Leaspy

Release 2.0.0-dev

Igor Koval, Raphael Couronne, Etienne Maheux, Arnaud Valladier,

Jun 14, 2023

GETTING STARTED


leaspy

CHAPTER

ONE

INSTALLATION

1.1 Package installation

1. Leaspy requires Python >= 3.8
2. Create a dedicated environment (optional):

Using `conda`:

```
conda create --name leaspy python=3.9
conda activate leaspy
```

Or using `pyenv`:

```
pyenv virtualenv leaspy
pyenv local leaspy
```

3. Install `leaspy` with `pip`:

```
pip install leaspy
```

It will automatically install all needed dependencies.

1.2 Notebook configuration

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` notebooks

```
conda install 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 first load synthetic data, using a `Loader` object, to get of a grasp of longitudinal data.

```
>>> 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
```

The data correspond to repeated visits (*TIME* index) of different participants (*ID* index). Each visit corresponds to the measurement of 4 different variables : the *MMSE*, the RAVLT, the FAQ and the FDG PET.

If plotted, the data would look like the following:

where each color corresponds to a variable, and the connected dots corresponds to the repeated visits of a single participant.

Not very engaging, right ? To go a step further, let's first encapsulate the data into the main `Data` container.

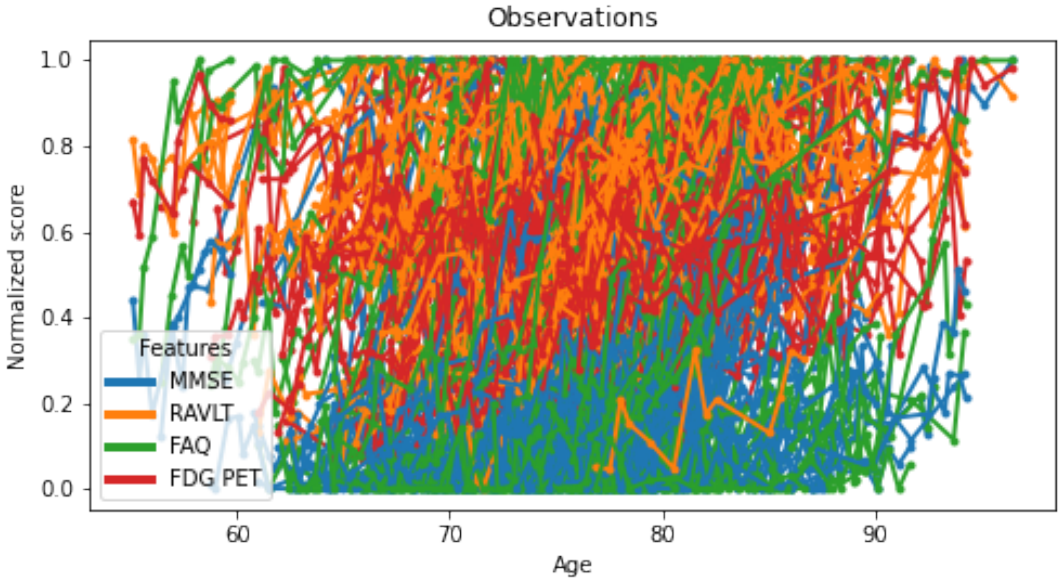
```
>>> data = Data.from_dataframe(alzheimer_df)
```

Leaspy core functionality is to estimate the group-average trajectory of the different variables that are measured in a population.

Let's initialize the `Leaspy` object:

```
>>> leaspy_logistic = Leaspy('logistic', source_dimension=2)
```

as well as the algorithm needed to estimate the group-average trajectory:




```
>>> fit_settings = AlgorithmSettings('mcmc_saem', seed=0, n_iter=8000)
```

We then use the `Leaspy.fit()` method to estimate the group average trajectory:

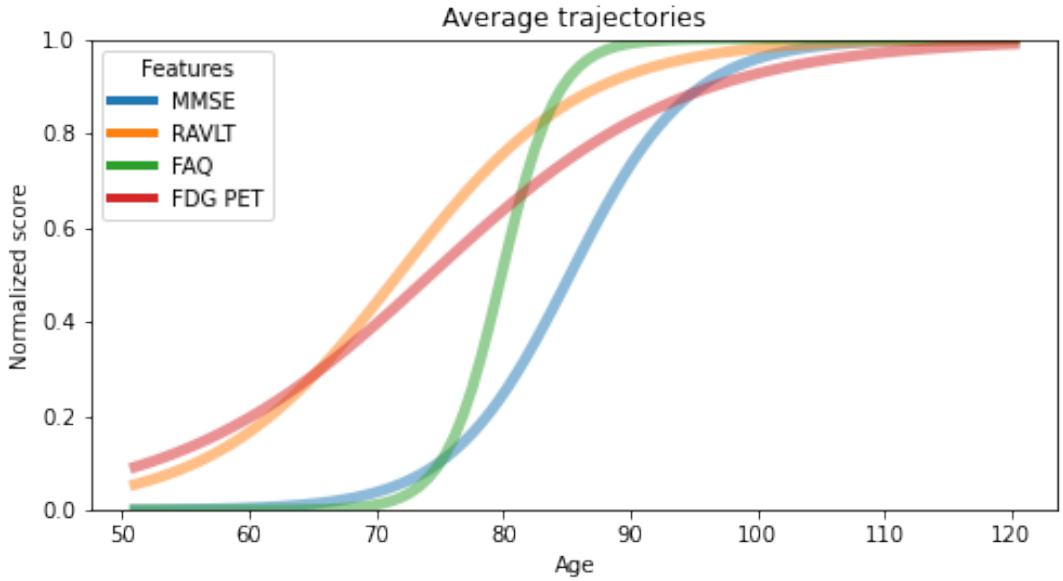
```
>>> leaspy_logistic.fit(data, fit_settings)
==> Setting seed to 0
|##################################################|   8000/8000 iterations
Fit with `mcmc_saem` took: 6m 57s
The standard deviation of the noise at the end of the fit is:
MMSE: 6.50%
RAVLT: 7.63%
FAQ: 6.67%
FDG PET: 7.87%
```

If we were to plot the measured average progression of the variables - see started example notebook for details - it would look like the following

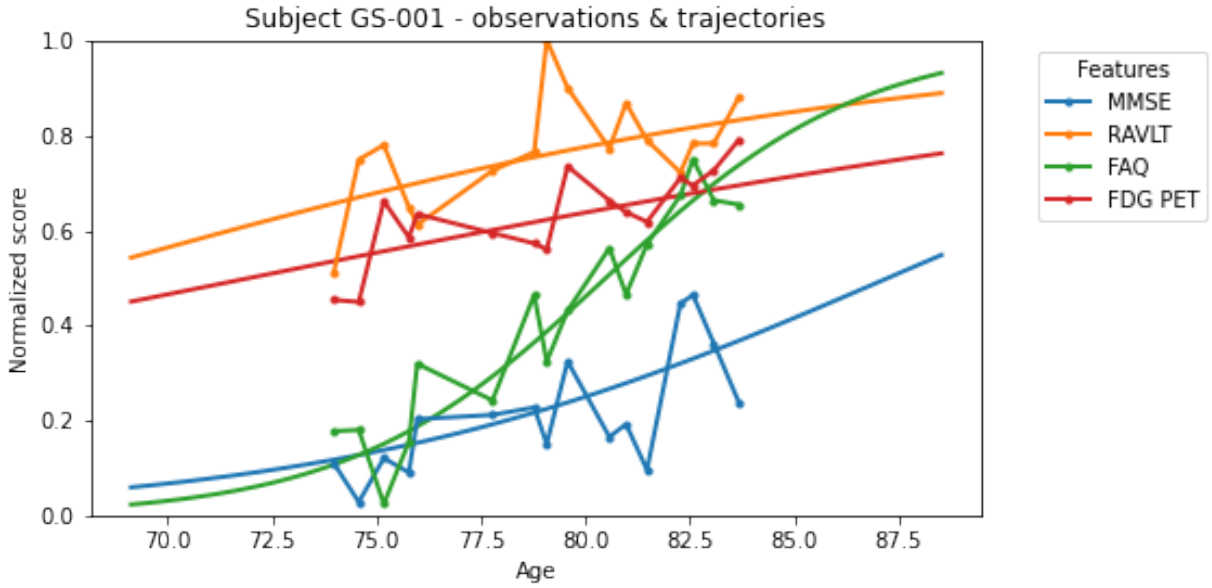
We can also derive the individual trajectory of each subject.

To do this, we use the `Leaspy.personalize()` method, again by providing the proper `AlgorithmSettings`.

```
>>> personalize_settings = AlgorithmSettings('scipy_minimize', seed=0)
>>> individual_parameters = leaspy_logistic.personalize(data, personalize_settings)
 ==> Setting seed to 0
|##################################################|   200/200 subjects
Personalize with `scipy_minimize` took: 9s
The standard deviation of the noise at the end of the personalize is:
MMSE: 6.32%
RAVLT: 7.27%
FAQ: 6.29%
FDG PET: 7.49%
```



Plotting the input participant data against its personalization would give the following - see started example notebook for details.



2.2 Using my own data

2.2.1 Data format

Leaspy uses 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 at 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
        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 observed variables (also named *features* or *endpoints*). In this fashion, you have one column per *feature* and one line per *visit*.

2.2.2 Data scale & constraints

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

2.2.3 Missing data

Leaspy automatically handles missing data as long as they are encoded as `nan` in your `pandas.DataFrame`, or as empty values in your *csv* file.

2.3 Going further

You can check the *User guide* and the *full API documentation*. You can also dive into the started example of the Leaspy repository. The Disease Progression Modelling website also hosts a mathematical introduction and tutorials for Leaspy.

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`(*model_name: str*, ***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.
- `'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.
- `'univariate_logistic'` - a 'logistic' model for a single modality.
- `'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.

noise_model
[str] *For manifold-like models*. Define the noise structure of the model, can be either:

- `'gaussian_scalar'`: gaussian error, with same standard deviation for all features
- `'gaussian_diagonal'`: gaussian error, with one standard deviation parameter per feature (default)
- `'bernoulli'`: for binary data (Bernoulli realization)
- `'ordinal'` or `'ordinal_ranking'`: for ordinal data. WARNING : make sure your dataset only contains positive integers.

source_dimension
[int, optional] *For multivariate models only*. Set the degrees of freedom for _spatial_ variability. This number MUST BE strictly lower than the number of features. By default, this number is equal to square root of the number of features. One can interpret this hyperparameter as a way to reduce the dimension of inter-individual _spatial_ variability between progressions.

batch_deltas_ordinal
[bool, optional] *For logistic models with ordinal noise model only*. If True, concatenates the deltas for each feature into a 2-dimensional Tensor "deltas" model parameter, which essentially allows faster sampling with new samplers. If False, each feature will induce a new model parameter "deltas_<feature_name>". The default is False but it is preferable to switch to True when ordinal items have many levels or when there are many items (when fit takes too long basically). Batching deltas will speed up the sampling part of the MCMC SAEM by trading for less accuracy in the estimation of deltas.

See also:

`leaspy.models`
`ModelFactory`
`Data`
`AlgorithmSettings`
`leaspy.algo`
`IndividualParameters`

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.

Methods

`calibrate`(data, 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 $(t_{i,j})_j$.
`estimate_ages_from_biomarker_values`(...[, ...])	For individuals characterized by their individual parameters z_i, returns the age $t_{i,j}$ at which a given feature value $y_{i,j,k}$ is reached.
`fit`(data, 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_loss])	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.

calibrate(*data: Data*, *settings: AlgorithmSettings*) → None

Duplicates of the `fit()` method.

check_if_initialized() → None

Check if model is initialized.

Raises

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

estimate(*timepoints: pd.MultiIndex | Dict[IDType, List[float]]*, *individual_parameters: IndividualParameters*, *, *to_dataframe: bool = None*, *ordinal_method: str = 'MLE'*) → pd.DataFrame | Dict[IDType, np.ndarray]

Return the model values for individuals characterized by their individual parameters z_i at time-points $(t_{i,j})_j$.

Parameters

timepoints
[dictionary {str/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*

ordinal_method
[str] <!> Only used for ordinal models. * 'MLE' or 'maximum_likelihood' returns maximum likelihood estimator for each point (int) * 'E' or 'expectation' returns expectation (float) * 'P' or 'probabilities' returns probabilities of all levels (array[float]).

Returns

individual_trajectory
[pandas.DataFrame or dict (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')
>>> df_train = Loader.load_dataset('parkinson-putamen-train_and_test').xs('train
↪', level='SPLIT')
>>> timepoints = {'GS-001': (70, 74, 80), 'GS-002': (71, 72)}  # as dict
>>> timepoints = df_train.sort_index().groupby('ID').tail(2).index  # as pandas␣
↪(ID, TIME) MultiIndex
>>> estimations = leaspy_logistic.estimate(timepoints, individual_parameters)
```

estimate_ages_from_biomarker_values(*individual_parameters: IndividualParameters*, *biomarker_values: Dict[str, List[float] | float]*, *feature: str | None = None*) → Dict[str, List[float] | float]

For individuals characterized by their individual parameters z_i, returns the age $t_{i,j}$ at which a given feature value $y_{i,j,k}$ is reached.

Parameters

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

biomarker_values
[Dict[Union[str, int], Union[List, float]]] Dictionary that associates to each patient (being a key of the dictionary) a value (float between 0 and 1, or a list of such floats) from which leaspy will estimate the age at which the value is reached. TODO? shouldn't we allow pandas.Series / pandas.DataFrame

feature
[str] For multivariate models only: feature name (indicates to which model feature the biomarker values belongs)

Returns

biomarker_ages
Dictionary that associates to each patient (being a key of the dictionary) the corresponding age (or ages) for which the value(s) from biomarker_values have been reached. Same format as biomarker values.

Raises

LeaspyTypeError
bad types for input

LeaspyInputError
inconsistent inputs

Examples

Given the individual parameters of two subjects, and the feature value of 0.2 for the first and 0.5 and 0.6 for the second, get the corresponding estimated ages at which these values will be reached.

```
>>> from leaspy.datasets import Loader
>>> leaspy_logistic = Loader.load_leaspy_instance('parkinson-putamen-train')
>>> individual_parameters = Loader.load_individual_parameters('parkinson-
↪putamen-train')
>>> biomarker_values = {'GS-001': [0.2], 'GS-002': [0.5, 0.6]}
# Here the 'feature' argument is optional, as the model is univariate
>>> estimated_ages = leaspy_logistic.estimate_ages_from_biomarker_
↪values(individual_parameters, biomarker_values,
>>> feature='PUTAMEN')
```

fit(*data: Data*, *settings: AlgorithmSettings*) → None

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 $(y_{i,j})$.

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

See also:

leaspy.algo.fit

Examples

Fit a logistic model on a longitudinal dataset, display the group parameters

```
>>> 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', seed=0)
>>> settings.set_logs('path/to/logs', console_print_periodicity=50)
>>> 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])
tau_mean : 68.56787872314453
tau_std : 10.12782096862793
```

(continues on next page)

(continued from previous page)

```
xi_mean : -2.3396952152252197
xi_std : 0.5421289801597595
noise_std : 0.021265486255288124
```

classmethod load(*path_to_model_settings: str*) → Leaspy

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: Data*, *settings: AlgorithmSettings*, *, *return_loss: bool = 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_loss
[bool (default False)] Returns a tuple (individual_parameters, loss) if True

Returns

ips
[`IndividualParameters`] Contains individual parameters

if return_loss is True
[tuple]

- ips : `IndividualParameters`
- loss : `torch.Tensor`

Raises

`LeaspyInputError`
if model is not initialized.

See also:

`leaspy.algo.personalize`

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', 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: str*, ***kwargs*) → None

Save Leaspy object as json model parameter file.

Parameters

path
[str] Path to store the model's parameters.

****kwargs**
Keyword arguments for `save()` (including those sent to `json.dump()` function).

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', seed=0)
```

(continues on next page)

(continued from previous page)

```
>>> 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')
```

simulate(*individual_parameters: IndividualParameters*, *data: Data*, *settings: AlgorithmSettings*)

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 distribution. 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.

See also:

`SimulationAlgorithm`

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 noise to these estimations, which is the same noise-model as the one from your model by default. But, you may customize it by setting the *noise* keyword.

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, noise=
↪'bernoulli')
>>> 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
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 example *'GS-001'*. Furthermore, you want to set the level of

noise around 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
```

class Leaspy(*model_name: str*, ***kwargs*)

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.
- `'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.
- `'univariate_logistic'` - a 'logistic' model for a single modality.
- `'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.

noise_model

[str] *For manifold-like models*. Define the noise structure of the model, can be either:

- `'gaussian_scalar'`: gaussian error, with same standard deviation for all features
- `'gaussian_diagonal'`: gaussian error, with one standard deviation parameter per feature (default)
- `'bernoulli'`: for binary data (Bernoulli realization)
- `'ordinal'` or `'ordinal_ranking'`: for ordinal data. WARNING : make sure your dataset only contains positive integers.

source_dimension

[int, optional] *For multivariate models only*. Set the degrees of freedom for _spatial_ variability. This number MUST BE strictly lower than the number of features. By default, this number is equal to square root of the number of features. One can interpret this

hyperparameter as a way to reduce the dimension of inter-individual _spatial_ variability between progressions.

batch_deltas_ordinal
[bool, optional] *For logistic models with ordinal noise model only*. If True, concatenates the deltas for each feature into a 2-dimensional Tensor "deltas" model parameter, which essentially allows faster sampling with new samplers. If False, each feature will induce a new model parameter "deltas_<feature_name>". The default is False but it is preferable to switch to True when ordinal items have many levels or when there are many items (when fit takes too long basically). Batching deltas will speed up the sampling part of the MCMC SAEM by trading for less accuracy in the estimation of deltas.

See also:

`leaspy.models`
`ModelFactory`
`Data`
`AlgorithmSettings`
`leaspy.algo`
`IndividualParameters`

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.

