value, in the shape (number of timepoints, number of features)

Examples

Given the individual parameters of two subjects, estimate the features of the first at 70, 74 and 80 years old and at 71 and 72 years old for the second.

```
>>> from leaspy.datasets import Loader
>>> leaspy_logistic = Loader.load_leaspy_instance('parkinson-putamen-train')
>>> individual_parameters = Loader.load_individual_parameters('parkinson-
↪putamen-train')
>>> timepoints = {'GS-001': (70, 74, 80), 'GS-002': (71, 72)}
>>> estimations = leaspy_logistic.estimate(timepoints, individual_parameters)
```

fit(*data*, *algorithm_settings*)

Estimate the model's parameters θ for a given dataset and a given algorithm. These model's parameters correspond to the fixed-effects of the mixed-effects model.

Parameters

data [*Data*] Contains the information of the individuals, in particular the time-points $(t_{i,j})$ and the observations $(y_{i,j})$.

algorithm_settings [*AlgorithmSettings*] Contains the algorithm's settings.

Examples

Fit a logistic model on a longitudinal dataset, display the group parameters and plot the group average trajectory.

```
>>> from leaspy import AlgorithmSettings, Data, Leaspy
>>> from leaspy.datasets import Loader
>>> putamen_df = Loader.load_dataset('parkinson-putamen')
>>> data = Data.from_dataframe(putamen_df)
>>> leaspy_logistic = Leaspy('univariate_logistic')
>>> settings = AlgorithmSettings('mcmc_saem', progress_bar=True, seed=0)
>>> leaspy_logistic.fit(data, settings)
 ==> Setting seed to 0
|##################################################|   10000/10000 iterations
The standard deviation of the noise at the end of the calibration is:
0.0213
Calibration took: 30s
>>> print(str(leaspy_logistic.model))
=== MODEL ===
g : tensor([-1.1744])
```

(continues on next page)

(continued from previous page)

```
tau_mean : 68.56787872314453
tau_std : 10.12782096862793
xi_mean : -2.3396952152252197
xi_std : 0.5421289801597595
noise_std : 0.021265486255288124
>>> leaspy_logistic.plotting.average_trajectory()
```

classmethod load(*path_to_model_settings*)

Instantiate a Leaspy object from json model parameter file or the corresponding dictionary

This function can be used to load a pre-trained model.

Parameters

path_to_model_settings: str or dict Path to the model's settings json file or dictionary of model parameters

Returns

Leaspy An instanced Leaspy object with the given population parameters θ.

Examples

Load a univariate logistic pre-trained model.

```
>>> from leaspy import Leaspy
>>> from leaspy.datasets.loader import model_paths
>>> leaspy_logistic = Leaspy.load(model_paths['parkinson-putamen-train'])
>>> print(str(leaspy_logistic.model))
=== MODEL ===
g : tensor([-0.7901])
tau_mean : 64.18125915527344
tau_std : 10.199116706848145
xi_mean : -2.346343994140625
xi_std : 0.5663877129554749
noise_std : 0.021229960024356842
```

personalize(*data*, *settings*, *return_noise=False*)

From a model, estimate individual parameters for each *ID* of a given dataset. These individual parameters correspond to the random-effects $(z_{i,j})$ of the mixed-effects model.

Parameters

data [*Data*] Contains the information of the individuals, in particular the time-points $(t_{i,j})$ and the observations $(y_{i,j})$.

settings [*AlgorithmSettings*] Contains the algorithm's settings.

return_noise: bool (default False) Returns a tuple (individual_parameters, noise_std) if True

Returns

ips [*IndividualParameters*] Contains individual parameters

if return_noise is True: ips : *IndividualParameters* noise_std : `torch.Tensor`

Examples

Compute the individual parameters for a given longitudinal dataset and calibrated model, then display the histogram of the log-acceleration:

```
>>> from leaspy import AlgorithmSettings, Data
>>> from leaspy.datasets import Loader
>>> leaspy_logistic = Loader.load_leaspy_instance('parkinson-putamen-train')
>>> putamen_df = Loader.load_dataset('parkinson-putamen')
>>> data = Data.from_dataframe(putamen_df)
>>> personalize_settings = AlgorithmSettings('scipy_minimize', progress_
↪bar=True, use_jacobian=True, seed=0)
>>> individual_parameters = leaspy_logistic.personalize(data, personalize_
↪settings)
 ==> Setting seed to 0
|##################################################|   200/200 subjects
The standard deviation of the noise at the end of the personalization is:
0.0191
Personalization scipy_minimize took: 5s
>>> ip_df = individual_parameters.to_dataframe()
>>> ip_df[['xi']].hist()
```

save(*path*, ***kwargs*)

Save Leaspy object as json model parameter file.

Parameters

path: str Path to store the model's parameters.

****kwargs** Keyword arguments for json.dump method.

Examples

Load the univariate dataset `'parkinson-putamen'`, calibrate the model & save it:

```
>>> from leaspy import AlgorithmSettings, Data, Leaspy
>>> from leaspy.datasets import Loader
>>> putamen_df = Loader.load_dataset('parkinson-putamen')
>>> data = Data.from_dataframe(putamen_df)
>>> leaspy_logistic = Leaspy('univariate_logistic')
>>> settings = AlgorithmSettings('mcmc_saem', progress_bar=True, seed=0)
>>> leaspy_logistic.fit(data, settings)
 ==> Setting seed to 0
|##################################################|   10000/10000 iterations
The standard deviation of the noise at the end of the calibration is:
0.0213
Calibration took: 30s
>>> leaspy_logistic.save('leaspy-logistic-model_parameters-seed0.json',␣
↪indent=2)
```

simulate(*individual_parameters*, *data*, *settings*)

Generate longitudinal synthetic patients data from a given model, a given collection of individual parameters and some given settings.

This procedure learn the joined distribution of the individual parameters and baseline age of the subjects present in `individual_parameters` and `data` respectively to sample new patients from this joined dis-

tribution. The model is used to compute for each patient their scores from the individual parameters. The number of visits per patients is set in `settings['parameters']['mean_number_of_visits']` and `settings['parameters']['std_number_of_visits']` which are set by default to 6 and 3 respectively.

Parameters

individual_parameters [*IndividualParameters*] Contains the individual parameters.

data [*Data*] Data object

settings [*AlgorithmSettings*] Contains the algorithm's settings.

Returns

simulated_data [Result] Contains the generated individual parameters & the corresponding generated scores.

Notes

To generate a new subject, first we estimate the joined distribution of the individual parameters and the reparametrized baseline ages. Then, we randomly pick a new point from this distribution, which define the individual parameters & baseline age of our new subjects. Then, we generate the timepoints following the baseline age. Then, from the model and the generated timepoints and individual parameters, we compute the corresponding values estimations. Then, we add some gaussian noise to these estimations. The level of noise is, by default, equal to the corresponding `'noise_std'` parameter of the model. You can choose to set your own noise value.

Examples

Use a calibrated model & individual parameters to simulate new subjects similar to the ones you have:

```
>>> from leaspy import AlgorithmSettings, Data
>>> from leaspy.datasets import Loader
>>> putamen_df = Loader.load_dataset('parkinson-putamen-train_and_test')
>>> data = Data.from_dataframe(putamen_df.xs('train', level='SPLIT'))
>>> leaspy_logistic = Loader.load_leaspy_instance('parkinson-putamen-train')
>>> individual_parameters = Loader.load_individual_parameters('parkinson-
↪putamen-train')
>>> simulation_settings = AlgorithmSettings('simulation', seed=0)
>>> simulated_data = leaspy_logistic.simulate(individual_parameters, data,␣
↪simulation_settings)
 ==> Setting seed to 0
>>> print(simulated_data.data.to_dataframe().set_index(['ID', 'TIME']).head())
                                      PUTAMEN
ID                    TIME
Generated_subject_001 63.611107  0.556399
                      64.111107  0.571381
                      64.611107  0.586279
                      65.611107  0.615718
                      66.611107  0.644518
>>> print(simulated_data.get_dataframe_individual_parameters().tail())
                             tau        xi
ID
Generated_subject_096  46.771028 -2.483644
```

(continues on next page)

(continued from previous page)

```
Generated_subject_097  73.189964 -2.513465
Generated_subject_098  57.874967 -2.175362
Generated_subject_099  54.889400 -2.069300
Generated_subject_100  50.046972 -2.259841
```

By default, you have simulate 100 subjects, with an average number of visit at 6 & and standard deviation is the number of visits equal to 3. Let's say you want to simulate 200 subjects, everyone of them having ten visits exactly:

```
>>> simulation_settings = AlgorithmSettings('simulation', seed=0, number_of_
↪subjects=200, \
mean_number_of_visits=10, std_number_of_visits=0)
 ==> Setting seed to 0
>>> simulated_data = leaspy_logistic.simulate(individual_parameters, data,␣
↪simulation_settings)
>>> print(simulated_data.data.to_dataframe().set_index(['ID', 'TIME']).tail())
                                 PUTAMEN
ID                     TIME
Generated_subject_200 72.119949  0.829185
                      73.119949  0.842113
                      74.119949  0.854271
                      75.119949  0.865680
                      76.119949  0.876363
```

By default, the generated subjects are named *'Generated_subject_001'*, *'Generated_subject_002'* and so on. Let's say you want a shorter name, for exemple *'GS-001'*. Furthermore, you want to set the level of noise arround the subject trajectory when generating the observations:

```
>>> simulation_settings = AlgorithmSettings('simulation', seed=0, prefix='GS-',␣
↪noise=.2)
>>> simulated_data = leaspy_logistic.simulate(individual_parameters, data,␣
↪simulation_settings)
 ==> Setting seed to 0
>>> print(simulated_data.get_dataframe_individual_parameters().tail())
              tau        xi
ID
GS-096  46.771028 -2.483644
GS-097  73.189964 -2.513465
GS-098  57.874967 -2.175362
GS-099  54.889400 -2.069300
GS-100  50.046972 -2.259841
```

3.2 `leaspy.algo`: Algorithms

Contains all algorithms used in the package.

abstract_algo.AbstractAlgo()	Abstract class containing common methods for all algorithm classes.
algo_factory.AlgoFactory()	Return the wanted algorithm given its name.

3.2.1 leaspy.algo.abstract_algo.AbstractAlgo

class leaspy.algo.abstract_algo.**AbstractAlgo**

Bases: abc.ABC

Abstract class containing common methods for all algorithm classes. These classes are child classes of *AbstractAlgo*.

Attributes

algo_parameters: dict Contains the algorithm's parameters. These ones are set by a *AlgorithmSettings* class object.

name: str Name of the algorithm.

seed: int, optional Seed used by numpy and torch.

output_manager [FitOutputManager] Optional output manager of the algorithm

Methods

convert_timer(d)	Convert a float representing computation time in seconds to a string giving time in hour, minutes and seconds %h %min %s.
display_progress_bar(iteration, n_iter, suffix)	Display a progression bar while running algorithm, simply based on *sys.stdout*.
load_parameters(parameters)	Update the algorithm's parameters by the ones in the given dictionary.
run(model, dataset)	Main method, run the algorithm.
set_output_manager(output_settings)	Set a FitOutputManager object for the run of the algorithm

__init__()

__weakref__
list of weak references to the object (if defined)

static _initialize_seed(*seed*)
Set numpy and torch seeds and display it (static method).

Notes - numpy seed is needed for reproducibility for the simulation algorithm which use the scipy kernel density estimation function. Indeed, scipy use numpy random seed.

Parameters

seed: int The wanted seed

static convert_timer(*d*)
Convert a float representing computation time in seconds to a string giving time in hour, minutes and seconds %h %min %s.

If less than one hour, do not return hours. If less than a minute, do not return minuts.

Parameters

d: float Computation time

Returns

res: str Time formating in hour, minutes and seconds.

static display_progress_bar(*iteration*, *n_iter*, *suffix*, *n_step_default=50*)
Display a progression bar while running algorithm, simply based on *sys.stdout*.

Parameters

iteration: int Current iteration of the algorithm.

n_iter: int Total iterations' number of the algorithm.

suffix: str

Used to differentiate types of algorithms:

- for fit algorithms: `suffix = 'iterations'`
- for personalization algorithms: `suffix = 'subjects'`.

n_step_default: int, default 50 The size of the progression bar.

load_parameters(*parameters*)
Update the algorithm's parameters by the ones in the given dictionary. The keys in the io which does not belong to the algorithm's parameters keys are ignored.

Parameters

parameters: dict Contains the pairs (key, value) of the wanted parameters

Examples

```
>>> settings = leaspy.io.settings.algorithm_settings.AlgorithmSettings("mcmc_
↪saem")
>>> my_algo = leaspy.algo.fit.tensor_mcmcsaem.TensorMCMCSAEM(settings)
>>> my_algo.algo_parameters
{'n_iter': 10000,
 'n_burn_in_iter': 9000,
 'eps': 0.001,
 'L': 10,
 'sampler_ind': 'Gibbs',
 'sampler_pop': 'Gibbs',
 'annealing': {'do_annealing': False,
  'initial_temperature': 10,
  'n_plateau': 10,
  'n_iter': 200}}
>>> parameters = {'n_iter': 5000, 'n_burn_in_iter': 4000}
>>> my_algo.load_parameters(parameters)
>>> my_algo.algo_parameters
{'n_iter': 5000,
 'n_burn_in_iter': 4000,
 'eps': 0.001,
 'L': 10,
 'sampler_ind': 'Gibbs',
 'sampler_pop': 'Gibbs',
 'annealing': {'do_annealing': False,
  'initial_temperature': 10,
  'n_plateau': 10,
  'n_iter': 200}}
```

abstract run(*model*, *dataset*)
Main method, run the algorithm.

TODO fix proper abstract class

Parameters

model [*AbstractModel*] The used model.

dataset [*Dataset*] Contains all the subjects' observations with corresponding timepoints, in torch format to speed up computations.

Returns

Depends on algorithm class: TODO change?

See also:

AbstractFitAlgo

AbstractPersonalizeAlgo

SimulationAlgorithm

set_output_manager(*output_settings*)
Set a `FitOutputManager` object for the run of the algorithm

Parameters

output_settings [*OutputsSettings*] Contains the logs settings for the computation run (console print periodicity, plot periodicity ...)

Examples

```
>>> from leaspy import AlgorithmSettings
>>> from leaspy.io.settings.outputs_settings import OutputsSettings
>>> from leaspy.algo.fit.tensor_mcmcsaem import TensorMCMCSAEM
>>> algo_settings = AlgorithmSettings("mcmc_saem")
>>> my_algo = TensorMCMCSAEM(algo_settings)
>>> settings = {'path': 'brouillons',
                'console_print_periodicity': 50,
                'plot_periodicity': 100,
                'save_periodicity': 50
                }
>>> my_algo.set_output_manager(OutputsSettings(settings))
```

3.2.2 leaspy.algo.algo_factory.AlgoFactory

class leaspy.algo.algo_factory.**AlgoFactory**
Bases: object

Return the wanted algorithm given its name.

Methods

algo(algorithm_class, settings)	Return the wanted algorithm given its name.

__weakref__
list of weak references to the object (if defined)

classmethod algo(*algorithm_class*, *settings*)
Return the wanted algorithm given its name.

Parameters

algorithm_class: str Task name, used to check if the algorithm within the input *settings* is compatible with this task. Must be one of the following api's name:

- *fit*
- *personalize*
- *simulate*

settings [*AlgorithmSettings*] The algorithm settings.

Returns

algorithm [child class of *AbstractAlgo*] The wanted algorithm if it exists and is compatible with algorithm class.

Raises

ValueError

- if the algorithm class is unknown
- if the algorithm name is unknown / does not belong to the wanted algorithm class

3.2.3 leaspy.algo.fit: Fit algorithms

Algorithms used to calibrate a model.

abstract_fit_algo.AbstractFitAlgo()	Abstract class containing common method for all *fit* algorithm classes.
abstract_mcmc.AbstractFitMCMC(settings)	Abstract class containing common method for all *fit* algorithm classes based on *Monte-Carlo Markov Chains* (MCMC).
tensor_mcmcsaem.TensorMCMCSAEM(settings)	Main algorithm for MCMC-SAEM.

leaspy.algo.fit.abstract_fit_algo.AbstractFitAlgo

class leaspy.algo.fit.abstract_fit_algo.**AbstractFitAlgo**
Bases: *leaspy.algo.abstract_algo.AbstractAlgo*

Abstract class containing common method for all *fit* algorithm classes.

See also:

Leaspy.fit()

Attributes

current_iteration: int, default 0 The number of the current iteration

Inherited attributes From *AbstractAlgo*

Methods

convert_timer(d)	Convert a float representing computation time in seconds to a string giving time in hour, minutes and seconds `%h %min %s`.
display_progress_bar(iteration, n_iter, suffix)	Display a progression bar while running algorithm, simply based on *sys.stdout*.
iteration(dataset, model, realizations)	Update the parameters (abstract method).
load_parameters(parameters)	Update the algorithm's parameters by the ones in the given dictionary.
run(model, dataset)	Main method, run the algorithm.
set_output_manager(output_settings)	Set a `FitOutputManager` object for the run of the algorithm

__init__()

__str__()
Return str(self).

__weakref__
list of weak references to the object (if defined)

abstract _initialize_algo(*dataset*, *model*, *realizations*)
Initialize the fit algorithm (abstract method).

Parameters

dataset [*Dataset*]

model [*AbstractModel*]

realizations [*CollectionRealization*]

static _initialize_seed(*seed*)
Set numpy and `torch` seeds and display it (static method).

Notes - numpy seed is needed for reproducibility for the simulation algorithm which use the scipy kernel density estimation function. Indeed, scipy use numpy random seed.

Parameters

seed: int The wanted seed

_is_burn_in()
Check if current iteration is in burn-in phase.

Returns

bool

_maximization_step(*dataset*, *model*, *realizations*)
Maximization step as in the EM algorithm. In practice parameters are set to current realizations (burn-in phase), or as a barycenter with previous realizations.

Parameters

dataset [*Dataset*]

model [*AbstractModel*]

realizations [*CollectionRealization*]

`static convert_timer`(*d*)

Convert a float representing computation time in seconds to a string giving time in hour, minutes and seconds `%h %min %s`.

If less than one hour, do not return hours. If less than a minute, do not return minuts.

Parameters

d: float Computation time

Returns

res: str Time formating in hour, minutes and seconds.

`static display_progress_bar`(*iteration*, *n_iter*, *suffix*, *n_step_default=50*)

Display a progression bar while running algorithm, simply based on *sys.stdout*.

Parameters

iteration: int Current iteration of the algorithm.

n_iter: int Total iterations' number of the algorithm.

suffix: str

Used to differentiate types of algorithms:

- for fit algorithms: `suffix = 'iterations'`
- for personalization algorithms: `suffix = 'subjects'`.

n_step_default: int, default 50 The size of the progression bar.

`abstract iteration`(*dataset*, *model*, *realizations*)

Update the parameters (abstract method).

Parameters

dataset [*Dataset*] Contains the subjects' obersvations in torch format to speed up computation.

model [*AbstractModel*] The used model.

realizations [*CollectionRealization*] The parameters.

`load_parameters`(*parameters*)

Update the algorithm's parameters by the ones in the given dictionary. The keys in the io which does not belong to the algorithm's parameters keys are ignored.

Parameters

parameters: dict Contains the pairs (key, value) of the wanted parameters

Examples

```
>>> settings = leaspy.io.settings.algorithm_settings.AlgorithmSettings("mcmc_
↪saem")
>>> my_algo = leaspy.algo.fit.tensor_mcmcsaem.TensorMCMCSAEM(settings)
>>> my_algo.algo_parameters
{'n_iter': 10000,
 'n_burn_in_iter': 9000,
 'eps': 0.001,
 'L': 10,
 'sampler_ind': 'Gibbs',
 'sampler_pop': 'Gibbs',
 'annealing': {'do_annealing': False,
  'initial_temperature': 10,
  'n_plateau': 10,
  'n_iter': 200}}
>>> parameters = {'n_iter': 5000, 'n_burn_in_iter': 4000}
>>> my_algo.load_parameters(parameters)
>>> my_algo.algo_parameters
{'n_iter': 5000,
 'n_burn_in_iter': 4000,
 'eps': 0.001,
 'L': 10,
 'sampler_ind': 'Gibbs',
 'sampler_pop': 'Gibbs',
 'annealing': {'do_annealing': False,
  'initial_temperature': 10,
  'n_plateau': 10,
  'n_iter': 200}}
```

run(*model*, *dataset*)

Main method, run the algorithm.

Basically, it initializes the *CollectionRealization* object, updates it using the *iteration* method then returns it.

TODO fix proper abstract class

Parameters

model [*AbstractModel*] The used model.

dataset [*Dataset*] Contains the subjects' observations in torch format to speed up computation.

Returns

realizations [*CollectionRealization*] The optimized parameters.

set_output_manager(*output_settings*)

Set a `FitOutputManager` object for the run of the algorithm

Parameters

output_settings [*OutputsSettings*] Contains the logs settings for the computation run (console print periodicity, plot periodicity ...)

Examples

```
>>> from leaspy import AlgorithmSettings
>>> from leaspy.io.settings.outputs_settings import OutputsSettings
>>> from leaspy.algo.fit.tensor_mcmcsaem import TensorMCMCSAEM
>>> algo_settings = AlgorithmSettings("mcmc_saem")
>>> my_algo = TensorMCMCSAEM(algo_settings)
>>> settings = {'path': 'brouillons',
                'console_print_periodicity': 50,
                'plot_periodicity': 100,
                'save_periodicity': 50
                }
>>> my_algo.set_output_manager(OutputsSettings(settings))
```

leaspy.algo.fit.abstract_mcmc.AbstractFitMCMC

class leaspy.algo.fit.abstract_mcmc.**AbstractFitMCMC**(*settings*)

Bases: *leaspy.algo.fit.abstract_fit_algo.AbstractFitAlgo*

Abstract class containing common method for all *fit* algorithm classes based on *Monte-Carlo Markov Chains* (MCMC).

Parameters

settings [*AlgorithmSettings*] MCMC fit algorithm settings

See also:

algo.samplers

Attributes

samplers [dict[str, *AbstractSampler*]] Dictionary of samplers per each variable

TODO add missing

Methods

convert_timer(d)	Convert a float representing computation time in seconds to a string giving time in hour, minutes and seconds %h %min %s.
display_progress_bar(iteration, n_iter, suffix)	Display a progression bar while running algorithm, simply based on *sys.stdout*.
iteration(data, model, realizations)	MCMC-SAEM iteration.
load_parameters(parameters)	Update the algorithm's parameters by the ones in the given dictionary.
run(model, dataset)	Main method, run the algorithm.
set_output_manager(output_settings)	Set a FitOutputManager object for the run of the algorithm

__init__(*settings*)

__str__()
Return str(self).

__weakref__
list of weak references to the object (if defined)

_initialize_algo(*data*, *model*, *realizations*)
Initialize the samplers, annealing, MCMC toolbox and sufficient statistics.

Parameters

data [*Dataset*]

model [*AbstractModel*]

realizations [*CollectionRealization*]

_initialize_annealing()
Initialize annealing, setting initial temperature and number of iterations.

_initialize_samplers(*model*, *data*)
Instanciate samplers for Gibbs / HMC sampling as a dictionnary samplers {variable_name: sampler}

Parameters

data [*Dataset*]

model [*AbstractModel*]

static _initialize_seed(*seed*)
Set numpy and torch seeds and display it (static method).

Notes - numpy seed is needed for reproducibility for the simulation algorithm which use the scipy kernel density estimation function. Indeed, scipy use numpy random seed.

Parameters

seed: int The wanted seed

_initialize_sufficient_statistics(*data*, *model*, *realizations*)
Initialize the sufficient statistics.

Parameters

data [*Dataset*]

model [*AbstractModel*]

realizations [*CollectionRealization*]

_is_burn_in()
Check if current iteration is in burn-in phase.

Returns

bool

_maximization_step(*dataset*, *model*, *realizations*)
Maximization step as in the EM algorith. In practice parameters are set to current realizations (burn-in phase), or as a barycenter with previous realizations.

Parameters

dataset [*Dataset*]

model [*AbstractModel*]

realizations [*CollectionRealization*]

`_update_temperature()`
Update the temperature according to a plateau annealing scheme.

`static convert_timer(d)`
Convert a float representing computation time in seconds to a string giving time in hour, minutes and seconds `%h %min %s`.

If less than one hour, do not return hours. If less than a minute, do not return minuts.

Parameters

d: float Computation time

Returns

res: str Time formating in hour, minutes and seconds.

`static display_progress_bar(iteration, n_iter, suffix, n_step_default=50)`
Display a progression bar while running algorithm, simply based on *sys.stdout*.

Parameters

iteration: int Current iteration of the algorithm.

n_iter: int Total iterations' number of the algorithm.

suffix: str

Used to differentiate types of algorithms:

- for fit algorithms: `suffix = 'iterations'`
- for personalization algorithms: `suffix = 'subjects'`.

n_step_default: int, default 50 The size of the progression bar.

`iteration(data, model, realizations)`
MCMC-SAEM iteration.

1. Sample : MC sample successively of the populatin and individual variales
2. Maximization step : update model parameters from current population/individual variables values.

Parameters

data [*Dataset*]

model [*AbstractModel*]

realizations [*CollectionRealization*]

`load_parameters(parameters)`
Update the algorithm's parameters by the ones in the given dictionary. The keys in the io which does not belong to the algorithm's parameters keys are ignored.

Parameters

parameters: dict Contains the pairs (key, value) of the wanted parameters

Examples

```
>>> settings = leaspy.io.settings.algorithm_settings.AlgorithmSettings("mcmc_
↪saem")
>>> my_algo = leaspy.algo.fit.tensor_mcmcsaem.TensorMCMCSAEM(settings)
>>> my_algo.algo_parameters
{'n_iter': 10000,
 'n_burn_in_iter': 9000,
 'eps': 0.001,
 'L': 10,
 'sampler_ind': 'Gibbs',
 'sampler_pop': 'Gibbs',
 'annealing': {'do_annealing': False,
  'initial_temperature': 10,
  'n_plateau': 10,
  'n_iter': 200}}
>>> parameters = {'n_iter': 5000, 'n_burn_in_iter': 4000}
>>> my_algo.load_parameters(parameters)
>>> my_algo.algo_parameters
{'n_iter': 5000,
 'n_burn_in_iter': 4000,
 'eps': 0.001,
 'L': 10,
 'sampler_ind': 'Gibbs',
 'sampler_pop': 'Gibbs',
 'annealing': {'do_annealing': False,
  'initial_temperature': 10,
  'n_plateau': 10,
  'n_iter': 200}}
```

run(*model*, *dataset*)

Main method, run the algorithm.

Basically, it initializes the *CollectionRealization* object, updates it using the *iteration* method then returns it.

TODO fix proper abstract class

Parameters

model [*AbstractModel*] The used model.

dataset [*Dataset*] Contains the subjects' observations in torch format to speed up computation.

Returns

realizations [*CollectionRealization*] The optimized parameters.

set_output_manager(*output_settings*)

Set a `FitOutputManager` object for the run of the algorithm

Parameters

output_settings [*OutputsSettings*] Contains the logs settings for the computation run (console print periodicity, plot periodicity ...)

Examples

```
>>> from leaspy import AlgorithmSettings
>>> from leaspy.io.settings.outputs_settings import OutputsSettings
>>> from leaspy.algo.fit.tensor_mcmcsaem import TensorMCMCSAEM
>>> algo_settings = AlgorithmSettings("mcmc_saem")
>>> my_algo = TensorMCMCSAEM(algo_settings)
>>> settings = {'path': 'brouillons',
                'console_print_periodicity': 50,
                'plot_periodicity': 100,
                'save_periodicity': 50
                }
>>> my_algo.set_output_manager(OutputsSettings(settings))
```

leaspy.algo.fit.tensor_mcmcsaem.TensorMCMCSAEM

class leaspy.algo.fit.tensor_mcmcsaem.**TensorMCMCSAEM**(*settings*)

Bases: *leaspy.algo.fit.abstract_mcmc.AbstractFitMCMC*

Main algorithm for MCMC-SAEM.

Methods

convert_timer(d)	Convert a float representing computation time in seconds to a string giving time in hour, minutes and seconds %h %min %s.
display_progress_bar(iteration, n_iter, suffix)	Display a progression bar while running algorithm, simply based on *sys.stdout*.
iteration(data, model, realizations)	MCMC-SAEM iteration.
load_parameters(parameters)	Update the algorithm's parameters by the ones in the given dictionary.
run(model, dataset)	Main method, run the algorithm.
set_output_manager(output_settings)	Set a FitOutputManager object for the run of the algorithm

__init__(*settings*)

__str__()

Return str(self).

__weakref__

list of weak references to the object (if defined)

_initialize_algo(*data*, *model*, *realizations*)

Initialize the samplers, annealing, MCMC toolbox and sufficient statistics.

Parameters

data [*Dataset*]

model [*AbstractModel*]

realizations [*CollectionRealization*]

_initialize_annealing()
Initialize annealing, setting initial temperature and number of iterations.

_initialize_samplers(*model*, *data*)
Instanciate samplers for Gibbs / HMC sampling as a dictionnary samplers {variable_name: sampler}

Parameters

data [*Dataset*]

model [*AbstractModel*]

static _initialize_seed(*seed*)
Set numpy and torch seeds and display it (static method).

Notes - numpy seed is needed for reproducibility for the simulation algorithm which use the scipy kernel density estimation function. Indeed, scipy use numpy random seed.

Parameters

seed: int The wanted seed

_initialize_sufficient_statistics(*data*, *model*, *realizations*)
Initialize the sufficient statistics.

Parameters

data [*Dataset*]

model [*AbstractModel*]

realizations [*CollectionRealization*]

_is_burn_in()
Check if current iteration is in burn-in phase.

Returns

bool

_maximization_step(*dataset*, *model*, *realizations*)
Maximization step as in the EM algorith. In practice parameters are set to current realizations (burn-in phase), or as a barycenter with previous realizations.

Parameters

dataset [*Dataset*]

model [*AbstractModel*]

realizations [*CollectionRealization*]

_update_temperature()
Update the temperature according to a plateau annealing scheme.

static convert_timer(*d*)
Convert a float representing computation time in seconds to a string giving time in hour, minutes and seconds %h %min %s.

If less than one hour, do not return hours. If less than a minute, do not return minuts.

Parameters

d: float Computation time

Returns

res: str Time formating in hour, minutes and seconds.

static display_progress_bar(*iteration*, *n_iter*, *suffix*, *n_step_default=50*)
Display a progression bar while running algorithm, simply based on *sys.stdout*.

Parameters

iteration: int Current iteration of the algorithm.

n_iter: int Total iterations' number of the algorithm.

suffix: str

Used to differentiate types of algorithms:

- for fit algorithms: `suffix = 'iterations'`
- for personalization algorithms: `suffix = 'subjects'`.

n_step_default: int, default 50 The size of the progression bar.

iteration(*data*, *model*, *realizations*)
MCMC-SAEM iteration.

1. Sample : MC sample successively of the populatin and individual variales
2. Maximization step : update model parameters from current population/individual variables values.

Parameters

data [*Dataset*]

model [*AbstractModel*]

realizations [*CollectionRealization*]

load_parameters(*parameters*)
Update the algorithm's parameters by the ones in the given dictionary. The keys in the io which does not belong to the algorithm's parameters keys are ignored.

Parameters

parameters: dict Contains the pairs (key, value) of the wanted parameters

Examples

```
>>> settings = leaspy.io.settings.algorithm_settings.AlgorithmSettings("mcmc_
↪saem")
>>> my_algo = leaspy.algo.fit.tensor_mcmcsaem.TensorMCMCSAEM(settings)
>>> my_algo.algo_parameters
{'n_iter': 10000,
 'n_burn_in_iter': 9000,
 'eps': 0.001,
 'L': 10,
 'sampler_ind': 'Gibbs',
 'sampler_pop': 'Gibbs',
 'annealing': {'do_annealing': False,
  'initial_temperature': 10,
  'n_plateau': 10,
  'n_iter': 200}}
>>> parameters = {'n_iter': 5000, 'n_burn_in_iter': 4000}
>>> my_algo.load_parameters(parameters)
```

(continues on next page)

(continued from previous page)

```
>>> my_algo.algo_parameters
{'n_iter': 5000,
 'n_burn_in_iter': 4000,
 'eps': 0.001,
 'L': 10,
 'sampler_ind': 'Gibbs',
 'sampler_pop': 'Gibbs',
 'annealing': {'do_annealing': False,
  'initial_temperature': 10,
  'n_plateau': 10,
  'n_iter': 200}}
```

run(*model*, *dataset*)

Main method, run the algorithm.

Basically, it initializes the *CollectionRealization* object, updates it using the *iteration* method then returns it.

TODO fix proper abstract class

Parameters

model [*AbstractModel*] The used model.

dataset [*Dataset*] Contains the subjects' observations in torch format to speed up computation.

Returns

realizations [*CollectionRealization*] The optimized parameters.

set_output_manager(*output_settings*)

Set a `FitOutputManager` object for the run of the algorithm

Parameters

output_settings [*OutputsSettings*] Contains the logs settings for the computation run (console print periodicity, plot periodicity ...)

Examples

```
>>> from leaspy import AlgorithmSettings
>>> from leaspy.io.settings.outputs_settings import OutputsSettings
>>> from leaspy.algo.fit.tensor_mcmcsaem import TensorMCMCSAEM
>>> algo_settings = AlgorithmSettings("mcmc_saem")
>>> my_algo = TensorMCMCSAEM(algo_settings)
>>> settings = {'path': 'brouillons',
                'console_print_periodicity': 50,
                'plot_periodicity': 100,
                'save_periodicity': 50
                }
>>> my_algo.set_output_manager(OutputsSettings(settings))
```

3.2.4 leaspy.algo.personalize: Personalization algorithms

Algorithms used to personalize a model to a given set of subjects.

abstract_personalize_algo.AbstractPersonalizeAlgo(...)	Abstract class for *personalize* algorithm.
scipy_minimize.ScipyMinimize(settings)	Gradient descent based algorithm to compute individual parameters, *i.e.* personalize a model to a given set of subjects.

leaspy.algo.personalize.abstract_personalize_algo.AbstractPersonalizeAlgo

class leaspy.algo.personalize.abstract_personalize_algo.**AbstractPersonalizeAlgo**(*settings*)

Bases: *leaspy.algo.abstract_algo.AbstractAlgo*

Abstract class for *personalize* algorithm. Estimation of individual parameters of a given *Data* file with a frozen model (already estimated, or loaded from known parameters).

Parameters

settings [*AlgorithmSettings*] Settings of the algorithm.

See also:

Leaspy.personalize()

Attributes

algo_parameters: dict Algorithm's parameters.

name: str Algorithm's name.

seed: int, optional Algorithm's seed (default None).

loss: str Loss to used during algo

Methods

convert_timer(d)	Convert a float representing computation time in seconds to a string giving time in hour, minutes and seconds `%h %min %s`.
display_progress_bar(iteration, n_iter, suffix)	Display a progression bar while running algorithm, simply based on *sys.stdout*.
load_parameters(parameters)	Update the algorithm's parameters by the ones in the given dictionary.
run(model, data)	Main personalize function, wraps the abstract *_get_individual_parameters()* method.
set_output_manager(output_settings)	Set a `FitOutputManager` object for the run of the algorithm

__init__(*settings*)
Initialize class object from settings object

__weakref__
list of weak references to the object (if defined)

abstract _get_individual_parameters(*model*, *data*)
Estimate individual parameters from a *Dataset*.

Parameters

model [*AbstractModel*] A subclass object of leaspy AbstractModel.

data [*Dataset*] Dataset object build with leaspy class objects Data, algo & model

Returns

IndividualParameters

static _initialize_seed(*seed*)
Set numpy and torch seeds and display it (static method).

Notes - numpy seed is needed for reproducibility for the simulation algorithm which use the scipy kernel density estimation function. Indeed, scipy use numpy random seed.

Parameters

seed: int The wanted seed

static convert_timer(*d*)
Convert a float representing computation time in seconds to a string giving time in hour, minutes and seconds `%h %min %s`.

If less than one hour, do not return hours. If less than a minute, do not return minuts.

Parameters

d: float Computation time

Returns

res: str Time formating in hour, minutes and seconds.

static display_progress_bar(*iteration*, *n_iter*, *suffix*, *n_step_default=50*)
Display a progression bar while running algorithm, simply based on *sys.stdout*.

Parameters

iteration: int Current iteration of the algorithm.

n_iter: int Total iterations' number of the algorithm.

suffix: str

Used to differentiate types of algorithms:

- for fit algorithms: `suffix = 'iterations'`
- for personalization algorithms: `suffix = 'subjects'`.

n_step_default: int, default 50 The size of the progression bar.

load_parameters(*parameters*)
Update the algorithm's parameters by the ones in the given dictionary. The keys in the io which does not belong to the algorithm's parameters keys are ignored.

Parameters

parameters: dict Contains the pairs (key, value) of the wanted parameters

Examples

```
>>> settings = leaspy.io.settings.algorithm_settings.AlgorithmSettings("mcmc_
↪saem")
>>> my_algo = leaspy.algo.fit.tensor_mcmcsaem.TensorMCMCSAEM(settings)
>>> my_algo.algo_parameters
{'n_iter': 10000,
 'n_burn_in_iter': 9000,
 'eps': 0.001,
 'L': 10,
 'sampler_ind': 'Gibbs',
 'sampler_pop': 'Gibbs',
 'annealing': {'do_annealing': False,
  'initial_temperature': 10,
  'n_plateau': 10,
  'n_iter': 200}}
>>> parameters = {'n_iter': 5000, 'n_burn_in_iter': 4000}
>>> my_algo.load_parameters(parameters)
>>> my_algo.algo_parameters
{'n_iter': 5000,
 'n_burn_in_iter': 4000,
 'eps': 0.001,
 'L': 10,
 'sampler_ind': 'Gibbs',
 'sampler_pop': 'Gibbs',
 'annealing': {'do_annealing': False,
  'initial_temperature': 10,
  'n_plateau': 10,
  'n_iter': 200}}
```

run(*model*, *data*)

Main personalize function, wraps the abstract *_get_individual_parameters()* method.

Parameters

model [*AbstractModel*] A subclass object of leaspy *AbstractModel*.

data [*Dataset*] Dataset object build with leaspy class objects Data, algo & model

Returns

individual_parameters [*IndividualParameters*] Contains individual parameters.

noise_std: float or `torch.FloatTensor` The estimated noise (is a tensor if `'diag_noise'` in *model.loss*)

$$= \frac{1}{n_{visits} \times n_{dim}} \sqrt{\sum_{i,j \in [1, n_{visits}] \times [1, n_{dim}]} \varepsilon_{i,j}}$$

where $\varepsilon_{i,j} = \left(f(\theta, (z_{i,j}), (t_{i,j})) - (y_{i,j})\right)^2$, where θ are the model's fixed effect, $(z_{i,j})$ the model's random effects, $(t_{i,j})$ the time-points and f the model's estimator.

set_output_manager(*output_settings*)

Set a `FitOutputManager` object for the run of the algorithm

Parameters

output_settings [*OutputsSettings*] Contains the logs settings for the computation run (console print periodicity, plot periodicity ...)

Examples

```
>>> from leaspy import AlgorithmSettings
>>> from leaspy.io.settings.outputs_settings import OutputsSettings
>>> from leaspy.algo.fit.tensor_mcmcsaem import TensorMCMCSAEM
>>> algo_settings = AlgorithmSettings("mcmc_saem")
>>> my_algo = TensorMCMCSAEM(algo_settings)
>>> settings = {'path': 'brouillons',
                'console_print_periodicity': 50,
                'plot_periodicity': 100,
                'save_periodicity': 50
                }
>>> my_algo.set_output_manager(OutputsSettings(settings))
```

leaspy.algo.personalize.scipy_minimize.ScipyMinimize

class leaspy.algo.personalize.scipy_minimize.**ScipyMinimize**(*settings*)

Bases: *leaspy.algo.personalize.abstract_personalize_algo.AbstractPersonalizeAlgo*

Gradient descent based algorithm to compute individual parameters, *i.e.* personalize a model to a given set of subjects.

Methods

convert_timer(d)	Convert a float representing computation time in seconds to a string giving time in hour, minutes and seconds %h %min %s.
display_progress_bar(iteration, n_iter, suffix)	Display a progression bar while running algorithm, simply based on *sys.stdout*.
load_parameters(parameters)	Update the algorithm's parameters by the ones in the given dictionary.
obj(x, *args)	Objective loss function to minimize in order to get patient's individual parameters
run(model, data)	Main personalize function, wraps the abstract *_get_individual_parameters()* method.
set_output_manager(output_settings)	Set a FitOutputManager object for the run of the algorithm

__init__(*settings*)

Initialize class object from settings object

__weakref__

list of weak references to the object (if defined)

_get_individual_parameters(*model*, *data*)

Compute individual parameters of all patients given a leaspy model & a leaspy dataset.

Parameters

model [*AbstractModel*] Model used to compute the group average parameters.

data [*Dataset* class object] Contains the individual scores.

Returns

IndividualParameters Contains the individual parameters of all patients.

_get_individual_parameters_patient(*model*, *times*, *values*, *, *patient_id=None*)

Compute the individual parameter by minimizing the objective loss function with scipy solver.

Parameters

model [*AbstractModel*] Model used to compute the group average parameters.

times [torch.Tensor [n_tpts]] Contains the individual ages corresponding to the given `values`.

values [torch.Tensor [n_tpts, n_fts]] Contains the individual true scores corresponding to the given `times`.

patient_id [str (or None)] ID of patient (essentially here for logging purposes when no convergence)

Returns

individual parameters [dict[str, torch.Tensor [1,n_dims_param]]] Individual parameters as a dict of tensors.

reconstruction error [torch.Tensor [n_tpts, n_features]] Model values minus real values.

_get_individual_parameters_patient_master(*it*, *data*, *model*, *, *patient_id=None*)

Compute individual parameters of all patients given a leaspy model & a leaspy dataset.

Parameters

it: int The iteration number.

model [*AbstractModel*] Model used to compute the group average parameters.

data [*Dataset*] Contains the individual scores.

Returns

IndividualParameters Contains the individual parameters of all patients.

_get_normalized_grad_tensor_from_grad_dict(*dict_grad_tensors*, *model*)

From a dict of gradient tensors per param (without normalization), returns the full tensor of gradients (= for all params, consecutively):

- concatenated with conventional order of x0
- normalized because we derive w.r.t. "standardized" parameter (adimensional gradient)

_get_reconstruction_error(*model*, *times*, *values*, *individual_parameters*)

Compute model values minus real values of a patient for a given model, timepoints, real values & individual parameters.

Parameters

model [*AbstractModel*] Model used to compute the group average parameters.

times [torch.Tensor [n_tpts]] Contains the individual ages corresponding to the given `values`.

values [torch.Tensor [n_tpts,n_fts]] Contains the individual true scores corresponding to the given `times`.

individual_parameters [dict[str, torch.Tensor [1,n_dims_param]]] Individual parameters as a dict

Returns

torch.Tensor [n_tpts,n_fts] Model values minus real values.

_get_regularity(*model*, *individual_parameters*)

Compute the regularity of a patient given his individual parameters for a given model.

Parameters

model [*AbstractModel*] Model used to compute the group average parameters.

individual_parameters [dict[str, torch.Tensor [n_ind,n_dims_param]]] Individual parameters as a dict

Returns

regularity [torch.Tensor [n_individuals]] Regularity of the patient(s) corresponding to the given individual parameters. (Sum on all parameters)

regularity_grads [dict[param_name: str, torch.Tensor [n_individuals, n_dims_param]]] Gradient of regularity term with respect to individual parameters.

_initialize_parameters(*model*)

Initialize individual parameters of one patient with group average parameter.

`x = [xi_mean/xi_std, tau_mean/tau_std] (+ [0.] * n_sources if multivariate model)`

Parameters

model [*AbstractModel*]

Returns

list [float] The individual **standardized** parameters to start with.

static _initialize_seed(*seed*)

Set numpy and `torch` seeds and display it (static method).

Notes - numpy seed is needed for reproducibility for the simulation algorithm which use the scipy kernel density estimation function. Indeed, scipy use numpy random seed.

Parameters

seed: int The wanted seed

_pull_individual_parameters(*x*, *model*)

Get individual parameters as a dict[param_name: str, torch.Tensor [1,n_dims_param]] from a condensed array-like version of it

(based on the conventional order defined in *_initialize_parameters()*)

static convert_timer(*d*)

Convert a float representing computation time in seconds to a string giving time in hour, minutes and seconds `%h %min %s`.

If less than one hour, do not return hours. If less than a minute, do not return minuts.

Parameters

d: float Computation time

Returns

res: str Time formating in hour, minutes and seconds.

`static display_progress_bar`(*iteration*, *n_iter*, *suffix*, *n_step_default=50*)

Display a progression bar while running algorithm, simply based on *sys.stdout*.

Parameters

iteration: int Current iteration of the algorithm.

n_iter: int Total iterations' number of the algorithm.

suffix: str

Used to differentiate types of algorithms:

- for fit algorithms: `suffix = 'iterations'`
- for personalization algorithms: `suffix = 'subjects'`.

n_step_default: int, default 50 The size of the progression bar.

`load_parameters`(*parameters*)

Update the algorithm's parameters by the ones in the given dictionary. The keys in the io which does not belong to the algorithm's parameters keys are ignored.

Parameters

parameters: dict Contains the pairs (key, value) of the wanted parameters

Examples

```
>>> settings = leaspy.io.settings.algorithm_settings.AlgorithmSettings("mcmc_
↪saem")
>>> my_algo = leaspy.algo.fit.tensor_mcmcsaem.TensorMCMCSAEM(settings)
>>> my_algo.algo_parameters
{'n_iter': 10000,
 'n_burn_in_iter': 9000,
 'eps': 0.001,
 'L': 10,
 'sampler_ind': 'Gibbs',
 'sampler_pop': 'Gibbs',
 'annealing': {'do_annealing': False,
  'initial_temperature': 10,
  'n_plateau': 10,
  'n_iter': 200}}
>>> parameters = {'n_iter': 5000, 'n_burn_in_iter': 4000}
>>> my_algo.load_parameters(parameters)
>>> my_algo.algo_parameters
{'n_iter': 5000,
 'n_burn_in_iter': 4000,
 'eps': 0.001,
 'L': 10,
 'sampler_ind': 'Gibbs',
 'sampler_pop': 'Gibbs',
 'annealing': {'do_annealing': False,
```

(continues on next page)

(continued from previous page)

```
'initial_temperature': 10,
'n_plateau': 10,
'n_iter': 200}}
```

obj(*x*, **args*)

Objective loss function to minimize in order to get patient's individual parameters

Parameters

x: array-like [float] Individual **standardized** parameters At initialization `x = [xi_mean/xi_std, tau_mean/tau_std] (+ [0.] * n_sources if multivariate model)`

args:

- **model** [*AbstractModel*] Model used to compute the group average parameters.
- **timepoints** [`torch.Tensor` [1,n_tpts]] Contains the individual ages corresponding to the given `values`
- **values** [`torch.Tensor` [n_tpts, n_fts]] Contains the individual true scores corresponding to the given `times`.
- **with_gradient: bool** If True: return (objective, gradient_objective) Else: simply return objective

Returns

objective: float Value of the loss function (opposite of log-likelihood).

if with_gradient is True:

2-tuple (as expected by `scipy.optimize.minimize()` when `jac=True`)

- objective: float
- gradient: array-like[float] of length n_dims_params

run(*model*, *data*)

Main personalize function, wraps the abstract *_get_individual_parameters()* method.

Parameters

model [*AbstractModel*] A subclass object of leaspy *AbstractModel*.

data [*Dataset*] Dataset object build with leaspy class objects Data, algo & model

Returns

individual_parameters [*IndividualParameters*] Contains individual parameters.

noise_std: float or `torch.FloatTensor` The estimated noise (is a tensor if `'diag_noise'` in *model.loss*)

$$= \frac{1}{n_{visits} \times n_{dim}} \sqrt{\sum_{i,j \in [1,n_{visits}] \times [1,n_{dim}]} \varepsilon_{i,j}}$$

where $\varepsilon_{i,j} = \left(f(\theta, (z_{i,j}), (t_{i,j})) - (y_{i,j})\right)^2$, where θ are the model's fixed effect, $(z_{i,j})$ the model's random effects, $(t_{i,j})$ the time-points and f the model's estimator.

set_output_manager(*output_settings*)

Set a `FitOutputManager` object for the run of the algorithm

Parameters

output_settings [*OutputsSettings*] Contains the logs settings for the computation run (console print periodicity, plot periodicity ...)

Examples

```
>>> from leaspy import AlgorithmSettings
>>> from leaspy.io.settings.outputs_settings import OutputsSettings
>>> from leaspy.algo.fit.tensor_mcmcsaem import TensorMCMCSAEM
>>> algo_settings = AlgorithmSettings("mcmc_saem")
>>> my_algo = TensorMCMCSAEM(algo_settings)
>>> settings = {'path': 'brouillons',
                'console_print_periodicity': 50,
                'plot_periodicity': 100,
                'save_periodicity': 50
                }
>>> my_algo.set_output_manager(OutputsSettings(settings))
```

3.2.5 leaspy.algo.samplers: Samplers

Samplers used by the algorithms.

abstract_sampler.AbstractSampler(info, ...)	Abstract sampler class.
gibbs_sampler.GibbsSampler(info, n_patients)	Gibbs sampler class.
hmc_sampler.HMCSampler(info, n_patients, eps)	Hamiltonian Monte Carlo sampler class.

leaspy.algo.samplers.abstract_sampler.AbstractSampler

class leaspy.algo.samplers.abstract_sampler.**AbstractSampler**(*info*, *n_patients*)

Bases: object

Abstract sampler class.

Attributes

acceptation_temp [torch.Tensor] Acceptation rate for the sampler in MCMC-SAEM algorithm Keep the history of the last *temp_length* last steps

name: str Name of variable

shape: tuple Shape of variable

temp_length: int Deepness of the history kept in the acceptation rate *acceptation_temp* Length of the *acceptation_temp* torch tensor

__init__(*info*, *n_patients*)

__weakref__
list of weak references to the object (if defined)

_group_metropolis_step(*alpha*)
Compute the acceptance decision (0. for False & 1. for True).

Parameters

alpha [torch.Tensor]

Returns

accepted [torch.Tensor] Acceptance decision (0. or 1.). The logs must be one dimensional (i.e. accepted.ndim = 1)

_metropolis_step(*alpha*)
Compute the Metropolis acceptance decision If better (alpha>=1) : accept If worse (alpha<1) : accept with probability alpha

Parameters

alpha [torch.Tensor]

Returns

int acceptance decision (0 or 1)

_update_acceptation_rate(*accepted*)
Update acceptation rate from history of boolean accepted values for each dimension of each variable (except sources)

Parameters

accepted [torch.Tensor]

leaspy.algo.samplers.gibbs_sampler.GibbsSampler

class leaspy.algo.samplers.gibbs_sampler.**GibbsSampler**(*info*, *n_patients*)
Bases: *leaspy.algo.samplers.abstract_sampler.AbstractSampler*

Gibbs sampler class.

Parameters

info: dict Informations on variable to be sampled

n_patients: int > 0 Number of individual (used for variable with `info['type'] == 'individual'`)

Methods

sample(data, model, realizations, ...)	Sample either as population or individual.

__init__(*info*, *n_patients*)

__weakref__
list of weak references to the object (if defined)

_group_metropolis_step(*alpha*)
Compute the acceptance decision (0. for False & 1. for True).

Parameters

alpha [torch.Tensor]

Returns

accepted [torch.Tensor] Acceptance decision (0. or 1.). The logs must be one dimensional (i.e. accepted.ndim = 1)

_metropolis_step(*alpha*)
Compute the Metropolis acceptance decision If better (alpha>=1) : accept If worse (alpha<1) : accept with probability alpha

Parameters

alpha [torch.Tensor]

Returns

int acceptance decision (0 or 1)

_proposal(*val*)
Proposal value around the current value with sampler standard deviation.

Parameters

val

Returns

value around *val*

_sample_individual_realizations(*data*, *model*, *realizations*, *temperature_inv*)
For each indivual variable, compute current patient-batched attachment and regularity. Propose a new value for the individual variable, and compute new patient-batched attachment and regularity. Do a MH step, keeping if better, or if worse with a probability.

Parameters

data [*Dataset*]

model [*AbstractModel*]

realizations [*CollectionRealization*]

temperature_inv [float > 0]

_sample_population_realizations(*data*, *model*, *realizations*, *temperature_inv*)
For each dimension (1D or 2D) of the population variable, compute current attachment and regularity. Propose a new value for the given dimension of the given population variable, and compute new attachment and regularity. Do a MH step, keeping if better, or if worse with a probability.

Parameters

data [*Dataset*]

model [*AbstractModel*]

realizations [*CollectionRealization*]

temperature_inv [float > 0]

_update_acceptation_rate(*accepted*)
Update acceptation rate from history of boolean accepted values for each dimension of each variable (except sources)

Parameters

accepted [torch.Tensor]

_update_std()
Update standard deviation of sampler according to current frequency of acceptation.

Adaptative std is known to improve sampling performances. Std is increased if frequency of acceptation > 40%, and decreased if <20%, so as to stay close to 30%.

sample(*data*, *model*, *realizations*, *temperature_inv*)
Sample either as population or individual.

Modifies in-place the realizations object.