Methods

`calibrate`(data, 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 $(t_{i,j})_j$.
`estimate_ages_from_biomarker_values`(...[, ...])	For individuals characterized by their individual parameters z_i, returns the age $t_{i,j}$ at which a given feature value $y_{i,j,k}$ is reached.
`fit`(data, 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_loss])	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.

3.2 leaspy.models: Models

Available models in *Leaspy*.

AbstractModel(name, *, noise_model[, ...])	Contains the common attributes & methods of the different models.
AbstractMultivariateModel(name, **kwargs)	Contains the common attributes & methods of the multivariate models.
BaseModel(name, **kwargs)	Base model class from which all Leaspy models should inherit.
ConstantModel(name, **kwargs)	*ConstantModel* is a benchmark model that predicts constant values (no matter what the patient's ages are).
GenericModel(name, **kwargs)	Generic model (temporary until AbstractModel is really **abstract**).
LMEModel(name, **kwargs)	LMEModel is a benchmark model that fits and personalize a linear mixed-effects model.
ModelFactory()	Return the wanted model given its name.
MultivariateModel(name, **kwargs)	Manifold model for multiple variables of interest (logistic or linear formulation).
MultivariateParallelModel(name, **kwargs)	Logistic model for multiple variables of interest, imposing same average evolution pace for all variables (logistic curves are only time-shifted).
UnivariateModel(name, **kwargs)	Univariate (logistic or linear) model for a single variable of interest.

3.2.1 leaspy.models.AbstractModel

class AbstractModel(*name: str*, *, *noise_model: str | BaseNoiseModel | Dict[str, Any]*, *fit_metrics: Dict[str, float] | None = None*, ***kwargs*)

Bases: BaseModel

Contains the common attributes & methods of the different models.

Parameters

name
[str] The name of the model.

noise_model
[str or BaseNoiseModel] The noise model for observations (keyword-only parameter).

fit_metrics
[dict] Metrics that should be measured during the fit of the model and reported back to the user.

****kwargs**
Hyperparameters for the model

Attributes

is_initialized
[bool] Indicates if the model is initialized.

name
[str] The model's name.

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

parameters
[dict] Contains the model's parameters

noise_model
[BaseNoiseModel] The noise model associated to the model.

regularization_distribution_factory
[function dist params -> torch.distributions.Distribution] Factory of torch distribution to compute log-likelihoods for *regularization* (gaussian by default) (Not used anymore)

fit_metrics
[dict] Contains the metrics that are measured during the fit of the model and reported to the user.

Methods

check_noise_model_compatibility(model)	Raise a LeaspyModelInputError is the provided noise model isn't compatible with the model instance.
compute_canonical_loss_tensorized(data, ...)	Compute canonical loss, which depends on the noise model.
compute_individual_ages_from_biomarker_va	For one individual, compute age(s) at which the given features values are reached (given the subject's individual parameters).
compute_individual_ages_from_biomarker_va	For one individual, compute age(s) at which the given features values are reached (given the subject's individual parameters), with tensorized inputs.
compute_individual_attachment_tensorized(.	Compute *attachment* term (per subject).
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_model_sufficient_statistics(data, ...)	Compute sufficient statistics from a CollectionRealization.
compute_regularity_individual_parameters(.	Compute the regularity terms (and their gradients if requested), per individual variable of the model.
compute_regularity_individual_realization(	Compute regularity term for IndividualRealization.
compute_regularity_population_realization(	Compute regularity term for PopulationRealization.
compute_regularity_realization(realization)	Compute regularity term for a AbstractRealization instance.
compute_regularity_variable(value, mean, std, *)	Compute regularity term (Gaussian distribution) and optionally its gradient wrt value.
compute_sufficient_statistics(data, realizations)	Compute sufficient statistics from realizations.
get_individual_random_variable_informatio	Return the information on individual random variables relative to the model.

continues on next page

Table 1 – continued from previous page

`get_individual_variable_names`()	Get the names of the individual variables of the model.
`get_population_random_variable_informatio`	Return the information on population random variables relative to the model.
`get_population_variable_names`()	Get the names of the population variables of the model.
`initialize`(dataset[, method])	Initialize the model given a `Dataset` and an initialization method.
`load_hyperparameters`(hyperparameters)	Load model's hyperparameters.
`load_parameters`(parameters)	Instantiate or update the model's parameters.
`move_to_device`(device)	Move a model and its relevant attributes to the specified `torch.device`.
`save`(path, **kwargs)	Save `Leaspy` object as json model parameter file.
`time_reparametrization`(timepoints, xi, tau)	Tensorized time reparametrization formula.
`to_dict`()	Export model as a dictionary ready for export.
`update_model_parameters_burn_in`(data, ...)	Update model parameters (burn-in phase).
`update_model_parameters_normal`(data, ...)	Update model parameters (after burn-in phase).
`update_parameters_burn_in`(data, ...)	Update model parameters (burn-in phase).
`update_parameters_normal`(data, ...)	Update model parameters (after burn-in phase).
`validate_compatibility_of_dataset`(dataset)	Raise if the given `Dataset` is not compatible with the current model.

check_noise_model_compatibility(*model: BaseNoiseModel*) → None

Raise a `LeaspyModelInputError` is the provided noise model isn't compatible with the model instance.

This needs to be implemented in subclasses.

Parameters

model
[`BaseNoiseModel`] The noise model with which to check compatibility.

compute_canonical_loss_tensorized(*data: Dataset*, *param_ind: Dict[str, Tensor]*, *, *attribute_type=None*) → Tensor

Compute canonical loss, which depends on the noise model.

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
[`str` or None (default)] Flag to ask for *MCMC* attributes instead of model's attributes.

Returns

loss
[`torch.Tensor`] shape = * (depending on noise-model, always summed over individuals & visits)

compute_individual_ages_from_biomarker_values(*value: float | List[float]*, *individual_parameters: Dict[str, Any]*, *feature: str | None = None*) → Tensor

For one individual, compute age(s) at which the given features values are reached (given the subject's individual parameters).

Consistency checks are done in the main *API* layer.

Parameters

value
[scalar or array_like[scalar] (`list`, `tuple`, `numpy.ndarray`)] Contains the *biomarker* value(s) of the subject.

individual_parameters
[`dict`] Contains the individual parameters. Each individual parameter should be a scalar or array_like.

feature
[`str` (or None)] Name of the considered *biomarker*.

Note: Optional for `UnivariateModel`, compulsory for `MultivariateModel`.

Returns

`torch.Tensor`
Contains the subject's ages computed at the given values(s). Shape of tensor is `(1, n_values)`.

Raises

`LeaspyModelInputError`
If computation is tried on more than 1 individual.

abstract compute_individual_ages_from_biomarker_values_tensorized(*value: Tensor*, *individual_parameters: Dict[str, Tensor]*, *feature: str | None*) → Tensor

For one individual, compute age(s) at which the given features values are reached (given the subject's individual parameters), with tensorized inputs.

Parameters

value
[`torch.Tensor` of shape `(1, n_values)`] Contains the *biomarker* value(s) of the subject.

individual_parameters
[DictParamsTorch] Contains the individual parameters. Each individual parameter should be a `torch.Tensor`.

feature
[`str` (or None)] Name of the considered *biomarker*.

Note: Optional for `UnivariateModel`, compulsory for `MultivariateModel`.

Returns

torch.Tensor
Contains the subject's ages computed at the given values(s). Shape of tensor is (n_values, 1).

compute_individual_attachment_tensorized(*data: Dataset*, *param_ind: Dict[str, Tensor]*, *, *attribute_type: str | None = None*) → Tensor

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
[DictParamsTorch] Contain the individual parameters.

attribute_type
[str or None] Flag to ask for *MCMC* attributes instead of model's attributes.

Returns

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

abstract compute_individual_tensorized(*timepoints: Tensor*, *individual_parameters: Dict[str, Tensor]*, *, *attribute_type: str | None = None*) → Tensor

Compute the individual values at timepoints according to the model.

Parameters

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

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

attribute_type
[str or 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: Dict[str, Any]*, *, *skip_ips_checks: bool = False*) → Tensor

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)`.

Raises

`LeaspyModelInputError`
If computation is tried on more than 1 individual.

`LeaspyIndividualParamsInputError`
if invalid individual parameters.

abstract compute_jacobian_tensorized(*timepoints: Tensor*, *individual_parameters: Dict[str, Tensor]*, **, attribute_type: str | None = None*) → Dict[str, Tensor]

Compute the jacobian of the model w.r.t. each individual parameter.

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

Note: As most of numerical operations are repeated when computing model & jacobian, we should create a single method that is able to compute model & jacobian "together" (= efficiently) when requested with a flag for instance.

Parameters

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

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

attribute_type
[`str` or None] Flag to ask for *MCMC* attributes instead of model's attributes.

Returns

`dict` **[param_name:** `str`**,** `torch.Tensor` **]**
Tensors are of shape `(n_individuals, n_timepoints, n_features, n_dims_param)`.

abstract compute_model_sufficient_statistics(*data: Dataset*, *realizations: CollectionRealization*) → Dict[str, Tensor]

Compute sufficient statistics from a `CollectionRealization`.

Parameters

data
[`Dataset`]

realizations
[`CollectionRealization`]

Returns

`dict` **[suff_stat:** `str`**,** `torch.Tensor` **]**

compute_regularity_individual_parameters(*individual_parameters: Dict[str, Tensor]*, *, *include_constant: bool = False*) → Tuple[Dict[str, Tensor], Dict[str, Tensor]]

Compute the regularity terms (and their gradients if requested), per individual variable of the model.

Parameters

individual_parameters
[dict [str, torch.Tensor]] Individual parameters as a dict of tensors of shape (n_ind, n_dims_param).

include_constant
[bool, optional] Whether to include a constant term or not. Default=False.

Returns

regularity
[dict [param_name: str, torch.Tensor]] Regularity of the patient(s) corresponding to the given individual parameters. Tensors have shape (n_individuals).

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

compute_regularity_individual_realization(*realization: IndividualRealization*) → Tensor

Compute regularity term for IndividualRealization.

Parameters

realization
[IndividualRealization]

Returns

torch.Tensor of the same shape as IndividualRealization.tensor

compute_regularity_population_realization(*realization: PopulationRealization*) → Tensor

Compute regularity term for PopulationRealization.

Parameters

realization
[PopulationRealization]

Returns

torch.Tensor of the same shape as PopulationRealization.tensor

compute_regularity_realization(*realization: AbstractRealization*) → Tensor

Compute regularity term for a AbstractRealization instance.

Parameters

realization
[AbstractRealization]

Returns

torch.Tensor of the same shape as AbstractRealization.tensor

compute_regularity_variable(*value: Tensor*, *mean: Tensor*, *std: Tensor*, *, *include_constant: bool = True*, *with_gradient: bool = False*) → Tensor | Tuple[Tensor, Tensor]

Compute regularity term (Gaussian distribution) and optionally its gradient wrt value.

Note: TODO: should be encapsulated in a `RandomVariableSpecification` class together with other specs of RV.

Parameters

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

include_constant
[`bool` (default `True`)] Whether we include or not additional terms constant with respect to *value*.

with_gradient
[`bool` (default `False`)] Whether we also return the gradient of *regularity* term with respect to *value*.

Returns

`torch.Tensor` of same shape than input

`compute_sufficient_statistics`(*data: Dataset*, *realizations: CollectionRealization*) → Dict[str, Tensor]

Compute sufficient statistics from realizations.

Parameters

data
[`Dataset`]

realizations
[`CollectionRealization`]

Returns

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

property dimension: `int` | `None`

The dimension of the model. If the private attribute is defined, then it takes precedence over the feature length. The associated setters are responsible for their coherence.

abstract `get_individual_random_variable_information`() → Dict[str, Any]

Return the information on individual random variables relative to the model.

Returns

DictParams
The information on the individual random variables.

`get_individual_variable_names`() → List[str]

Get the names of the individual variables of the model.

Returns

`list` of `str`

abstract `get_population_random_variable_information`() → Dict[str, Any]

Return the information on population random variables relative to the model.

Returns

DictParams
The information on the population random variables.

`get_population_variable_names()` → List[str]

Get the names of the population variables of the model.

Returns

`list` **of** `str`

`initialize`(*dataset: Dataset*, *method: str = 'default'*) → None

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: Dict[str, Any]*) → None

Load model's hyperparameters.

Parameters

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

Raises

`LeaspyModelInputError`
If any of the consistency checks fail.

`load_parameters`(*parameters: Dict[str, Any]*) → None

Instantiate or update the model's parameters.

Parameters

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

`move_to_device`(*device: device*) → None

Move a model and its relevant attributes to the specified `torch.device`.

Parameters

device
[`torch.device`]

`save`(*path: str*, ***kwargs*) → None

Save `Leaspy` object as json model parameter file.

TODO move logic upstream?

Parameters

path
[`str`] Path to store the model's parameters.

****kwargs**
Keyword arguments for `AbstractModel.to_dict()` child method and `json.dump` function (default to indent=2).

static time_reparametrization(*timepoints: Tensor*, *xi: Tensor*, *tau: Tensor*) → Tensor

Tensorized time reparametrization formula.

> **Warning:** 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 to_dict() → Dict[str, Any]

Export model as a dictionary ready for export.

Returns

KwargsType
The model instance serialized as a dictionary.

abstract update_model_parameters_burn_in(*data: Dataset*, *sufficient_statistics: Dict[str, Tensor]*) → None

Update model parameters (burn-in phase).

Parameters

data
[Dataset]

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

abstract update_model_parameters_normal(*data: Dataset*, *sufficient_statistics: Dict[str, Tensor]*) → None

Update model parameters (after burn-in phase).

Parameters

data
[Dataset]

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

update_parameters_burn_in(*data: Dataset*, *sufficient_statistics: Dict[str, Tensor]*) → None

Update model parameters (burn-in phase).

Parameters

data
[Dataset]

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

update_parameters_normal(*data: Dataset*, *sufficient_statistics: Dict[str, Tensor]*) → None

Update model parameters (after burn-in phase).

Parameters

data
[Dataset]

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

validate_compatibility_of_dataset(*dataset: Dataset*) → None

Raise if the given Dataset is not compatible with the current model.

Parameters

dataset
[Dataset] The Dataset we want to model.

Raises

LeaspyModelInputError

- If the Dataset has a number of dimensions smaller than 2.
- If the Dataset does not have the same dimensionality as the model.
- If the Dataset's headers do not match the model's.

3.2.2 leaspy.models.AbstractMultivariateModel

class AbstractMultivariateModel(*name: str*, ***kwargs*)

Bases: OrdinalModelMixin, AbstractModel

Contains the common attributes & methods of the multivariate models.

Parameters

name
[str] Name of the model.

****kwargs**
Hyperparameters for the model (including *noise_model*).

Raises

LeaspyModelInputError
If inconsistent hyperparameters.

Attributes

dimension
The dimension of the model.

features
is_ordinal
Property to check if the model is of ordinal sub-type.

is_ordinal_ranking
Property to check if the model is of ordinal-ranking sub-type (working with survival functions).

noise_model
ordinal_infos
Property to return the ordinal info dictionary.

Methods

check_noise_model_compatibility(model)	Check compatibility between the model instance and provided noise model.
compute_appropriate_ordinal_model(...)	Post-process model values (or their gradient) if needed.
compute_canonical_loss_tensorized(data, ...)	Compute canonical loss, which depends on the noise model.
compute_individual_ages_from_biomarker_va	For one individual, compute age(s) at which the given features values are reached (given the subject's individual parameters).
compute_individual_ages_from_biomarker_va	For one individual, compute age(s) at which the given features values are reached (given the subject's individual parameters), with tensorized inputs.
compute_individual_attachment_tensorized(.	Compute *attachment* term (per subject).
compute_individual_tensorized(timepoints, ...)	Compute the individual trajectories.
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_model_sufficient_statistics(data, ...)	Compute sufficient statistics from a CollectionRealization.
compute_ordinal_model_sufficient_statisti	Compute the sufficient statistics given realizations.
compute_ordinal_pdf_from_ordinal_sf(ordinal	Computes the probability density (or its jacobian) of an ordinal model [P(X = l), l=0..L] from *ordinal_sf* which are the survival function probabilities [P(X > l), i.e. P(X >= l+1), l=0..L-1] (or its jacobian).
compute_regularity_individual_parameters(.	Compute the regularity terms (and their gradients if requested), per individual variable of the model.
compute_regularity_individual_realization(	Compute regularity term for IndividualRealization.
compute_regularity_population_realization(	Compute regularity term for PopulationRealization.
compute_regularity_realization(realization)	Compute regularity term for a AbstractRealization instance.
compute_regularity_variable(value, mean, std, *)	Compute regularity term (Gaussian distribution) and optionally its gradient wrt value.
compute_sufficient_statistics(data, realizations)	Compute sufficient statistics from realizations.
get_additional_ordinal_population_random_	Return the information of additional population random variables for the ordinal model.

continues on next page

Table 2 – continued from previous page

get_individual_random_variable_informatio	Return the information on individual random variables relative to the model.
get_individual_variable_names()	Get the names of the individual variables of the model.
get_ordinal_parameters_updates_from_suffi	Return a dictionary computed from provided sufficient statistics for updating the parameters.
get_population_random_variable_informatio	Return the information on population random variables relative to the model.
get_population_variable_names()	Get the names of the population variables of the model.
initialize(dataset[, method])	Overloads base initialization of model (base method takes care of features consistency checks).
initialize_MCMC_toolbox()	Initialize *MCMC* toolbox for calibration of model.
load_hyperparameters(hyperparameters)	Updates all model hyperparameters from the provided hyperparameters.
load_parameters(parameters)	Updates all model parameters from the provided parameters.
move_to_device(device)	Move a model and its relevant attributes to the specified `torch.device`.
postprocess_model_estimation(estimation, *)	Extra layer of processing used to output nice estimated values in main API *Leaspy.estimate*.
save(path, **kwargs)	Save `Leaspy` object as json model parameter file.
time_reparametrization(timepoints, xi, tau)	Tensorized time reparametrization formula.
to_dict(*[, with_mixing_matrix])	Export `Leaspy` object as dictionary ready for *JSON* saving.
update_MCMC_toolbox(vars_to_update, realizations)	Update the *MCMC* toolbox with a `CollectionRealization` of model population parameters.
update_model_parameters_burn_in(data, ...)	Update model parameters (burn-in phase).
update_model_parameters_normal(data, ...)	Update model parameters (after burn-in phase).
update_ordinal_population_random_variable_	Update (in-place) the provided variable information dictionary.
update_parameters_burn_in(data, ...)	Update model parameters (burn-in phase).
update_parameters_normal(data, ...)	Update model parameters (after burn-in phase).
validate_compatibility_of_dataset(dataset)	Raise if the given `Dataset` is not compatible with the current model.

check_noise_model_compatibility(*model: BaseNoiseModel*) → None

Check compatibility between the model instance and provided noise model.

compute_appropriate_ordinal_model(*model_or_model_grad: Tensor*) → Tensor

Post-process model values (or their gradient) if needed.

compute_canonical_loss_tensorized(*data: Dataset*, *param_ind: Dict[str, Tensor]*, *, *attribute_type=None*) → Tensor

Compute canonical loss, which depends on the noise model.

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
[str or None (default)] Flag to ask for *MCMC* attributes instead of model's attributes.

Returns

loss
[torch.Tensor] shape = * (depending on noise-model, always summed over individuals & visits)

compute_individual_ages_from_biomarker_values(*value: float | List[float]*, *individual_parameters: Dict[str, Any]*, *feature: str | None = None*) → Tensor

For one individual, compute age(s) at which the given features values are reached (given the subject's individual parameters).

Consistency checks are done in the main *API* layer.

Parameters

value
[scalar or array_like[scalar] (list, tuple, numpy.ndarray)] Contains the *biomarker* value(s) of the subject.

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

feature
[str (or None)] Name of the considered *biomarker*.

Note: Optional for UnivariateModel, compulsory for MultivariateModel.

Returns

torch.Tensor
Contains the subject's ages computed at the given values(s). Shape of tensor is (1, n_values).

Raises

LeaspyModelInputError
If computation is tried on more than 1 individual.

abstract compute_individual_ages_from_biomarker_values_tensorized(*value: Tensor*, *individual_parameters: Dict[str, Tensor]*, *feature: str | None*) → Tensor

For one individual, compute age(s) at which the given features values are reached (given the subject's individual parameters), with tensorized inputs.

Parameters

value
[torch.Tensor of shape (1, n_values)] Contains the *biomarker* value(s) of the subject.

individual_parameters
[DictParamsTorch] Contains the individual parameters. Each individual parameter should be a `torch.Tensor`.

feature
[`str` (or None)] Name of the considered *biomarker*.

Note: Optional for `UnivariateModel`, compulsory for `MultivariateModel`.

Returns

`torch.Tensor`
Contains the subject's ages computed at the given values(s). Shape of tensor is `(n_values, 1)`.

compute_individual_attachment_tensorized(*data: Dataset*, *param_ind: Dict[str, Tensor]*, *, *attribute_type: str | None = None*) → Tensor

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
[DictParamsTorch] Contain the individual parameters.

attribute_type
[`str` or None] Flag to ask for *MCMC* attributes instead of model's attributes.

Returns

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

abstract compute_individual_tensorized(*timepoints: Tensor*, *individual_parameters: Dict[str, Tensor]*, *, *attribute_type=None*) → Tensor

Compute the individual trajectories.

Parameters

timepoints
[`torch.Tensor`] The time points for which to compute the trajectory.

individual_parameters
[DictParamsTorch] The individual parameters to use.

attribute_type
[Any, optional]

Returns

`torch.Tensor`
Individual trajectories.

compute_individual_trajectory(*timepoints*, *individual_parameters: Dict[str, Any]*, *, *skip_ips_checks: bool = False*) → Tensor

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)`.

Raises

`LeaspyModelInputError`
If computation is tried on more than 1 individual.

`LeaspyIndividualParamsInputError`
if invalid individual parameters.

abstract compute_jacobian_tensorized(*timepoints: Tensor*, *individual_parameters: Dict[str, Tensor]*, **, attribute_type: str | None = None*) → Dict[str, Tensor]

Compute the jacobian of the model w.r.t. each individual parameter.

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

Note: As most of numerical operations are repeated when computing model & jacobian, we should create a single method that is able to compute model & jacobian "together" (= efficiently) when requested with a flag for instance.

Parameters

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

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

attribute_type
[`str` or None] Flag to ask for *MCMC* attributes instead of model's attributes.

Returns

`dict` **[param_name:** `str`**,** `torch.Tensor` **]**
Tensors are of shape `(n_individuals, n_timepoints, n_features, n_dims_param)`.

compute_mean_traj(*timepoints: Tensor*, **, attribute_type: str | None = None*) → Tensor

Compute trajectory of the model with individual parameters being the group-average ones.

TODO check dimensions of io?

Parameters

timepoints
[torch.Tensor of shape (1, n_timepoints)]

attribute_type
[str or None] If a string, should be "MCMC".

Returns

torch.Tensor of shape (1, n_timepoints, dimension)
The group-average values at given timepoints.

abstract compute_model_sufficient_statistics(*data: Dataset*, *realizations: CollectionRealization*) → Dict[str, Tensor]

Compute sufficient statistics from a CollectionRealization.

Parameters

data
[Dataset]

realizations
[CollectionRealization]

Returns

dict [suff_stat: str, torch.Tensor]

compute_ordinal_model_sufficient_statistics(*realizations: CollectionRealization*) → Dict[str, Tensor]

Compute the sufficient statistics given realizations.

static compute_ordinal_pdf_from_ordinal_sf(*ordinal_sf: Tensor*, *dim_ordinal_levels: int = 3*) → Tensor

Computes the probability density (or its jacobian) of an ordinal model [P(X = l), l=0..L] from *ordinal_sf* which are the survival function probabilities [P(X > l), i.e. P(X >= l+1), l=0..L-1] (or its jacobian).

Parameters

ordinal_sf
[*torch.FloatTensor*] Survival function values : ordinal_sf[..., l] is the proba to be superior or equal to l+1 Dimensions are: * 0=individual * 1=visit * 2=feature * 3=ordinal_level [l=0..L-1] * [4=individual_parameter_dim_when_gradient]

dim_ordinal_levels
[int, default = 3] The dimension of the tensor where the ordinal levels are.

Returns

ordinal_pdf
[*torch.FloatTensor* (same shape as input, except for dimension 3 which has one more element)] ordinal_pdf[..., l] is the proba to be equal to l (l=0..L)

compute_regularity_individual_parameters(*individual_parameters: Dict[str, Tensor]*, ***, *include_constant: bool = False*) → Tuple[Dict[str, Tensor], Dict[str, Tensor]]

Compute the regularity terms (and their gradients if requested), per individual variable of the model.

Parameters

individual_parameters
[dict [str, torch.Tensor]] Individual parameters as a dict of tensors of shape (n_ind, n_dims_param).

include_constant
[bool, optional] Whether to include a constant term or not. Default=False.

Returns

regularity
[dict [param_name: str, torch.Tensor]] Regularity of the patient(s) corresponding to the given individual parameters. Tensors have shape `(n_individuals)`.

regularity_grads
[dict [param_name: str, torch.Tensor]] Gradient of regularity term with respect to individual parameters. Tensors have shape `(n_individuals, n_dims_param)`.

compute_regularity_individual_realization(*realization: IndividualRealization*) → Tensor

Compute regularity term for `IndividualRealization`.

Parameters

realization
[IndividualRealization]

Returns

torch.Tensor of the same shape as IndividualRealization.tensor

compute_regularity_population_realization(*realization: PopulationRealization*) → Tensor

Compute regularity term for `PopulationRealization`.

Parameters

realization
[PopulationRealization]

Returns

torch.Tensor of the same shape as PopulationRealization.tensor

compute_regularity_realization(*realization: AbstractRealization*) → Tensor

Compute regularity term for a `AbstractRealization` instance.

Parameters

realization
[AbstractRealization]

Returns

torch.Tensor of the same shape as AbstractRealization.tensor

compute_regularity_variable(*value: Tensor*, *mean: Tensor*, *std: Tensor*, *, *include_constant: bool = True*, *with_gradient: bool = False*) → Tensor | Tuple[Tensor, Tensor]

Compute regularity term (Gaussian distribution) and optionally its gradient wrt value.

Note: TODO: should be encapsulated in a `RandomVariableSpecification` class together with other specs of RV.

Parameters

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

include_constant
[bool (default True)] Whether we include or not additional terms constant with respect to *value*.

with_gradient
[bool (default False)] Whether we also return the gradient of *regularity* term with respect to *value*.

Returns

torch.Tensor of same shape than input

compute_sufficient_statistics(*data: Dataset*, *realizations: CollectionRealization*) → Dict[str, Tensor]

Compute sufficient statistics from realizations.

Parameters

data
[Dataset]

realizations
[CollectionRealization]

Returns

dict [suff_stat: str, torch.Tensor]

property dimension: int | None

The dimension of the model. If the private attribute is defined, then it takes precedence over the feature length. The associated setters are responsible for their coherence.

get_additional_ordinal_population_random_variable_information() → Dict[str, Any]

Return the information of additional population random variables for the ordinal model.

abstract get_individual_random_variable_information() → Dict[str, Any]

Return the information on individual random variables relative to the model.

Returns

DictParams
The information on the individual random variables.

get_individual_variable_names() → List[str]

Get the names of the individual variables of the model.

Returns

list of str

get_ordinal_parameters_updates_from_sufficient_statistics(*sufficient_statistics: Dict[str, Tensor]*) → Dict[str, Tensor]

Return a dictionary computed from provided sufficient statistics for updating the parameters.

abstract get_population_random_variable_information() → Dict[str, Any]

Return the information on population random variables relative to the model.

Returns

DictParams
The information on the population random variables.

get_population_variable_names() → List[str]

Get the names of the population variables of the model.

Returns

list of str

initialize(*dataset: Dataset*, *method: str = 'default'*) → None

Overloads base initialization of model (base method takes care of features consistency checks).

Parameters

dataset
[Dataset] Input Dataset from which to initialize the model.

method
[str, optional] The initialization method to be used. Default='default'.

abstract initialize_MCMC_toolbox() → None

Initialize *MCMC* toolbox for calibration of model.

property is_ordinal: bool

Property to check if the model is of ordinal sub-type.

property is_ordinal_ranking: bool

Property to check if the model is of ordinal-ranking sub-type (working with survival functions).

load_hyperparameters(*hyperparameters: Dict[str, Any]*) → None

Updates all model hyperparameters from the provided hyperparameters.

Parameters

hyperparameters
[KwargsType] The hyperparameters to be loaded.

load_parameters(*parameters: Dict[str, Any]*) → None

Updates all model parameters from the provided parameters.

Parameters

parameters
[KwargsType] The parameters to be loaded.

move_to_device(*device: device*) → None

Move a model and its relevant attributes to the specified torch.device.

Parameters

device
[torch.device]

property ordinal_infos: dict | None

Property to return the ordinal info dictionary.

postprocess_model_estimation(*estimation: ndarray*, *, *ordinal_method: str = 'MLE'*, ***kws*) → ndarray | Dict[Hashable, ndarray]

Extra layer of processing used to output nice estimated values in main API *Leaspy.estimate*.

Parameters

estimation
[numpy.ndarray[float]] The raw estimated values by model (from *compute_individual_trajectory*)

ordinal_method
[str] <!> Only used for ordinal models. * 'MLE' or 'maximum_likelihood' returns maximum likelihood estimator for each point (int) * 'E' or 'expectation' returns expectation (float) * 'P' or 'probabilities' returns probabilities of all-possible levels for a given feature:

{feature_name: array[float]<0..max_level_ft>}

****kws**
Some extra keywords arguments that may be handled in the future.

Returns

numpy.ndarray[float] or dict[str, numpy.ndarray[float]]
Post-processed values. In case using 'probabilities' mode, the values are a dictionary with keys being: *(feature_name: str, feature_level: int<0..max_level_for_feature>)* Otherwise it is a standard numpy.ndarray corresponding to different model features (in order)

`save`(*path: str*, ***kwargs*) → None

Save `Leaspy` object as json model parameter file.

TODO move logic upstream?

Parameters

path
[`str`] Path to store the model's parameters.

****kwargs**
Keyword arguments for `AbstractModel.to_dict()` child method and `json.dump` function (default to indent=2).

`static time_reparametrization`(*timepoints: Tensor*, *xi: Tensor*, *tau: Tensor*) → Tensor

Tensorized time reparametrization formula.

Warning: 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*

`to_dict`(*, *with_mixing_matrix: bool = True*) → Dict[str, Any]

Export `Leaspy` object as dictionary ready for *JSON* saving.

Parameters

with_mixing_matrix
[`bool` (default `True`)] Save the *mixing matrix* in the exported file in its 'parameters' section.

> **Warning:** 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…

Returns

KwargsType
The object as a dictionary.

abstract update_MCMC_toolbox(*vars_to_update: Set[str]*, *realizations: CollectionRealization*) → None

Update the *MCMC* toolbox with a `CollectionRealization` of model population parameters.

Parameters

vars_to_update
[`set` of `str`] Names of the population parameters to update in *MCMC* toolbox.

realizations
[`CollectionRealization`] All the realizations to update *MCMC* toolbox with.

abstract update_model_parameters_burn_in(*data: Dataset*, *sufficient_statistics: Dict[str, Tensor]*) → None

Update model parameters (burn-in phase).

Parameters

data
[`Dataset`]

sufficient_statistics
[`dict` [suff_stat: `str`, `torch.Tensor`]]

abstract update_model_parameters_normal(*data: Dataset*, *sufficient_statistics: Dict[str, Tensor]*) → None

Update model parameters (after burn-in phase).

Parameters

data
[`Dataset`]

sufficient_statistics
[`dict` [suff_stat: `str`, `torch.Tensor`]]

update_ordinal_population_random_variable_information(*variables_info: Dict[str, Any]*) → None

Update (in-place) the provided variable information dictionary.

Nota: this is needed due to different signification of *v0* in ordinal model (common per-level velocity)

Parameters

variables_info
[DictParams] The variables information to be updated with ordinal logic.

update_parameters_burn_in(*data: Dataset*, *sufficient_statistics: Dict[str, Tensor]*) → None

Update model parameters (burn-in phase).

Parameters

data
[Dataset]

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

update_parameters_normal(*data: Dataset*, *sufficient_statistics: Dict[str, Tensor]*) → None

Update model parameters (after burn-in phase).

Parameters

data
[Dataset]

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

validate_compatibility_of_dataset(*dataset: Dataset*) → None

Raise if the given Dataset is not compatible with the current model.

Parameters

dataset
[Dataset] The Dataset we want to model.

Raises

LeaspyModelInputError

- If the Dataset has a number of dimensions smaller than 2.
- If the Dataset does not have the same dimensionality as the model.
- If the Dataset's headers do not match the model's.

3.2.3 leaspy.models.BaseModel

class BaseModel(*name: str*, ***kwargs*)

Bases: ABC

Base model class from which all Leaspy models should inherit.

It defines the interface that a model should implement to be compatible with Leaspy.

Parameters

name
[str] The name of the model.

****kwargs**
Hyperparameters of the model

Attributes

name
[str] The name of the model.

is_initialized
[bool] True``if the model is initialized, ``False otherwise.

features
[list of str] List of model features (None if not initialization).

dimension
[int] The dimension of the model.

Methods

initialize(dataset[, method])	Initialize the model given a Dataset and an initialization method.
save(path, **kwargs)	Save Leaspy object as json model parameter file.
validate_compatibility_of_dataset(dataset)	Raise if the given Dataset is not compatible with the current model.

property dimension: int | None

The dimension of the model. If the private attribute is defined, then it takes precedence over the feature length. The associated setters are responsible for their coherence.

initialize(*dataset: Dataset*, *method: str = 'default'*) → None

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 save(*path: str*, ***kwargs*) → None

Save Leaspy object as json model parameter file.

Parameters

path
[str] Path to store the model's parameters.

****kwargs**
Additional parameters for writing.

validate_compatibility_of_dataset(*dataset: Dataset*) → None

Raise if the given Dataset is not compatible with the current model.

Parameters

dataset
[Dataset] The Dataset we want to model.

Raises

LeaspyModelInputError

- If the Dataset has a number of dimensions smaller than 2.
- If the Dataset does not have the same dimensionality as the model.
- If the Dataset's headers do not match the model's.

3.2.4 leaspy.models.ConstantModel

class ConstantModel(*name: str*, ***kwargs*)

Bases: GenericModel

ConstantModel is a benchmark model that predicts constant values (no matter what the patient's ages are).

These constant values depend on the algorithm setting and the patient's values provided during *calibration*.

It could predict:

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

Warning: Depending on `features`, the `last_known` / `max` value may correspond to different visits.

Warning: For a given feature, value will be `NaN` if and only if all values for this feature were `NaN`.

Parameters

name
[`str`] The model's name.

****kwargs**
Hyperparameters for the model. None supported for now.

See also:

`ConstantPredictionAlgorithm`

Attributes

name
[`str`] The model's name.

is_initialized
[`bool`] Always `True` (no true initialization needed for constant model).

features
[`list` of `str`] List of the model features. Unlike most models features will be determined at *personalization* only (because it does not needed any *fit*).

dimension
[`int`] The dimension of the model.

parameters
[`dict`] The model has no parameters: empty dictionary. The `prediction_type` parameter should be defined during *personalization*. Example:

```
>>> AlgorithmSettings('constant_prediction', prediction_type='last_
↪known')
```

Methods

`compute_individual_trajectory`(timepoints, ...)	Compute scores values at the given time-point(s) given a subject's individual parameters.
`get_hyperparameters`(*[, with_features, ...])	Get all model hyperparameters.
`hyperparameters_ok`()	Check all model hyperparameters are ok.
`initialize`(dataset[, method])	Initialize the model given a `Dataset` and an initialization method.
`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.
`validate_compatibility_of_dataset`(dataset)	Raise if the given `Dataset` is not compatible with the current model.

compute_individual_trajectory(*timepoints: Tensor*, *individual_parameters: dict*) → Tensor

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` [`str`, Any]] Contains the individual parameters. Each individual parameter should be a scalar or array_like.