Parameters

data [*Dataset*]

model [*AbstractModel*]

realizations [*CollectionRealization*]

temperature_inv [float > 0]

leaspy.algo.samplers.hmc_sampler.HMCSampler

class leaspy.algo.samplers.hmc_sampler.**HMCSampler**(*info*, *n_patients*, *eps*)
Bases: *leaspy.algo.samplers.abstract_sampler.AbstractSampler*

Hamiltonian Monte Carlo sampler class.

Attributes

eps: float wanted level of noise (?)

L: int … (?)

name: str Name of the wanted variable to be sampled

std [float] … (?)

type: str = 'pop' for population variable = 'ind' for individual variable

Methods

sample(data, model, realizations, ...)	Sample new realization for a given realization state, dataset, model and temperature

__init__(*info*, *n_patients*, *eps*)

Parameters

info: dict Information concerning the given variable to sample (defined in the leaspy model) Example : v0_infos = {

"name": "v0", "shape": torch.Size([self.dimension]), "type": "population", "rv_type": "multigaussian"

}

n_patients: int Number of patients

eps: float … (noise level prior ?)

__weakref__
list of weak references to the object (if defined)

_compute_U(*realizations*, *data*, *model*, *temperature_inv*)
…

Parameters

realizations

data

model

temperature_inv

Returns

…

_compute_ind_hamiltonian(*model*, *data*, *p*, *realizations*, *temperature_inv*)

…

Parameters

model

data

p

realizations

temperature_inv

Returns

torch.Tensor

_compute_pop_hamiltonian(*model*, *data*, *p*, *realizations*, *temperature_inv*)

…

Parameters

model

data

p

realizations

temperature_inv

Returns

torch.Tensor

_group_metropolis_step(*alpha*)

Compute the acceptance decision (0. for False & 1. for True).

Parameters

alpha [torch.Tensor]

Returns

accepted [torch.Tensor] Acceptance decision (0. or 1.). The logs must be one dimensional (i.e. accepted.ndim = 1)

_initialize_momentum(*old_real*)

Initialization of the momenta given …

Parameters

old_real: … …

Returns

torch.Tensor

_leapfrog_step(*p*, *realizations*, *model*, *data*, *temperature_inv*)

...

Parameters

p

realizations

model

data

temperature_inv

_metropolis_step(*alpha*)

Compute the Metropolis acceptance decision If better (alpha>=1) : accept If worse (alpha<1) : accept with probability alpha

Parameters

alpha [torch.Tensor]

Returns

int acceptance decision (0 or 1)

_proposal(*p*, *realizations*, *model*, *data*, *temperature_inv*)

Compute a proposition for the new state of the given variable

p [torch.Tensor] Current momenta

realizations [*CollectionRealization*] Contain the current state & informations of all the variables of interest

model [*AbstractModel*] Model used by the algorithm

data [*Dataset*] Dataset class object build with leaspy class object Data, model & algo

temperature_inv: float > 0 Inverse of the temperature used in tempered MCMC-SAEM

Returns

torch.Tensor

_sample_individual_realizations(*data*, *model*, *realizations*, *temperature_inv*)

...

Parameters

data

model

realizations

temperature_inv

_sample_pop_realizations(*data*, *model*, *realizations*, *temperature_inv*)

Parameters

data [*Dataset*] Dataset class object build with leaspy class object Data, model & algo

model [*AbstractModel*] Model used by the algorithm

realizations [*CollectionRealization*] Contain the current state & informations of all the variables of interest

temperature_inv: float > 0 Inverse of the temperature used in tempered MCMC-SAEM

_update_acceptation_rate(*accepted*)

Update acceptation rate from history of boolean accepted values for each dimension of each variable (except sources)

Parameters

accepted [torch.Tensor]

_update_p(*p*, *realizations*)

...

Parameters

p

realizations

Returns

bool Always True

sample(*data*, *model*, *realizations*, *temperature_inv*)

Sample new realization for a given realization state, dataset, model and temperature

Parameters

data [*Dataset*] Dataset class object build with leaspy class object Data, model & algo

model [*AbstractModel*] Model used by the algorithm

realizations [*CollectionRealization*] Contain the current state & informations of all the variables of interest

temperature_inv: float > 0 Inverse of the temperature used in tempered MCMC-SAEM

3.2.6 leaspy.algo.simulate: Simulation algorithms

Algorithm to simulate synthetic observations and individual parameters.

simulate.SimulationAlgorithm(settings)	To simulate new data given existing one by learning the individual parameters joined distribution.

leaspy.algo.simulate.simulate.SimulationAlgorithm

class leaspy.algo.simulate.simulate.**SimulationAlgorithm**(*settings*)

Bases: *leaspy.algo.abstract_algo.AbstractAlgo*

To simulate new data given existing one by learning the individual parameters joined distribution.

You can choose to only learn the distribution of a group of patient. To do so, choose the cofactor and the cofactor state of the wanted patient in the settings. For instance, for an Alzheimer's disease patient, you can load a genetic cofactor informative of the APOE4 carriers. Choose cofactor 'genetic' and cofactor_state 'APOE4' to simulate only APOE4 carriers.

Notes

One can choose to set the interval of the reparametrized baseline age of the simulated subjects. By doing so, the baseline age are no more jointly learned with individual parameters. Instead, the baseline ages are derived from the simulated individual parameters and the reparametrized baseline age which is sample from an uniform distribution in the set interval.

By definition, the relation between age and reparametrized age is:

$$\psi_i(t) = e^{\xi_i}(t - \tau_i) + \bar{\tau}$$

with t the real age, $\psi_i(t)$ the reparametrized age, ξ_i the individual log-acceleration parameter, τ_i the individual time-shift parameter and $\bar{\tau}$ the mean conversion age derivated by the *model* object.

Attributes

algo_parameters [dict] Contains the algorithm's parameters.

bandwidth_method [float or str or callable, optional] Bandwidth argument used in scipy.stats.gaussian_kde in order to learn the patients' distribution.

cofactor [str, optional (default = None)] The cofactor used to select the wanted group of patients (ex - 'genes'). It must correspond to an existing cofactor in the attribute *Data* of the input *result* of the *run()* method.

cofactor_state [str, optional (default None)] The cofactor state used to select the wanted group of patients (ex - 'APOE4'). It must correspond to an existing cofactor state in the attribute *Data* of the input *result* of the *run()* method.

features_bounds [bool or dict[str, (float, float)] (default False)] Specify if the scores of the generated subjects must be bounded. This parameter can express in two way:

- *bool* : the bounds are the maximum and minimum scores observed in the `results.data` object.
- *dict* : the user has to set the min and max bounds for every features. For example: `{'feature1': (score_min, score_max), 'feature2': (score_min, score_max), ...}`

mean_number_of_visits [int (default 6)] Average number of visits of the simulated patients. Examples - choose 5 => in average, a simulated patient will have 5 visits.

name `['simulation']` Algorithm's name.

noise ['default' (default) or float or array-like[float], optional]

Wanted level of gaussian noise in the generated scores.

- Set to `'default'`, the noise added to each feature score correspond to the reconstruction error for each feature (MSE on all visits, per feature).
- Set noise to `None` will lead to patients having "perfect progression" of their scores, i.e. following exactly a logistic curve.
- Set a float will add for each feature's scores a noise of standard deviation the given float.
- Set an array-like[float] (1D of length *n_features*) will add for the feature *j* a noise of standard deviation `noise[j]`.

number_of_subjects [int] Number of subject to simulate.

reparametrized_age_bounds [tuple[float, float], optional (default None)] Set the minimum and maximum age of the generated reparametrized subjects' ages. See Notes section. Example - reparametrized_age_bounds = (65, 70)

seed [int] Used by `numpy.random` & `torch.random` for reproducibility.

sources_method [str in {'full_kde', 'normal_sources'}]

- `'full_kde'` : the sources are also learned with the gaussian kernel density estimation.
- `'normal_sources'` : the sources are generated as multivariate normal distribution linked with the other individual parameters.

std_number_of_visits [int] Standard deviation used into the generation of the number of visits per simulated patient.

Methods

convert_timer(d)	Convert a float representing computation time in seconds to a string giving time in hour, minutes and seconds `%h %min %s`.
display_progress_bar(iteration, n_iter, suffix)	Display a progression bar while running algorithm, simply based on *sys.stdout*.
load_parameters(parameters)	Update the algorithm's parameters by the ones in the given dictionary.
run(model, individual_parameters, data)	Run simulation - learn joined distribution of patients' individual parameters and return a results object containing the simulated individual parameters and the simulated scores.
set_output_manager(output_settings)	Set a `FitOutputManager` object for the run of the algorithm

__init__(*settings*)
Process initializer function that is called by Leaspy().simulate.

Parameters

settings [*AlgorithmSettings*] Set the class attributes.

Raises

ValueError If `settings.parameters['sources_method']` is not one of the two option allowed - "full_kde" or "normal_sources".

TypeError If the type of `settings.parameters['features_bounds']` is not *bool* or *dict*.

__weakref__
list of weak references to the object (if defined)

_check_cofactors(*data*)
Check the value.

Parameters

data [*Data*] Contains the cofactors and cofactors' states.

Raises

ValueError Raised if the parameters "cofactor" and "cofactor_state" do not receive a valid value.

`static _get_bounded_subject(features_values, features_min, features_max)`
Select the subject whose scores are within the features boundaries.

Parameters

features_values [list [numpy.ndarray]] Contains the scores of all the subjects of all their visits. Each element correspond to a simulated subject, these elements are of shape n_vists x n_features.

features_min [numpy.ndarray] Lowest score allowed per feature - sorted accordingly to the features in `result.data.headers`.

features_max [numpy.ndarray] Highest score allowed per feature - sorted accordingly to the features in `result.data.headers`.

Returns

list [int] Indices of accepted simulated subjects

list [numpy.ndarray] Contains the scores of all the subjects whose scores are within the features boundaries.

`_get_features_bounds(results_object)`
Get the bound of the baseline scores of the generated patients. Each generated patient whose baseline is outside these bounds are discarded.

Parameters

results_object [Result]

Returns

features_min [numpy.ndarray] Lowest score allowed per feature - sorted accordingly to the features in `result.data.headers`.

features_max [numpy.ndarray] Highest score allowed per feature - sorted accordingly to the features in `result.data.headers`.

`static _get_mean_and_covariance_matrix(m)`
Compute the empirical mean and covariance matrix of the input. Twice faster than *numpy.cov*.

Parameters

m [torch.Tensor, shape = (n_individual_parameters, n_subjects)] Input matrix - one row per individual parameter distribution (xi, tau etc).

Returns

mean [torch.Tensor] Mean by variable, shape = (n_individual_parameters,).

covariance [torch.Tensor] Covariance matrix, shape = (n_individual_parameters, n_individual_parameters).

`_get_noise_generator(model, results)`
Compute the level of L2 error per feature and return a noise generator or size n_features.

Parameters

model [*AbstractModel*] Subclass object of *AbstractModel*.

results [Result] Object containing the computed individual parameters.

Returns

torch.distributions.Normal or None A gaussian noise generator. If self.noise is None, the function returns None.

Raises

ValueError If the attribute self.noise is an iterable of float of a length different than the number of features.

`_get_number_of_visits()`
Simulate number of visits for a new simulated patient based of attributes 'mean_number_of_visits' & 'std_number_of_visits'.

Returns

number_of_visits [int] Number of visits.

`static _get_real_age(repam_ages, tau, xi, tau_mean)`
Returns the subjects' real ages.

Parameters

repam_ages [numpy.ndarray, shape = (n_subjects,)] Reparametrized ages of the subjects.

tau [numpy.ndarray, shape = (n_subjects,)] Individual time-shifts.

xi [numpy.ndarray, shape = (n_subjects,)] Individual log-acceleration.

tau_mean [float] The mean conversion age derivated by the model.

Returns

numpy.ndarray, shape = (n_subjects,)

`static _get_reparametrized_age(timepoints, tau, xi, tau_mean)`
Returns the subjects' reparametrized ages.

Parameters

timepoints [numpy.ndarray, shape = (n_subjects,)] Real ages of the subjects.

tau [numpy.ndarray, shape = (n_subjects,)] Individual time-shifts.

xi [numpy.ndarray, shape = (n_subjects,)] Individual log-acceleration.

tau_mean [float] The mean conversion age derivated by the model.

Returns

numpy.ndarray, shape = (n_subjects,)

`_get_timepoints(bl)`
Generate the time points of a subject given his baseline age.

Parameters

bl [float] The subject's baseline age.

Returns

ages [list [float]] Contains the subject's time points.

`static _initialize_seed(seed)`
Set numpy and torch seeds and display it (static method).

Notes - numpy seed is needed for reproducibility for the simulation algorithm which use the scipy kernel density estimation function. Indeed, scipy use numpy random seed.

Parameters

seed: int The wanted seed

static _sample_sources(*bl*, *tau*, *xi*, *source_dimension*, *df_mean*, *df_cov*)
Simulate individual sources given baseline age bl, time-shift tau, log-acceleration xi & sources dimension.

Parameters

bl [float] Baseline age of the simulated patient.

tau [float] Time-shift of the simulated patient.

xi [float] Log-acceleration of the simulated patient.

source_dimension [int] Sources' dimension of the simulated patient.

df_mean [torch.Tensor, shape = (n_individual_parameters,)] Mean values per individual parameter type (bl_mean, tau_mean, xi_mean & sources_means) (1-dimensional).

df_cov [:class:`torch.Tensor`r, shape = (n_individual_parameters, n_individual_parameters)] Empirical covariance matrix of the individual parameters (2-dimensional).

Returns

t:class:*torch.Tensor* Sources of the simulated patient, shape = (n_sources,).

_simulate_individual_parameters(*model*, *number_of_simulated_subjects*, *kernel*, *ss*, *df_mean*, *df_cov*)
Compute the simulated individual parameters and timepoints.

Parameters

model [*AbstractModel*] A subclass object of leaspy *AbstractModel*.

number_of_simulated_subjects [int]

kernel [scipy.stats.gaussian_kde]

ss [sklearn.preprocessing.StandardScaler]

df_mean [torch.Tensor, shape = (n_individual_parameters,)] Mean values per individual parameter type.

df_cov [torch.Tensor, shape = (n_individual_parameters, n_individual_parameters)] Empirical covariance matrix of the individual parameters.

Returns

simulated_parameters [dict [str, numpy.ndarray]] Contains the simulated parameters.

timepoints [list [float]] Contains the ages of the subjects for all their visits - 2D list with one row per simulated subject.

static _simulate_subjects(*simulated_parameters*, *timepoints*, *model*, *noise_generator*)
Compute the simulated scores given the simulated individual parameters, timepoints & noise generator.

Parameters

model [*AbstractModel*] A subclass object of leaspy *AbstractModel*.

simulated_parameters [dict [str, numpy.ndarray]] Contains the simulated parameters.

timepoints [list [float]] Contains the ages of the subjects for all their visits - 2D list with one row per simulated subject.

noise_generator [`torch.distributions.Normal`, optional] A gaussian noise generator. If self.noise is None, the features' score are exactly the ones derived from the individual parameters by the model.

Returns

features_values [list [`numpy.ndarray`]] Contains the scores of all the subjects for all their visits. One entry per subject, each of them is a 2D *numpy.ndarray* of shape (n_visits, n_features).

`static convert_timer`(*d*)

Convert a float representing computation time in seconds to a string giving time in hour, minutes and seconds `%h %min %s`.

If less than one hour, do not return hours. If less than a minute, do not return minuts.

Parameters

d: float Computation time

Returns

res: str Time formating in hour, minutes and seconds.

`static display_progress_bar`(*iteration*, *n_iter*, *suffix*, *n_step_default=50*)

Display a progression bar while running algorithm, simply based on *sys.stdout*.

Parameters

iteration: int Current iteration of the algorithm.

n_iter: int Total iterations' number of the algorithm.

suffix: str

Used to differentiate types of algorithms:

- for fit algorithms: `suffix = 'iterations'`
- for personalization algorithms: `suffix = 'subjects'`.

n_step_default: int, default 50 The size of the progression bar.

`load_parameters`(*parameters*)

Update the algorithm's parameters by the ones in the given dictionary. The keys in the io which does not belong to the algorithm's parameters keys are ignored.

Parameters

parameters: dict Contains the pairs (key, value) of the wanted parameters

Examples

```
>>> settings = leaspy.io.settings.algorithm_settings.AlgorithmSettings("mcmc_
↪saem")
>>> my_algo = leaspy.algo.fit.tensor_mcmcsaem.TensorMCMCSAEM(settings)
>>> my_algo.algo_parameters
{'n_iter': 10000,
 'n_burn_in_iter': 9000,
 'eps': 0.001,
 'L': 10,
 'sampler_ind': 'Gibbs',
```

(continues on next page)

(continued from previous page)

```
 'sampler_pop': 'Gibbs',
 'annealing': {'do_annealing': False,
  'initial_temperature': 10,
  'n_plateau': 10,
  'n_iter': 200}}
>>> parameters = {'n_iter': 5000, 'n_burn_in_iter': 4000}
>>> my_algo.load_parameters(parameters)
>>> my_algo.algo_parameters
{'n_iter': 5000,
 'n_burn_in_iter': 4000,
 'eps': 0.001,
 'L': 10,
 'sampler_ind': 'Gibbs',
 'sampler_pop': 'Gibbs',
 'annealing': {'do_annealing': False,
  'initial_temperature': 10,
  'n_plateau': 10,
  'n_iter': 200}}
```

run(*model*, *individual_parameters*, *data*)

Run simulation - learn joined distribution of patients' individual parameters and return a results object containing the simulated individual parameters and the simulated scores.

Parameters

model [*AbstractModel*] Subclass object of *AbstractModel*. Model used to compute the population & individual parameters. It contains the population parameters.

individual_parameters [*IndividualParameters*] Object containing the computed individual parameters.

Returns

Result Contains the simulated individual parameters & individual scores.

Notes

In simulation_settings, one can specify in the parameters the cofactor & cofactor_state. By doing so, one can simulate based only on the subject for the given cofactor & cofactor's state.

By default, all the subject in results.data are used to estimate the joined distribution.

set_output_manager(*output_settings*)

Set a `FitOutputManager` object for the run of the algorithm

Parameters

output_settings [*OutputsSettings*] Contains the logs settings for the computation run (console print periodicity, plot periodicity ...)

Examples

```
>>> from leaspy import AlgorithmSettings
>>> from leaspy.io.settings.outputs_settings import OutputsSettings
>>> from leaspy.algo.fit.tensor_mcmcsaem import TensorMCMCSAEM
>>> algo_settings = AlgorithmSettings("mcmc_saem")
>>> my_algo = TensorMCMCSAEM(algo_settings)
>>> settings = {'path': 'brouillons',
                'console_print_periodicity': 50,
                'plot_periodicity': 100,
                'save_periodicity': 50
                }
>>> my_algo.set_output_manager(OutputsSettings(settings))
```

3.2.7 `leaspy.algo.others`: Other algorithms

Reference algorithms to use with reference models (for benchmarks).

constant_prediction_algo.ConstantPredictionAlgorithm(...)	Personalization algorithm associated to *ConstantModel*
lme_fit.LMEFitAlgorithm(settings)	Calibration algorithm associated to *LMEModel*
lme_personalize.LMEPersonalizeAlgorithm(settings)	Personalization algorithm associated to *LMEModel*

leaspy.algo.others.constant_prediction_algo.ConstantPredictionAlgorithm

class leaspy.algo.others.constant_prediction_algo.**ConstantPredictionAlgorithm**(*settings*)

Bases: object

Personalization algorithm associated to *ConstantModel*

It could predict:

- *last_known*: last non NaN value seen during calibration*§,
- *last*: last value seen during calibration (even if NaN),
- *max*: maximum (=worst) value seen during calibration*§,
- *mean*: average of values seen during calibration§.

* <!> depending on features, the *last_known* / *max* value may correspond to different visits.

§ <!> for a given feature, value will be NaN if and only if all values for this feature were NaN.

Methods

run(model, dataset)	Main method, refer to abstract definition in *run()*.
set_output_manager(settings)	Not implemented.

__init__(*settings*)
ConstantPredictionAlgorithm is the algorithm that outputs a constant prediction

__weakref__
list of weak references to the object (if defined)

_get_individual_last_values(*times*, *values*)

Parameters

times [numpy.ndarray [float]] shape (n_visits,)

values [numpy.ndarray [float]] shape (n_visits, n_features)

Returns

dict[ft_name: str, constant_value_to_be_padded]

run(*model*, *dataset*)
Main method, refer to abstract definition in *run()*.

TODO fix proper inheritance

Parameters

model [*ConstantModel*] A subclass object of leaspy *ConstantModel*.

dataset [*Dataset*] Dataset object build with leaspy class objects Data, algo & model

Returns

individual_parameters [*IndividualParameters*] Contains individual parameters.

noise_std: float TODO: always 0 for now

set_output_manager(*settings*)
Not implemented.

leaspy.algo.others.lme_fit.LMEFitAlgorithm

class leaspy.algo.others.lme_fit.**LMEFitAlgorithm**(*settings*)
Bases: *leaspy.algo.abstract_algo.AbstractAlgo*

Calibration algorithm associated to *LMEModel*

Parameters

settings [*AlgorithmSettings*]

- **with_random_slope_age** [bool] If False: only varying intercepts If True: random intercept & random slope w.r.t ages
- **force_independent_random_effects** [bool] Force independence of random intercept & random slope
- other keyword arguments passed to statsmodels.regression.mixed_linear_model.MixedLM.fit()

See also:

`statsmodels.regression.mixed_linear_model.MixedLM`

Methods

convert_timer(d)	Convert a float representing computation time in seconds to a string giving time in hour, minutes and seconds `%h %min %s`.
display_progress_bar(iteration, n_iter, suffix)	Display a progression bar while running algorithm, simply based on *sys.stdout*.
load_parameters(parameters)	Update the algorithm's parameters by the ones in the given dictionary.
run(model, dataset)	Main method, refer to abstract definition in *run()*.
set_output_manager(settings)	Not implemented.

__init__(*settings*)

__weakref__
list of weak references to the object (if defined)

static _initialize_seed(*seed*)
Set `numpy` and `torch` seeds and display it (static method).

Notes - numpy seed is needed for reproducibility for the simulation algorithm which use the scipy kernel density estimation function. Indeed, scipy use numpy random seed.

Parameters

seed: int The wanted seed

static convert_timer(*d*)
Convert a float representing computation time in seconds to a string giving time in hour, minutes and seconds `%h %min %s`.

If less than one hour, do not return hours. If less than a minute, do not return minuts.

Parameters

d: float Computation time

Returns

res: str Time formating in hour, minutes and seconds.

static display_progress_bar(*iteration*, *n_iter*, *suffix*, *n_step_default=50*)
Display a progression bar while running algorithm, simply based on *sys.stdout*.

Parameters

iteration: int Current iteration of the algorithm.

n_iter: int Total iterations' number of the algorithm.

suffix: str

Used to differentiate types of algorithms:

- for fit algorithms: `suffix = 'iterations'`

- for personalization algorithms: `suffix = 'subjects'`.

n_step_default: int, default 50 The size of the progression bar.

load_parameters(*parameters*)

Update the algorithm's parameters by the ones in the given dictionary. The keys in the io which does not belong to the algorithm's parameters keys are ignored.

Parameters

parameters: dict Contains the pairs (key, value) of the wanted parameters

Examples

```
>>> settings = leaspy.io.settings.algorithm_settings.AlgorithmSettings("mcmc_
↪saem")
>>> my_algo = leaspy.algo.fit.tensor_mcmcsaem.TensorMCMCSAEM(settings)
>>> my_algo.algo_parameters
{'n_iter': 10000,
 'n_burn_in_iter': 9000,
 'eps': 0.001,
 'L': 10,
 'sampler_ind': 'Gibbs',
 'sampler_pop': 'Gibbs',
 'annealing': {'do_annealing': False,
  'initial_temperature': 10,
  'n_plateau': 10,
  'n_iter': 200}}
>>> parameters = {'n_iter': 5000, 'n_burn_in_iter': 4000}
>>> my_algo.load_parameters(parameters)
>>> my_algo.algo_parameters
{'n_iter': 5000,
 'n_burn_in_iter': 4000,
 'eps': 0.001,
 'L': 10,
 'sampler_ind': 'Gibbs',
 'sampler_pop': 'Gibbs',
 'annealing': {'do_annealing': False,
  'initial_temperature': 10,
  'n_plateau': 10,
  'n_iter': 200}}
```

run(*model*, *dataset*)

Main method, refer to abstract definition in *run()*.

TODO fix proper inheritance

Parameters

model [*LMEModel*] A subclass object of leaspy *LMEModel*.

dataset [*Dataset*] Dataset object build with leaspy class objects Data, algo & model

set_output_manager(*settings*)

Not implemented.

leaspy.algo.others.lme_personalize.LMEPersonalizeAlgorithm

class leaspy.algo.others.lme_personalize.**LMEPersonalizeAlgorithm**(*settings*)

Bases: *leaspy.algo.abstract_algo.AbstractAlgo*

Personalization algorithm associated to *LMEModel*

Attributes

name ['lme_personalize']

Methods

convert_timer(d)	Convert a float representing computation time in seconds to a string giving time in hour, minutes and seconds %h %min %s.
display_progress_bar(iteration, n_iter, suffix)	Display a progression bar while running algorithm, simply based on *sys.stdout*.
load_parameters(parameters)	Update the algorithm's parameters by the ones in the given dictionary.
run(model, dataset)	Main method, refer to abstract definition in *run()*.
set_output_manager(settings)	Not implemented.

__init__(*settings*)

__weakref__

list of weak references to the object (if defined)

static _generic_get_random_effects(*resid*, *Z*, *cov_re_unscaled_inv*)

The conditional means of random effects given the data. cf. http://sia.webpopix.org/lme.html#estimation-of-the-random-effects

Parameters

resid [numpy.ndarray (n_i,)] endog - fixed_effects * exog

Z [numpy.ndarray (n_i, k_re)] exog_re

cov_re_unscaled_inv [numpy.ndarray (k_re, k_re)] inverse

Returns

random_effects [numpy.ndarray (k_re,)] For a given individual

static _initialize_seed(*seed*)

Set numpy and torch seeds and display it (static method).

Notes - numpy seed is needed for reproducibility for the simulation algorithm which use the scipy kernel density estimation function. Indeed, scipy use numpy random seed.

Parameters

seed: int The wanted seed

static convert_timer(*d*)

Convert a float representing computation time in seconds to a string giving time in hour, minutes and seconds %h %min %s.

If less than one hour, do not return hours. If less than a minute, do not return minuts.

Parameters

d: float Computation time

Returns

res: str Time formating in hour, minutes and seconds.

`static display_progress_bar`(*iteration*, *n_iter*, *suffix*, *n_step_default=50*)

Display a progression bar while running algorithm, simply based on *sys.stdout*.

Parameters

iteration: int Current iteration of the algorithm.

n_iter: int Total iterations' number of the algorithm.

suffix: str

Used to differentiate types of algorithms:

- for fit algorithms: `suffix = 'iterations'`
- for personalization algorithms: `suffix = 'subjects'`.

n_step_default: int, default 50 The size of the progression bar.