Returns

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

property dimension: int | None

The dimension of the model. If the private attribute is defined, then it takes precedence over the feature length. The associated setters are responsible for their coherence.

get_hyperparameters(*, *with_features=True*, *with_properties=True*, *default=None*) → Dict[str, Any]

Get all model hyperparameters.

Parameters

with_features, with_properties
[`bool` (default `True`)] Whether to include *features* and respectively all *_properties* (i.e. _dynamic_ hyperparameters) in the returned dictionary.

default
[Any] Default value is something is an hyperparameter is missing (should not!).

Returns

:obj:`dict` { hyperparam_name
[`str` -> hyperparam_value][Any }]

hyperparameters_ok() → bool

Check all model hyperparameters are ok.

Returns

bool

initialize(*dataset: Dataset*, *method: str = 'default'*) → None

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

load_hyperparameters(*hyperparameters: Dict[str, Any]*, *, *with_defaults: bool = False*) → None

Load model hyperparameters from a dict.

Parameters

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

with_defaults
[bool (default False)] If True, it also resets hyperparameters that are part of the model but not included in *hyperparameters* to their default value.

Raises

LeaspyModelInputError
If inconsistent 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.

list_converter
[callable] The function to convert list objects.

save(*path: str*, ***kwargs*) → None

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.

validate_compatibility_of_dataset(*dataset: Dataset*) → None

Raise if the given Dataset is not compatible with the current model.

Parameters

dataset
[Dataset] The Dataset we want to model.

Raises

LeaspyModelInputError

- If the `Dataset` has a number of dimensions smaller than 2.
- If the `Dataset` does not have the same dimensionality as the model.
- If the `Dataset`'s headers do not match the model's.

3.2.5 `leaspy.models.GenericModel`

class GenericModel(*name: str*, ***kwargs*)

Bases: `BaseModel`

Generic model (temporary until `AbstractModel` is really **abstract**).

TODO: change naming after AbstractModel was renamed?

Parameters

name
[`str`] The name of the model.

****kwargs**
Hyperparameters of the model.

Attributes

name
[`str`] The name of the model.

is_initialized
[`bool`] `True` if the model is initialized, `False` otherwise.

features
[`list` of `str`] List of model features (None if not initialization).

dimension
[`int` (read-only)] The dimension of the model.

parameters
[`dict`] Contains internal parameters of the model.

Methods

`compute_individual_trajectory`(timepoints, ...)	Compute scores values at the given time-point(s) given a subject's individual parameters.
`get_hyperparameters`(*[, with_features, ...])	Get all model hyperparameters.
`hyperparameters_ok`()	Check all model hyperparameters are ok.
`initialize`(dataset[, method])	Initialize the model given a `Dataset` and an initialization method.
`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.
`validate_compatibility_of_dataset`(dataset)	Raise if the given `Dataset` is not compatible with the current model.

abstract compute_individual_trajectory(*timepoints*, *individual_parameters: dict*) → Tensor

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` [`str`, Any]] Contains the individual parameters. Each individual parameter should be a scalar or array_like.

Returns

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

property dimension: int | None

The dimension of the model. If the private attribute is defined, then it takes precedence over the feature length. The associated setters are responsible for their coherence.

get_hyperparameters(*, *with_features=True*, *with_properties=True*, *default=None*) → Dict[str, Any]

Get all model hyperparameters.

Parameters

with_features, with_properties
[`bool` (default `True`)] Whether to include *features* and respectively all *_properties* (i.e. _dynamic_ hyperparameters) in the returned dictionary.

default
[Any] Default value is something is an hyperparameter is missing (should not!).

Returns

:obj:`dict` { hyperparam_name
[`str` -> hyperparam_value][Any }]

hyperparameters_ok() → bool

Check all model hyperparameters are ok.

Returns

`bool`

initialize(*dataset: Dataset*, *method: str = 'default'*) → None

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

load_hyperparameters(*hyperparameters: Dict[str, Any]*, *, *with_defaults: bool = False*) → None

Load model hyperparameters from a `dict`.

Parameters

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

with_defaults
[bool (default False)] If True, it also resets hyperparameters that are part of the model but not included in *hyperparameters* to their default value.

Raises

LeaspyModelInputError
If inconsistent 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.

list_converter
[callable] The function to convert list objects.

save(*path: str*, ***kwargs*) → None

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.

validate_compatibility_of_dataset(*dataset: Dataset*) → None

Raise if the given Dataset is not compatible with the current model.

Parameters

dataset
[Dataset] The Dataset we want to model.

Raises

LeaspyModelInputError

- If the Dataset has a number of dimensions smaller than 2.
- If the Dataset does not have the same dimensionality as the model.
- If the Dataset's headers do not match the model's.

3.2.6 leaspy.models.LMEModel

class LMEModel(*name: str*, ***kwargs*)

Bases: GenericModel

LMEModel is a benchmark model that fits and personalize a linear mixed-effects model.

The model specification is the following:

$$y_{ij} = fixed_{intercept} + random_{intercept_i} + (fixed_{slopeAge} + random_{slopeAge_i}) * age_{ij} + \epsilon_{ij}$$

with:

- y_{ij}: value of the feature of the i-th subject at his j-th visit,
- age_{ij}: age of the i-th subject at his j-th visit.
- ϵ_{ij}: residual Gaussian noise (independent between visits)

Warning: This model must be fitted on one feature only (univariate model).

TODO? should inherit from AbstractModel. TODO? add some covariates in this very simple model.

Parameters

name
[str] The model's name.

****kwargs**

Model hyperparameters:

- with_random_slope_age : bool (default True).

See also:

LMEFitAlgorithm
LMEPersonalizeAlgorithm

Attributes

name
[str] The model's name.

is_initialized
[bool] True if the model is initialized, False otherwise.

with_random_slope_age
[bool (default True)] Has the LME a random slope for subject's age? Otherwise it only has a random intercept per subject.

features
[list of str] List of the model features.

Warning: LME has only one feature.

dimension
[int] The dimension of the model.

parameters
[dict]

Contains the model parameters. In particular:

- **ages_mean**
  [float] Mean of ages (for normalization).
- **ages_std**
  [float] Std-dev of ages (for normalization).
- **fe_params**
  [np.ndarray of float] Fixed effects.
- **cov_re**
  [np.ndarray] Variance-covariance matrix of random-effects.
- **cov_re_unscaled_inv**
  [np.ndarray] Inverse of unscaled (= divided by variance of noise) variance-covariance matrix of random-effects. This matrix is used for personalization to new subjects.
- **noise_std**
  [float] Std-dev of Gaussian noise.
- **bse_fe, bse_re**
  [np.ndarray of float] Standard errors on fixed-effects and random-effects respectively (not used in Leaspy).

Methods

compute_individual_trajectory(timepoints, ...)	Compute scores values at the given time-point(s) given a subject's individual parameters.
get_hyperparameters(*[, with_features, ...])	Get all model hyperparameters.
hyperparameters_ok()	Check all model hyperparameters are ok.
initialize(dataset[, method])	Initialize the model given a Dataset and an initialization method.
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.
validate_compatibility_of_dataset(dataset)	Raise if the given Dataset is not compatible with the current model.

compute_individual_trajectory(*timepoints*, *individual_parameters: dict*)

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.

individual_parameters
[dict]

Individual parameters:

- random_intercept
- random_slope_age (if with_random_slope_age == True)

Returns

`torch.Tensor` **of** `float`
The individual trajectories. The shape of the tensor is `(n_individuals == 1, n_tpts == len(timepoints), n_features == 1)`.

property dimension: `int | None`

The dimension of the model. If the private attribute is defined, then it takes precedence over the feature length. The associated setters are responsible for their coherence.

get_hyperparameters(*, *with_features=True*, *with_properties=True*, *default=None*) → Dict[str, Any]

Get all model hyperparameters.

Parameters

with_features, with_properties
[`bool` (default `True`)] Whether to include *features* and respectively all *_properties* (i.e. _dynamic_ hyperparameters) in the returned dictionary.

default
[Any] Default value is something is an hyperparameter is missing (should not!).

Returns

:obj:`dict` { hyperparam_name
[`str` -> hyperparam_value][Any }]

hyperparameters_ok() → bool

Check all model hyperparameters are ok.

Returns

`bool`

initialize(*dataset: Dataset*, *method: str = 'default'*) → None

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

load_hyperparameters(*hyperparameters: Dict[str, Any]*, *, *with_defaults: bool = False*) → None

Load model hyperparameters from a `dict`.

Parameters

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

with_defaults
[`bool` (default `False`)] If `True`, it also resets hyperparameters that are part of the model but not included in *hyperparameters* to their default value.

Raises

LeaspyModelInputError
If inconsistent 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.

list_converter
[callable] The function to convert list objects.

save(*path: str*, ***kwargs*) → None

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.

validate_compatibility_of_dataset(*dataset: Dataset*) → None

Raise if the given Dataset is not compatible with the current model.

Parameters

dataset
[Dataset] The Dataset we want to model.

Raises

LeaspyModelInputError

- If the Dataset has a number of dimensions smaller than 2.
- If the Dataset does not have the same dimensionality as the model.
- If the Dataset's headers do not match the model's.

3.2.7 leaspy.models.ModelFactory

class 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.

static model(*name: str*, ***kwargs*) → BaseModel

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 inappropriate for the given model's name.

Returns

`BaseModel`
A child class object of `models.BaseModel` class object determined by `name`.

Raises

`LeaspyModelInputError`
If an incorrect model is requested.

See also:

`Leaspy`

3.2.8 leaspy.models.MultivariateModel

`class MultivariateModel(`*name: str*, ***kwargs*`)`

Bases: `AbstractMultivariateModel`

Manifold model for multiple variables of interest (logistic or linear formulation).

Parameters

name
[str] The name of the model.

****kwargs**
Hyperparameters of the model (including *noise_model*)

Raises

`LeaspyModelInputError`

- If *name* is not one of allowed sub-type: 'univariate_linear' or 'univariate_logistic'
- If hyperparameters are inconsistent

Attributes

`dimension`
The dimension of the model.

features
`is_ordinal`
Property to check if the model is of ordinal sub-type.

`is_ordinal_ranking`
Property to check if the model is of ordinal-ranking sub-type (working with survival functions).

noise_model
`ordinal_infos`
Property to return the ordinal info dictionary.

Methods

check_noise_model_compatibility(model)	Check compatibility between the model instance and provided noise model.
compute_appropriate_ordinal_model(...)	Post-process model values (or their gradient) if needed.
compute_canonical_loss_tensorized(data, ...)	Compute canonical loss, which depends on the noise model.
compute_individual_ages_from_biomarker_va	For one individual, compute age(s) at which the given features values are reached (given the subject's individual parameters).
compute_individual_ages_from_biomarker_va	For one individual, compute age(s) at which the given features values are reached (given the subject's individual parameters), with tensorized inputs.
compute_individual_ages_from_biomarker_va	For one individual, compute age(s) at which the given features values are reached (given the subject's individual parameters), with tensorized inputs.
compute_individual_attachment_tensorized(.	Compute *attachment* term (per subject).
compute_individual_tensorized(timepoints, ...)	Compute the individual trajectories.
compute_individual_tensorized_linear(...[, ...])	Compute the individual trajectories.
compute_individual_tensorized_logistic(...)	Compute the individual trajectories.
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_mean_traj(timepoints, *[, ...])	Compute trajectory of the model with individual parameters being the group-average ones.
compute_model_sufficient_statistics(data, ...)	Compute the model's *sufficient statistics*.
compute_ordinal_model_sufficient_statisti	Compute the sufficient statistics given realizations.
compute_ordinal_pdf_from_ordinal_sf(ordinal	Computes the probability density (or its jacobian) of an ordinal model [P(X = l), l=0..L] from *ordinal_sf* which are the survival function probabilities [P(X > l), i.e. P(X >= l+1), l=0..L-1] (or its jacobian).
compute_regularity_individual_parameters(.	Compute the regularity terms (and their gradients if requested), per individual variable of the model.
compute_regularity_individual_realization(	Compute regularity term for `IndividualRealization`.
compute_regularity_population_realization(	Compute regularity term for `PopulationRealization`.
compute_regularity_realization(realization)	Compute regularity term for a `AbstractRealization` instance.
compute_regularity_variable(value, mean, std, *)	Compute regularity term (Gaussian distribution) and optionally its gradient wrt value.
compute_sufficient_statistics(data, realizations)	Compute sufficient statistics from realizations.

continues on next page

Table 3 – continued from previous page

get_additional_ordinal_population_random_	Return the information of additional population random variables for the ordinal model.
get_individual_random_variable_informatio	Return the information on individual random variables relative to the model.
get_individual_variable_names()	Get the names of the individual variables of the model.
get_ordinal_parameters_updates_from_suffi	Return a dictionary computed from provided sufficient statistics for updating the parameters.
get_population_random_variable_informatio	Return the information on population random variables relative to the model.
get_population_variable_names()	Get the names of the population variables of the model.
initialize(dataset[, method])	Overloads base initialization of model (base method takes care of features consistency checks).
initialize_MCMC_toolbox()	Initialize the model's *MCMC* toolbox attribute.
load_hyperparameters(hyperparameters)	Updates all model hyperparameters from the provided hyperparameters.
load_parameters(parameters)	Updates all model parameters from the provided parameters.
move_to_device(device)	Move a model and its relevant attributes to the specified `torch.device`.
postprocess_model_estimation(estimation, *)	Extra layer of processing used to output nice estimated values in main API *Leaspy.estimate*.
save(path, **kwargs)	Save `Leaspy` object as json model parameter file.
time_reparametrization(timepoints, xi, tau)	Tensorized time reparametrization formula.
to_dict(*[, with_mixing_matrix])	Export `Leaspy` object as dictionary ready for *JSON* saving.
update_MCMC_toolbox(vars_to_update, realizations)	Update the model's *MCMC* toolbox attribute with the provided `vars_to_update`.
update_model_parameters_burn_in(data, ...)	Update the model's parameters during the burn in phase.
update_model_parameters_normal(data, ...)	Stochastic *sufficient statistics* used to update the parameters of the model.
update_ordinal_population_random_variable_	Update (in-place) the provided variable information dictionary.
update_parameters_burn_in(data, ...)	Update model parameters (burn-in phase).
update_parameters_normal(data, ...)	Update model parameters (after burn-in phase).
validate_compatibility_of_dataset(dataset)	Raise if the given `Dataset` is not compatible with the current model.

check_noise_model_compatibility(*model: BaseNoiseModel*) → None

Check compatibility between the model instance and provided noise model.

compute_appropriate_ordinal_model(*model_or_model_grad: Tensor*) → Tensor

Post-process model values (or their gradient) if needed.

compute_canonical_loss_tensorized(*data: Dataset*, *param_ind: Dict[str, Tensor]*, *, *attribute_type=None*) → Tensor

Compute canonical loss, which depends on the noise model.

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
[str or None (default)] Flag to ask for *MCMC* attributes instead of model's attributes.

Returns

loss
[torch.Tensor] shape = * (depending on noise-model, always summed over individuals & visits)

compute_individual_ages_from_biomarker_values(*value: float | List[float]*, *individual_parameters: Dict[str, Any]*, *feature: str | None = None*) → Tensor

For one individual, compute age(s) at which the given features values are reached (given the subject's individual parameters).

Consistency checks are done in the main *API* layer.

Parameters

value
[scalar or array_like[scalar] (list, tuple, numpy.ndarray)] Contains the *biomarker* value(s) of the subject.

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

feature
[str (or None)] Name of the considered *biomarker*.

Note: Optional for UnivariateModel, compulsory for MultivariateModel.

Returns

torch.Tensor
Contains the subject's ages computed at the given values(s). Shape of tensor is (1, n_values).

Raises

LeaspyModelInputError
If computation is tried on more than 1 individual.

compute_individual_ages_from_biomarker_values_tensorized(*value: Tensor*, *individual_parameters: dict*, *feature: str*) → Tensor

For one individual, compute age(s) at which the given features values are reached (given the subject's individual parameters), with tensorized inputs.

Parameters

value
[torch.Tensor of shape (1, n_values)] Contains the *biomarker* value(s) of the subject.

individual_parameters
[DictParamsTorch] Contains the individual parameters. Each individual parameter should be a `torch.Tensor`.

feature
[`str` (or None)] Name of the considered *biomarker*.

Note: Optional for `UnivariateModel`, compulsory for `MultivariateModel`.

Returns

`torch.Tensor`
Contains the subject's ages computed at the given values(s). Shape of tensor is `(n_values, 1)`.

compute_individual_ages_from_biomarker_values_tensorized_logistic(*value: Tensor*, *individual_parameters: dict*, *feature: str*) → Tensor

For one individual, compute age(s) at which the given features values are reached (given the subject's individual parameters), with tensorized inputs.

Parameters

value
[`torch.Tensor` of shape `(1, n_values)`] Contains the *biomarker* value(s) of the subject.

individual_parameters
[DictParamsTorch] Contains the individual parameters. Each individual parameter should be a `torch.Tensor`.

feature
[`str` (or None)] Name of the considered *biomarker*.

Note: Optional for `UnivariateModel`, compulsory for `MultivariateModel`.