`load_parameters`(*parameters*)

Update the algorithm's parameters by the ones in the given dictionary. The keys in the io which does not belong to the algorithm's parameters keys are ignored.

Parameters

parameters: dict Contains the pairs (key, value) of the wanted parameters

Examples

```
>>> settings = leaspy.io.settings.algorithm_settings.AlgorithmSettings("mcmc_
↪saem")
>>> my_algo = leaspy.algo.fit.tensor_mcmcsaem.TensorMCMCSAEM(settings)
>>> my_algo.algo_parameters
{'n_iter': 10000,
 'n_burn_in_iter': 9000,
 'eps': 0.001,
 'L': 10,
 'sampler_ind': 'Gibbs',
 'sampler_pop': 'Gibbs',
 'annealing': {'do_annealing': False,
  'initial_temperature': 10,
  'n_plateau': 10,
  'n_iter': 200}}
>>> parameters = {'n_iter': 5000, 'n_burn_in_iter': 4000}
>>> my_algo.load_parameters(parameters)
>>> my_algo.algo_parameters
{'n_iter': 5000,
 'n_burn_in_iter': 4000,
 'eps': 0.001,
 'L': 10,
 'sampler_ind': 'Gibbs',
 'sampler_pop': 'Gibbs',
```

(continues on next page)

(continued from previous page)

```
 'annealing': {'do_annealing': False,
  'initial_temperature': 10,
  'n_plateau': 10,
  'n_iter': 200}}
```

run(*model*, *dataset*)
Main method, refer to abstract definition in *run()*.

TODO fix proper inheritance

Parameters

model [*LMEModel*] A subclass object of leaspy *LMEModel*.

dataset [*Dataset*] Dataset object build with leaspy class objects Data, algo & model

Returns

individual_parameters [*IndividualParameters*] Contains individual parameters.

noise_std: float The estimated noise

set_output_manager(*settings*)
Not implemented.

3.3 leaspy.dataset: Datasets

Give access to some synthetic longitudinal observations mimicking cohort of subjects with neurodegenerative disorders, as well as calibrated models and computed individual parameters.

loader.Loader()	Contains static methods to load synthetic longitudinal dataset, calibrated *Leaspy* instances & *IndividualParameters*.

3.3.1 leaspy.datasets.loader.Loader

class leaspy.datasets.loader.**Loader**
Bases: object

Contains static methods to load synthetic longitudinal dataset, calibrated *Leaspy* instances & *IndividualParameters*.

Notes

- A *Leaspy* instance named <name> have been calibrated on the dataset <name>.
- An *IndividualParameters* name <name> have been computed by personalizing the *Leaspy* instance named <name> on the dataset <name>.

See the documentation of each method to get their respective available names.

Attributes

data_paths: dict [str, str] Contains the datasets' names and their respective path within `leaspy.datasets` subpackage.

model_paths: dict [str, str] Contains the *Leaspy* instances' names and their respective path within `leaspy.datasets` subpackage.

ip_paths: dict [str, str] Contains the individual parameters' names and their respective path within `leaspy.datasets` subpackage.

Methods

load_dataset(dataset_name)	Load synthetic longitudinal observations mimicking cohort of subjects with neurodegenerative disorders.
load_individual_parameters(ip_name)	Load a Leaspy instance with a model allready calibrated on the synthetic dataset corresponding to the name of the instance.
load_leaspy_instance(instance_name)	Load a Leaspy instance with a model allready calibrated on the synthetic dataset corresponding to the name of the instance.

__init__()

__weakref__
list of weak references to the object (if defined)

static load_dataset(*dataset_name*)
Load synthetic longitudinal observations mimicking cohort of subjects with neurodegenerative disorders.

Parameters

dataset_name: {'parkinson-multivariate', 'alzheimer-multivariate', 'parkinson-putamen', 'parkinson-puta
Name of the dataset.

Returns

`pandas.DataFrame` DataFrame containing the IDs, timepoints and observations.

Notes

All *DataFrames* have the same structures.

- Index: a `pandas.MultiIndex` - `['ID', 'TIME']` which contain IDs and timepoints. The *DataFrame* is sorted by index. So, one line corresponds to one visit for one subject. The *DataFrame* having *'train_and_test'* in their name also have `'SPLIT'` as the third index level. It differenciate *train* and *test* data.
- Columns: One column correspond to one feature (or score).

static load_individual_parameters(*ip_name*)
Load a Leaspy instance with a model allready calibrated on the synthetic dataset corresponding to the name of the instance.

Parameters

ip_name: {'parkinson-putamen-train'} Name of the individual parameters.

Returns

IndividualParameters Leaspy instance with a model allready calibrated.

static load_leaspy_instance(*instance_name*)
Load a Leaspy instance with a model allready calibrated on the synthetic dataset corresponding to the name of the instance.

Parameters

instance_name: {'parkinson-putamen-train'} Name of the instance.

Returns

Leaspy Leaspy instance with a model allready calibrated.

3.4 leaspy.io: Inputs / Outputs

Containers class objects used as input / ouputs in the *Leaspy* package.

3.4.1 leaspy.io.data: Data containers

data.Data()	Main data container, initialized from a *csv file* or a pandas.DataFrame.
dataset.Dataset(data[, model, algo])	Data container based on torch.Tensor, used to run algorithms.

leaspy.io.data.data.Data

class leaspy.io.data.data.**Data**
Bases: object

Main data container, initialized from a *csv file* or a pandas.DataFrame.

Methods

from_csv_file(path)	Create a *Data* object from a CSV file.
from_dataframe(df)	Create a *Data* object from a pandas.DataFrame.
from_individuals(indices, timepoints, ...)	Create a *Data* class object from lists of *ID*, *timepoints* and the corresponding *values*.
get_by_idx(idx)	Get the IndividualData of a an individual identified by its ID.
load_cofactors(df, cofactors)	Load cofactors from a *pandas.DataFrame* to the *Data* object
to_dataframe([cofactors])	Return the subjects' observations in a pandas.DataFrame along their ID and ages at all visits.

__init__()

__weakref__
list of weak references to the object (if defined)

static from_csv_file(*path*)
Create a *Data* object from a CSV file.

Returns

Data

`static from_dataframe`(*df*)
Create a *Data* object from a `pandas.DataFrame`.

Returns

Data

`static from_individuals`(*indices*, *timepoints*, *values*, *headers*)
Create a *Data* class object from lists of *ID*, *timepoints* and the corresponding *values*.

Parameters

indices: list[str] Contains the individuals' ID.

timepoints: list[array-like 1D] For each individual `i`, list of ages at visits. Number of timepoints is refered below as `n_timepoints_i`

values: list[array-like 2D] For each individual `i`, all values at visits. Shape is `(n_timepoints_i, n_features)`.

headers: list[str] Contains the features' names.

Returns

Data Data class object with all ID, timepoints, values and feautures' names.

`get_by_idx`(*idx*)
Get the `IndividualData` of a an individual identified by its ID.

Returns

`IndividualData`

`load_cofactors`(*df*, *cofactors*)
Load cofactors from a *pandas.DataFrame* to the *Data* object

Parameters

df [`pandas.DataFrame`] the index is the list of subject ids

cofactors: list[str] names of the column(s) of df which shall be loaded as cofactors

`to_dataframe`(*cofactors=None*)
Return the subjects' observations in a `pandas.DataFrame` along their ID and ages at all visits.

Parameters

cofactors [str, list [str], optional (default None)] Contains the cofactors' names to be included in the DataFrame. By default, no cofactors are returned. If cofactors == "all", all the available cofactors are returned.

Returns

`pandas.DataFrame` Contains the subjects's ID, age and scores (optional - and cofactors) for each timepoint.

leaspy.io.data.dataset.Dataset

class leaspy.io.data.dataset.**Dataset**(*data*, *model=None*, *algo=None*)
Bases: object

Data container based on torch.Tensor, used to run algorithms.

Parameters

data [*Data*] Create *Dataset* from *Data* object

model [*AbstractModel* (optional)] If not None, will check compatibility of model and data

algo [*AbstractAlgo* (optional)] If not None, will check compatibility of algo and data

Attributes

headers [list[str]] Features names

dimension [int] Number of features

n_individuals [int] Number of individuals

nb_observations_per_individuals [list[int]] Number of observations per individual

max_observations [int] Maximum number of observation for one individual

n_visits [int] Total number of visits

indices [list[ID]] Order of patients

timepoints [torch.FloatTensor, shape (n_individuals, max_observations)] Ages of patients at their different visits

values [torch.FloatTensor, shape (n_individuals, max_observations, dimension,)] Values of patients for each visit for each feature

mask [torch.FloatTensor, shape (n_individuals, max_observations, dimension,)] Binary mask associated to values. If 1: value is meaningful If 0: value is meaningless (either was nan or does not correspond to a real visit - only here for padding)

L2_norm_per_ft [torch.Tensor, shape (dimension,)] Sum of all non-nan squared values, feature per feature

L2_norm [scalar torch.Tensor] Sum of all non-nan squared values

Methods

get_times_patient(i)	Get ages for patient number i
get_values_patient(i)	Get values for patient number i
to_pandas()	Convert dataset to a *DataFrame*.

__init__(*data*, *model=None*, *algo=None*)

__weakref__
list of weak references to the object (if defined)

get_times_patient(*i*)
Get ages for patient number i

Returns

torch.Tensor, shape (n_obs_of_patient,) Contains float

get_values_patient(*i*)
Get values for patient number i

Returns

torch.Tensor, shape (n_obs_of_patient, dimension) Contains float or nans

to_pandas()
Convert dataset to a *DataFrame*.

Returns

pandas.DataFrame

3.4.2 leaspy.io.outputs: Outputs class objects

individual_parameters.IndividualParameters()	Data container for individual parameters, contains IDs, timepoints and observations values.

leaspy.io.outputs.individual_parameters.IndividualParameters

class leaspy.io.outputs.individual_parameters.**IndividualParameters**
Bases: object

Data container for individual parameters, contains IDs, timepoints and observations values. Ouput of the *Leaspy.personalize()* method, contains the *random effects*.

There are used as ouput of the *personalization algorithms* and as input/ouput of the *simulation algorithm*, to provide an initial distribution of individual parameters.

Attributes

_indices: list List of the patient indices

_individual_parameters: dict Individual indices (key) with their corresponding individual parameters {parameter name: parameter value}

_parameters_shape: dict Shape of each individual parameter

_default_saving_type: str Default extension for saving when none is provided

Methods

add_individual_parameters(index, ...)	Add the individual parameter of an individual to the IndividualParameters object
from_dataframe(df)	Static method that returns an IndividualParameters object from the dataframe
from_pytorch(indices, dict_pytorch)	Static method that returns an IndividualParameters object from the indices and pytorch dictionary
get_mean(parameter)	Returns the mean value of a parameter across all patients
get_std(parameter)	Returns the stardard deviation of a parameter across all patients

continues on next page

Table 28 – continued from previous page

items()	items() method of dict _individual_parameters
load(path)	Static method that loads the individual parameters (json or csv) existing at the path locatio
save(path, **kwargs)	Saves the individual parameters (json or csv) at the path location
subset(indices)	Returns IndividualParameters object with a subset of the initial individuals
to_dataframe()	Returns the dataframe of individual parameters
to_pytorch()	Returns the indices and pytorch dictionary of individual parameters

__init__()

__weakref__
list of weak references to the object (if defined)

add_individual_parameters(*index*, *individual_parameters*)
Add the individual parameter of an individual to the IndividualParameters object

Parameters

index: str Index of the individual

individual_parameters: dict Individual parameters of the individual {name: value:}

Raises

ValueError If the index is not a string or has already been added

Examples

Add two individual with tau, xi and sources parameters

```
>>> ip = IndividualParameters()
>>> ip.add_individual_parameters('index-1', {"xi": 0.1, "tau": 70, "sources":␣
↪[0.1, -0.3]})
>>> ip.add_individual_parameters('index-2', {"xi": 0.2, "tau": 73, "sources":␣
↪[-0.4, -0.1]})
```

static from_dataframe(*df*)
Static method that returns an IndividualParameters object from the dataframe

Parameters

df [pandas.DataFrame] Dataframe of the invidual parameters. Each row must correspond to one individual. The index corresponds to the individual index. The columns are the names of the parameters.

Returns

IndividualParameters

static from_pytorch(*indices*, *dict_pytorch*)
Static method that returns an IndividualParameters object from the indices and pytorch dictionary

Parameters

indices: list[ID] List of the patients indices

dict_pytorch: dict[parameter:str, `torch.Tensor`] Dictionary of the individual parameters

Returns

IndividualParameters

Examples

```
>>> indices = ['index-1', 'index-2', 'index-3']
>>> ip_pytorch = {
>>>     "xi": torch.tensor([[0.1], [0.2], [0.3]], dtype=torch.float32),
>>>     "tau": torch.tensor([[70], [73], [58.]], dtype=torch.float32),
>>>     "sources": torch.tensor([[0.1, -0.3], [-0.4, 0.1], [-0.6, 0.2]],␣
↪dtype=torch.float32)
>>> }
>>> ip_pytorch = IndividualParameters.from_pytorch(indices, ip_pytorch)
```

get_mean(*parameter*)

Returns the mean value of a parameter across all patients

Parameters

parameter: str Name of the parameter

Returns

list or float (depending on parameter shape) Mean value of the parameter

Raises

ValueError If the parameter is not in the IndividualParameters

Examples

```
>>> ip = IndividualParameters.load("path/to/individual_parameters")
>>> tau_mean = ip.get_mean("tau")
```

get_std(*parameter*)

Returns the stardard deviation of a parameter across all patients

Parameters

parameter: str Name of the parameter

Returns

list or float (depending on parameter shape) Standard value of the parameter

Raises

ValueError If the parameter is not in the IndividualParameters

Examples

```
>>> ip = IndividualParameters.load("path/to/individual_parameters")
>>> tau_std = ip.get_std("tau")
```

items()
items() method of dict _individual_parameters

static load(*path*)
Static method that loads the individual parameters (json or csv) existing at the path locatio

Parameters

path: str Path and file name of the individual parameters.

Returns

IndividualParameters Individual parameters object load from the file

Raises

ValueError If the provided extension is not *csv* or not *json*.

Examples

```
>>> ip = IndividualParameters.load('/path/to/individual_parameters_1.json')
>>> ip2 = IndividualParameters.load('/path/to/individual_parameters_2.csv')
```

save(*path*, ***kwargs*)
Saves the individual parameters (json or csv) at the path location

TODO? save leaspy version as well for retro/future-compatibility issues?

Parameters

path: str Path and file name of the individual parameters. The extension can be json or csv. If no extension, default extension (csv) is used

****kwargs:** Additional keyword arguments argument to pass to either: - pandas.to_csv - json.dump depending on saving format requested

subset(*indices*)
Returns IndividualParameters object with a subset of the initial individuals

Parameters

indices: list[ID] List of strings that corresponds to the indices of the individuals to return

Returns

IndividualParameters An instance of the IndividualParameters object with the selected list of individuals

Raises

ValueError Raise an error if one of the index is not in the IndividualParameters

Examples

```
>>> ip = IndividualParameters()
>>> ip.add_individual_parameters('index-1', {"xi": 0.1, "tau": 70, "sources":␣
↪[0.1, -0.3]})
>>> ip.add_individual_parameters('index-2', {"xi": 0.2, "tau": 73, "sources":␣
↪[-0.4, -0.1]})
>>> ip.add_individual_parameters('index-3', {"xi": 0.3, "tau": 58, "sources":␣
↪[-0.6, 0.2]})
>>> ip_sub = ip.subset(['index-1', 'index-3'])
```

to_dataframe()

Returns the dataframe of individual parameters

Returns

pandas.DataFrame Each row corresponds to one individual. The index corresponds to the individual index. The columns are the names of the parameters.

Examples

Convert the individual parameters object into a dataframe

```
>>> ip = IndividualParameters.load("path/to/individual_parameters")
>>> ip_df = ip.to_dataframe()
```

to_pytorch()

Returns the indices and pytorch dictionary of individual parameters

Returns

indices: list[ID] List of patient indices

pytorch_dict: dict[parameter:str, *torch.Tensor*] Dictionary of the individual parameters {parameter name: pytorch tensor of values across individuals}

Examples

Convert the individual parameters object into a dataframe

```
>>> ip = IndividualParameters.load("path/to/individual_parameters")
>>> indices, ip_pytorch = ip.to_pytorch()
```

3.4.3 leaspy.io.realizations: Realizations class objects

realization.Realization(name, shape, ...)	Contains the realization of a given parameter.
collection_realization.CollectionRealization()	Realizations of population and individual parameters.

leaspy.io.realizations.realization.Realization

class leaspy.io.realizations.realization.**Realization**(*name*, *shape*, *variable_type*)

Bases: object

Contains the realization of a given parameter.

Parameters

name: str Variable name

shape: tuple of int Shape of variable (multiple dimensions allowed)

variable_type: str `'individual'` or `'population'` variable?

Attributes

name: str Variable name

shape: tuple of int Shape of variable (multiple dimensions allowed)

variable_type: str `'individual'` or `'population'` variable?

tensor_realizations [torch.Tensor] Actual realizations, whose shape is given by *shape*

Methods

copy()	Copy the Realization object
from_tensor(name, shape, variable_type, ...)	Create realization from variable infos and torch tensor object
initialize(n_individuals, model[, ...])	Initialize realization from a given model.
set_autograd()	Set autograd for tensor of realizations
set_tensor_realizations_element(element, dim)	Manually change the value (in-place) of *tensor_realizations* at dimension *dim*.
unset_autograd()	Unset autograd for tensor of realizations

__init__(*name*, *shape*, *variable_type*)

__str__()
Return str(self).

__weakref__
list of weak references to the object (if defined)

copy()
Copy the Realization object

Notes

From PyTorch `torch.Tensor.clone()` doc: Unlike copy_(), this function is recorded in the computation graph. Gradients propagating to the cloned tensor will propagate to the original tensor.

classmethod from_tensor(*name*, *shape*, *variable_type*, *tensor_realization*)
Create realization from variable infos and torch tensor object

Parameters

name: str Variable name

shape: tuple of int Shape of variable (multiple dimensions allowed)

variable_type: str `'individual'` or `'population'` variable?

tensor_realization [`torch.Tensor`] Actual realizations, whose shape is given by *shape*

Returns

Realization

initialize(*n_individuals*, *model*, *scale_individual=1.0*)
Initialize realization from a given model.

Parameters

n_individuals [int > 0]

model [*AbstractModel*]

scale_individual [float > 0] Multiplicative factor to scale the std-dev as given by model parameters

set_autograd()
Set autograd for tensor of realizations

See also:

`torch.Tensor.requires_grad_`

set_tensor_realizations_element(*element*, *dim*)
Manually change the value (in-place) of *tensor_realizations* at dimension *dim*.

unset_autograd()
Unset autograd for tensor of realizations

See also:

`torch.Tensor.requires_grad_`

leaspy.io.realizations.collection_realization.CollectionRealization

class leaspy.io.realizations.collection_realization.**CollectionRealization**
Bases: `object`

Realizations of population and individual parameters.

Methods

copy()	Copy of self instance
initialize(n_individuals, model)	Initialize the Collection Realization with a model.
initialize_from_values(n_individuals, model)	Idem that *initialize()* method except it calls *Realization.initialize()* with `scale_individual=.01`
keys()	Return all variable names
to_dict()	Returns 2 dictionaries with realizations

__init__()

__weakref__
list of weak references to the object (if defined)

copy()
Copy of self instance

Returns

CollectionRealization

initialize(*n_individuals*, *model*)
Initialize the Collection Realization with a model.

Idem that *initialize_from_values()* method except it calls *Realization.initialize()* with `scale_individual=1.`

Parameters

n_individuals [int]

model [*AbstractModel*]

initialize_from_values(*n_individuals*, *model*)
Idem that *initialize()* method except it calls *Realization.initialize()* with `scale_individual=.01`

keys()
Return all variable names

to_dict()
Returns 2 dictionaries with realizations

Returns

reals_pop [dict[var_name: str, *Realization*]] Realizations of population variables

reals_ind [dict[var_name: str, *Realization*]] Realizations of individual variables

3.4.4 leaspy.io.settings: Settings class objects

model_settings.ModelSettings(...)	Used in *Leaspy.load()* to create a *Leaspy* class object from a *json* file.
algorithm_settings.AlgorithmSettings(name, ...)	Used to set the algorithms' settings.
outputs_settings.OutputsSettings(settings)	Used to create the *logs* folder to monitor the convergence of the calibration algorithm.

leaspy.io.settings.model_settings.ModelSettings

class leaspy.io.settings.model_settings.**ModelSettings**(*path_to_model_settings*)

Bases: object

Used in *Leaspy.load()* to create a *Leaspy* class object from a *json* file.

__init__(*path_to_model_settings*)

__weakref__

list of weak references to the object (if defined)

leaspy.io.settings.algorithm_settings.AlgorithmSettings

class leaspy.io.settings.algorithm_settings.**AlgorithmSettings**(*name*, ***kwargs*)

Bases: object

Used to set the algorithms' settings. All parameters, except the choice of the algorithm, is set by default. The user can overwrite all default settings.

Parameters

name: str

The algorithm's name. Must be in:

- **For *fit* algorithms:**
  - *mcmc_saem*
  - *lme_fit*
- **For *personalize* algorithms:**
  - *scipy_minimize*
  - *mean_real*
  - *mode_real*
  - *gradient_descent_personalize*
  - *constant_prediction*
  - *lme_personalize*
- **For *simulate* algorithms:**
  - *simulation*

model_initialization_method: str, optional For fit algorithms, give a model initialization method, according to those possible in *initialize_parameters()*.

algo_initialization_method: str, optional Personalize the algorithm initialization method, according to those possible for the given algorithm (refer to its documentation in `algo`).

n_iter: int, optional Number of iteration. There is no stopping criteria for the all the MCMC SAEM algorithms.

n_burn_in_iter: int, optional Number of iteration during burning phase, used for the MCMC SAEM algorithms.

seed: int, optional, default None Used for stochastic algorithms.

use_jacobian: bool, optional, default False Used in `scipy_minimize` algorithm to perform a *LBFGS* instead of a *Powell* algorithm.

n_jobs: int, optional, default 1 Used in `scipy_minimize` algorithm to accelerate calculation with parallel derivation using joblib.

loss: {'MSE', 'MSE_diag_noise', 'crossentropy'}, optional, default 'MSE'

The wanted loss.

- `'MSE'`: MSE of all features
- `'MSE_diag_noise'`: MSE per feature
- `'crossentropy'`: used when the features are binary

progress_bar: bool, optional, default False Used to display a progress bar during computation.

See also:

`leaspy.algo`

Attributes

name: {'mcmc_saem', 'scipy_minimize', 'simulation', 'mean_real', 'gradient_descent_personalize', 'mode_rea
The algorithm's name.

model_initialization_method: str, optional For fit algorithms, give a model initialization method, according to those possible in *initialize_parameters()*.

algo_initialization_method: str, optional Personalize the algorithm initialization method, according to those possible for the given algorithm (refer to its documentation in `algos`).

seed: int, optional, default None Used for stochastic algorithms.

loss: {'MSE', 'MSE_diag_noise', 'crossentropy'}, optional, default 'MSE'

The wanted loss.

- `'MSE'`: MSE of all features
- `'MSE_diag_noise'`: MSE per feature
- `'crossentropy'`: used when the features are binary

parameters: dict Contains the other parameters: *n_iter*, *n_burn_in_iter*, *use_jacobian*, *n_jobs* & *progress_bar*.

logs: :class:`.OutputsSettings`, optional Used to create a `logs` file during a model calibration containing convergence information.

Methods

load(path_to_algorithm_settings)	Instantiate a AlgorithmSettings object a from json file.
save(path, **kwargs)	Save an AlgorithmSettings object in a json file.
set_logs(path, **kwargs)	Use this method to monitor the convergence of a model callibration.

__init__(*name*, ***kwargs*)

__weakref__
list of weak references to the object (if defined)

static _get_nested_dict(*nested_dict: dict*, *nested_levels*, *default=None*)
Get a nested key of a dict or default if any previous level is missing.

classmethod _set_nested_dict(*nested_dict: dict*, *nested_levels*, *val*)
Set a nested key of a dict. Precondition: all intermediate levels must exist.

classmethod load(*path_to_algorithm_settings*)
Instantiate a AlgorithmSettings object a from json file.

Parameters

path_to_algorithm_settings: str Path of the json file.

Returns

AlgorithmSettings An instanced of AlgorithmSettings with specified parameters.

Examples

```
>>> from leaspy import AlgorithmSettings
>>> leaspy_univariate = AlgorithmSettings.load('outputs/leaspy-univariate_
↪model-settings.json')
```

save(*path*, ***kwargs*)
Save an AlgorithmSettings object in a json file.

TODO? save leaspy version as well for retro/future-compatibility issues?

Parameters

path: str Path to store the AlgorithmSettings.

****kwargs** Keyword arguments for json.dump method.

Examples

```
>>> from leaspy import AlgorithmSettings
>>> settings = AlgorithmSettings('scipy_minimize', seed=42, loss='MSE_diag_
↪noise', n_jobs=-1, use_jacobian=True, progress_bar=True)
>>> settings.save('outputs/scipy_minimize-settings.json', indent=2)
```

set_logs(*path*, ***kwargs*)

Use this method to monitor the convergence of a model callibration.

It create graphs and csv files of the values of the population parameters (fixed effects) during the callibration

Parameters

path: str The path of the folder to store the graphs and csv files.

****kwargs:**

- **console_print_periodicity: int, optional, default 50** Display logs in the console/terminal every N iterations.
- **plot_periodicity: int, optional, default 100** Saves the values to display in pdf every N iterations.
- **save_periodicity: int, optional, default 50** Saves the values in csv files every N iterations.
- **overwrite_logs_folder: bool, optionl, default False** Set it to `True` to overwrite the content of the folder in `path`.

Raises

ValueError If the folder given in `path` allready exists and if `overwrite_logs_folder` is set to `False`.

Notes

By default, if the folder given in `path` allready exists, the method will raise an error. To overwrite the content of the folder, set `overwrite_logs_folder` it to `True`.

leaspy.io.settings.outputs_settings.OutputsSettings

class leaspy.io.settings.outputs_settings.**OutputsSettings**(*settings*)

Bases: `object`

Used to create the *logs* folder to monitor the convergence of the calibration algorithm.

__init__(*settings*)

__weakref__

list of weak references to the object (if defined)

3.5 leaspy.models: Models

Available models in *Leaspy*.

model_factory.ModelFactory()	Return the wanted model given its name.
abstract_model.AbstractModel(name)	Contains the common attributes & methods of the different models.
abstract_multivariate_model.AbstractMultivariateModel(...)	Contains the common attributes & methods of the multivariate models.
multivariate_model.MultivariateModel(name, ...)	Manifold model for multiple variables of interest (logistic or linear formulation).
multivariate_parallel_model.MultivariateParallelModel(...)	Logistic model for multiple variables of interest, imposing same average evolution pace for all variables (logistic curves are only time-shifted).
univariate_model.UnivariateModel(name, **kwargs)	Univariate (logistic or linear) model for a single variable of interest.
lme_model.LMEModel(name)	LMEModel is a benchmark model that fits and personalize a linear mixed-effects model
constant_model.ConstantModel(name)	*ConstantModel* is a benchmark model that predicts constant values no matter of the patient's ages.

3.5.1 leaspy.models.model_factory.ModelFactory

class leaspy.models.model_factory.**ModelFactory**

Bases: object

Return the wanted model given its name.

Methods

model(name, **kwargs)	Return the model object corresponding to 'name' arg with possible *kwargs*

__weakref__

list of weak references to the object (if defined)

static model(*name*, ***kwargs*)

Return the model object corresponding to 'name' arg with possible *kwargs*

Check name type and value.

Parameters

name: str The model's name.

****kwargs:** Contains model's hyper-parameters. Raise an error if the keyword is inapropriate for the given model's name.

Returns

AbstractModel A child class object of `models.AbstractModel` class object determined by 'name'.

See also:

leaspy.api.Leaspy

3.5.2 leaspy.models.abstract_model.AbstractModel

class leaspy.models.abstract_model.**AbstractModel**(*name: str*)

Bases: abc.ABC

Contains the common attributes & methods of the different models.

Attributes

is_initialized [bool] Indicates if the model is initialized

name [str] The model's name

features [list[str]] Names of the model features

parameters [dict] Contains the model's parameters

loss [str] The loss to optimize ('MSE', 'MSE_diag_noise' or 'crossentropy')

distribution [torch.distributions.normal.Normal] Gaussian distribution object to compute variables regularization

Methods

compute_individual_attachment_tensorized(...)	Compute attachment term (per subject)
compute_individual_attachment_tensorized_mcmc(...)	Compute MCMC attachment of all subjects? One subject? One visit? TODO: complete
compute_individual_tensorized(timepoints, ...)	Compute the individual values at timepoints according to the model.
compute_individual_trajectory(timepoints, ...)	Compute scores values at the given time-point(s) given a subject's individual parameters.
compute_jacobian_tensorized(timepoints, ...)	Compute the jacobian of the model w.r.t.
compute_regularity_realization(realization)	Compute regularity term for a *Realization* instance.
compute_regularity_variable(value, mean, std)	Compute regularity term (Gaussian distribution), low-level.
compute_sufficient_statistics(data, realizations)	Compute sufficient statistics from realizations
compute_sum_squared_per_ft_tensorized(data, ...)	Compute the square of the residuals per subject per feature
compute_sum_squared_tensorized(data, param_ind)	Compute the square of the residuals per subject
get_individual_realization_names()	Get names of individual variales of the model.
get_individual_variable_name()	Return list of names of the individual variables from the model.
get_param_from_real(realizations)	Get individual parameters realizations from all model realizations
get_population_realization_names()	Get names of population variales of the model.
get_realization_object(n_individuals)	Initialization of a *CollectionRealization* used during model fitting.
initialize(dataset[, method])	Initialize the model given a dataset and an initialization method.

continues on next page

Table 36 – continued from previous page

load_hyperparameters(hyperparameters)	Load model's hyperparameters
load_parameters(parameters)	Instantiate or update the model's parameters.
random_variable_informations()	Informations on model's random variables.
save(path, **kwargs)	Save Leaspy object as json model parameter file.
smart_initialization_realizations(data, ...)	Smart initialization of realizations if needed.
time_reparametrization(timepoints, xi, tau)	Tensorized time reparametrization formula
update_model_parameters(data, ...[, ...])	Update model parameters (high-level function)
update_model_parameters_burn_in(data, ...)	Update model parameters (burn-in phase)
update_model_parameters_normal(data, suff_stats)	Update model parameters (after burn-in phase)

__init__(*name: str*)

__str__()
Return str(self).

__weakref__
list of weak references to the object (if defined)

_audit_individual_parameters(*ips*)
Perform various consistency and compatibility (with current model) checks on an individual parameters dict and outputs qualified information about it.

TODO? move to IndividualParameters class?

Parameters

ips: dict Contains some untrusted individual parameters. If representing only one individual (in a multivariate model) it could be:

{'tau':0.1, 'xi':-0.3, 'sources':[0.1,...]}

Or for multiple individuals: {'tau':[0.1,0.2,...], 'xi':[-0.3,0.2,...], 'sources':[[0.1,...],[0,...],...]}

In particular, a sources vector (if present) should always be a array_like, even if it is 1D

Returns

ips_info: dict 'nb_inds': number of individuals present (int >= 0) 'tensorized_ips': tensorized version of individual parameters 'tensorized_ips_gen': generator providing for all individuals present (ordered as is)

their own tensorized individual parameters

Raises

ValueError: if any of the consistency/compatibility checks fail

static _tensorize_2D(*x*, *unsqueeze_dim*, *dtype=torch.float32*)
Helper to convert a scalar or array_like into an, at least 2D, dtype tensor

Parameters

x: scalar or array_like element to be tensorized

unsqueeze_dim: 0 or -1 dimension to be unsqueezed; meaningful for 1D array-like only >>> _tensorize_2D([1, 2], 0) == tensor([[1, 2]]) >>> _tensorize_2D([1, 2], -1) == tensor([[1], [2]]) for scalar or vector of length 1 it has no matter

compute_individual_attachment_tensorized(*data*, *param_ind*, *attribute_type*)
Compute attachment term (per subject)

Parameters

data [*Dataset*] Contains the data of the subjects, in particular the subjects' time-points and the mask for nan values & padded visits

param_ind: dict Contain the individual parameters

attribute_type: Any, optional Flag to ask for MCMC attributes instead of model's attributes.

Returns

attachment [torch.Tensor] Negative Log-likelihood, shape = (n_subjects,)

compute_individual_attachment_tensorized_mcmc(*data*, *realizations*)
Compute MCMC attachment of all subjects? One subject? One visit? TODO: complete

Parameters

data [*Dataset*] Contains the data of the subjects, in particular the subjects' time-points and the mask (?)

realizations [*CollectionRealization*]

Returns

attachment [torch.Tensor] The subject attachment (?)

abstract compute_individual_tensorized(*timepoints*, *individual_parameters*, *attribute_type=None*)
Compute the individual values at timepoints according to the model.

Parameters

timepoints [torch.Tensor of shape (n_individuals, n_timepoints)]

individual_parameters [dict[param_name: str, torch.Tensor of shape (n_individuals, n_dims_param)]]

attribute_type: Any (default None) Flag to ask for MCMC attributes instead of model's attributes.

Returns

torch.Tensor of shape (n_individuals, n_timepoints, n_features)

compute_individual_trajectory(*timepoints*, *individual_parameters*, ***, *skip_ips_checks=False*)
Compute scores values at the given time-point(s) given a subject's individual parameters.

Parameters

timepoints [scalar or array_like[scalar] (list, tuple, numpy.ndarray)] Contains the age(s) of the subject.

individual_parameters: dict Contains the individual parameters. Each individual parameter should be a scalar or array_like

skip_ips_checks: bool (default: False) Flag to skip consistency/compatibility checks and tensorization of individual_parameters when it was done earlier (speed-up)

Returns

torch.Tensor Contains the subject's scores computed at the given age(s) Shape of tensor is (1, n_tpts, n_features)

abstract compute_jacobian_tensorized(*timepoints*, *ind_parameters*, *attribute_type=None*)
Compute the jacobian of the model w.r.t. each individual parameter.

This function aims to be used in *ScipyMinimize* to speed up optimization.

Parameters

timepoints [torch.Tensor of shape (n_individuals, n_timepoints)]

individual_parameters [dict[param_name: str, torch.Tensor of shape (n_individuals, n_dims_param)]]

attribute_type: Any (default None) Flag to ask for MCMC attributes instead of model's attributes.

Returns

dict[param_name: str, torch.Tensor of shape (n_individuals, n_timepoints, n_features, n_dims_param)

compute_regularity_realization(*realization*)
Compute regularity term for a *Realization* instance.

Parameters

realization [*Realization*]

compute_regularity_variable(*value*, *mean*, *std*)
Compute regularity term (Gaussian distribution), low-level.

Parameters

value, mean, std [torch.Tensor of same shapes]

abstract compute_sufficient_statistics(*data*, *realizations*)
Compute sufficient statistics from realizations

Parameters

data [*Dataset*]

realizations [*CollectionRealization*]

Returns

dict[suff_stat: str, torch.Tensor]

compute_sum_squared_per_ft_tensorized(*data*, *param_ind*, *attribute_type=None*)
Compute the square of the residuals per subject per feature

Parameters

data [*Dataset*] Contains the data of the subjects, in particular the subjects' time-points and the mask (?)

param_ind [dict] Contain the individual parameters

attribute_type [Any (default None)] Flag to ask for MCMC attributes instead of model's attributes.

Returns

torch.Tensor of shape (n_individuals,dimension) Contains L2 residual for each subject and each feature

compute_sum_squared_tensorized(*data*, *param_ind*, *attribute_type=None*)
Compute the square of the residuals per subject

Parameters

data [*Dataset*] Contains the data of the subjects, in particular the subjects' time-points and the mask (?)

param_ind [dict] Contain the individual parameters

attribute_type [Any (default None)] Flag to ask for MCMC attributes instead of model's attributes.

Returns

torch.Tensor of shape (n_individuals,) Contains L2 residual for each subject

`get_individual_realization_names()`
Get names of individual variales of the model.

Returns

list[str]

`get_individual_variable_name()`
Return list of names of the individual variables from the model.

Duplicate of *get_individual_realization_names()* TODO delete one of them

Returns

individual_variable_name [list [str]] Contains the individual variables' names

`get_param_from_real(realizations)`
Get individual parameters realizations from all model realizations

Parameters

realizations [*CollectionRealization*]

Returns

dict[param_name: str, torch.Tensor [n_individuals, dims_param]] Individual parameters

`get_population_realization_names()`
Get names of population variales of the model.

Returns

list[str]

`get_realization_object(n_individuals)`
Initialization of a *CollectionRealization* used during model fitting.

Parameters

n_individuals [int] Number of individuals to track

Returns

CollectionRealization

`abstract initialize(dataset, method='default')`
Initialize the model given a dataset and an initialization method.

After calling this method `is_initialized` should be True and model should be ready for use.

Parameters

dataset [*Dataset*] The dataset we want to initialize from.

method [str] A custom method to initialize the model

abstract load_hyperparameters(*hyperparameters*)
Load model's hyperparameters

Parameters

hyperparameters [dict[str, Any]] Contains the model's hyperparameters

load_parameters(*parameters*)
Instantiate or update the model's parameters.

Parameters

parameters: dict[str, Any] Contains the model's parameters

abstract random_variable_informations()
Informations on model's random variables.

Returns

dict[str, Any]

abstract save(*path*, ***kwargs*)
Save Leaspy object as json model parameter file.

Parameters

path: str Path to store the model's parameters.

****kwargs** Keyword arguments for json.dump method.

smart_initialization_realizations(*data*, *realizations*)
Smart initialization of realizations if needed.

Default behavior to return *realizations* as they are (no smart trick).

Parameters

data [*Dataset*]

realizations [*CollectionRealization*]

Returns

CollectionRealization

static time_reparametrization(*timepoints*, *xi*, *tau*)
Tensorized time reparametrization formula

<!> Shapes of tensors must be compatible between them.

Parameters

timepoints [torch.Tensor] Timepoints to reparametrize

xi [torch.Tensor] Log-acceleration of individual(s)

tau [torch.Tensor] Time-shift(s)

Returns

torch.Tensor of same shape as *timepoints*

update_model_parameters(*data*, *reals_or_suff_stats*, *burn_in_phase=True*)
Update model parameters (high-level function)

Under-the-hood call *update_model_parameters_burn_in()* or *update_model_parameters_normal()* depending on the phase of the fit algorithm

Parameters

data [*Dataset*]

reals_or_suff_stats :

If during burn-in phase will be realizations: *CollectionRealization*

If after burn-in phase will be sufficient statistics: dict[suff_stat: str, torch.Tensor]

abstract update_model_parameters_burn_in(*data*, *realizations*)
Update model parameters (burn-in phase)

Parameters

data [*Dataset*]

realizations [*CollectionRealization*]

abstract update_model_parameters_normal(*data*, *suff_stats*)
Update model parameters (after burn-in phase)

Parameters

data [*Dataset*]

suff_stats [dict[suff_stat: str, torch.Tensor]]

3.5.3 leaspy.models.abstract_multivariate_model.AbstractMultivariateModel

class leaspy.models.abstract_multivariate_model.**AbstractMultivariateModel**(*name*, ***kwargs*)
Bases: *leaspy.models.abstract_model.AbstractModel*

Contains the common attributes & methods of the multivariate models.

Methods

compute_individual_attachment_tensorized(...)	Compute attachment term (per subject)
compute_individual_attachment_tensorized_mcmc(...)	Compute MCMC attachment of all subjects? One subject? One visit? TODO: complete
compute_individual_tensorized(timepoints, ...)	Compute the individual values at timepoints according to the model.
compute_individual_trajectory(timepoints, ...)	Compute scores values at the given time-point(s) given a subject's individual parameters.
compute_jacobian_tensorized(timepoints, ...)	Compute the jacobian of the model w.r.t.
compute_mean_traj(timepoints)	Compute trajectory of the model with individual parameters being the group-average ones.
compute_regularity_realization(realization)	Compute regularity term for a *Realization* instance.
compute_regularity_variable(value, mean, std)	Compute regularity term (Gaussian distribution), low-level.
compute_sufficient_statistics(data, realizations)	Compute sufficient statistics from realizations
compute_sum_squared_per_ft_tensorized(data, ...)	Compute the square of the residuals per subject per feature

continues on next page

Table 37 – continued from previous page

compute_sum_squared_tensorized(data, param_ind)	Compute the square of the residuals per subject
get_individual_realization_names()	Get names of individual variales of the model.
get_individual_variable_name()	Return list of names of the individual variables from the model.
get_param_from_real(realizations)	Get individual parameters realizations from all model realizations
get_population_realization_names()	Get names of population variales of the model.
get_realization_object(n_individuals)	Initialization of a *CollectionRealization* used during model fitting.
initialize(dataset[, method])	Initialize the model given a dataset and an initialization method.
initialize_MCMC_toolbox()	Initialize Monte-Carlo Markov-Chain toolbox for calibration of model
load_hyperparameters(hyperparameters)	Load model's hyperparameters
load_parameters(parameters)	Instantiate or update the model's parameters.
random_variable_informations()	Informations on model's random variables.
save(path[, with_mixing_matrix])	Save Leaspy object as json model parameter file.
smart_initialization_realizations(data, ...)	Smart initialization of realizations if needed.
time_reparametrization(timepoints, xi, tau)	Tensorized time reparametrization formula
update_MCMC_toolbox(...)	Update the MCMC toolbox with a collection of realizations of model population parameters.
update_model_parameters(data, ...[, ...])	Update model parameters (high-level function)
update_model_parameters_burn_in(data, ...)	Update model parameters (burn-in phase)
update_model_parameters_normal(data, suff_stats)	Update model parameters (after burn-in phase)

__init__(*name*, ***kwargs*)

__str__()
Return str(self).

__weakref__
list of weak references to the object (if defined)

_audit_individual_parameters(*ips*)
Perform various consistency and compatibility (with current model) checks on an individual parameters dict and outputs qualified information about it.

TODO? move to IndividualParameters class?

Parameters

ips: dict Contains some untrusted individual parameters. If representing only one individual (in a multivariate model) it could be:

{'tau':0.1, 'xi':-0.3, 'sources':[0.1,...]}

Or for multiple individuals: {'tau':[0.1,0.2,...], 'xi':[-0.3,0.2,...], 'sources':[[0.1,...],[0,...],...]}

In particular, a sources vector (if present) should always be a array_like, even if it is 1D

Returns

ips_info: dict 'nb_inds': number of individuals present (int >= 0) 'tensorized_ips': tensorized version of individual parameters 'tensorized_ips_gen': generator providing for all individuals present (ordered as is)

their own tensorized individual parameters

Raises

ValueError: if any of the consistency/compatibility checks fail

`static _tensorize_2D`(*x*, *unsqueeze_dim*, *dtype=torch.float32*)
Helper to convert a scalar or array_like into an, at least 2D, dtype tensor

Parameters

x: scalar or array_like element to be tensorized

unsqueeze_dim: 0 or -1 dimension to be unsqueezed; meaningful for 1D array-like only >>> _tensorize_2D([1, 2], 0) == tensor([[1, 2]]) >>> _tensorize_2D([1, 2], -1) == tensor([[1], [2]]) for scalar or vector of length 1 it has no matter

`compute_individual_attachment_tensorized`(*data*, *param_ind*, *attribute_type*)
Compute attachment term (per subject)

Parameters

data [*Dataset*] Contains the data of the subjects, in particular the subjects' time-points and the mask for nan values & padded visits

param_ind: dict Contain the individual parameters

attribute_type: Any, optional Flag to ask for MCMC attributes instead of model's attributes.

Returns

attachment [torch.Tensor] Negative Log-likelihood, shape = (n_subjects,)

`compute_individual_attachment_tensorized_mcmc`(*data*, *realizations*)
Compute MCMC attachment of all subjects? One subject? One visit? TODO: complete

Parameters

data [*Dataset*] Contains the data of the subjects, in particular the subjects' time-points and the mask (?)

realizations [*CollectionRealization*]

Returns

attachment [torch.Tensor] The subject attachment (?)

`abstract compute_individual_tensorized`(*timepoints*, *individual_parameters*, *attribute_type=None*)
Compute the individual values at timepoints according to the model.

Parameters

timepoints [torch.Tensor of shape (n_individuals, n_timepoints)]

individual_parameters [dict[param_name: str, torch.Tensor of shape (n_individuals, n_dims_param)]]

attribute_type: Any (default None) Flag to ask for MCMC attributes instead of model's attributes.

Returns

torch.Tensor of shape (n_individuals, n_timepoints, n_features)

compute_individual_trajectory(*timepoints*, *individual_parameters*, *, *skip_ips_checks=False*)
Compute scores values at the given time-point(s) given a subject's individual parameters.

Parameters

timepoints [scalar or array_like[scalar] (list, tuple, numpy.ndarray)] Contains the age(s) of the subject.

individual_parameters: dict Contains the individual parameters. Each individual parameter should be a scalar or array_like

skip_ips_checks: bool (default: False) Flag to skip consistency/compatibility checks and tensorization of individual_parameters when it was done earlier (speed-up)

Returns

torch.Tensor Contains the subject's scores computed at the given age(s) Shape of tensor is (1, n_tpts, n_features)

abstract compute_jacobian_tensorized(*timepoints*, *ind_parameters*, *attribute_type=None*)
Compute the jacobian of the model w.r.t. each individual parameter.

This function aims to be used in *ScipyMinimize* to speed up optimization.

Parameters

timepoints [torch.Tensor of shape (n_individuals, n_timepoints)]

individual_parameters [dict[param_name: str, torch.Tensor of shape (n_individuals, n_dims_param)]]

attribute_type: Any (default None) Flag to ask for MCMC attributes instead of model's attributes.

Returns

dict[param_name: str, torch.Tensor of shape (n_individuals, n_timepoints, n_features, n_dims_param)

compute_mean_traj(*timepoints*)
Compute trajectory of the model with individual parameters being the group-average ones.

TODO check dimensions of io?

Parameters

timepoints [torch.Tensor [1, n_timepoints]]

Returns

torch.Tensor **[1, n_timepoints, dimension]** The group-average values at given timepoints

compute_regularity_realization(*realization*)
Compute regularity term for a *Realization* instance.

Parameters

realization [*Realization*]

compute_regularity_variable(*value*, *mean*, *std*)
Compute regularity term (Gaussian distribution), low-level.

Parameters

value, mean, std [torch.Tensor of same shapes]

abstract compute_sufficient_statistics(*data*, *realizations*)
Compute sufficient statistics from realizations

Parameters

data [*Dataset*]

realizations [*CollectionRealization*]

Returns

dict[suff_stat: str, torch.Tensor]

compute_sum_squared_per_ft_tensorized(*data*, *param_ind*, *attribute_type=None*)
Compute the square of the residuals per subject per feature

Parameters

data [*Dataset*] Contains the data of the subjects, in particular the subjects' time-points and the mask (?)

param_ind [dict] Contain the individual parameters

attribute_type [Any (default None)] Flag to ask for MCMC attributes instead of model's attributes.

Returns

torch.Tensor of shape (n_individuals,dimension) Contains L2 residual for each subject and each feature

compute_sum_squared_tensorized(*data*, *param_ind*, *attribute_type=None*)
Compute the square of the residuals per subject

Parameters

data [*Dataset*] Contains the data of the subjects, in particular the subjects' time-points and the mask (?)

param_ind [dict] Contain the individual parameters

attribute_type [Any (default None)] Flag to ask for MCMC attributes instead of model's attributes.

Returns

torch.Tensor of shape (n_individuals,) Contains L2 residual for each subject

get_individual_realization_names()
Get names of individual variales of the model.

Returns

list[str]

get_individual_variable_name()
Return list of names of the individual variables from the model.

Duplicate of *get_individual_realization_names()* TODO delete one of them

Returns

individual_variable_name [list [str]] Contains the individual variables' names

get_param_from_real(*realizations*)
Get individual parameters realizations from all model realizations

Parameters

realizations [*CollectionRealization*]

Returns

dict[param_name: str, torch.Tensor [n_individuals, dims_param]] Individual parameters

`get_population_realization_names()`
Get names of population variales of the model.

Returns

list[str]

`get_realization_object(n_individuals)`
Initialization of a *CollectionRealization* used during model fitting.

Parameters

n_individuals [int] Number of individuals to track

Returns

CollectionRealization

`initialize(dataset, method='default')`
Initialize the model given a dataset and an initialization method.

After calling this method `is_initialized` should be True and model should be ready for use.

Parameters

dataset [*Dataset*] The dataset we want to initialize from.

method [str] A custom method to initialize the model

`abstract initialize_MCMC_toolbox()`
Initialize Monte-Carlo Markov-Chain toolbox for calibration of model

TODO to move in a "MCMC-model interface"

`load_hyperparameters(hyperparameters)`
Load model's hyperparameters

Parameters

hyperparameters [dict[str, Any]] Contains the model's hyperparameters

`load_parameters(parameters)`
Instantiate or update the model's parameters.

Parameters

parameters: dict[str, Any] Contains the model's parameters

`abstract random_variable_informations()`
Informations on model's random variables.

Returns

dict[str, Any]

`save(path, with_mixing_matrix=True, **kwargs)`
Save Leaspy object as json model parameter file.

Parameters

path: str Path to store the model's parameters.

with_mixing_matrix: bool (default True) Save the mixing matrix in the exported file in its 'parameters' section. <!> It is not a real parameter and its value will be overwritten at model loading

(orthonormal basis is recomputed from other "true" parameters and mixing matrix is then deduced from this orthonormal basis and the betas)!

It was integrated historically because it is used for convenience in browser webtool and only there…

****kwargs** Keyword arguments for json.dump method.

smart_initialization_realizations(*data*, *realizations*)
Smart initialization of realizations if needed.

Default behavior to return *realizations* as they are (no smart trick).

Parameters

data [*Dataset*]

realizations [*CollectionRealization*]

Returns

CollectionRealization

static time_reparametrization(*timepoints*, *xi*, *tau*)
Tensorized time reparametrization formula

<!> Shapes of tensors must be compatible between them.

Parameters

timepoints [torch.Tensor] Timepoints to reparametrize

xi [torch.Tensor] Log-acceleration of individual(s)

tau [torch.Tensor] Time-shift(s)

Returns

torch.Tensor of same shape as *timepoints*

abstract update_MCMC_toolbox(*name_of_the_variables_that_have_been_changed*, *realizations*)
Update the MCMC toolbox with a collection of realizations of model population parameters.

TODO to move in a "MCMC-model interface"

Parameters

name_of_the_variables_that_have_been_changed: container[str] (list, tuple, …) Names of the population parameters to update in MCMC toolbox

realizations [*CollectionRealization*] All the realizations to update MCMC toolbox with

update_model_parameters(*data*, *reals_or_suff_stats*, *burn_in_phase=True*)
Update model parameters (high-level function)

Under-the-hood call *update_model_parameters_burn_in()* or *update_model_parameters_normal()* depending on the phase of the fit algorithm

Parameters

data [*Dataset*]

reals_or_suff_stats :

If during burn-in phase will be realizations: *CollectionRealization*

If after burn-in phase will be sufficient statistics: dict[suff_stat: str, torch.Tensor]

abstract update_model_parameters_burn_in(*data*, *realizations*)
Update model parameters (burn-in phase)

Parameters

data [*Dataset*]

realizations [*CollectionRealization*]

abstract update_model_parameters_normal(*data*, *suff_stats*)
Update model parameters (after burn-in phase)

Parameters

data [*Dataset*]

suff_stats [dict[suff_stat: str, torch.Tensor]]

3.5.4 leaspy.models.multivariate_model.MultivariateModel

class leaspy.models.multivariate_model.**MultivariateModel**(*name*, ***kwargs*)
Bases: *leaspy.models.abstract_multivariate_model.AbstractMultivariateModel*

Manifold model for multiple variables of interest (logistic or linear formulation).