Returns

`torch.Tensor`
Contains the subject's ages computed at the given values(s). Shape of tensor is `(n_values, 1)`.

compute_individual_attachment_tensorized(*data: Dataset*, *param_ind: Dict[str, Tensor]*, *, *attribute_type: str | None = None*) → Tensor

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
[DictParamsTorch] Contain the individual parameters.

attribute_type
[`str` or None] Flag to ask for *MCMC* attributes instead of model's attributes.

Returns

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

compute_individual_tensorized(*timepoints: Tensor*, *individual_parameters: dict*, *, *attribute_type=None*) → Tensor

Compute the individual trajectories.

Parameters

timepoints
[torch.Tensor] The time points for which to compute the trajectory.

individual_parameters
[DictParamsTorch] The individual parameters to use.

attribute_type
[Any, optional]

Returns

torch.Tensor
Individual trajectories.

compute_individual_tensorized_linear(*timepoints: Tensor*, *individual_parameters: dict*, *, *attribute_type=None*) → Tensor

Compute the individual trajectories.

Parameters

timepoints
[torch.Tensor] The time points for which to compute the trajectory.

individual_parameters
[DictParamsTorch] The individual parameters to use.

attribute_type
[Any, optional]

Returns

torch.Tensor
Individual trajectories.

compute_individual_tensorized_logistic(*timepoints: Tensor*, *individual_parameters: dict*, *, *attribute_type=None*) → Tensor

Compute the individual trajectories.

Parameters

timepoints
[torch.Tensor] The time points for which to compute the trajectory.

individual_parameters
[DictParamsTorch] The individual parameters to use.

attribute_type
[Any, optional]

Returns

torch.Tensor
Individual trajectories.

compute_individual_trajectory(*timepoints*, *individual_parameters: Dict[str, Any]*, ***, *skip_ips_checks: bool = False*) → Tensor

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)`.

Raises

LeaspyModelInputError
If computation is tried on more than 1 individual.

LeaspyIndividualParamsInputError
if invalid individual parameters.

compute_jacobian_tensorized(*timepoints: Tensor*, *individual_parameters: dict*, ***, *attribute_type=None*) → Dict[str, Tensor]

Compute the jacobian of the model w.r.t. each individual parameter.

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

Note: As most of numerical operations are repeated when computing model & jacobian, we should create a single method that is able to compute model & jacobian "together" (= efficiently) when requested with a flag for instance.

Parameters

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

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

attribute_type
[`str` or None] Flag to ask for *MCMC* attributes instead of model's attributes.

Returns

`dict` **[param_name:** `str`**,** `torch.Tensor` **]**
Tensors are of shape `(n_individuals, n_timepoints, n_features, n_dims_param)`.

compute_jacobian_tensorized_linear(*timepoints: Tensor*, *individual_parameters: dict*, *, *attribute_type=None*) → Dict[str, Tensor]

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.

Note: As most of numerical operations are repeated when computing model & jacobian, we should create a single method that is able to compute model & jacobian "together" (= efficiently) when requested with a flag for instance.

Parameters

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

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

attribute_type
[`str` or None] Flag to ask for *MCMC* attributes instead of model's attributes.

Returns

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

compute_jacobian_tensorized_logistic(*timepoints: Tensor*, *individual_parameters: dict*, *, *attribute_type=None*) → Dict[str, Tensor]

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.

Note: As most of numerical operations are repeated when computing model & jacobian, we should create a single method that is able to compute model & jacobian "together" (= efficiently) when requested with a flag for instance.

Parameters

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

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

attribute_type
[`str` or None] Flag to ask for *MCMC* attributes instead of model's attributes.

Returns

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

compute_mean_traj(*timepoints: Tensor*, *, *attribute_type: str | None = None*) → Tensor

Compute trajectory of the model with individual parameters being the group-average ones.

TODO check dimensions of io?

Parameters

timepoints
[torch.Tensor of shape (1, n_timepoints)]

attribute_type
[str or None] If a string, should be "MCMC".

Returns

torch.Tensor of shape (1, n_timepoints, dimension)
The group-average values at given timepoints.

compute_model_sufficient_statistics(*data: Dataset*, *realizations: CollectionRealization*) → Dict[str, Tensor]

Compute the model's *sufficient statistics*.

Parameters

data
[Dataset] The input dataset.

realizations
[CollectionRealization] The realizations from which to compute the model's *sufficient statistics*.

Returns

DictParamsTorch
The computed *sufficient statistics*.

compute_ordinal_model_sufficient_statistics(*realizations: CollectionRealization*) → Dict[str, Tensor]

Compute the sufficient statistics given realizations.

static compute_ordinal_pdf_from_ordinal_sf(*ordinal_sf: Tensor*, *dim_ordinal_levels: int = 3*) → Tensor

Computes the probability density (or its jacobian) of an ordinal model [P(X = l), l=0..L] from *ordinal_sf* which are the survival function probabilities [P(X > l), i.e. P(X >= l+1), l=0..L-1] (or its jacobian).

Parameters

ordinal_sf
[*torch.FloatTensor*] Survival function values : ordinal_sf[..., l] is the proba to be superior or equal to l+1 Dimensions are: * 0=individual * 1=visit * 2=feature * 3=ordinal_level [l=0..L-1] * [4=individual_parameter_dim_when_gradient]

dim_ordinal_levels
[int, default = 3] The dimension of the tensor where the ordinal levels are.

Returns

ordinal_pdf
[*torch.FloatTensor* (same shape as input, except for dimension 3 which has one more element)] ordinal_pdf[..., l] is the proba to be equal to l (l=0..L)

compute_regularity_individual_parameters(*individual_parameters: Dict[str, Tensor]*, *, *include_constant: bool = False*) → Tuple[Dict[str, Tensor], Dict[str, Tensor]]

Compute the regularity terms (and their gradients if requested), per individual variable of the model.

Parameters

individual_parameters
[dict [str, torch.Tensor]] Individual parameters as a dict of tensors of shape (n_ind, n_dims_param).

include_constant
[bool, optional] Whether to include a constant term or not. Default=False.

Returns

regularity
[dict [param_name: str, torch.Tensor]] Regularity of the patient(s) corresponding to the given individual parameters. Tensors have shape (n_individuals).

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

compute_regularity_individual_realization(*realization: IndividualRealization*) → Tensor

Compute regularity term for IndividualRealization.

Parameters

realization
[IndividualRealization]

Returns

torch.Tensor of the same shape as IndividualRealization.tensor

compute_regularity_population_realization(*realization: PopulationRealization*) → Tensor

Compute regularity term for PopulationRealization.

Parameters

realization
[PopulationRealization]

Returns

torch.Tensor of the same shape as PopulationRealization.tensor

compute_regularity_realization(*realization: AbstractRealization*) → Tensor

Compute regularity term for a AbstractRealization instance.

Parameters

realization
[AbstractRealization]

Returns

torch.Tensor of the same shape as AbstractRealization.tensor

compute_regularity_variable(*value: Tensor*, *mean: Tensor*, *std: Tensor*, *, *include_constant: bool = True*, *with_gradient: bool = False*) → Tensor | Tuple[Tensor, Tensor]

Compute regularity term (Gaussian distribution) and optionally its gradient wrt value.

Note: TODO: should be encapsulated in a `RandomVariableSpecification` class together with other specs of RV.

Parameters

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

include_constant
[`bool` (default `True`)] Whether we include or not additional terms constant with respect to *value*.

with_gradient
[`bool` (default `False`)] Whether we also return the gradient of *regularity* term with respect to *value*.

Returns

`torch.Tensor` of same shape than input

`compute_sufficient_statistics`(*data: Dataset*, *realizations: CollectionRealization*) → Dict[str, Tensor]

Compute sufficient statistics from realizations.

Parameters

data
[`Dataset`]

realizations
[`CollectionRealization`]

Returns

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

property dimension: `int` | `None`

The dimension of the model. If the private attribute is defined, then it takes precedence over the feature length. The associated setters are responsible for their coherence.

`get_additional_ordinal_population_random_variable_information`() → Dict[str, Any]

Return the information of additional population random variables for the ordinal model.

`get_individual_random_variable_information`() → Dict[str, Any]

Return the information on individual random variables relative to the model.

Returns

DictParams
The information on the individual random variables.

`get_individual_variable_names`() → List[str]

Get the names of the individual variables of the model.

Returns

`list` of `str`

get_ordinal_parameters_updates_from_sufficient_statistics(*sufficient_statistics: Dict[str, Tensor]*) → Dict[str, Tensor]

Return a dictionary computed from provided sufficient statistics for updating the parameters.

get_population_random_variable_information() → Dict[str, Any]

Return the information on population random variables relative to the model.

Returns

DictParams
The information on the population random variables.

get_population_variable_names() → List[str]

Get the names of the population variables of the model.

Returns

list of str

initialize(*dataset: Dataset*, *method: str = 'default'*) → None

Overloads base initialization of model (base method takes care of features consistency checks).

Parameters

dataset
[Dataset] Input Dataset from which to initialize the model.

method
[str, optional] The initialization method to be used. Default='default'.

initialize_MCMC_toolbox() → None

Initialize the model's *MCMC* toolbox attribute.

property is_ordinal: bool

Property to check if the model is of ordinal sub-type.

property is_ordinal_ranking: bool

Property to check if the model is of ordinal-ranking sub-type (working with survival functions).

load_hyperparameters(*hyperparameters: Dict[str, Any]*) → None

Updates all model hyperparameters from the provided hyperparameters.

Parameters

hyperparameters
[KwargsType] The hyperparameters to be loaded.

load_parameters(*parameters: Dict[str, Any]*) → None

Updates all model parameters from the provided parameters.

Parameters

parameters
[KwargsType] The parameters to be loaded.

move_to_device(*device: device*) → None

Move a model and its relevant attributes to the specified torch.device.

Parameters

device
[torch.device]

property ordinal_infos: dict | None

Property to return the ordinal info dictionary.

postprocess_model_estimation(*estimation: ndarray*, *, *ordinal_method: str = 'MLE'*, ***kws*) → ndarray | Dict[Hashable, ndarray]

Extra layer of processing used to output nice estimated values in main API *Leaspy.estimate*.

Parameters

estimation
[numpy.ndarray[float]] The raw estimated values by model (from *compute_individual_trajectory*)

ordinal_method
[str] <!> Only used for ordinal models. * 'MLE' or 'maximum_likelihood' returns maximum likelihood estimator for each point (int) * 'E' or 'expectation' returns expectation (float) * 'P' or 'probabilities' returns probabilities of all-possible levels for a given feature:

{feature_name: array[float]<0..max_level_ft>}

****kws**
Some extra keywords arguments that may be handled in the future.

Returns

numpy.ndarray[float] or dict[str, numpy.ndarray[float]]
Post-processed values. In case using 'probabilities' mode, the values are a dictionary with keys being: *(feature_name: str, feature_level: int<0..max_level_for_feature>)* Otherwise it is a standard numpy.ndarray corresponding to different model features (in order)

save(*path: str*, ***kwargs*) → None

Save `Leaspy` object as json model parameter file.

TODO move logic upstream?

Parameters

path
[`str`] Path to store the model's parameters.

****kwargs**
Keyword arguments for `AbstractModel.to_dict()` child method and `json.dump` function (default to indent=2).

static time_reparametrization(*timepoints: Tensor*, *xi: Tensor*, *tau: Tensor*) → Tensor

Tensorized time reparametrization formula.

Warning: 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*

to_dict(*, *with_mixing_matrix: bool = True*) → Dict[str, Any]

Export Leaspy object as dictionary ready for *JSON* saving.

Parameters

with_mixing_matrix
[bool (default True)] Save the *mixing matrix* in the exported file in its 'parameters' section.

> **Warning:** 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...

Returns

KwargsType
The object as a dictionary.

update_MCMC_toolbox(*vars_to_update: set*, *realizations: CollectionRealization*) → None

Update the model's *MCMC* toolbox attribute with the provided vars_to_update.

Parameters

vars_to_update
[set of str] The set of variable names to be updated.

realizations
[CollectionRealization] The realizations to use for updating the *MCMC* toolbox.

update_model_parameters_burn_in(*data: Dataset*, *sufficient_statistics: Dict[str, Tensor]*) → None

Update the model's parameters during the burn in phase.

During the burn-in phase, we only need to store the following parameters (cf. !66 and #60)

- noise_std
- *_mean/std for *regularization* of individual variables
- others population parameters for *regularization* of population variables

We don't need to update the model "attributes" (never used during burn-in!).

Parameters

data
[Dataset] The input dataset.

sufficient_statistics
[DictParamsTorch] The *sufficient statistics* to use for parameter update.

update_model_parameters_normal(*data: Dataset*, *sufficient_statistics: Dict[str, Tensor]*) → None

Stochastic *sufficient statistics* used to update the parameters of the model.

Note:

TODOs:

- factorize `update_model_parameters_***` methods?
- add a true, configurable, validation for all parameters? (e.g.: bounds on tau_var/std but also on tau_mean, ...)
- check the SS, especially the issue with mean(xi) and v_k
- Learn the mean of xi and v_k
- Set the mean of xi to 0 and add it to the mean of V_k

Parameters

data
[`Dataset`] The input dataset.

sufficient_statistics
[DictParamsTorch] The *sufficient statistics* to use for parameter update.

update_ordinal_population_random_variable_information(*variables_info: Dict[str, Any]*) → None

Update (in-place) the provided variable information dictionary.

Nota: this is needed due to different signification of *v0* in ordinal model (common per-level velocity)

Parameters

variables_info
[DictParams] The variables information to be updated with ordinal logic.

update_parameters_burn_in(*data: Dataset*, *sufficient_statistics: Dict[str, Tensor]*) → None

Update model parameters (burn-in phase).

Parameters

data
[`Dataset`]

sufficient_statistics
[`dict` [suff_stat: `str`, `torch.Tensor`]]

update_parameters_normal(*data: Dataset*, *sufficient_statistics: Dict[str, Tensor]*) → None

Update model parameters (after burn-in phase).

Parameters

data
[`Dataset`]

sufficient_statistics
[`dict` [suff_stat: `str`, `torch.Tensor`]]

validate_compatibility_of_dataset(*dataset: Dataset*) → None

Raise if the given `Dataset` is not compatible with the current model.

Parameters

dataset
[Dataset] The Dataset we want to model.

Raises

LeaspyModelInputError

- If the Dataset has a number of dimensions smaller than 2.
- If the Dataset does not have the same dimensionality as the model.
- If the Dataset's headers do not match the model's.

3.2.9 leaspy.models.MultivariateParallelModel

class MultivariateParallelModel(*name: str*, ***kwargs*)

Bases: AbstractMultivariateModel

Logistic model for multiple variables of interest, imposing same average evolution pace for all variables (logistic curves are only time-shifted).

Parameters

name
[str] The name of the model.

****kwargs**
Hyperparameters of the model.

Attributes

dimension
The dimension of the model.

features
is_ordinal
Property to check if the model is of ordinal sub-type.

is_ordinal_ranking
Property to check if the model is of ordinal-ranking sub-type (working with survival functions).

noise_model
ordinal_infos
Property to return the ordinal info dictionary.

Methods

check_noise_model_compatibility(model)	Check compatibility between the model instance and provided noise model.
compute_appropriate_ordinal_model(...)	Post-process model values (or their gradient) if needed.
compute_canonical_loss_tensorized(data, ...)	Compute canonical loss, which depends on the noise model.
compute_individual_ages_from_biomarker_va	For one individual, compute age(s) at which the given features values are reached (given the subject's individual parameters).

continues on next page

Table 4 – continued from previous page

compute_individual_ages_from_biomarker_va	For one individual, compute age(s) at which the given features values are reached (given the subject's individual parameters), with tensorized inputs.
compute_individual_attachment_tensorized(.	Compute *attachment* term (per subject).
compute_individual_tensorized(timepoints, ...)	Compute the individual trajectories.
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_model_sufficient_statistics(data, ...)	Compute sufficient statistics from a CollectionRealization.
compute_ordinal_model_sufficient_statisti	Compute the sufficient statistics given realizations.
compute_ordinal_pdf_from_ordinal_sf(ordinal	Computes the probability density (or its jacobian) of an ordinal model [P(X = l), l=0..L] from *ordinal_sf* which are the survival function probabilities [P(X > l), i.e. P(X >= l+1), l=0..L-1] (or its jacobian).
compute_regularity_individual_parameters(.	Compute the regularity terms (and their gradients if requested), per individual variable of the model.
compute_regularity_individual_realization(	Compute regularity term for IndividualRealization.
compute_regularity_population_realization(	Compute regularity term for PopulationRealization.
compute_regularity_realization(realization)	Compute regularity term for a AbstractRealization instance.
compute_regularity_variable(value, mean, std, *)	Compute regularity term (Gaussian distribution) and optionally its gradient wrt value.
compute_sufficient_statistics(data, realizations)	Compute sufficient statistics from realizations.
get_additional_ordinal_population_random_	Return the information of additional population random variables for the ordinal model.
get_individual_random_variable_informatio	Return the information on individual random variables relative to the model.
get_individual_variable_names()	Get the names of the individual variables of the model.
get_ordinal_parameters_updates_from_suffi	Return a dictionary computed from provided sufficient statistics for updating the parameters.
get_population_random_variable_informatio	Return the information on population random variables relative to the model.
get_population_variable_names()	Get the names of the population variables of the model.
initialize(dataset[, method])	Overloads base initialization of model (base method takes care of features consistency checks).
initialize_MCMC_toolbox()	Initialize *MCMC* toolbox for calibration of model.
load_hyperparameters(hyperparameters)	Updates all model hyperparameters from the provided hyperparameters.
load_parameters(parameters)	Updates all model parameters from the provided parameters.
move_to_device(device)	Move a model and its relevant attributes to the specified torch.device.

continues on next page

Table 4 – continued from previous page

`postprocess_model_estimation`(estimation, *)	Extra layer of processing used to output nice estimated values in main API *Leaspy.estimate*.
`save`(path, **kwargs)	Save `Leaspy` object as json model parameter file.
`time_reparametrization`(timepoints, xi, tau)	Tensorized time reparametrization formula.
`to_dict`(*[, with_mixing_matrix])	Export `Leaspy` object as dictionary ready for *JSON* saving.
`update_MCMC_toolbox`(vars_to_update, realizations)	Update the *MCMC* toolbox with a `CollectionRealization` of model population parameters.
`update_model_parameters_burn_in`(data, ...)	Update model parameters (burn-in phase).
`update_model_parameters_normal`(data, ...)	Update model parameters (after burn-in phase).
`update_ordinal_population_random_variable`.	Update (in-place) the provided variable information dictionary.
`update_parameters_burn_in`(data, ...)	Update model parameters (burn-in phase).
`update_parameters_normal`(data, ...)	Update model parameters (after burn-in phase).
`validate_compatibility_of_dataset`(dataset)	Raise if the given `Dataset` is not compatible with the current model.

check_noise_model_compatibility(*model: BaseNoiseModel*) → None

Check compatibility between the model instance and provided noise model.

compute_appropriate_ordinal_model(*model_or_model_grad: Tensor*) → Tensor

Post-process model values (or their gradient) if needed.

compute_canonical_loss_tensorized(*data: Dataset*, *param_ind: Dict[str, Tensor]*, *, *attribute_type=None*) → Tensor

Compute canonical loss, which depends on the noise model.