Methods

compute_individual_attachment_tensorized(...)	Compute attachment term (per subject)
compute_individual_attachment_tensorized_mcmc(...)	Compute MCMC attachment of all subjects? One subject? One visit? TODO: complete
compute_individual_tensorized(timepoints, ...)	Compute the individual values at timepoints according to the model.
compute_individual_tensorized_linear(...[, ...])	Compute the individual values at timepoints according to the model (linear).
compute_individual_tensorized_logistic(...)	Compute the individual values at timepoints according to the model (logistic).
compute_individual_tensorized_mixed(...[, ...])	Compute the individual values at timepoints according to the model (mixed logistic-linear).
compute_individual_trajectory(timepoints, ...)	Compute scores values at the given time-point(s) given a subject's individual parameters.
compute_jacobian_tensorized(timepoints, ...)	Compute the jacobian of the model w.r.t.
compute_jacobian_tensorized_linear(...[, ...])	Compute the jacobian of the model (linear) w.r.t.
compute_jacobian_tensorized_logistic(...[, ...])	Compute the jacobian of the model (logistic) w.r.t.
compute_jacobian_tensorized_mixed(...[, ...])	Compute the jacobian of the model (mixed logistic-linear) w.r.t.
compute_mean_traj(timepoints)	Compute trajectory of the model with individual parameters being the group-average ones.

continues on next page

Table 38 – continued from previous page

compute_regularity_realization(realization)	Compute regularity term for a *Realization* instance.
compute_regularity_variable(value, mean, std)	Compute regularity term (Gaussian distribution), low-level.
compute_sufficient_statistics(data, realizations)	Compute sufficient statistics from realizations
compute_sum_squared_per_ft_tensorized(data, ...)	Compute the square of the residuals per subject per feature
compute_sum_squared_tensorized(data, param_ind)	Compute the square of the residuals per subject
get_individual_realization_names()	Get names of individual variales of the model.
get_individual_variable_name()	Return list of names of the individual variables from the model.
get_param_from_real(realizations)	Get individual parameters realizations from all model realizations
get_population_realization_names()	Get names of population variales of the model.
get_realization_object(n_individuals)	Initialization of a *CollectionRealization* used during model fitting.
initialize(dataset[, method])	Initialize the model given a dataset and an initialization method.
initialize_MCMC_toolbox([set_v0_prior])	Initialize Monte-Carlo Markov-Chain toolbox for calibration of model
load_hyperparameters(hyperparameters)	Load model's hyperparameters
load_parameters(parameters)	Instantiate or update the model's parameters.
random_variable_informations()	Informations on model's random variables.
save(path[, with_mixing_matrix])	Save Leaspy object as json model parameter file.
smart_initialization_realizations(data, ...)	Smart initialization of realizations if needed.
time_reparametrization(timepoints, xi, tau)	Tensorized time reparametrization formula
update_MCMC_toolbox(...)	Update the MCMC toolbox with a collection of realizations of model population parameters.
update_model_parameters(data, ...[, ...])	Update model parameters (high-level function)
update_model_parameters_burn_in(data, ...)	Update model parameters (burn-in phase)
update_model_parameters_normal(data, suff_stats)	Update model parameters (after burn-in phase)

__init__(*name*, ***kwargs*)

__str__()
Return str(self).

__weakref__
list of weak references to the object (if defined)

_audit_individual_parameters(*ips*)
Perform various consistency and compatibility (with current model) checks on an individual parameters dict and outputs qualified information about it.

TODO? move to IndividualParameters class?

Parameters

ips: dict Contains some untrusted individual parameters. If representing only one individual (in a multivariate model) it could be:

{'tau':0.1, 'xi':-0.3, 'sources':[0.1,...]}

Or for multiple individuals: {'tau':[0.1,0.2,...], 'xi':[-0.3,0.2,...], 'sources':[[0.1,...],[0,...],...]}

In particular, a sources vector (if present) should always be a array_like, even if it is 1D

Returns

ips_info: dict 'nb_inds': number of individuals present (int >= 0) 'tensorized_ips': tensorized version of individual parameters 'tensorized_ips_gen': generator providing for all individuals present (ordered as is)

their own tensorized individual parameters

Raises

ValueError: if any of the consistency/compatibility checks fail

`static _tensorize_2D`(*x*, *unsqueeze_dim*, *dtype=torch.float32*)

Helper to convert a scalar or array_like into an, at least 2D, dtype tensor

Parameters

x: scalar or array_like element to be tensorized

unsqueeze_dim: 0 or -1 dimension to be unsqueezed; meaningful for 1D array-like only >>> _tensorize_2D([1, 2], 0) == tensor([[1, 2]]) >>> _tensorize_2D([1, 2], -1) == tensor([[1], [2]]) for scalar or vector of length 1 it has no matter

`compute_individual_attachment_tensorized`(*data*, *param_ind*, *attribute_type*)

Compute attachment term (per subject)

Parameters

data [*Dataset*] Contains the data of the subjects, in particular the subjects' time-points and the mask for nan values & padded visits

param_ind: dict Contain the individual parameters

attribute_type: Any, optional Flag to ask for MCMC attributes instead of model's attributes.

Returns

attachment [torch.Tensor] Negative Log-likelihood, shape = (n_subjects,)

`compute_individual_attachment_tensorized_mcmc`(*data*, *realizations*)

Compute MCMC attachment of all subjects? One subject? One visit? TODO: complete

Parameters

data [*Dataset*] Contains the data of the subjects, in particular the subjects' time-points and the mask (?)

realizations [*CollectionRealization*]

Returns

attachment [torch.Tensor] The subject attachment (?)

`compute_individual_tensorized`(*timepoints*, *ind_parameters*, *attribute_type=None*)

Compute the individual values at timepoints according to the model.

Parameters

timepoints [torch.Tensor of shape (n_individuals, n_timepoints)]

individual_parameters [dict[param_name: str, torch.Tensor of shape (n_individuals, n_dims_param)]]

attribute_type: Any (default None) Flag to ask for MCMC attributes instead of model's attributes.

Returns

torch.Tensor of shape (n_individuals, n_timepoints, n_features)

compute_individual_tensorized_linear(*timepoints*, *ind_parameters*, *attribute_type=None*)
Compute the individual values at timepoints according to the model (linear).

Parameters

timepoints [torch.Tensor of shape (n_individuals, n_timepoints)]

individual_parameters [dict[param_name: str, torch.Tensor of shape (n_individuals, n_dims_param)]]

attribute_type: Any (default None) Flag to ask for MCMC attributes instead of model's attributes.

Returns

torch.Tensor of shape (n_individuals, n_timepoints, n_features)

compute_individual_tensorized_logistic(*timepoints*, *ind_parameters*, *attribute_type=None*)
Compute the individual values at timepoints according to the model (logistic).

Parameters

timepoints [torch.Tensor of shape (n_individuals, n_timepoints)]

individual_parameters [dict[param_name: str, torch.Tensor of shape (n_individuals, n_dims_param)]]

attribute_type: Any (default None) Flag to ask for MCMC attributes instead of model's attributes.

Returns

torch.Tensor of shape (n_individuals, n_timepoints, n_features)

compute_individual_tensorized_mixed(*timepoints*, *ind_parameters*, *attribute_type=None*)
Compute the individual values at timepoints according to the model (mixed logistic-linear).

Parameters

timepoints [torch.Tensor of shape (n_individuals, n_timepoints)]

individual_parameters [dict[param_name: str, torch.Tensor of shape (n_individuals, n_dims_param)]]

attribute_type: Any (default None) Flag to ask for MCMC attributes instead of model's attributes.

Returns

torch.Tensor of shape (n_individuals, n_timepoints, n_features)

compute_individual_trajectory(*timepoints*, *individual_parameters*, ***, *skip_ips_checks=False*)
Compute scores values at the given time-point(s) given a subject's individual parameters.

Parameters

timepoints [scalar or array_like[scalar] (list, tuple, numpy.ndarray)] Contains the age(s) of the subject.

individual_parameters: dict Contains the individual parameters. Each individual parameter should be a scalar or array_like

skip_ips_checks: bool (default: False) Flag to skip consistency/compatibility checks and tensorization of individual_parameters when it was done earlier (speed-up)

Returns

torch.Tensor Contains the subject's scores computed at the given age(s) Shape of tensor is (1, n_tpts, n_features)

compute_jacobian_tensorized(*timepoints*, *ind_parameters*, *attribute_type=None*)

Compute the jacobian of the model w.r.t. each individual parameter.

This function aims to be used in *ScipyMinimize* to speed up optimization.

Parameters

timepoints [torch.Tensor of shape (n_individuals, n_timepoints)]

individual_parameters [dict[param_name: str, torch.Tensor of shape (n_individuals, n_dims_param)]]

attribute_type: Any (default None) Flag to ask for MCMC attributes instead of model's attributes.

Returns

dict[param_name: str, torch.Tensor of shape (n_individuals, n_timepoints, n_features, n_dims_param)

compute_jacobian_tensorized_linear(*timepoints*, *ind_parameters*, *attribute_type=None*)

Compute the jacobian of the model (linear) w.r.t. each individual parameter.

This function aims to be used in *ScipyMinimize* to speed up optimization.

Parameters

timepoints [torch.Tensor of shape (n_individuals, n_timepoints)]

individual_parameters [dict[param_name: str, torch.Tensor of shape (n_individuals, n_dims_param)]]

attribute_type: Any (default None) Flag to ask for MCMC attributes instead of model's attributes.

Returns

dict[param_name: str, torch.Tensor of shape (n_individuals, n_timepoints, n_features, n_dims_param)

compute_jacobian_tensorized_logistic(*timepoints*, *ind_parameters*, *attribute_type=None*)

Compute the jacobian of the model (logistic) w.r.t. each individual parameter.

This function aims to be used in *ScipyMinimize* to speed up optimization.

Parameters

timepoints [torch.Tensor of shape (n_individuals, n_timepoints)]

individual_parameters [dict[param_name: str, torch.Tensor of shape (n_individuals, n_dims_param)]]

attribute_type: Any (default None) Flag to ask for MCMC attributes instead of model's attributes.

Returns

dict[param_name: str, torch.Tensor of shape (n_individuals, n_timepoints, n_features, n_dims_param)

compute_jacobian_tensorized_mixed(*timepoints*, *ind_parameters*, *attribute_type=None*)
Compute the jacobian of the model (mixed logistic-linear) w.r.t. each individual parameter.

This function aims to be used in *ScipyMinimize* to speed up optimization.

Parameters

timepoints [torch.Tensor of shape (n_individuals, n_timepoints)]

individual_parameters [dict[param_name: str, torch.Tensor of shape (n_individuals, n_dims_param)]]

attribute_type: Any (default None) Flag to ask for MCMC attributes instead of model's attributes.

Returns

dict[param_name: str, torch.Tensor of shape (n_individuals, n_timepoints, n_features, n_dims_param)

compute_mean_traj(*timepoints*)
Compute trajectory of the model with individual parameters being the group-average ones.

TODO check dimensions of io?

Parameters

timepoints [torch.Tensor [1, n_timepoints]]

Returns

torch.Tensor [1, n_timepoints, dimension] The group-average values at given timepoints

compute_regularity_realization(*realization*)
Compute regularity term for a *Realization* instance.

Parameters

realization [*Realization*]

compute_regularity_variable(*value*, *mean*, *std*)
Compute regularity term (Gaussian distribution), low-level.

Parameters

value, mean, std [torch.Tensor of same shapes]

compute_sufficient_statistics(*data*, *realizations*)
Compute sufficient statistics from realizations

Parameters

data [*Dataset*]

realizations [*CollectionRealization*]

Returns

dict[suff_stat: str, torch.Tensor]

compute_sum_squared_per_ft_tensorized(*data*, *param_ind*, *attribute_type=None*)
Compute the square of the residuals per subject per feature

Parameters

data [*Dataset*] Contains the data of the subjects, in particular the subjects' time-points and the mask (?)

param_ind [dict] Contain the individual parameters

attribute_type [Any (default None)] Flag to ask for MCMC attributes instead of model's attributes.

Returns

torch.Tensor of shape (n_individuals,dimension) Contains L2 residual for each subject and each feature

compute_sum_squared_tensorized(*data*, *param_ind*, *attribute_type=None*)
Compute the square of the residuals per subject

Parameters

data [*Dataset*] Contains the data of the subjects, in particular the subjects' time-points and the mask (?)

param_ind [dict] Contain the individual parameters

attribute_type [Any (default None)] Flag to ask for MCMC attributes instead of model's attributes.

Returns

torch.Tensor of shape (n_individuals,) Contains L2 residual for each subject

get_individual_realization_names()
Get names of individual variales of the model.

Returns

list[str]

get_individual_variable_name()
Return list of names of the individual variables from the model.

Duplicate of *get_individual_realization_names()* TODO delete one of them

Returns

individual_variable_name [list [str]] Contains the individual variables' names

get_param_from_real(*realizations*)
Get individual parameters realizations from all model realizations

Parameters

realizations [*CollectionRealization*]

Returns

dict[param_name: str, torch.Tensor [n_individuals, dims_param]] Individual parameters

get_population_realization_names()
Get names of population variales of the model.

Returns

list[str]

`get_realization_object`(*n_individuals*)
Initialization of a *CollectionRealization* used during model fitting.

Parameters

n_individuals [int] Number of individuals to track

Returns

CollectionRealization

`initialize`(*dataset*, *method='default'*)
Initialize the model given a dataset and an initialization method.

After calling this method `is_initialized` should be True and model should be ready for use.

Parameters

dataset [*Dataset*] The dataset we want to initialize from.

method [str] A custom method to initialize the model

`initialize_MCMC_toolbox`(*set_v0_prior=False*)
Initialize Monte-Carlo Markov-Chain toolbox for calibration of model

TODO to move in a "MCMC-model interface"

`load_hyperparameters`(*hyperparameters*)
Load model's hyperparameters

Parameters

hyperparameters [dict[str, Any]] Contains the model's hyperparameters

`load_parameters`(*parameters*)
Instantiate or update the model's parameters.

Parameters

parameters: dict[str, Any] Contains the model's parameters

`random_variable_informations`()
Informations on model's random variables.

Returns

dict[str, Any]

`save`(*path*, *with_mixing_matrix=True*, ***kwargs*)
Save Leaspy object as json model parameter file.

Parameters

path: str Path to store the model's parameters.

with_mixing_matrix: bool (default True) Save the mixing matrix in the exported file in its 'parameters' section. <!> It is not a real parameter and its value will be overwritten at model loading

(orthonormal basis is recomputed from other "true" parameters and mixing matrix is then deduced from this orthonormal basis and the betas)!

It was integrated historically because it is used for convenience in browser webtool and only there…

****kwargs** Keyword arguments for json.dump method.

smart_initialization_realizations(*data*, *realizations*)
Smart initialization of realizations if needed.

Default behavior to return *realizations* as they are (no smart trick).

Parameters

data [*Dataset*]

realizations [*CollectionRealization*]

Returns

CollectionRealization

static time_reparametrization(*timepoints*, *xi*, *tau*)
Tensorized time reparametrization formula

<!> Shapes of tensors must be compatible between them.

Parameters

timepoints [torch.Tensor] Timepoints to reparametrize

xi [torch.Tensor] Log-acceleration of individual(s)

tau [torch.Tensor] Time-shift(s)

Returns

torch.Tensor of same shape as *timepoints*

update_MCMC_toolbox(*name_of_the_variables_that_have_been_changed*, *realizations*)
Update the MCMC toolbox with a collection of realizations of model population parameters.

TODO to move in a "MCMC-model interface"

Parameters

name_of_the_variables_that_have_been_changed: container[str] (list, tuple, ...)
Names of the population parameters to update in MCMC toolbox

realizations [*CollectionRealization*] All the realizations to update MCMC toolbox with

update_model_parameters(*data*, *reals_or_suff_stats*, *burn_in_phase=True*)
Update model parameters (high-level function)

Under-the-hood call *update_model_parameters_burn_in()* or *update_model_parameters_normal()* depending on the phase of the fit algorithm

Parameters

data [*Dataset*]

reals_or_suff_stats :

If during burn-in phase will be realizations: *CollectionRealization*

If after burn-in phase will be sufficient statistics: dict[suff_stat: str, torch.Tensor]

update_model_parameters_burn_in(*data*, *realizations*)
Update model parameters (burn-in phase)

Parameters

data [*Dataset*]

realizations [*CollectionRealization*]

update_model_parameters_normal(*data*, *suff_stats*)
Update model parameters (after burn-in phase)

Parameters

data [*Dataset*]

suff_stats [dict[suff_stat: str, torch.Tensor]]

3.5.5 leaspy.models.multivariate_parallel_model.MultivariateParallelModel

class leaspy.models.multivariate_parallel_model.**MultivariateParallelModel**(*name*, ***kwargs*)
Bases: *leaspy.models.abstract_multivariate_model.AbstractMultivariateModel*

Logistic model for multiple variables of interest, imposing same average evolution pace for all variables (logistic curves are only time-shifted).

Methods

compute_individual_attachment_tensorized(...)	Compute attachment term (per subject)
compute_individual_attachment_tensorized_mcmc(...)	Compute MCMC attachment of all subjects? One subject? One visit? TODO: complete
compute_individual_tensorized(timepoints, ...)	Compute the individual values at timepoints according to the model.
compute_individual_trajectory(timepoints, ...)	Compute scores values at the given time-point(s) given a subject's individual parameters.
compute_jacobian_tensorized(timepoints, ...)	Compute the jacobian of the model w.r.t.
compute_mean_traj(timepoints)	Compute trajectory of the model with individual parameters being the group-average ones.
compute_regularity_realization(realization)	Compute regularity term for a *Realization* instance.
compute_regularity_variable(value, mean, std)	Compute regularity term (Gaussian distribution), low-level.
compute_sufficient_statistics(data, realizations)	Compute sufficient statistics from realizations
compute_sum_squared_per_ft_tensorized(data, ...)	Compute the square of the residuals per subject per feature
compute_sum_squared_tensorized(data, param_ind)	Compute the square of the residuals per subject
get_individual_realization_names()	Get names of individual variales of the model.
get_individual_variable_name()	Return list of names of the individual variables from the model.
get_param_from_real(realizations)	Get individual parameters realizations from all model realizations
get_population_realization_names()	Get names of population variales of the model.
get_realization_object(n_individuals)	Initialization of a *CollectionRealization* used during model fitting.
initialize(dataset[, method])	Initialize the model given a dataset and an initialization method.

continues on next page

Table 39 – continued from previous page

initialize_MCMC_toolbox()	Initialize Monte-Carlo Markov-Chain toolbox for calibration of model
load_hyperparameters(hyperparameters)	Load model's hyperparameters
load_parameters(parameters)	Instantiate or update the model's parameters.
random_variable_informations()	Informations on model's random variables.
save(path[, with_mixing_matrix])	Save Leaspy object as json model parameter file.
smart_initialization_realizations(data, ...)	Smart initialization of realizations if needed.
time_reparametrization(timepoints, xi, tau)	Tensorized time reparametrization formula
update_MCMC_toolbox(...)	Update the MCMC toolbox with a collection of realizations of model population parameters.
update_model_parameters(data, ...[, ...])	Update model parameters (high-level function)
update_model_parameters_burn_in(data, ...)	Update model parameters (burn-in phase)
update_model_parameters_normal(data, suff_stats)	Update model parameters (after burn-in phase)

__init__(*name*, ***kwargs*)

__str__()
Return str(self).

__weakref__
list of weak references to the object (if defined)

_audit_individual_parameters(*ips*)
Perform various consistency and compatibility (with current model) checks on an individual parameters dict and outputs qualified information about it.

TODO? move to IndividualParameters class?

Parameters

ips: dict Contains some untrusted individual parameters. If representing only one individual (in a multivariate model) it could be:

{'tau':0.1, 'xi':-0.3, 'sources':[0.1,...]}

Or for multiple individuals: {'tau':[0.1,0.2,...], 'xi':[-0.3,0.2,...], 'sources':[[0.1,...],[0,...],...]}

In particular, a sources vector (if present) should always be a array_like, even if it is 1D

Returns

ips_info: dict 'nb_inds': number of individuals present (int >= 0) 'tensorized_ips': tensorized version of individual parameters 'tensorized_ips_gen': generator providing for all individuals present (ordered as is)

their own tensorized individual parameters

Raises

ValueError: if any of the consistency/compatibility checks fail

static _tensorize_2D(*x*, *unsqueeze_dim*, *dtype=torch.float32*)
Helper to convert a scalar or array_like into an, at least 2D, dtype tensor

Parameters

x: scalar or array_like element to be tensorized

unsqueeze_dim: 0 or -1 dimension to be unsqueezed; meaningful for 1D array-like only >>> _tensorize_2D([1, 2], 0) == tensor([[1, 2]]) >>> _tensorize_2D([1, 2], -1) == tensor([[1], [2]]) for scalar or vector of length 1 it has no matter

`compute_individual_attachment_tensorized`(*data*, *param_ind*, *attribute_type*)
Compute attachment term (per subject)

Parameters

data [*Dataset*] Contains the data of the subjects, in particular the subjects' time-points and the mask for nan values & padded visits

param_ind: dict Contain the individual parameters

attribute_type: Any, optional Flag to ask for MCMC attributes instead of model's attributes.

Returns

attachment [torch.Tensor] Negative Log-likelihood, shape = (n_subjects,)

`compute_individual_attachment_tensorized_mcmc`(*data*, *realizations*)
Compute MCMC attachment of all subjects? One subject? One visit? TODO: complete

Parameters

data [*Dataset*] Contains the data of the subjects, in particular the subjects' time-points and the mask (?)

realizations [*CollectionRealization*]

Returns

attachment [torch.Tensor] The subject attachment (?)

`compute_individual_tensorized`(*timepoints*, *ind_parameters*, *attribute_type=None*)
Compute the individual values at timepoints according to the model.

Parameters

timepoints [torch.Tensor of shape (n_individuals, n_timepoints)]

individual_parameters [dict[param_name: str, torch.Tensor of shape (n_individuals, n_dims_param)]]

attribute_type: Any (default None) Flag to ask for MCMC attributes instead of model's attributes.

Returns

torch.Tensor of shape (n_individuals, n_timepoints, n_features)

`compute_individual_trajectory`(*timepoints*, *individual_parameters*, *, *skip_ips_checks=False*)
Compute scores values at the given time-point(s) given a subject's individual parameters.

Parameters

timepoints [scalar or array_like[scalar] (list, tuple, numpy.ndarray)] Contains the age(s) of the subject.

individual_parameters: dict Contains the individual parameters. Each individual parameter should be a scalar or array_like

skip_ips_checks: bool (default: False) Flag to skip consistency/compatibility checks and tensorization of individual_parameters when it was done earlier (speed-up)

Returns

`torch.Tensor` Contains the subject's scores computed at the given age(s) Shape of tensor is (1, n_tpts, n_features)

`compute_jacobian_tensorized`(*timepoints*, *ind_parameters*, *attribute_type=None*)
Compute the jacobian of the model w.r.t. each individual parameter.

This function aims to be used in *ScipyMinimize* to speed up optimization.

Parameters

timepoints [`torch.Tensor` of shape (n_individuals, n_timepoints)]

individual_parameters [dict[param_name: str, `torch.Tensor` of shape (n_individuals, n_dims_param)]]

attribute_type: Any (default None) Flag to ask for MCMC attributes instead of model's attributes.

Returns

dict[param_name: str, `torch.Tensor` of shape (n_individuals, n_timepoints, n_features, n_dims_param)

`compute_mean_traj`(*timepoints*)
Compute trajectory of the model with individual parameters being the group-average ones.

TODO check dimensions of io?

Parameters

timepoints [`torch.Tensor` [1, n_timepoints]]

Returns

`torch.Tensor` **[1, n_timepoints, dimension]** The group-average values at given timepoints

`compute_regularity_realization`(*realization*)
Compute regularity term for a *Realization* instance.

Parameters

realization [*Realization*]

`compute_regularity_variable`(*value*, *mean*, *std*)
Compute regularity term (Gaussian distribution), low-level.

Parameters

value, mean, std [`torch.Tensor` of same shapes]

`compute_sufficient_statistics`(*data*, *realizations*)
Compute sufficient statistics from realizations

Parameters

data [*Dataset*]

realizations [*CollectionRealization*]

Returns

dict[suff_stat: str, `torch.Tensor`]

`compute_sum_squared_per_ft_tensorized`(*data*, *param_ind*, *attribute_type=None*)
Compute the square of the residuals per subject per feature

Parameters

data [*Dataset*] Contains the data of the subjects, in particular the subjects' time-points and the mask (?)

param_ind [dict] Contain the individual parameters

attribute_type [Any (default None)] Flag to ask for MCMC attributes instead of model's attributes.

Returns

torch.Tensor of shape (n_individuals,dimension) Contains L2 residual for each subject and each feature

compute_sum_squared_tensorized(*data*, *param_ind*, *attribute_type=None*)

Compute the square of the residuals per subject

Parameters

data [*Dataset*] Contains the data of the subjects, in particular the subjects' time-points and the mask (?)

param_ind [dict] Contain the individual parameters

attribute_type [Any (default None)] Flag to ask for MCMC attributes instead of model's attributes.

Returns

torch.Tensor of shape (n_individuals,) Contains L2 residual for each subject

get_individual_realization_names()

Get names of individual variales of the model.

Returns

list[str]

get_individual_variable_name()

Return list of names of the individual variables from the model.

Duplicate of *get_individual_realization_names()* TODO delete one of them

Returns

individual_variable_name [list [str]] Contains the individual variables' names

get_param_from_real(*realizations*)

Get individual parameters realizations from all model realizations

Parameters

realizations [*CollectionRealization*]

Returns

dict[param_name: str, torch.Tensor [n_individuals, dims_param]] Individual parameters

get_population_realization_names()

Get names of population variales of the model.

Returns

list[str]

`get_realization_object`(*n_individuals*)
Initialization of a *CollectionRealization* used during model fitting.

Parameters

n_individuals [int] Number of individuals to track

Returns

CollectionRealization

`initialize`(*dataset*, *method='default'*)
Initialize the model given a dataset and an initialization method.

After calling this method `is_initialized` should be True and model should be ready for use.

Parameters

dataset [*Dataset*] The dataset we want to initialize from.

method [str] A custom method to initialize the model

`initialize_MCMC_toolbox`()
Initialize Monte-Carlo Markov-Chain toolbox for calibration of model

TODO to move in a "MCMC-model interface"

`load_hyperparameters`(*hyperparameters*)
Load model's hyperparameters

Parameters

hyperparameters [dict[str, Any]] Contains the model's hyperparameters

`load_parameters`(*parameters*)
Instantiate or update the model's parameters.

Parameters

parameters: dict[str, Any] Contains the model's parameters

`random_variable_informations`()
Informations on model's random variables.

Returns

dict[str, Any]

`save`(*path*, *with_mixing_matrix=True*, ***kwargs*)
Save Leaspy object as json model parameter file.

Parameters

path: str Path to store the model's parameters.

with_mixing_matrix: bool (default True) Save the mixing matrix in the exported file in its 'parameters' section. <!> It is not a real parameter and its value will be overwritten at model loading

(orthonormal basis is recomputed from other "true" parameters and mixing matrix is then deduced from this orthonormal basis and the betas)!

It was integrated historically because it is used for convenience in browser webtool and only there…

****kwargs** Keyword arguments for json.dump method.

smart_initialization_realizations(*data*, *realizations*)
Smart initialization of realizations if needed.

Default behavior to return *realizations* as they are (no smart trick).

Parameters

data [*Dataset*]

realizations [*CollectionRealization*]

Returns

CollectionRealization

static time_reparametrization(*timepoints*, *xi*, *tau*)
Tensorized time reparametrization formula

<!> Shapes of tensors must be compatible between them.

Parameters

timepoints [torch.Tensor] Timepoints to reparametrize

xi [torch.Tensor] Log-acceleration of individual(s)

tau [torch.Tensor] Time-shift(s)

Returns

torch.Tensor of same shape as *timepoints*

update_MCMC_toolbox(*name_of_the_variables_that_have_been_changed*, *realizations*)
Update the MCMC toolbox with a collection of realizations of model population parameters.