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
[`str` or None (default)] Flag to ask for *MCMC* attributes instead of model's attributes.

Returns

loss
[`torch.Tensor`] shape = * (depending on noise-model, always summed over individuals & visits)

compute_individual_ages_from_biomarker_values(*value: float | List[float]*, *individual_parameters: Dict[str, Any]*, *feature: str | None = None*) → Tensor

For one individual, compute age(s) at which the given features values are reached (given the subject's individual parameters).

Consistency checks are done in the main *API* layer.

Parameters

value
[scalar or array_like[scalar] (`list`, `tuple`, `numpy.ndarray`)] Contains the *biomarker* value(s) of the subject.

individual_parameters
[`dict`] Contains the individual parameters. Each individual parameter should be a scalar or array_like.

feature
[`str` (or None)] Name of the considered *biomarker*.

Note: Optional for `UnivariateModel`, compulsory for `MultivariateModel`.

Returns

`torch.Tensor`
Contains the subject's ages computed at the given values(s). Shape of tensor is `(1, n_values)`.

Raises

`LeaspyModelInputError`
If computation is tried on more than 1 individual.

compute_individual_ages_from_biomarker_values_tensorized(*value: Tensor*, *individual_parameters: dict*, *feature: str*) → Tensor

For one individual, compute age(s) at which the given features values are reached (given the subject's individual parameters), with tensorized inputs.

Parameters

value
[`torch.Tensor` of shape `(1, n_values)`] Contains the *biomarker* value(s) of the subject.

individual_parameters
[DictParamsTorch] Contains the individual parameters. Each individual parameter should be a `torch.Tensor`.

feature
[`str` (or None)] Name of the considered *biomarker*.

Note: Optional for `UnivariateModel`, compulsory for `MultivariateModel`.

Returns

`torch.Tensor`
Contains the subject's ages computed at the given values(s). Shape of tensor is `(n_values, 1)`.

compute_individual_attachment_tensorized(*data: Dataset*, *param_ind: Dict[str, Tensor]*, *, *attribute_type: str | None = None*) → Tensor

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
[DictParamsTorch] Contain the individual parameters.

attribute_type
[str or None] Flag to ask for *MCMC* attributes instead of model's attributes.

Returns

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

compute_individual_tensorized(*timepoints: Tensor*, *individual_parameters: dict*, *, *attribute_type=None*) → Tensor

Compute the individual trajectories.

Parameters

timepoints
[torch.Tensor] The time points for which to compute the trajectory.

individual_parameters
[DictParamsTorch] The individual parameters to use.

attribute_type
[Any, optional]

Returns

torch.Tensor
Individual trajectories.

compute_individual_trajectory(*timepoints*, *individual_parameters: Dict[str, Any]*, *, *skip_ips_checks: bool = False*) → Tensor

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).

Raises

LeaspyModelInputError
If computation is tried on more than 1 individual.

LeaspyIndividualParamsInputError
if invalid individual parameters.

compute_jacobian_tensorized(*timepoints: Tensor*, *individual_parameters: dict*, ***, *attribute_type=None*) → Dict[str, Tensor]

Compute the jacobian of the model w.r.t. each individual parameter.

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

Note: As most of numerical operations are repeated when computing model & jacobian, we should create a single method that is able to compute model & jacobian "together" (= efficiently) when requested with a flag for instance.

Parameters

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

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

attribute_type
[`str` or None] Flag to ask for *MCMC* attributes instead of model's attributes.

Returns

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

compute_mean_traj(*timepoints: Tensor*, ***, *attribute_type: str | None = None*) → Tensor

Compute trajectory of the model with individual parameters being the group-average ones.

TODO check dimensions of io?

Parameters

timepoints
[`torch.Tensor` of shape `(1, n_timepoints)`]

attribute_type
[`str` or None] If a string, should be "MCMC".

Returns

`torch.Tensor` of shape `(1, n_timepoints, dimension)`
The group-average values at given timepoints.

compute_model_sufficient_statistics(*data: Dataset*, *realizations: CollectionRealization*) → Dict[str, Tensor]

Compute sufficient statistics from a `CollectionRealization`.

Parameters

data
[`Dataset`]

realizations
[`CollectionRealization`]

Returns

`dict` [**suff_stat:** `str`, `torch.Tensor`]

compute_ordinal_model_sufficient_statistics(*realizations: CollectionRealization*) → Dict[str, Tensor]

Compute the sufficient statistics given realizations.

static compute_ordinal_pdf_from_ordinal_sf(*ordinal_sf: Tensor*, *dim_ordinal_levels: int = 3*) → Tensor

Computes the probability density (or its jacobian) of an ordinal model [P(X = l), l=0..L] from *ordinal_sf* which are the survival function probabilities [P(X > l), i.e. P(X >= l+1), l=0..L-1] (or its jacobian).

Parameters

ordinal_sf
[*torch.FloatTensor*] Survival function values : ordinal_sf[..., l] is the proba to be superior or equal to l+1 Dimensions are: * 0=individual * 1=visit * 2=feature * 3=ordinal_level [l=0..L-1] * [4=individual_parameter_dim_when_gradient]

dim_ordinal_levels
[int, default = 3] The dimension of the tensor where the ordinal levels are.

Returns

ordinal_pdf
[*torch.FloatTensor* (same shape as input, except for dimension 3 which has one more element)] ordinal_pdf[..., l] is the proba to be equal to l (l=0..L)

compute_regularity_individual_parameters(*individual_parameters: Dict[str, Tensor]*, *, *include_constant: bool = False*) → Tuple[Dict[str, Tensor], Dict[str, Tensor]]

Compute the regularity terms (and their gradients if requested), per individual variable of the model.

Parameters

individual_parameters
[`dict` [`str`, `torch.Tensor`]] Individual parameters as a dict of tensors of shape `(n_ind, n_dims_param)`.

include_constant
[`bool`, optional] Whether to include a constant term or not. Default=False.

Returns

regularity
[`dict` [param_name: `str`, `torch.Tensor`]] Regularity of the patient(s) corresponding to the given individual parameters. Tensors have shape `(n_individuals)`.

regularity_grads
[`dict` [param_name: `str`, `torch.Tensor`]] Gradient of regularity term with respect to individual parameters. Tensors have shape `(n_individuals, n_dims_param)`.

compute_regularity_individual_realization(*realization: IndividualRealization*) → Tensor

Compute regularity term for `IndividualRealization`.

Parameters

realization
[`IndividualRealization`]

Returns

torch.Tensor of the same shape as IndividualRealization.tensor

compute_regularity_population_realization(*realization: PopulationRealization*) → Tensor

Compute regularity term for PopulationRealization.

Parameters

realization
[PopulationRealization]

Returns

torch.Tensor of the same shape as PopulationRealization.tensor

compute_regularity_realization(*realization: AbstractRealization*) → Tensor

Compute regularity term for a AbstractRealization instance.

Parameters

realization
[AbstractRealization]

Returns

torch.Tensor of the same shape as AbstractRealization.tensor

compute_regularity_variable(*value: Tensor*, *mean: Tensor*, *std: Tensor*, *, *include_constant: bool = True*, *with_gradient: bool = False*) → Tensor | Tuple[Tensor, Tensor]

Compute regularity term (Gaussian distribution) and optionally its gradient wrt value.

Note: TODO: should be encapsulated in a RandomVariableSpecification class together with other specs of RV.

Parameters

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

include_constant
[bool (default True)] Whether we include or not additional terms constant with respect to *value*.

with_gradient
[bool (default False)] Whether we also return the gradient of *regularity* term with respect to *value*.

Returns

torch.Tensor of same shape than input

compute_sufficient_statistics(*data: Dataset*, *realizations: CollectionRealization*) → Dict[str, Tensor]

Compute sufficient statistics from realizations.

Parameters

data
[Dataset]

realizations
[CollectionRealization]

Returns

`dict` [**suff_stat:** `str`, `torch.Tensor`]

property dimension: `int` | `None`

The dimension of the model. If the private attribute is defined, then it takes precedence over the feature length. The associated setters are responsible for their coherence.

get_additional_ordinal_population_random_variable_information() → Dict[str, Any]

Return the information of additional population random variables for the ordinal model.

get_individual_random_variable_information() → Dict[str, Any]

Return the information on individual random variables relative to the model.

Returns

DictParams
The information on the individual random variables.

get_individual_variable_names() → List[str]

Get the names of the individual variables of the model.

Returns

`list` **of** `str`

get_ordinal_parameters_updates_from_sufficient_statistics(*sufficient_statistics: Dict[str, Tensor]*) → Dict[str, Tensor]

Return a dictionary computed from provided sufficient statistics for updating the parameters.

get_population_random_variable_information() → Dict[str, Any]

Return the information on population random variables relative to the model.

Returns

DictParams
The information on the population random variables.

get_population_variable_names() → List[str]

Get the names of the population variables of the model.

Returns

`list` **of** `str`

initialize(*dataset: Dataset*, *method: str = 'default'*) → None

Overloads base initialization of model (base method takes care of features consistency checks).

Parameters

dataset
[`Dataset`] Input `Dataset` from which to initialize the model.

method
[`str`, optional] The initialization method to be used. Default='default'.

initialize_MCMC_toolbox() → None

Initialize *MCMC* toolbox for calibration of model.

property is_ordinal: `bool`

Property to check if the model is of ordinal sub-type.

property is_ordinal_ranking: bool

Property to check if the model is of ordinal-ranking sub-type (working with survival functions).

load_hyperparameters(*hyperparameters: Dict[str, Any]*) → None

Updates all model hyperparameters from the provided hyperparameters.

Parameters

hyperparameters
[KwargsType] The hyperparameters to be loaded.

load_parameters(*parameters: Dict[str, Any]*) → None

Updates all model parameters from the provided parameters.

Parameters

parameters
[KwargsType] The parameters to be loaded.

move_to_device(*device: device*) → None

Move a model and its relevant attributes to the specified `torch.device`.

Parameters

device
[`torch.device`]

property ordinal_infos: dict | None

Property to return the ordinal info dictionary.

postprocess_model_estimation(*estimation: ndarray*, *, *ordinal_method: str = 'MLE'*, ***kws*) → ndarray | Dict[Hashable, ndarray]

Extra layer of processing used to output nice estimated values in main API *Leaspy.estimate*.

Parameters

estimation
[numpy.ndarray[float]] The raw estimated values by model (from *compute_individual_trajectory*)

ordinal_method
[str] <!> Only used for ordinal models. * 'MLE' or 'maximum_likelihood' returns maximum likelihood estimator for each point (int) * 'E' or 'expectation' returns expectation (float) * 'P' or 'probabilities' returns probabilities of all-possible levels for a given feature:

{feature_name: array[float]<0..max_level_ft>}

****kws**
Some extra keywords arguments that may be handled in the future.

Returns

numpy.ndarray[float] or dict[str, numpy.ndarray[float]]
Post-processed values. In case using 'probabilities' mode, the values are a dictionary with keys being: *(feature_name: str, feature_level: int<0..max_level_for_feature>)* Otherwise it is a standard numpy.ndarray corresponding to different model features (in order)

save(*path: str*, ***kwargs*) → None

Save `Leaspy` object as json model parameter file.

TODO move logic upstream?

Parameters

path
[`str`] Path to store the model's parameters.

****kwargs**
Keyword arguments for `AbstractModel.to_dict()` child method and `json.dump` function (default to indent=2).

static time_reparametrization(*timepoints: Tensor*, *xi: Tensor*, *tau: Tensor*) → Tensor

Tensorized time reparametrization formula.

> **Warning:** 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*

to_dict(*, *with_mixing_matrix: bool = True*) → Dict[str, Any]

Export `Leaspy` object as dictionary ready for *JSON* saving.

Parameters

with_mixing_matrix
[`bool` (default `True`)] Save the *mixing matrix* in the exported file in its 'parameters' section.

> **Warning:** 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…

Returns

KwargsType
The object as a dictionary.

update_MCMC_toolbox(*vars_to_update: set*, *realizations: CollectionRealization*) → None

Update the *MCMC* toolbox with a `CollectionRealization` of model population parameters.

Parameters

vars_to_update
[set of str] Names of the population parameters to update in *MCMC* toolbox.

realizations
[CollectionRealization] All the realizations to update *MCMC* toolbox with.

update_model_parameters_burn_in(*data: Dataset*, *sufficient_statistics: Dict[str, Tensor]*) → None

Update model parameters (burn-in phase).

Parameters

data
[Dataset]

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

update_model_parameters_normal(*data: Dataset*, *sufficient_statistics: Dict[str, Tensor]*) → None

Update model parameters (after burn-in phase).

Parameters

data
[Dataset]

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

update_ordinal_population_random_variable_information(*variables_info: Dict[str, Any]*) → None

Update (in-place) the provided variable information dictionary.

Nota: this is needed due to different signification of *v0* in ordinal model (common per-level velocity)

Parameters

variables_info
[DictParams] The variables information to be updated with ordinal logic.

update_parameters_burn_in(*data: Dataset*, *sufficient_statistics: Dict[str, Tensor]*) → None

Update model parameters (burn-in phase).

Parameters

data
[Dataset]

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

update_parameters_normal(*data: Dataset*, *sufficient_statistics: Dict[str, Tensor]*) → None

Update model parameters (after burn-in phase).

Parameters

data
[Dataset]

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

validate_compatibility_of_dataset(*dataset: Dataset*) → None

Raise if the given Dataset is not compatible with the current model.

Parameters

dataset
[Dataset] The Dataset we want to model.

Raises

LeaspyModelInputError

- If the Dataset has a number of dimensions smaller than 2.
- If the Dataset does not have the same dimensionality as the model.
- If the Dataset's headers do not match the model's.

3.2.10 leaspy.models.UnivariateModel

class UnivariateModel(*name: str*, ***kwargs*)

Bases: MultivariateModel

Univariate (logistic or linear) model for a single variable of interest.

Parameters

name
[str] The name of the model.

****kwargs**
Hyperparameters of the model.

Raises

LeaspyModelInputError

- If *name* is not one of allowed sub-type: 'univariate_linear' or 'univariate_logistic'
- If hyperparameters are inconsistent

Attributes

dimension
The dimension of the model.

features
is_ordinal
Property to check if the model is of ordinal sub-type.

is_ordinal_ranking
Property to check if the model is of ordinal-ranking sub-type (working with survival functions).

noise_model
ordinal_infos
Property to return the ordinal info dictionary.