TODO to move in a "MCMC-model interface"

Parameters

name_of_the_variables_that_have_been_changed: container[str] (list, tuple, ...)
Names of the population parameters to update in MCMC toolbox

realizations [*CollectionRealization*] All the realizations to update MCMC toolbox with

update_model_parameters(*data*, *reals_or_suff_stats*, *burn_in_phase=True*)
Update model parameters (high-level function)

Under-the-hood call *update_model_parameters_burn_in()* or *update_model_parameters_normal()* depending on the phase of the fit algorithm

Parameters

data [*Dataset*]

reals_or_suff_stats :

If during burn-in phase will be realizations: *CollectionRealization*

If after burn-in phase will be sufficient statistics: dict[suff_stat: str, torch.Tensor]

update_model_parameters_burn_in(*data*, *realizations*)
Update model parameters (burn-in phase)

Parameters

data [*Dataset*]

realizations [*CollectionRealization*]

update_model_parameters_normal(*data*, *suff_stats*)
Update model parameters (after burn-in phase)

Parameters

data [*Dataset*]

suff_stats [dict[suff_stat: str, torch.Tensor]]

3.5.6 leaspy.models.univariate_model.UnivariateModel

class leaspy.models.univariate_model.**UnivariateModel**(*name*, ***kwargs*)
Bases: *leaspy.models.abstract_model.AbstractModel*

Univariate (logistic or linear) model for a single variable of interest.

Methods

compute_individual_attachment_tensorized(...)	Compute attachment term (per subject)
compute_individual_attachment_tensorized_mcmc(...)	Compute MCMC attachment of all subjects? One subject? One visit? TODO: complete
compute_individual_tensorized(timepoints, ...)	Compute the individual values at timepoints according to the model.
compute_individual_tensorized_linear(...[, ...])	Compute the individual values at timepoints according to the model (linear).
compute_individual_tensorized_logistic(...)	Compute the individual values at timepoints according to the model (logistic).
compute_individual_trajectory(timepoints, ...)	Compute scores values at the given time-point(s) given a subject's individual parameters.
compute_jacobian_tensorized(timepoints, ...)	Compute the jacobian of the model w.r.t.
compute_jacobian_tensorized_linear(...[, ...])	Compute the jacobian of the model (linear) w.r.t.
compute_jacobian_tensorized_logistic(...[, MCMC])	Compute the jacobian of the model (logistic) w.r.t.
compute_mean_traj(timepoints)	Compute trajectory of the model with individual parameters being the group-average ones.
compute_regularity_realization(realization)	Compute regularity term for a *Realization* instance.
compute_regularity_variable(value, mean, std)	Compute regularity term (Gaussian distribution), low-level.
compute_sufficient_statistics(data, realizations)	Compute sufficient statistics from realizations
compute_sum_squared_per_ft_tensorized(data, ...)	Compute the square of the residuals per subject per feature
compute_sum_squared_tensorized(data, param_ind)	Compute the square of the residuals per subject
get_individual_realization_names()	Get names of individual variales of the model.
get_individual_variable_name()	Return list of names of the individual variables from the model.

continues on next page

Table 40 – continued from previous page

get_param_from_real(realizations)	Get individual parameters realizations from all model realizations
get_population_realization_names()	Get names of population variales of the model.
get_realization_object(n_individuals)	Initialization of a *CollectionRealization* used during model fitting.
initialize(dataset[, method])	Initialize the model given a dataset and an initialization method.
initialize_MCMC_toolbox()	Initialize Monte-Carlo Markov-Chain toolbox for calibration of model
load_hyperparameters(hyperparameters)	Load model's hyperparameters
load_parameters(parameters)	Instantiate or update the model's parameters.
random_variable_informations()	Informations on model's random variables.
save(path, **kwargs)	Save Leaspy object as json model parameter file.
smart_initialization_realizations(data, ...)	Smart initialization of realizations if needed.
time_reparametrization(timepoints, xi, tau)	Tensorized time reparametrization formula
update_MCMC_toolbox(...)	Update the MCMC toolbox with a collection of realizations of model population parameters.
update_model_parameters(data, ...[, ...])	Update model parameters (high-level function)
update_model_parameters_burn_in(data, ...)	Update model parameters (burn-in phase)
update_model_parameters_normal(data, suff_stats)	Update model parameters (after burn-in phase)

__init__(*name*, ***kwargs*)

__str__()
Return str(self).

__weakref__
list of weak references to the object (if defined)

_audit_individual_parameters(*ips*)
Perform various consistency and compatibility (with current model) checks on an individual parameters dict and outputs qualified information about it.

TODO? move to IndividualParameters class?

Parameters

ips: dict Contains some untrusted individual parameters. If representing only one individual (in a multivariate model) it could be:

{'tau':0.1, 'xi':-0.3, 'sources':[0.1,...]}

Or for multiple individuals: {'tau':[0.1,0.2,...], 'xi':[-0.3,0.2,...], 'sources':[[0.1,...],[0,...],...]}

In particular, a sources vector (if present) should always be a array_like, even if it is 1D

Returns

ips_info: dict 'nb_inds': number of individuals present (int >= 0) 'tensorized_ips': tensorized version of individual parameters 'tensorized_ips_gen': generator providing for all individuals present (ordered as is)

their own tensorized individual parameters

Raises

ValueError: if any of the consistency/compatibility checks fail

`static _tensorize_2D`(*x*, *unsqueeze_dim*, *dtype=torch.float32*)
Helper to convert a scalar or array_like into an, at least 2D, dtype tensor

Parameters

x: scalar or array_like element to be tensorized

unsqueeze_dim: 0 or -1 dimension to be unsqueezed; meaningful for 1D array-like only >>> _tensorize_2D([1, 2], 0) == tensor([[1, 2]]) >>> _tensorize_2D([1, 2], -1) == tensor([[1], [2]]) for scalar or vector of length 1 it has no matter

`compute_individual_attachment_tensorized`(*data*, *param_ind*, *attribute_type*)
Compute attachment term (per subject)

Parameters

data [*Dataset*] Contains the data of the subjects, in particular the subjects' time-points and the mask for nan values & padded visits

param_ind: dict Contain the individual parameters

attribute_type: Any, optional Flag to ask for MCMC attributes instead of model's attributes.

Returns

attachment [torch.Tensor] Negative Log-likelihood, shape = (n_subjects,)

`compute_individual_attachment_tensorized_mcmc`(*data*, *realizations*)
Compute MCMC attachment of all subjects? One subject? One visit? TODO: complete

Parameters

data [*Dataset*] Contains the data of the subjects, in particular the subjects' time-points and the mask (?)

realizations [*CollectionRealization*]

Returns

attachment [torch.Tensor] The subject attachment (?)

`compute_individual_tensorized`(*timepoints*, *ind_parameters*, *attribute_type=None*)
Compute the individual values at timepoints according to the model.

Parameters

timepoints [torch.Tensor of shape (n_individuals, n_timepoints)]

individual_parameters [dict[param_name: str, torch.Tensor of shape (n_individuals, n_dims_param)]]

attribute_type: Any (default None) Flag to ask for MCMC attributes instead of model's attributes.

Returns

torch.Tensor of shape (n_individuals, n_timepoints, n_features)

`compute_individual_tensorized_linear`(*timepoints*, *ind_parameters*, *attribute_type=False*)
Compute the individual values at timepoints according to the model (linear).

Parameters

timepoints [torch.Tensor of shape (n_individuals, n_timepoints)]

individual_parameters [dict[param_name: str, torch.Tensor of shape (n_individuals, n_dims_param)]]

attribute_type: Any (default None) Flag to ask for MCMC attributes instead of model's attributes.

Returns

torch.Tensor of shape (n_individuals, n_timepoints, n_features)

compute_individual_tensorized_logistic(*timepoints*, *ind_parameters*, *attribute_type=False*)
Compute the individual values at timepoints according to the model (logistic).

Parameters

timepoints [torch.Tensor of shape (n_individuals, n_timepoints)]

individual_parameters [dict[param_name: str, torch.Tensor of shape (n_individuals, n_dims_param)]]

attribute_type: Any (default None) Flag to ask for MCMC attributes instead of model's attributes.

Returns

torch.Tensor of shape (n_individuals, n_timepoints, n_features)

compute_individual_trajectory(*timepoints*, *individual_parameters*, ***, *skip_ips_checks=False*)
Compute scores values at the given time-point(s) given a subject's individual parameters.

Parameters

timepoints [scalar or array_like[scalar] (list, tuple, numpy.ndarray)] Contains the age(s) of the subject.

individual_parameters: dict Contains the individual parameters. Each individual parameter should be a scalar or array_like

skip_ips_checks: bool (default: False) Flag to skip consistency/compatibility checks and tensorization of individual_parameters when it was done earlier (speed-up)

Returns

torch.Tensor Contains the subject's scores computed at the given age(s) Shape of tensor is (1, n_tpts, n_features)

compute_jacobian_tensorized(*timepoints*, *ind_parameters*, *attribute_type=None*)
Compute the jacobian of the model w.r.t. each individual parameter.

This function aims to be used in *ScipyMinimize* to speed up optimization.

Parameters

timepoints [torch.Tensor of shape (n_individuals, n_timepoints)]

individual_parameters [dict[param_name: str, torch.Tensor of shape (n_individuals, n_dims_param)]]

attribute_type: Any (default None) Flag to ask for MCMC attributes instead of model's attributes.

Returns

dict[param_name: str, torch.Tensor of shape (n_individuals, n_timepoints, n_features, n_dims_param)

compute_jacobian_tensorized_linear(*timepoints*, *ind_parameters*, *attribute_type=None*)
Compute the jacobian of the model (linear) w.r.t. each individual parameter.

This function aims to be used in *ScipyMinimize* to speed up optimization.

Parameters

timepoints [torch.Tensor of shape (n_individuals, n_timepoints)]

individual_parameters [dict[param_name: str, torch.Tensor of shape (n_individuals, n_dims_param)]]

attribute_type: Any (default None) Flag to ask for MCMC attributes instead of model's attributes.

Returns

dict[param_name: str, torch.Tensor of shape (n_individuals, n_timepoints, n_features, n_dims_param)

compute_jacobian_tensorized_logistic(*timepoints*, *ind_parameters*, *MCMC=False*)
Compute the jacobian of the model (logistic) w.r.t. each individual parameter.

This function aims to be used in *ScipyMinimize* to speed up optimization.

Parameters

timepoints [torch.Tensor of shape (n_individuals, n_timepoints)]

individual_parameters [dict[param_name: str, torch.Tensor of shape (n_individuals, n_dims_param)]]

attribute_type: Any (default None) Flag to ask for MCMC attributes instead of model's attributes.

Returns

dict[param_name: str, torch.Tensor of shape (n_individuals, n_timepoints, n_features, n_dims_param)

compute_mean_traj(*timepoints*)
Compute trajectory of the model with individual parameters being the group-average ones.

TODO check dimensions of io? TODO generalize in abstract manifold model

Parameters

timepoints [torch.Tensor [1, n_timepoints]]

Returns

torch.Tensor [1, n_timepoints, dimension] The group-average values at given timepoints

compute_regularity_realization(*realization*)
Compute regularity term for a *Realization* instance.

Parameters

realization [*Realization*]

compute_regularity_variable(*value*, *mean*, *std*)
Compute regularity term (Gaussian distribution), low-level.

Parameters

value, mean, std [torch.Tensor of same shapes]

compute_sufficient_statistics(*data*, *realizations*)
Compute sufficient statistics from realizations

Parameters

data [*Dataset*]

realizations [*CollectionRealization*]

Returns

dict[suff_stat: str, torch.Tensor]

compute_sum_squared_per_ft_tensorized(*data*, *param_ind*, *attribute_type=None*)
Compute the square of the residuals per subject per feature

Parameters

data [*Dataset*] Contains the data of the subjects, in particular the subjects' time-points and the mask (?)

param_ind [dict] Contain the individual parameters

attribute_type [Any (default None)] Flag to ask for MCMC attributes instead of model's attributes.

Returns

torch.Tensor of shape (n_individuals,dimension) Contains L2 residual for each subject and each feature

compute_sum_squared_tensorized(*data*, *param_ind*, *attribute_type=None*)
Compute the square of the residuals per subject

Parameters

data [*Dataset*] Contains the data of the subjects, in particular the subjects' time-points and the mask (?)

param_ind [dict] Contain the individual parameters

attribute_type [Any (default None)] Flag to ask for MCMC attributes instead of model's attributes.

Returns

torch.Tensor of shape (n_individuals,) Contains L2 residual for each subject

get_individual_realization_names()
Get names of individual variales of the model.

Returns

list[str]

get_individual_variable_name()
Return list of names of the individual variables from the model.

Duplicate of *get_individual_realization_names()* TODO delete one of them

Returns

individual_variable_name [list [str]] Contains the individual variables' names

get_param_from_real(*realizations*)
Get individual parameters realizations from all model realizations

Parameters

realizations [*CollectionRealization*]

Returns

dict[param_name: str, torch.Tensor [n_individuals, dims_param]] Individual parameters

`get_population_realization_names()`
Get names of population variales of the model.

Returns

list[str]

`get_realization_object(n_individuals)`
Initialization of a *CollectionRealization* used during model fitting.

Parameters

n_individuals [int] Number of individuals to track

Returns

CollectionRealization

`initialize(dataset, method='default')`
Initialize the model given a dataset and an initialization method.

After calling this method `is_initialized` should be True and model should be ready for use.

Parameters

dataset [*Dataset*] The dataset we want to initialize from.

method [str] A custom method to initialize the model

`initialize_MCMC_toolbox()`
Initialize Monte-Carlo Markov-Chain toolbox for calibration of model

TODO to move in a "MCMC-model interface"

`load_hyperparameters(hyperparameters)`
Load model's hyperparameters

Parameters

hyperparameters [dict[str, Any]] Contains the model's hyperparameters

`load_parameters(parameters)`
Instantiate or update the model's parameters.

Parameters

parameters: dict[str, Any] Contains the model's parameters

`random_variable_informations()`
Informations on model's random variables.

Returns

dict[str, Any]

`save(path, **kwargs)`
Save Leaspy object as json model parameter file.

Parameters

path: str Path to store the model's parameters.

****kwargs** Keyword arguments for json.dump method.

smart_initialization_realizations(*data*, *realizations*)
Smart initialization of realizations if needed.

Default behavior to return *realizations* as they are (no smart trick).

Parameters

data [*Dataset*]

realizations [*CollectionRealization*]

Returns

CollectionRealization

static time_reparametrization(*timepoints*, *xi*, *tau*)
Tensorized time reparametrization formula

<!> Shapes of tensors must be compatible between them.

Parameters

timepoints [torch.Tensor] Timepoints to reparametrize

xi [torch.Tensor] Log-acceleration of individual(s)

tau [torch.Tensor] Time-shift(s)

Returns

torch.Tensor of same shape as *timepoints*

update_MCMC_toolbox(*name_of_the_variables_that_have_been_changed*, *realizations*)
Update the MCMC toolbox with a collection of realizations of model population parameters.

TODO to move in a "MCMC-model interface"

Parameters

name_of_the_variables_that_have_been_changed: container[str] (list, tuple, ...)
Names of the population parameters to update in MCMC toolbox

realizations [*CollectionRealization*] All the realizations to update MCMC toolbox with

update_model_parameters(*data*, *reals_or_suff_stats*, *burn_in_phase=True*)
Update model parameters (high-level function)

Under-the-hood call *update_model_parameters_burn_in()* or *update_model_parameters_normal()* depending on the phase of the fit algorithm

Parameters

data [*Dataset*]

reals_or_suff_stats :

If during burn-in phase will be realizations: *CollectionRealization*

If after burn-in phase will be sufficient statistics: dict[suff_stat: str, torch.Tensor]

update_model_parameters_burn_in(*data*, *realizations*)
Update model parameters (burn-in phase)

Parameters

data [*Dataset*]

realizations [*CollectionRealization*]

update_model_parameters_normal(*data*, *suff_stats*)
Update model parameters (after burn-in phase)

Parameters

data [*Dataset*]

suff_stats [dict[suff_stat: str, torch.Tensor]]

3.5.7 leaspy.models.lme_model.LMEModel

class leaspy.models.lme_model.**LMEModel**(*name*)
Bases: leaspy.models.generic_model.GenericModel

LMEModel is a benchmark model that fits and personalize a linear mixed-effects model

The model specification is the following: $y_{ij} = fixed_{intercept} + random_{intercept_i} + (fixed_{slope} + random_{slope_{age_i}}) * ages_{ij}$ with:
- y_ij: feature array of the i-th patient (n_i visits),
- ages_ij: ages array of the i-th patient (n_i visits)

This model must be fitted on one feature only (univariate model).

See also:

leaspy.algo.others.lme_fit.LMEFitAlgorithm
leaspy.algo.others.lme_personalize.LMEPersonalizeAlgorithm

Attributes

name: str The model's name

parameters: dict Contains the model parameters

features: list[str] List of the model features

Methods

compute_individual_trajectory(timepoints, ip)	Compute scores values at the given time-point(s) given a subject's individual parameters.
get_hyperparameters(*[, with_properties, ...])	Get all model hyperparameters
hyperparameters_ok()	Check all model hyperparameters are ok
load_hyperparameters(hyperparameters)	Load model hyperparameters from a dict
load_parameters(parameters[, list_converter])	Instantiate or update the model's parameters.
save(path, **kwargs)	Save Leaspy object as json model parameter file.

__init__(*name*)

__str__()
Return str(self).

__weakref__
list of weak references to the object (if defined)

compute_individual_trajectory(*timepoints*, *ip*)
Compute scores values at the given time-point(s) given a subject's individual parameters.

Parameters

timepoints: array-like of ages (not normalized) Timepoints to compute individual trajectory at

ip: dict

Individual parameters:

- random_intercept
- random_slope_age (if `with_random_slope_age == True`)

Returns

`torch.Tensor` **of float of shape (n_individuals == 1, n_tpts == len(timepoints), n_features == 1)**

get_hyperparameters(***, *with_properties=True*, *default=None*)
Get all model hyperparameters

Returns

dict

hyperparameters_ok() → bool
Check all model hyperparameters are ok

Returns

bool

load_hyperparameters(*hyperparameters*) → None
Load model hyperparameters from a dict

Parameters

hyperparameters: dict Contains the model's hyperparameters

load_parameters(*parameters*, *list_converter=<built-in function array>*) → None
Instantiate or update the model's parameters.

Parameters

parameters: dict Contains the model's parameters

save(*path*, ***kwargs*)
Save Leaspy object as json model parameter file.

Default save method: it can be overwritten in child class but should be generic…

Parameters

path: str Path to store the model's parameters.

****kwargs** Keyword arguments for json.dump method.

3.5.8 leaspy.models.constant_model.ConstantModel

class leaspy.models.constant_model.**ConstantModel**(*name*)

Bases: leaspy.models.generic_model.GenericModel

ConstantModel is a benchmark model that predicts constant values no matter of the patient's ages.

These constant values depend on the algorithm setting and the patient's values provided during calibration. It could predict:

- *last_known*: last non NaN value seen during calibration*§,
- *last*: last value seen during calibration (even if NaN),
- *max*: maximum (=worst) value seen during calibration*§,
- *mean*: average of values seen during calibration§.

* <!> depending on features, the *last_known* / *max* value may correspond to different visits.

§ <!> for a given feature, value will be NaN if and only if all values for this feature were NaN.

See also:

leaspy.algo.others.constant_prediction_algo.ConstantPredictionAlgorithm

Attributes

name: str The model's name

features: list[str] List of the model features

dimension: int Number of features (read-only)

parameters: dict Population parameters: empty dictionary.

Methods

compute_individual_trajectory(timepoints, ip)	Compute scores values at the given time-point(s) given a subject's individual parameters.
get_hyperparameters(*[, with_properties, ...])	Get all model hyperparameters
hyperparameters_ok()	Check all model hyperparameters are ok
load_hyperparameters(hyperparameters)	Load model hyperparameters from a dict
load_parameters(parameters[, list_converter])	Instantiate or update the model's parameters.
save(path, **kwargs)	Save Leaspy object as json model parameter file.

__init__(*name*)

__str__()

Return str(self).

__weakref__

list of weak references to the object (if defined)

compute_individual_trajectory(*timepoints*, *ip*)

Compute scores values at the given time-point(s) given a subject's individual parameters.

Parameters

timepoints [scalar or array_like[scalar] (list, tuple, numpy.ndarray)] Contains the

age(s) of the subject.

individual_parameters: dict Contains the individual parameters. Each individual parameter should be a scalar or array_like

****kws: Any** extra model specific keyword-arguments

Returns

`torch.Tensor` Contains the subject's scores computed at the given age(s) Shape of tensor is (1, n_tpts, n_features)

`get_hyperparameters`(*, *with_properties=True*, *default=None*)
Get all model hyperparameters

Returns

dict

`hyperparameters_ok`() → bool
Check all model hyperparameters are ok

Returns

bool

`load_hyperparameters`(*hyperparameters*) → None
Load model hyperparameters from a dict

Parameters

hyperparameters: dict Contains the model's hyperparameters

`load_parameters`(*parameters*, *list_converter=<built-in function array>*) → None
Instantiate or update the model's parameters.

Parameters

parameters: dict Contains the model's parameters

`save`(*path*, ***kwargs*)
Save Leaspy object as json model parameter file.

Default save method: it can be overwritten in child class but should be generic…

Parameters

path: str Path to store the model's parameters.

****kwargs** Keyword arguments for json.dump method.

3.5.9 `leaspy.models.utils.attributes`: Models' attributes

Attributes used by the models.

attributes_factory.AttributesFactory()	Return an *Attributes* class object based on the given parameters.
abstract_attributes.AbstractAttributes(name)	Abstract base class for attributes of models.
abstract_manifold_model_attributes.AbstractManifoldModelAttributes(...)	Abstract base class for attributes of leaspy manifold models.
linear_attributes.LinearAttributes(name, ...)	Attributes of leaspy linear models.

continues on next page

Table 43 – continued from previous page

logistic_attributes.LogisticAttributes(name, ...)	Attributes of leaspy logistic models.
logistic_parallel_attributes.LogisticParallelAttributes(...)	Attributes of leaspy logistic parallel models.

leaspy.models.utils.attributes.attributes_factory.AttributesFactory

class leaspy.models.utils.attributes.attributes_factory.**AttributesFactory**

Bases: object

Return an *Attributes* class object based on the given parameters.

Methods

attributes(name, dimension[, source_dimension])	Class method to build correct model attributes depending on model *name*.

__weakref__

list of weak references to the object (if defined)

classmethod attributes(*name*, *dimension*, *source_dimension=None*)

Class method to build correct model attributes depending on model *name*.

Parameters

name: str

dimension [int]

source_dimension [int, optional (default None)]

Returns

AbstractAttributes

leaspy.models.utils.attributes.abstract_attributes.AbstractAttributes

class leaspy.models.utils.attributes.abstract_attributes.**AbstractAttributes**(*name*, *dimension=None*, *source_dimension=None*)

Bases: object

Abstract base class for attributes of models.

Contains the common attributes & methods of the different attributes classes. Such classes are used to update the models' attributes.

Parameters

name: str

dimension: int (default None)

source_dimension: int (default None)

univariate: bool Whether model is univariate or not (i.e. dimension == 1)

has_sources: bool Whether model has sources or not (not univariate and source_dimension >= 1)

Attributes

name: str Name of the associated leaspy model.

dimension: int Number of features of the model

source_dimension: int Number of sources of the model TODO? move to AbstractManifoldModelAttributes?

univariate: bool Whether model is univariate or not (i.e. dimension == 1)

has_sources: bool Whether model has sources or not (not univariate and source_dimension >= 1) TODO? move to AbstractManifoldModelAttributes?

update_possibilities: tuple[str] (default empty) Contains the available parameters to update. Different models have different parameters.

Methods

get_attributes()	Returns the essential attributes of a given model.
update(names_of_changes_values, values)	Update model group average parameter(s).

__init__(*name*, *dimension=None*, *source_dimension=None*)
Instantiate a AbstractAttributes class object.

__weakref__
list of weak references to the object (if defined)

_check_names(*names_of_changed_values*)
Check if the name of the parameter(s) to update are in the possibilities allowed by the model.

Parameters

names_of_changed_values: list [str]

Raises

ValueError

get_attributes()
Returns the essential attributes of a given model.

Returns

Depends on the subclass, please refer to each specific class.

update(*names_of_changes_values*, *values*)
Update model group average parameter(s).

Parameters

names_of_changed_values: list [str] Values to be updated

values: dict [str, `torch.Tensor`] New values used to update the model's group average parameters

Raises

ValueError If *names_of_changed_values* contains unknown values to update.

leaspy.models.utils.attributes.abstract_manifold_model_attributes.AbstractManifoldModelAttributes

class leaspy.models.utils.attributes.abstract_manifold_model_attributes.**AbstractManifoldModelAttributes**

Bases: *leaspy.models.utils.attributes.abstract_attributes.AbstractAttributes*

Abstract base class for attributes of leaspy manifold models.

Contains the common attributes & methods of the different attributes classes. Such classes are used to update the models' attributes.

Parameters

dimension: int (default None)

source_dimension: int (default None)

Attributes

name: str (default None) Name of the associated leaspy model.

dimension: int

source_dimension: int

univariate: bool Whether model is univariate or not (i.e. dimension == 1)

has_sources: bool Whether model has sources or not (not univariate and source_dimension >= 1)

update_possibilities: tuple [str], (default ('all', 'g', 'v0', 'betas')) Contains the available parameters to update. Different models have different parameters.

positions [torch.Tensor [dimension] (default None)] <!> Depending on the submodel it does not correspond to the same thing.

velocities [torch.Tensor [dimension] (default None)] Vector of velocities for each feature (positive components).

orthonormal_basis [torch.Tensor [dimension, dimension - 1] (default None)]

betas [torch.Tensor [dimension - 1, source_dimension] (default None)]

mixing_matrix [torch.Tensor [dimension, source_dimension] (default None)] Matrix A such that w_i = A * s_i.

Methods

get_attributes()	Returns the following attributes: `positions`, `velocities` & `mixing_matrix`.
update(names_of_changes_values, values)	Update model group average parameter(s).

__init__(*name*, *dimension*, *source_dimension*)

Instantiate a AbstractManifoldModelAttributes class object.

Parameters

name: str

dimension: int

source_dimension: int

__weakref__
list of weak references to the object (if defined)

_check_names(*names_of_changed_values*)
Check if the name of the parameter(s) to update are in the possibilities allowed by the model.

Parameters

names_of_changed_values: list [str]

Raises

ValueError

_compute_Q(*dgamma_t0*, *G_metric*, *strip_col=0*)
Householder decomposition, adapted for a non-Euclidean inner product defined by: (1) $< x, y > Metric(p) =< x, G(p)y > Eucl = xTG(p)y$, where: $G(p)$ is the symmetric positive-definite (SPD) matrix defining the metric at point *p*.

The Euclidean case is the special case where *G* is the identity matrix. Product-metric is a special case where *G(p)* is a diagonal matrix (identified to a vector) whose components are all > 0.