Methods

check_noise_model_compatibility(model)	Check compatibility between the model instance and provided noise model.
compute_appropriate_ordinal_model(...)	Post-process model values (or their gradient) if needed.
compute_canonical_loss_tensorized(data, ...)	Compute canonical loss, which depends on the noise model.
compute_individual_ages_from_biomarker_va	For one individual, compute age(s) at which the given features values are reached (given the subject's individual parameters).
compute_individual_ages_from_biomarker_va	For one individual, compute age(s) at which the given features values are reached (given the subject's individual parameters), with tensorized inputs.
compute_individual_ages_from_biomarker_va	For one individual, compute age(s) at which the given features values are reached (given the subject's individual parameters), with tensorized inputs.
compute_individual_attachment_tensorized(.	Compute *attachment* term (per subject).
compute_individual_tensorized(timepoints, ...)	Compute the individual trajectories.
compute_individual_tensorized_linear(...[, ...])	Compute the individual trajectories.
compute_individual_tensorized_logistic(...)	Compute the individual trajectories.
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_mean_traj(timepoints, *[, ...])	Compute trajectory of the model with individual parameters being the group-average ones.
compute_model_sufficient_statistics(data, ...)	Compute the model's *sufficient statistics*.
compute_ordinal_model_sufficient_statisti	Compute the sufficient statistics given realizations.
compute_ordinal_pdf_from_ordinal_sf(ordinal	Computes the probability density (or its jacobian) of an ordinal model [P(X = l), l=0..L] from *ordinal_sf* which are the survival function probabilities [P(X > l), i.e. P(X >= l+1), l=0..L-1] (or its jacobian).
compute_regularity_individual_parameters(.	Compute the regularity terms (and their gradients if requested), per individual variable of the model.
compute_regularity_individual_realization(	Compute regularity term for `IndividualRealization`.
compute_regularity_population_realization(	Compute regularity term for `PopulationRealization`.
compute_regularity_realization(realization)	Compute regularity term for a `AbstractRealization` instance.
compute_regularity_variable(value, mean, std, *)	Compute regularity term (Gaussian distribution) and optionally its gradient wrt value.
compute_sufficient_statistics(data, realizations)	Compute sufficient statistics from realizations.

continues on next page

Table 5 – continued from previous page

`get_additional_ordinal_population_random_`	Return the information of additional population random variables for the ordinal model.
`get_individual_random_variable_informatio`	Return the information on individual random variables relative to the model.
`get_individual_variable_names`()	Get the names of the individual variables of the model.
`get_ordinal_parameters_updates_from_suffi`	Return a dictionary computed from provided sufficient statistics for updating the parameters.
`get_population_random_variable_informatio`	Return the information on population random variables relative to the model.
`get_population_variable_names`()	Get the names of the population variables of the model.
`initialize`(dataset[, method])	Overloads base initialization of model (base method takes care of features consistency checks).
`initialize_MCMC_toolbox`()	Initialize the model's *MCMC* toolbox attribute.
`load_hyperparameters`(hyperparameters)	Updates all model hyperparameters from the provided hyperparameters.
`load_parameters`(parameters)	Updates all model parameters from the provided parameters.
`move_to_device`(device)	Move a model and its relevant attributes to the specified `torch.device`.
`postprocess_model_estimation`(estimation, *)	Extra layer of processing used to output nice estimated values in main API *Leaspy.estimate*.
`save`(path, **kwargs)	Save `Leaspy` object as json model parameter file.
`time_reparametrization`(timepoints, xi, tau)	Tensorized time reparametrization formula.
`to_dict`(*[, with_mixing_matrix])	Export `Leaspy` object as dictionary ready for *JSON* saving.
`update_MCMC_toolbox`(vars_to_update, realizations)	Update the model's *MCMC* toolbox attribute with the provided `vars_to_update`.
`update_model_parameters_burn_in`(data, ...)	Update the model's parameters during the burn in phase.
`update_model_parameters_normal`(data, ...)	Stochastic *sufficient statistics* used to update the parameters of the model.
`update_ordinal_population_random_variable_`	Update (in-place) the provided variable information dictionary.
`update_parameters_burn_in`(data, ...)	Update model parameters (burn-in phase).
`update_parameters_normal`(data, ...)	Update model parameters (after burn-in phase).
`validate_compatibility_of_dataset`(dataset)	Raise if the given `Dataset` is not compatible with the current model.

check_noise_model_compatibility(*model: BaseNoiseModel*) → None

Check compatibility between the model instance and provided noise model.

compute_appropriate_ordinal_model(*model_or_model_grad: Tensor*) → Tensor

Post-process model values (or their gradient) if needed.

compute_canonical_loss_tensorized(*data: Dataset*, *param_ind: Dict[str, Tensor]*, *, *attribute_type=None*) → Tensor

Compute canonical loss, which depends on the noise model.

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
[str or None (default)] Flag to ask for *MCMC* attributes instead of model's attributes.

Returns

loss
[torch.Tensor] shape = * (depending on noise-model, always summed over individuals & visits)

compute_individual_ages_from_biomarker_values(*value: float | List[float]*, *individual_parameters: Dict[str, Any]*, *feature: str | None = None*) → Tensor

For one individual, compute age(s) at which the given features values are reached (given the subject's individual parameters).

Consistency checks are done in the main *API* layer.

Parameters

value
[scalar or array_like[scalar] (list, tuple, numpy.ndarray)] Contains the *biomarker* value(s) of the subject.

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

feature
[str (or None)] Name of the considered *biomarker*.

Note: Optional for UnivariateModel, compulsory for MultivariateModel.

Returns

torch.Tensor
Contains the subject's ages computed at the given values(s). Shape of tensor is (1, n_values).

Raises

LeaspyModelInputError
If computation is tried on more than 1 individual.

compute_individual_ages_from_biomarker_values_tensorized(*value: Tensor*, *individual_parameters: dict*, *feature: str*) → Tensor

For one individual, compute age(s) at which the given features values are reached (given the subject's individual parameters), with tensorized inputs.

Parameters

value
[torch.Tensor of shape (1, n_values)] Contains the *biomarker* value(s) of the subject.

individual_parameters
[DictParamsTorch] Contains the individual parameters. Each individual parameter should be a `torch.Tensor`.

feature
[`str` (or None)] Name of the considered *biomarker*.

Note: Optional for `UnivariateModel`, compulsory for `MultivariateModel`.

Returns

`torch.Tensor`
Contains the subject's ages computed at the given values(s). Shape of tensor is `(n_values, 1)`.

compute_individual_ages_from_biomarker_values_tensorized_logistic(*value: Tensor*, *individual_parameters: dict*, *feature: str*) → Tensor

For one individual, compute age(s) at which the given features values are reached (given the subject's individual parameters), with tensorized inputs.

Parameters

value
[`torch.Tensor` of shape `(1, n_values)`] Contains the *biomarker* value(s) of the subject.

individual_parameters
[DictParamsTorch] Contains the individual parameters. Each individual parameter should be a `torch.Tensor`.

feature
[`str` (or None)] Name of the considered *biomarker*.

Note: Optional for `UnivariateModel`, compulsory for `MultivariateModel`.

Returns

`torch.Tensor`
Contains the subject's ages computed at the given values(s). Shape of tensor is `(n_values, 1)`.

compute_individual_attachment_tensorized(*data: Dataset*, *param_ind: Dict[str, Tensor]*, *, *attribute_type: str | None = None*) → Tensor

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
[DictParamsTorch] Contain the individual parameters.

attribute_type
[`str` or None] Flag to ask for *MCMC* attributes instead of model's attributes.

Returns

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

compute_individual_tensorized(*timepoints: Tensor*, *individual_parameters: dict*, *, *attribute_type=None*) → Tensor

Compute the individual trajectories.

Parameters

timepoints
[torch.Tensor] The time points for which to compute the trajectory.

individual_parameters
[DictParamsTorch] The individual parameters to use.

attribute_type
[Any, optional]

Returns

torch.Tensor
Individual trajectories.

compute_individual_tensorized_linear(*timepoints: Tensor*, *individual_parameters: dict*, *, *attribute_type=None*) → Tensor

Compute the individual trajectories.

Parameters

timepoints
[torch.Tensor] The time points for which to compute the trajectory.

individual_parameters
[DictParamsTorch] The individual parameters to use.

attribute_type
[Any, optional]

Returns

torch.Tensor
Individual trajectories.

compute_individual_tensorized_logistic(*timepoints: Tensor*, *individual_parameters: dict*, *, *attribute_type=None*) → Tensor

Compute the individual trajectories.

Parameters

timepoints
[torch.Tensor] The time points for which to compute the trajectory.

individual_parameters
[DictParamsTorch] The individual parameters to use.

attribute_type
[Any, optional]

Returns

torch.Tensor
Individual trajectories.

compute_individual_trajectory(*timepoints*, *individual_parameters: Dict[str, Any]*, ***, *skip_ips_checks: bool = False*) → Tensor

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)`.

Raises

LeaspyModelInputError
If computation is tried on more than 1 individual.

LeaspyIndividualParamsInputError
if invalid individual parameters.

compute_jacobian_tensorized(*timepoints: Tensor*, *individual_parameters: dict*, ***, *attribute_type=None*) → Dict[str, Tensor]

Compute the jacobian of the model w.r.t. each individual parameter.

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

Note: As most of numerical operations are repeated when computing model & jacobian, we should create a single method that is able to compute model & jacobian "together" (= efficiently) when requested with a flag for instance.

Parameters

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

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

attribute_type
[`str` or None] Flag to ask for *MCMC* attributes instead of model's attributes.

Returns

`dict` **[param_name:** `str`**,** `torch.Tensor` **]**
Tensors are of shape `(n_individuals, n_timepoints, n_features, n_dims_param)`.

compute_jacobian_tensorized_linear(*timepoints: Tensor*, *individual_parameters: dict*, *, *attribute_type=None*) → Dict[str, Tensor]

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.

Note: As most of numerical operations are repeated when computing model & jacobian, we should create a single method that is able to compute model & jacobian "together" (= efficiently) when requested with a flag for instance.

Parameters

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

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

attribute_type
[`str` or None] Flag to ask for *MCMC* attributes instead of model's attributes.

Returns

`dict` **[param_name:** `str`**,** `torch.Tensor` **]**
Tensors are of shape `(n_individuals, n_timepoints, n_features, n_dims_param)`.

compute_jacobian_tensorized_logistic(*timepoints: Tensor*, *individual_parameters: dict*, *, *attribute_type=None*) → Dict[str, Tensor]

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.

Note: As most of numerical operations are repeated when computing model & jacobian, we should create a single method that is able to compute model & jacobian "together" (= efficiently) when requested with a flag for instance.

Parameters

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

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

attribute_type
[`str` or None] Flag to ask for *MCMC* attributes instead of model's attributes.

Returns

`dict` **[param_name:** `str`**,** `torch.Tensor` **]**
Tensors are of shape `(n_individuals, n_timepoints, n_features, n_dims_param)`.

compute_mean_traj(*timepoints: Tensor*, *, *attribute_type: str | None = None*) → Tensor

Compute trajectory of the model with individual parameters being the group-average ones.

TODO check dimensions of io?

Parameters

timepoints
[torch.Tensor of shape (1, n_timepoints)]

attribute_type
[str or None] If a string, should be "MCMC".

Returns

torch.Tensor of shape (1, n_timepoints, dimension)
The group-average values at given timepoints.

compute_model_sufficient_statistics(*data: Dataset*, *realizations: CollectionRealization*) → Dict[str, Tensor]

Compute the model's *sufficient statistics*.

Parameters

data
[Dataset] The input dataset.

realizations
[CollectionRealization] The realizations from which to compute the model's *sufficient statistics*.

Returns

DictParamsTorch
The computed *sufficient statistics*.

compute_ordinal_model_sufficient_statistics(*realizations: CollectionRealization*) → Dict[str, Tensor]

Compute the sufficient statistics given realizations.

static compute_ordinal_pdf_from_ordinal_sf(*ordinal_sf: Tensor*, *dim_ordinal_levels: int = 3*) → Tensor

Computes the probability density (or its jacobian) of an ordinal model [P(X = l), l=0..L] from *ordinal_sf* which are the survival function probabilities [P(X > l), i.e. P(X >= l+1), l=0..L-1] (or its jacobian).

Parameters

ordinal_sf
[*torch.FloatTensor*] Survival function values : ordinal_sf[..., l] is the proba to be superior or equal to l+1 Dimensions are: * 0=individual * 1=visit * 2=feature * 3=ordinal_level [l=0..L-1] * [4=individual_parameter_dim_when_gradient]

dim_ordinal_levels
[int, default = 3] The dimension of the tensor where the ordinal levels are.

Returns

ordinal_pdf
[*torch.FloatTensor* (same shape as input, except for dimension 3 which has one more element)] ordinal_pdf[..., l] is the proba to be equal to l (l=0..L)

compute_regularity_individual_parameters(*individual_parameters: Dict[str, Tensor]*, *, *include_constant: bool = False*) → Tuple[Dict[str, Tensor], Dict[str, Tensor]]

Compute the regularity terms (and their gradients if requested), per individual variable of the model.

Parameters

individual_parameters
[dict [str, torch.Tensor]] Individual parameters as a dict of tensors of shape (n_ind, n_dims_param).

include_constant
[bool, optional] Whether to include a constant term or not. Default=False.

Returns

regularity
[dict [param_name: str, torch.Tensor]] Regularity of the patient(s) corresponding to the given individual parameters. Tensors have shape (n_individuals).

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

compute_regularity_individual_realization(*realization: IndividualRealization*) → Tensor

Compute regularity term for IndividualRealization.

Parameters

realization
[IndividualRealization]

Returns

torch.Tensor of the same shape as IndividualRealization.tensor

compute_regularity_population_realization(*realization: PopulationRealization*) → Tensor

Compute regularity term for PopulationRealization.

Parameters

realization
[PopulationRealization]

Returns

torch.Tensor of the same shape as PopulationRealization.tensor

compute_regularity_realization(*realization: AbstractRealization*) → Tensor

Compute regularity term for a AbstractRealization instance.

Parameters

realization
[AbstractRealization]

Returns

torch.Tensor of the same shape as AbstractRealization.tensor

compute_regularity_variable(*value: Tensor*, *mean: Tensor*, *std: Tensor*, *, *include_constant: bool = True*, *with_gradient: bool = False*) → Tensor | Tuple[Tensor, Tensor]

Compute regularity term (Gaussian distribution) and optionally its gradient wrt value.

Note: TODO: should be encapsulated in a `RandomVariableSpecification` class together with other specs of RV.

Parameters

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

include_constant
[`bool` (default `True`)] Whether we include or not additional terms constant with respect to *value*.

with_gradient
[`bool` (default `False`)] Whether we also return the gradient of *regularity* term with respect to *value*.

Returns

`torch.Tensor` of same shape than input

`compute_sufficient_statistics`(*data: Dataset*, *realizations: CollectionRealization*) → Dict[str, Tensor]

Compute sufficient statistics from realizations.

Parameters

data
[`Dataset`]

realizations
[`CollectionRealization`]

Returns

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

property dimension: `int` | `None`

The dimension of the model. If the private attribute is defined, then it takes precedence over the feature length. The associated setters are responsible for their coherence.

`get_additional_ordinal_population_random_variable_information`() → Dict[str, Any]

Return the information of additional population random variables for the ordinal model.

`get_individual_random_variable_information`() → Dict[str, Any]

Return the information on individual random variables relative to the model.

Returns

DictParams
The information on the individual random variables.

`get_individual_variable_names`() → List[str]

Get the names of the individual variables of the model.

Returns

`list` of `str`

get_ordinal_parameters_updates_from_sufficient_statistics(*sufficient_statistics: Dict[str, Tensor]*) → Dict[str, Tensor]

Return a dictionary computed from provided sufficient statistics for updating the parameters.

get_population_random_variable_information() → Dict[str, Any]

Return the information on population random variables relative to the model.

Returns

DictParams
The information on the population random variables.

get_population_variable_names() → List[str]

Get the names of the population variables of the model.

Returns

list of str

initialize(*dataset: Dataset*, *method: str = 'default'*) → None

Overloads base initialization of model (base method takes care of features consistency checks).

Parameters

dataset
[Dataset] Input Dataset from which to initialize the model.

method
[str, optional] The initialization method to be used. Default='default'.

initialize_MCMC_toolbox() → None

Initialize the model's *MCMC* toolbox attribute.

property is_ordinal: bool

Property to check if the model is of ordinal sub-type.

property is_ordinal_ranking: bool

Property to check if the model is of ordinal-ranking sub-type (working with survival functions).

load_hyperparameters(*hyperparameters: Dict[str, Any]*) → None

Updates all model hyperparameters from the provided hyperparameters.

Parameters

hyperparameters
[KwargsType] The hyperparameters to be loaded.

load_parameters(*parameters: Dict[str, Any]*) → None

Updates all model parameters from the provided parameters.

Parameters

parameters
[KwargsType] The parameters to be loaded.

move_to_device(*device: device*) → None

Move a model and its relevant attributes to the specified torch.device.

Parameters

device
[torch.device]

property ordinal_infos: dict | None

Property to return the ordinal info dictionary.

postprocess_model_estimation(*estimation: ndarray*, *, *ordinal_method: str = 'MLE'*, ***kws*) → ndarray | Dict[Hashable, ndarray]

Extra layer of processing used to output nice estimated values in main API *Leaspy.estimate*.

Parameters

estimation
[numpy.ndarray[float]] The raw estimated values by model (from *compute_individual_trajectory*)

ordinal_method
[str] <!> Only used for ordinal models. * 'MLE' or 'maximum_likelihood' returns maximum likelihood estimator for each point (int) * 'E' or 'expectation' returns expectation (float) * 'P' or 'probabilities' returns probabilities of all-possible levels for a given feature:

{feature_name: array[float]<0..max_level_ft>}

****kws**
Some extra keywords arguments that may be handled in the future.

Returns

numpy.ndarray[float] or dict[str, numpy.ndarray[float]]
Post-processed values. In case using 'probabilities' mode, the values are a dictionary with keys being: *(feature_name: str, feature_level: int<0..max_level_for_feature>)* Otherwise it is a standard numpy.ndarray corresponding to different model features (in order)

save(*path: str*, ***kwargs*) → None

Save `Leaspy` object as json model parameter file.

TODO move logic upstream?

Parameters

path
[`str`] Path to store the model's parameters.

****kwargs**
Keyword arguments for `AbstractModel.to_dict()` child method and `json.dump` function (default to indent=2).

static time_reparametrization(*timepoints: Tensor*, *xi: Tensor*, *tau: Tensor*) → Tensor

Tensorized time reparametrization formula.

Warning: 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*

to_dict(*, *with_mixing_matrix: bool = True*) → Dict[str, Any]

Export Leaspy object as dictionary ready for *JSON* saving.

Parameters

with_mixing_matrix
[bool (default True)] Save the *mixing matrix* in the exported file in its 'parameters' section.

> **Warning:** 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…

Returns

KwargsType
The object as a dictionary.

update_MCMC_toolbox(*vars_to_update: set*, *realizations: CollectionRealization*) → None

Update the model's *MCMC* toolbox attribute with the provided vars_to_update.

Parameters

vars_to_update
[set of str] The set of variable names to be updated.

realizations
[CollectionRealization] The realizations to use for updating the *MCMC* toolbox.

update_model_parameters_burn_in(*data: Dataset*, *sufficient_statistics: Dict[str, Tensor]*) → None

Update the model's parameters during the burn in phase.

During the burn-in phase, we only need to store the following parameters (cf. !66 and #60)

- noise_std
- *_mean/std for *regularization* of individual variables
- others population parameters for *regularization* of population variables

We don't need to update the model "attributes" (never used during burn-in!).

Parameters

data
[Dataset] The input dataset.

sufficient_statistics
[DictParamsTorch] The *sufficient statistics* to use for parameter update.

update_model_parameters_normal(*data: Dataset*, *sufficient_statistics: Dict[str, Tensor]*) → None

Stochastic *sufficient statistics* used to update the parameters of the model.