It is used in child classes to compute and set in-place the `orthonormal_basis` attribute given the time-derivative of the geodesic at initial time and the *G_metric*. The first component of the full orthonormal basis is a vector collinear *G_metric x dgamma_t0* that we get rid of.

The orthonormal basis we construct is always orthonormal for the Euclidean canonical inner product. But all (but first) vectors of it lie in the sub-space orthogonal (for canonical inner product) to *G_metric * dgamma_t0* which is the same thing that being orthogonal to *dgamma_t0* for the inner product implied by the metric.

[We could do otherwise if we'd like a full orthonormal basis, w.r.t. the non-Euclidean inner product.
But it'd imply to compute G^(-1/2) & G^(1/2) which may be computationally costly in case we don't have direct access to them (for the special case of product-metric it is easy - just the component-wise inverse (sqrt'ed) of diagonal) TODO are there any advantages/drawbacks of one method over the other except this one?

are there any biases between features when only considering Euclidean orthonormal basis?]

Parameters

dgamma_t0: `torch.Tensor` 1D Time-derivative of the geodesic at initial time

G_metric: scalar, `torch.Tensor` 0D, 1D or 2D-square The *G(p)* defining the metric as refered in equation (1) just before. If 0D / scalar: *G* is proportional to the identity matrix If 1D (vector): *G* is a diagonal matrix (diagonal components > 0) If 2D (square matrix): *G* is general (SPD)

strip_col: int in 0..model_dimension-1 (default 0) Which column of the basis should be the one collinear to *dgamma_t0* (that we get rid of)

_compute_betas(*values*)
Update the attribute `betas`.

Parameters

values: dict [str, `torch.Tensor`]

_compute_mixing_matrix()
Update the attribute `mixing_matrix`.

`_compute_velocities`(*values*)
Update the attribute `velocities`.

Parameters

values: dict [str, `torch.Tensor`]

`static _mixing_matrix_utils`(*linear_combination_values*, *matrix*)
Intermediate function used to test the good behaviour of the class' methods.

Parameters

linear_combination_values: `torch.Tensor`

matrix: `torch.Tensor`

Returns

torch.Tensor

`get_attributes`()
Returns the following attributes: `positions`, `velocities` & `mixing_matrix`.

Returns

For univariate models: positions: *torch.Tensor*

For not univariate models:

- positions: *torch.Tensor*
- velocities: *torch.Tensor*
- mixing_matrix: *torch.Tensor*

`update`(*names_of_changes_values*, *values*)
Update model group average parameter(s).

Parameters

names_of_changed_values: list [str] Values to be updated

values: dict [str, `torch.Tensor`] New values used to update the model's group average parameters

Raises

ValueError If *names_of_changed_values* contains unknown values to update.

leaspy.models.utils.attributes.linear_attributes.LinearAttributes

class leaspy.models.utils.attributes.linear_attributes.**LinearAttributes**(*name*, *dimension*, *source_dimension*)

Bases: *leaspy.models.utils.attributes.abstract_manifold_model_attributes.AbstractManifoldModelAttributes*

Attributes of leaspy linear models.

Contains the common attributes & methods to update the linear model's attributes.

See also:

leaspy.models.univariate_model.UnivariateModel
leaspy.models.multivariate_model.MultivariateModel

Attributes

name: str (default 'linear') Name of the associated leaspy model.

dimension: int

source_dimension: int

univariate: bool Whether model is univariate or not (i.e. dimension == 1)

has_sources: bool Whether model has sources or not (not univariate and source_dimension >= 1)

update_possibilities: tuple [str] (default ('all', 'g', 'v0', 'betas')) Contains the available parameters to update. Different models have different parameters.

positions: :class:`torch.Tensor` [dimension] (default None) positions = realizations['g'] such that "p0" = positions

velocities: :class:`torch.Tensor` [dimension] (default None) Always positive: exp(realizations['v0'])

orthonormal_basis [`torch.Tensor` [dimension, dimension - 1] (default None)]

betas [`torch.Tensor` [dimension - 1, source_dimension] (default None)]

mixing_matrix [`torch.Tensor` [dimension, source_dimension] (default None)] Matrix A such that w_i = A * s_i.

Methods

get_attributes()	Returns the following attributes: `positions`, `velocities` & `mixing_matrix`.
update(names_of_changed_values, values)	Update model group average parameter(s).

__init__(*name*, *dimension*, *source_dimension*)
Instantiate a LinearAttributes class object.

Parameters

name: str

dimension: int

source_dimension: int

__weakref__
list of weak references to the object (if defined)

_check_names(*names_of_changed_values*)
Check if the name of the parameter(s) to update are in the possibilities allowed by the model.

Parameters

names_of_changed_values: list [str]

Raises

ValueError

_compute_Q(*dgamma_t0*, *G_metric*, *strip_col=0*)
Householder decomposition, adapted for a non-Euclidean inner product defined by: (1) $< x, y > Metric(p) =< x, G(p)y > Eucl = xTG(p)y$, where: $G(p)$ is the symmetric positive-definite (SPD) matrix defining the metric at point p.

The Euclidean case is the special case where *G* is the identity matrix. Product-metric is a special case where *G(p)* is a diagonal matrix (identified to a vector) whose components are all > 0.

It is used in child classes to compute and set in-place the `orthonormal_basis` attribute given the time-derivative of the geodesic at initial time and the *G_metric*. The first component of the full orthonormal basis is a vector collinear *G_metric x dgamma_t0* that we get rid of.

The orthonormal basis we construct is always orthonormal for the Euclidean canonical inner product. But all (but first) vectors of it lie in the sub-space orthogonal (for canonical inner product) to *G_metric * dgamma_t0* which is the same thing that being orthogonal to *dgamma_t0* for the inner product implied by the metric.

[We could do otherwise if we'd like a full orthonormal basis, w.r.t. the non-Euclidean inner product.
But it'd imply to compute G^(-1/2) & G^(1/2) which may be computationally costly in case we don't have direct access to them (for the special case of product-metric it is easy - just the component-wise inverse (sqrt'ed) of diagonal) TODO are there any advantages/drawbacks of one method over the other except this one?

are there any biases between features when only considering Euclidean orthonormal basis?]

Parameters

dgamma_t0: `torch.Tensor` 1D Time-derivative of the geodesic at initial time

G_metric: scalar, `torch.Tensor` 0D, 1D or 2D-square The *G(p)* defining the metric as refered in equation (1) just before. If 0D / scalar: *G* is proportional to the identity matrix If 1D (vector): *G* is a diagonal matrix (diagonal components > 0) If 2D (square matrix): *G* is general (SPD)

strip_col: int in 0..model_dimension-1 (default 0) Which column of the basis should be the one collinear to *dgamma_t0* (that we get rid of)

`_compute_betas`(*values*)
Update the attribute `betas`.

Parameters

values: dict [str, `torch.Tensor`]

`_compute_mixing_matrix`()
Update the attribute `mixing_matrix`.

`_compute_orthonormal_basis`()
Compute the attribute `orthonormal_basis` which is an orthonormal basis, w.r.t the canonical inner product, of the sub-space orthogonal, w.r.t the inner product implied by the metric, to the time-derivative of the geodesic at initial time. In linear case, this inner product corresponds to canonical Euclidean one.

`_compute_positions`(*values*)
Update the attribute `positions`.

Parameters

values: dict [str, `torch.Tensor`]

`_compute_velocities`(*values*)
Update the attribute `velocities`.

Parameters

values: dict [str, `torch.Tensor`]

static _mixing_matrix_utils(*linear_combination_values*, *matrix*)
Intermediate function used to test the good behaviour of the class' methods.

Parameters

linear_combination_values: `torch.Tensor`

matrix: `torch.Tensor`

Returns

torch.Tensor

get_attributes()
Returns the following attributes: `positions`, `velocities` & `mixing_matrix`.

Returns

For univariate models: positions: *torch.Tensor*

For not univariate models:

- positions: *torch.Tensor*
- velocities: *torch.Tensor*
- mixing_matrix: *torch.Tensor*

update(*names_of_changed_values*, *values*)
Update model group average parameter(s).

Parameters

names_of_changed_values: list [str]

Elements of list must be either:

- `all` (update everything)
- `g` correspond to the attribute `positions`.
- `v0` (`xi_mean` if univariate) correspond to the attribute `velocities`.
- `betas` correspond to the linear combinaison of columns from the orthonormal basis so to derive the `mixing_matrix`.

values: dict [str, `torch.Tensor`] New values used to update the model's group average parameters

Raises

ValueError If *names_of_changed_values* contains unknown parameters.

leaspy.models.utils.attributes.logistic_attributes.LogisticAttributes

class leaspy.models.utils.attributes.logistic_attributes.**LogisticAttributes**(*name*, *dimension*, *source_dimension*)

Bases: *leaspy.models.utils.attributes.abstract_manifold_model_attributes.AbstractManifoldModelAttributes*

Attributes of leaspy logistic models.

Contains the common attributes & methods to update the logistic model's attributes.

See also:

leaspy.models.univariate_model.UnivariateModel

leaspy.models.multivariate_model.MultivariateModel

Attributes

name: str (default 'logistic') Name of the associated leaspy model.

dimension: int

source_dimension: int

univariate: bool Whether model is univariate or not (i.e. dimension == 1)

has_sources: bool Whether model has sources or not (not univariate and source_dimension >= 1)

update_possibilities: tuple [str] (default ('all', 'g', 'v0', 'betas')) Contains the available parameters to update. Different models have different parameters.

positions: :class:`torch.Tensor` [dimension] (default None) positions = exp(realizations['g']) such that "p0" = 1 / (1 + positions)

velocities: :class:`torch.Tensor` [dimension] (default None) Always positive: exp(realizations['v0'])

orthonormal_basis [`torch.Tensor` [dimension, dimension - 1] (default None)]

betas [`torch.Tensor` [dimension - 1, source_dimension] (default None)]

mixing_matrix [`torch.Tensor` [dimension, source_dimension] (default None)] Matrix A such that w_i = A * s_i.

Methods

get_attributes()	Returns the following attributes: `positions`, `velocities` & `mixing_matrix`.
update(names_of_changed_values, values)	Update model group average parameter(s).

__init__(*name*, *dimension*, *source_dimension*)
Instantiate a *LogisticAttributes* class object.

Parameters

name: str

dimension: int

source_dimension: int

__weakref__
list of weak references to the object (if defined)

_check_names(*names_of_changed_values*)
Check if the name of the parameter(s) to update are in the possibilities allowed by the model.

Parameters

names_of_changed_values: list [str]

Raises

ValueError

_compute_Q(*dgamma_t0*, *G_metric*, *strip_col=0*)

Householder decomposition, adapted for a non-Euclidean inner product defined by: $(1) < x, y > Metric(p) =< x, G(p)y > Eucl = xTG(p)y$, where: $G(p)$ is the symmetric positive-definite (SPD) matrix defining the metric at point *p*.

The Euclidean case is the special case where *G* is the identity matrix. Product-metric is a special case where *G(p)* is a diagonal matrix (identified to a vector) whose components are all > 0.

It is used in child classes to compute and set in-place the `orthonormal_basis` attribute given the time-derivative of the geodesic at initial time and the *G_metric*. The first component of the full orthonormal basis is a vector collinear *G_metric x dgamma_t0* that we get rid of.

The orthonormal basis we construct is always orthonormal for the Euclidean canonical inner product. But all (but first) vectors of it lie in the sub-space orthogonal (for canonical inner product) to *G_metric * dgamma_t0* which is the same thing that being orthogonal to *dgamma_t0* for the inner product implied by the metric.

[We could do otherwise if we'd like a full orthonormal basis, w.r.t. the non-Euclidean inner product.

But it'd imply to compute G^(-1/2) & G^(1/2) which may be computationally costly in case we don't have direct access to them (for the special case of product-metric it is easy - just the component-wise inverse (sqrt'ed) of diagonal) TODO are there any advantages/drawbacks of one method over the other except this one?

are there any biases between features when only considering Euclidean orthonormal basis?]

Parameters

dgamma_t0: `torch.Tensor` 1D Time-derivative of the geodesic at initial time

G_metric: scalar, `torch.Tensor` 0D, 1D or 2D-square The *G(p)* defining the metric as refered in equation (1) just before. If 0D / scalar: *G* is proportional to the identity matrix If 1D (vector): *G* is a diagonal matrix (diagonal components > 0) If 2D (square matrix): *G* is general (SPD)

strip_col: int in 0..model_dimension-1 (default 0) Which column of the basis should be the one collinear to *dgamma_t0* (that we get rid of)

_compute_betas(*values*)

Update the attribute `betas`.

Parameters

values: dict [str, `torch.Tensor`]

_compute_mixing_matrix()

Update the attribute `mixing_matrix`.

_compute_orthonormal_basis()

Compute the attribute `orthonormal_basis` which is an orthonormal basis, w.r.t the canonical inner product, of the sub-space orthogonal, w.r.t the inner product implied by the metric, to the time-derivative of the geodesic at initial time.

_compute_positions(*values*)

Update the attribute `positions`.

Parameters

values: dict [str, `torch.Tensor`]

_compute_velocities(*values*)

Update the attribute `velocities`.

Parameters

values: dict [str, `torch.Tensor`]

`static _mixing_matrix_utils`(*linear_combination_values*, *matrix*)
Intermediate function used to test the good behaviour of the class' methods.

Parameters

linear_combination_values: `torch.Tensor`

matrix: `torch.Tensor`

Returns

torch.Tensor

`get_attributes`()
Returns the following attributes: `positions`, `velocities` & `mixing_matrix`.

Returns

For univariate models: positions: *torch.Tensor*

For not univariate models:

- positions: *torch.Tensor*
- velocities: *torch.Tensor*
- mixing_matrix: *torch.Tensor*

`update`(*names_of_changed_values*, *values*)
Update model group average parameter(s).

Parameters

names_of_changed_values: list [str]

Elements of list must be either:

- `all` (update everything)
- `g` correspond to the attribute `positions`.
- `v0` (`xi_mean` if univariate) correspond to the attribute `velocities`.
- `betas` correspond to the linear combinaison of columns from the orthonormal basis so to derive the `mixing_matrix`.

values: dict [str, `torch.Tensor`] New values used to update the model's group average parameters

Raises

ValueError If *names_of_changed_values* contains unknown parameters.

leaspy.models.utils.attributes.logistic_parallel_attributes.LogisticParallelAttributes

class leaspy.models.utils.attributes.logistic_parallel_attributes.**LogisticParallelAttributes**(*name*, *dimension*, *source_dimen*

Bases: *leaspy.models.utils.attributes.abstract_manifold_model_attributes.AbstractManifoldModelAttributes*

Attributes of leaspy logistic parallel models.

Contains the common attributes & methods of the logistic parallel models' attributes.

See also:

leaspy.models.multivariate_parallel_model.MultivariateParallelModel

Attributes

name: str (default 'logistic_parallel') Name of the associated leaspy model.

dimension: int

source_dimension: int

has_sources: bool Whether model has sources or not (source_dimension >= 1)

update_possibilities: tuple [str] (default ('all', 'g', 'xi_mean', 'deltas', 'betas')) Contains the available parameters to update. Different models have different parameters.

positions: :class:`torch.Tensor` (scalar) (default None) positions = exp(realizations['g']) such that "p0" = 1 / (1 + positions * exp(-deltas))

deltas: :class:`torch.Tensor` [dimension] (default None) deltas = [0, delta_2_realization, ..., delta_n_realization]

velocities: :class:`torch.Tensor` (scalar) (default None) Always positive: exp(realizations['xi_mean'])

orthonormal_basis [torch.Tensor [dimension, dimension - 1] (default None)]

betas [torch.Tensor [dimension - 1, source_dimension] (default None)]

mixing_matrix [torch.Tensor [dimension, source_dimension] (default None)] Matrix A such that w_i = A * s_i.

Methods

get_attributes()	Returns the following attributes: `positions`, `deltas` & `mixing_matrix`.
update(names_of_changed_values, values)	Update model group average parameter(s).

__init__(*name*, *dimension*, *source_dimension*)

Instantiate a *LogisticParallelAttributes* class object.

Parameters

name: str

dimension: int

source_dimension: int

__weakref__
list of weak references to the object (if defined)

_check_names(*names_of_changed_values*)
Check if the name of the parameter(s) to update are in the possibilities allowed by the model.

Parameters

names_of_changed_values: list [str]

Raises

ValueError

_compute_Q(*dgamma_t0*, *G_metric*, *strip_col=0*)
Householder decomposition, adapted for a non-Euclidean inner product defined by: (1) $< x, y > Metric(p) =< x, G(p)y > Eucl = xTG(p)y$, where: $G(p)$ is the symmetric positive-definite (SPD) matrix defining the metric at point p.

The Euclidean case is the special case where *G* is the identity matrix. Product-metric is a special case where *G(p)* is a diagonal matrix (identified to a vector) whose components are all > 0.

It is used in child classes to compute and set in-place the `orthonormal_basis` attribute given the time-derivative of the geodesic at initial time and the *G_metric*. The first component of the full orthonormal basis is a vector collinear *G_metric x dgamma_t0* that we get rid of.

The orthonormal basis we construct is always orthonormal for the Euclidean canonical inner product. But all (but first) vectors of it lie in the sub-space orthogonal (for canonical inner product) to *G_metric * dgamma_t0* which is the same thing that being orthogonal to *dgamma_t0* for the inner product implied by the metric.

[We could do otherwise if we'd like a full orthonormal basis, w.r.t. the non-Euclidean inner product.
But it'd imply to compute G^(-1/2) & G^(1/2) which may be computationally costly in case we don't have direct access to them (for the special case of product-metric it is easy - just the component-wise inverse (sqrt'ed) of diagonal) TODO are there any advantages/drawbacks of one method over the other except this one?

are there any biases between features when only considering Euclidean orthonormal basis?]

Parameters

dgamma_t0: `torch.Tensor` 1D Time-derivative of the geodesic at initial time

G_metric: scalar, `torch.Tensor` 0D, 1D or 2D-square The *G(p)* defining the metric as refered in equation (1) just before. If 0D / scalar: *G* is proportional to the identity matrix If 1D (vector): *G* is a diagonal matrix (diagonal components > 0) If 2D (square matrix): *G* is general (SPD)

strip_col: int in 0..model_dimension-1 (default 0) Which column of the basis should be the one collinear to *dgamma_t0* (that we get rid of)

_compute_betas(*values*)
Update the attribute `betas`.

Parameters

values: dict [str, `torch.Tensor`]

_compute_deltas(*values*)
Update` the attribute `deltas`.

Parameters

values: dict [str, `torch.Tensor`]

`_compute_gamma_dgamma_t0()`
Computes both gamma: - value at t0 - derivative w.r.t. time at time t0

Returns

2-tuple: gamma_t0: *torch.Tensor* 1D dgamma_t0: *torch.Tensor* 1D

`_compute_mixing_matrix()`
Update the attribute `mixing_matrix`.

`_compute_orthonormal_basis()`
Compute the attribute `orthonormal_basis` which is an orthonormal basis, w.r.t the canonical inner product, of the sub-space orthogonal, w.r.t the inner product implied by the metric, to the time-derivative of the geodesic at initial time.

`_compute_positions(values)`
Update the attribute `positions`.

Parameters

values: dict [str, `torch.Tensor`]

`_compute_velocities(values)`
Update the attribute `velocities`.

Parameters

values: dict [str, `torch.Tensor`]

`static _mixing_matrix_utils(linear_combination_values, matrix)`
Intermediate function used to test the good behaviour of the class' methods.

Parameters

linear_combination_values: `torch.Tensor`

matrix: `torch.Tensor`

Returns

torch.Tensor

`get_attributes()`
Returns the following attributes: `positions`, `deltas` & `mixing_matrix`.

Returns

positions: *torch.Tensor*

deltas: *torch.Tensor*

mixing_matrix: *torch.Tensor*

`update(names_of_changed_values, values)`
Update model group average parameter(s).

Parameters

names_of_changed_values: list [str]

Elements of list must be either:

- `all` (update everything)

- `g` correspond to the attribute `positions`.
- `xi_mean` correspond to the attribute `velocities`.
- `deltas` correspond to the attribute `deltas`.
- `betas` correspond to the linear combinaison of columns from the orthonormal basis so to derive the `mixing_matrix`.

values: dict [str, `torch.Tensor`] New values used to update the model's group average parameters

Raises

ValueError If *names_of_changed_values* contains unknown parameters.

3.5.10 `leaspy.models.utils.initialization`: Initialization methods

model_initialization.initialize_parameters(...)	Initialize the model's group parameters given its name & the scores of all subjects.

leaspy.models.utils.initialization.model_initialization.initialize_parameters

leaspy.models.utils.initialization.model_initialization.**initialize_parameters**(*model*, *dataset*, *method='default'*)

Initialize the model's group parameters given its name & the scores of all subjects.
Under-the-hood it calls an initialization function dedicated for the *model*:

- `initialize_linear()` (including when *univariate*)
- `initialize_logistic()` (including when *univariate*)
- `initialize_logistic_parallel()`

It is automatically called during *Leaspy.fit()*.

Parameters

model [*AbstractModel*] The model to initialize.

dataset [*Dataset*] Contains the individual scores.

method: str

Must be one of:

- `'default'`: initialize at mean.
- `'random'`: initialize with a gaussian realization with same mean and variance.

Returns

parameters: dict [str, torch.Tensor] Contains the initialized model's group parameters.

CHAPTER

FOUR

USER GUIDE

TODO

4.1 Mathematical aspects

4.1.1 Introduction

TODO

4.1.2 Mathematical formulation

TODO

4.1.3 Riemanian framework

TODO

4.1.4 Missing data

4.2 Leaspy's tutorial

4.2.1 What do I need?

TODO

4.2.2 Derive the population parameters

TODO

4.2.3 Derive the individual parameters

TODO

4.2.4 Cofactor analysis

TODO

4.2.5 What about missing values?

TODO

4.2.6 Predictions

TODO

4.2.7 Simulations

TODO

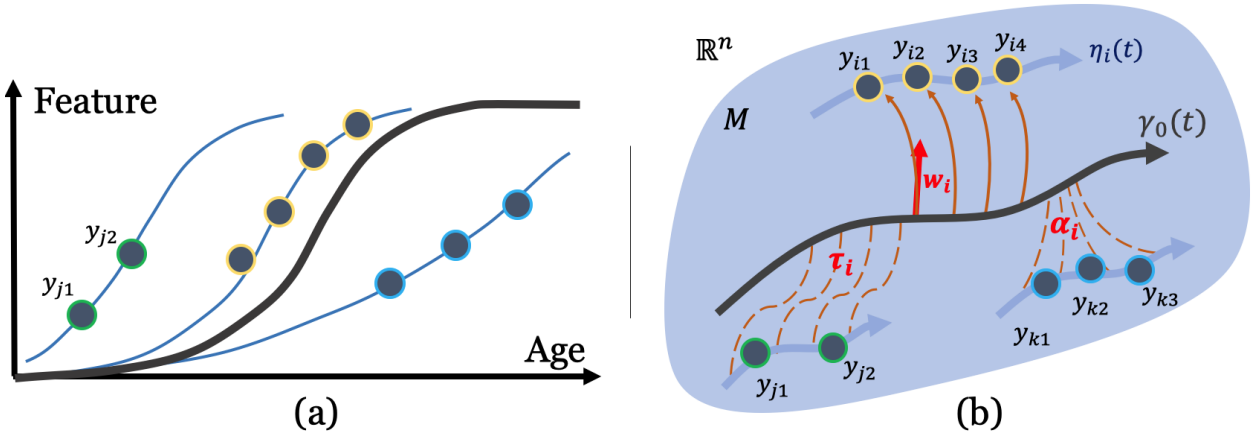
CHAPTER

FIVE

LEARNING SPATIOTEMPORAL PATTERNS IN PYTHON

5.1 Description

Leaspy is a software package for the statistical analysis of **longitudinal data**, particularly **medical** data that comes in a form of **repeated observations** of patients at different time-points.



Considering these series of short-term data, the software aims at :

- Recombining them to reconstruct the long-term spatio-temporal trajectory of evolution
- Positioning each patient observations relatively to the group-average timeline, in term of both temporal differences (time shift and acceleration factor) and spatial differences (diffent sequences of events, spatial pattern of progression, ...)
- Quantifying impact of cofactors (gender, genetic mutation, environmental factors, ...) on the evolution of the signal
- Imputing missing values
- Predicting future observations
- Simulating virtual patients to unbias the initial cohort or mimic its characteristics

The software package can be used with scalar multivariate data whose progression can be modeled by a logistic shape, an exponential decay or a linear progression. The simplest type of data handled by the software are scalar data: they correspond to one (univariate) or multiple (multivariate) measurement(s) per patient observation. This includes, for instance, clinical scores, cognitive assessments, physiological measurements (e.g. blood markers, radioactive markers) but also imaging-derived data that are rescaled, for instance, between 0 and 1 to describe a logistic progression.

5.2 Getting started

Information to install, test, and contribute to the package.

5.3 User Guide

The main documentation. This contains an in-depth description of all algorithms and how to apply them.

5.4 API Documentation

The exact API of all functions and classes, as given in the docstrings. The API documents expected types and allowed features for all functions, and all parameters available for the algorithms.

5.5 License

5.6 Further information

More detailed explanations about the models themselves and about the estimation procedure can be found in the following articles :

- **Mathematical framework**: *A Bayesian mixed-effects model to learn trajectories of changes from repeated manifold-valued observations.* Jean-Baptiste Schiratti, Stéphanie Allassonnière, Olivier Colliot, and Stanley Durrleman. The Journal of Machine Learning Research, 18:1–33, December 2017. Open Access
- **Application to imaging data**: *Statistical learning of spatiotemporal patterns from longitudinal manifold-valued networks.* I. Koval, J.-B. Schiratti, A. Routier, M. Bacci, O. Colliot, S. Allassonnière and S. Durrleman. MICCAI, September 2017. Open Access
- **Application to imaging data**: *Spatiotemporal Propagation of the Cortical Atrophy: Population and Individual Patterns.* Igor Koval, Jean-Baptiste Schiratti, Alexandre Routier, Michael Bacci, Olivier Colliot, Stéphanie Allassonnière, and Stanley Durrleman. Front Neurol. 2018 May 4;9:235. Open Access
- **Application to data with missing values**: *Learning disease progression models with longitudinal data and missing values*. R. Couronne, M. Vidailhet, JC. Corvol, S. Lehéricy, S. Durrleman. ISBI, April 2019. Open Access
- **Intensive application for Alzheimer's Disease progression**: *AD Course Map charts Alzheimer's disease progression*, I. Koval, A. Bone, M. Louis, S. Bottani, A. Marcoux, J. Samper-Gonzalez, N. Burgos, B. Charlier, A. Bertrand, S. Epelbaum, O. Colliot, S. Allassonniere & S. Durrleman, Under review Open Access
- www.digital-brain.org : Website related to the application of the model for Alzheimer's disease.

INDEX

Symbols

A

C

D

E

F

G

H

I

K

L

M

O

P

R

S

T

U