Note:

TODOs:

- factorize `update_model_parameters_***` methods?
- add a true, configurable, validation for all parameters? (e.g.: bounds on tau_var/std but also on tau_mean, ...)
- check the SS, especially the issue with mean(xi) and v_k
- Learn the mean of xi and v_k
- Set the mean of xi to 0 and add it to the mean of V_k

Parameters

data
[`Dataset`] The input dataset.

sufficient_statistics
[DictParamsTorch] The *sufficient statistics* to use for parameter update.

update_ordinal_population_random_variable_information(*variables_info: Dict[str, Any]*) → None

Update (in-place) the provided variable information dictionary.

Nota: this is needed due to different signification of *v0* in ordinal model (common per-level velocity)

Parameters

variables_info
[DictParams] The variables information to be updated with ordinal logic.

update_parameters_burn_in(*data: Dataset*, *sufficient_statistics: Dict[str, Tensor]*) → None

Update model parameters (burn-in phase).

Parameters

data
[`Dataset`]

sufficient_statistics
[`dict` [suff_stat: `str`, `torch.Tensor`]]

update_parameters_normal(*data: Dataset*, *sufficient_statistics: Dict[str, Tensor]*) → None

Update model parameters (after burn-in phase).

Parameters

data
[`Dataset`]

sufficient_statistics
[`dict` [suff_stat: `str`, `torch.Tensor`]]

validate_compatibility_of_dataset(*dataset: Dataset*) → None

Raise if the given `Dataset` is not compatible with the current model.

Parameters

dataset
[Dataset] The Dataset we want to model.

Raises

LeaspyModelInputError

- If the Dataset has a number of dimensions smaller than 2.
- If the Dataset does not have the same dimensionality as the model.
- If the Dataset's headers do not match the model's.

3.2.11 leaspy.models.noise_models: Noise Models

Available noise models in *Leaspy*.

DistributionFamily([parameters])	Base class for a distribution family being able to sample "around" user-provided values.
AbstractGaussianNoiseModel([parameters, ...])	Base class for Gaussian noise models.
AbstractOrdinalNoiseModel([parameters, ...])	Base class for Ordinal noise models.
BaseNoiseModel([parameters])	Base class for valid noise models that may be used in probabilistic models.
BernoulliNoiseModel([parameters])	Class implementing Bernoulli noise models.
GaussianDiagonalNoiseModel([parameters, ...])	Class implementing diagonal Gaussian noise models.
GaussianScalarNoiseModel([parameters, ...])	Class implementing scalar Gaussian noise models.
OrdinalNoiseModel([parameters, max_levels])	Class implementing ordinal noise models (likelihood is based on PDF).
OrdinalRankingNoiseModel([parameters, ...])	Class implementing OrdinalRankingNoiseModel (likelihood is based on SF).

leaspy.models.noise_models.DistributionFamily

class DistributionFamily(*parameters: Dict[str, Tensor] | None = None*)

Bases: object

Base class for a distribution family being able to sample "around" user-provided values.

Parameters

parameters
[dict [str, torch.Tensor] or None] Values for all the free parameters of the distribution family. All of them must have values before using the sampling methods.

Attributes

free_parameters
[frozenset of str] Name of all the free parameters (but *loc*) needed to characterize the distribution. Nota: for each parameter, if a method named "validate_xxx" exists (torch.Tensor -> torch.Tensor), then it will be used for user-input validation of parameter "xxx".

factory
[None or function(free parameters values) -> torch.distributions.distribution.Distribution] The factory for the distribution family.

parameters
[dict [str, torch.Tensor] or None] Values for all the free parameters of the distribution family. All of them must have values before using the sampling methods.

Methods

move_to_device(device)	Move all torch tensors stored in this instance to the provided device (parameters & hyperparameters).
raise_if_partially_defined()	Raise an error if some of the free parameters are not defined.
raise_if_unknown_parameters(params)	Raise an error if the provided parameters are not part of the free parameters.
rv_around(loc)	Return the torch distribution centred around values (only if noise is not None).
sample_around(loc)	Realization around *loc* with respect to partially defined distribution.
sampler_around(loc)	Return the sampling function around input values.
to_dict()	Serialize instance as dictionary.
update_parameters(*[, validate])	(Partial) update of the free parameters of the distribution family.
validate(**params)	Validation function for parameters (based on 'validate_xxx' methods).

move_to_device(*device: device*) → None

Move all torch tensors stored in this instance to the provided device (parameters & hyperparameters).

Parameters

device
[torch.device] Torch device on which to move the tensors.

raise_if_partially_defined() → None

Raise an error if some of the free parameters are not defined.

classmethod raise_if_unknown_parameters(*params: Iterable | None*) → None

Raise an error if the provided parameters are not part of the free parameters.

Parameters

params
[Iterable, optional] The list of parameters to analyze.

rv_around(*loc: Tensor*) → Distribution

Return the torch distribution centred around values (only if noise is not None).

Parameters

loc
[torch.Tensor] The loc around which to sample.

Returns

torch.distributions.distribution.Distribution
The torch distribution centered around the loc.

`sample_around`(*loc: Tensor*) → Tensor

Realization around *loc* with respect to partially defined distribution.

Parameters

loc
[`torch.Tensor`] The loc around which to sample.

Returns

`torch.Tensor`
The requested sample.

`sampler_around`(*loc: Tensor*) → Callable[[], Tensor]

Return the sampling function around input values.

Parameters

loc
[`torch.Tensor`] The loc around which to sample.

Returns

Callable
The sampler.

`to_dict`() → Dict[str, Any]

Serialize instance as dictionary.

Returns

`KwargsType`
The instance serialized as a dictionary.

`update_parameters`(*, *validate: bool = False*, ***parameters: Tensor*) → None

(Partial) update of the free parameters of the distribution family.

Parameters

validate
[`bool`, optional] If True, the provided parameters are validated before being updated. Default=False.

****parameters**
[`torch.Tensor`] The new parameters.

`validate`(***params: Any*) → Dict[str, Tensor]

Validation function for parameters (based on 'validate_xxx' methods).

Parameters

****params**
[Any] The parameters to validate.

Returns

`DictParamsTorch`
The validated parameters.

leaspy.models.noise_models.AbstractGaussianNoiseModel

class AbstractGaussianNoiseModel(*parameters: Dict[str, Tensor] | None = None*, *scale_dimension: int | None = None*)

Bases: GaussianFamily, BaseNoiseModel

Base class for Gaussian noise models.

Parameters

parameters
[dict [str, torch.Tensor] or None] Values for all the free parameters of the distribution family. All of them must have values before using the sampling methods.

scale_dimension
[int, optional] The scale dimension.

Attributes

scale_dimension
[int, optional] The scale dimension.

parameters
[dict] Contains the parameters relative to the noise model.

canonical_loss_properties
[tuple] The properties for the canonical loss.

Methods

compute_canonical_loss(data, predictions)	Compute a human-friendly overall loss (RMSE).
compute_l2_residuals(data, predictions)	Compute the squared residuals of the given predictions.
compute_nll(data, predictions, *[, ...])	Negative log-likelihood without summation (and its gradient w.r.t.
compute_residuals(data, predictions)	Compute the residuals of the given predictions.
compute_rmse(data, predictions)	Computes root mean squared error of provided data vs.
compute_sufficient_statistics(data, predictions)	Compute the specific sufficient statistics and metrics for this noise-model.
factory	alias of Normal
move_to_device(device)	Move all torch tensors stored in this instance to the provided device (parameters & hyperparameters).
raise_if_partially_defined()	Raise an error if some of the free parameters are not defined.
raise_if_unknown_parameters(params)	Raise an error if the provided parameters are not part of the free parameters.
rv_around(loc)	Return the torch distribution centred around values (only if noise is not None).
sample_around(loc)	Realization around *loc* with respect to partially defined distribution.
sampler_around(loc)	Return the sampling function around input values.
to_dict()	Serialize instance as dictionary.
update_parameters(*[, validate])	(Partial) update of the free parameters of the distribution family.
update_parameters_from_predictions(data, ...)	In-place update of free parameters from provided predictions.
update_parameters_from_sufficient_statist	In-place update of free parameters from provided sufficient statistics.
validate(**params)	Validation function for parameters (based on 'validate_xxx' methods).
validate_scale(scale)	Add a size-validation for scale parameter.

classmethod compute_canonical_loss(*data: Dataset*, *predictions: Tensor*) → Tensor

Compute a human-friendly overall loss (RMSE).

Parameters

data
[Dataset] The dataset related to the computation of the log likelihood.

predictions
[torch.Tensor] The model's predictions from which to compute the canonical loss.

Returns

torch.Tensor
The computed loss.

classmethod compute_l2_residuals(*data: Dataset*, *predictions: Tensor*) → Tensor

Compute the squared residuals of the given predictions.

Parameters

data
[Dataset] The dataset related to the computation of the squared residuals.

predictions
[torch.Tensor] The model's predictions from which to compute the squared residuals.

Returns

torch.Tensor
The squared residuals.

compute_nll(*data: Dataset*, *predictions: Tensor*, *, *with_gradient: bool = False*) → Tensor | Tuple[Tensor, Tensor]

Negative log-likelihood without summation (and its gradient w.r.t. predictions if requested).

Parameters

data
[Dataset] The dataset related to the computation of the log likelihood.

predictions
[torch.Tensor] The model's predictions from which to compute the log likelihood.

with_gradient
[bool, optional] If True, returns also the gradient of the negative log likelihood wrt the predictions. If False, only returns the negative log likelihood. Default=False.

Returns

torch.Tensor or tuple of torch.Tensor
The negative log likelihood (and its jacobian if requested).

static compute_residuals(*data: Dataset*, *predictions: Tensor*) → Tensor

Compute the residuals of the given predictions.

Parameters

data
[Dataset] The dataset related to the computation of the residuals.

predictions
[torch.Tensor] The model's predictions from which to compute the residuals.

Returns

torch.Tensor
The residuals.

classmethod compute_rmse(*data: Dataset*, *predictions: Tensor*) → Tensor

Computes root mean squared error of provided data vs. predictions.

Parameters

data
[Dataset] The dataset related to the computation of the root mean squared error.

predictions
[torch.Tensor] The model's predictions from which to compute the root mean squared error.

Returns

torch.Tensor
The root mean squared error.

compute_sufficient_statistics(*data: Dataset*, *predictions: Tensor*) → Dict[str, Tensor]

Compute the specific sufficient statistics and metrics for this noise-model.

Parameters

data
[Dataset] The dataset related to the computation of the sufficient statistics.

predictions
[torch.Tensor] The model's predictions from which to compute the sufficient statistics.

Returns

DictParamsTorch
The sufficient statistics.

factory
alias of Normal

move_to_device(*device: device*) → None

Move all torch tensors stored in this instance to the provided device (parameters & hyperparameters).

Parameters

device
[torch.device] Torch device on which to move the tensors.

raise_if_partially_defined() → None

Raise an error if some of the free parameters are not defined.

classmethod raise_if_unknown_parameters(*params: Iterable | None*) → None

Raise an error if the provided parameters are not part of the free parameters.

Parameters

params
[Iterable, optional] The list of parameters to analyze.

rv_around(*loc: Tensor*) → Distribution

Return the torch distribution centred around values (only if noise is not None).

Parameters

loc
[torch.Tensor] The loc around which to sample.

Returns

torch.distributions.distribution.Distribution
The torch distribution centered around the loc.

sample_around(*loc: Tensor*) → Tensor

Realization around *loc* with respect to partially defined distribution.

Parameters

loc
[torch.Tensor] The loc around which to sample.

Returns

torch.Tensor
The requested sample.

sampler_around(*loc: Tensor*) → Callable[[], Tensor]

Return the sampling function around input values.

Parameters

loc
[torch.Tensor] The loc around which to sample.

Returns

Callable
The sampler.

to_dict() → Dict[str, Any]

Serialize instance as dictionary.

Returns

KwargsType
The instance serialized as a dictionary.

update_parameters(*, *validate: bool = False*, ***parameters: Tensor*) → None

(Partial) update of the free parameters of the distribution family.

Parameters

validate
[bool, optional] If True, the provided parameters are validated before being updated. Default=False.

****parameters**
[torch.Tensor] The new parameters.

update_parameters_from_predictions(*data: Dataset*, *predictions: Tensor*) → None

In-place update of free parameters from provided predictions.

Parameters

data
[Dataset] The dataset related to the computation of the log likelihood.

predictions
[torch.Tensor] The model's predictions from which to update the parameters.

update_parameters_from_sufficient_statistics(*data: Dataset*, *sufficient_statistics: Dict[str, Tensor]*) → None

In-place update of free parameters from provided sufficient statistics.

Parameters

data
[Dataset] The dataset related to the computation of the log likelihood.

sufficient_statistics
[DictParamsTorch] The sufficient statistics to use for parameter update.

validate(***params: Any*) → Dict[str, Tensor]

Validation function for parameters (based on 'validate_xxx' methods).

Parameters

****params**
[Any] The parameters to validate.

Returns

DictParamsTorch
The validated parameters.

`validate_scale`(*scale: Tensor*) → Tensor

Add a size-validation for scale parameter.

Parameters

scale
[`torch.Tensor`] The scale to validate.

Returns

`torch.Tensor`
The validated scale.

`leaspy.models.noise_models.AbstractOrdinalNoiseModel`

`class AbstractOrdinalNoiseModel`(*parameters: DictParamsTorch | None = None*, *max_levels: Dict[FeatureType, int] | None = None*)

Bases: `BaseNoiseModel`

Base class for Ordinal noise models.

Parameters

parameters
[`dict` [`str`, `torch.Tensor`] or None] Values for all the free parameters of the distribution family. All of them must have values before using the sampling methods.

max_levels
[`dict`, optional] Maximum levels for ordinal noise.

Attributes

max_levels
[`dict`, optional] Maximum levels for ordinal noise.

Methods

compute_canonical_loss(data, predictions)	Compute a human-friendly overall loss (independent from instance parameters), useful as a measure of goodness-of-fit after personalization (nll by default - assuming no free parameters).
compute_nll(data, predictions, *[, ...])	Compute negative log-likelihood of data given model predictions (no summation), and its gradient w.r.t.
compute_sufficient_statistics(data, predictions)	Computes the set of noise-related sufficient statistics and metrics (to be extended in child class).
move_to_device(device)	Move all torch tensors stored in this instance to the provided device (parameters & hyperparameters).
raise_if_partially_defined()	Raise an error if some of the free parameters are not defined.
raise_if_unknown_parameters(params)	Raise an error if the provided parameters are not part of the free parameters.
rv_around(loc)	Return the torch distribution centred around values (only if noise is not None).
sample_around(loc)	Realization around *loc* with respect to partially defined distribution.
sampler_around(loc)	Return the sampling function around input values.
to_dict()	Serialize instance as dictionary.
update_parameters(*[, validate])	(Partial) update of the free parameters of the distribution family.
update_parameters_from_predictions(data, ...)	Updates noise-model parameters in-place (nothing done by default).
update_parameters_from_sufficient_statist	Updates noise-model parameters in-place (nothing done by default).
validate(**params)	Validation function for parameters (based on 'validate_xxx' methods).

compute_canonical_loss(*data: Dataset*, *predictions: Tensor*) → Tensor

Compute a human-friendly overall loss (independent from instance parameters), useful as a measure of goodness-of-fit after personalization (nll by default - assuming no free parameters).

Parameters

data
[Dataset] The dataset related to the computation of the log likelihood.

predictions
[torch.Tensor] The model's predictions from which to compute the canonical loss.

Returns

torch.Tensor
The computed loss.

abstract compute_nll(*data: Dataset*, *predictions: Tensor*, *, *with_gradient: bool = False*) → Tensor | Tuple[Tensor, Tensor]

Compute negative log-likelihood of data given model predictions (no summation), and its gradient w.r.t. predictions if requested.

Parameters

data
[Dataset] The dataset related to the computation of the log likelihood.

predictions
[torch.Tensor] The model's predictions from which to compute the log likelihood.

with_gradient
[bool, optional] If True, returns also the gradient of the negative log likelihood wrt the predictions. If False, only returns the negative log likelihood. Default=False.

Returns

torch.Tensor or tuple of torch.Tensor
The negative log likelihood (and its jacobian if requested).

compute_sufficient_statistics(*data: Dataset*, *predictions: Tensor*) → Dict[str, Tensor]

Computes the set of noise-related sufficient statistics and metrics (to be extended in child class).

Parameters

data
[Dataset] The dataset related to the computation of the sufficient statistics.

predictions
[torch.Tensor] The model's predictions from which to compute the sufficient statistics.

Returns

DictParamsTorch
The sufficient statistics.

move_to_device(*device: device*) → None

Move all torch tensors stored in this instance to the provided device (parameters & hyperparameters).

Parameters

device
[torch.device] Torch device on which to move the tensors.

raise_if_partially_defined() → None

Raise an error if some of the free parameters are not defined.

classmethod raise_if_unknown_parameters(*params: Iterable | None*) → None

Raise an error if the provided parameters are not part of the free parameters.

Parameters

params
[Iterable, optional] The list of parameters to analyze.

rv_around(*loc: Tensor*) → Distribution

Return the torch distribution centred around values (only if noise is not None).

Parameters

loc
[torch.Tensor] The loc around which to sample.

Returns

torch.distributions.distribution.Distribution
The torch distribution centered around the loc.

sample_around(*loc: Tensor*) → Tensor

Realization around *loc* with respect to partially defined distribution.

Parameters

loc
[torch.Tensor] The loc around which to sample.

Returns

torch.Tensor
The requested sample.

sampler_around(*loc: Tensor*) → Callable[[], Tensor]

Return the sampling function around input values.

Parameters

loc
[torch.Tensor] The loc around which to sample.

Returns

Callable
The sampler.

to_dict() → Dict[str, Any]

Serialize instance as dictionary.

> **Warning:** Do NOT export hyper-parameters that are derived (error-prone and boring checks when re-creating).

Returns

KwargsType
The instance serialized as a dictionary.

update_parameters(*, *validate: bool = False*, ***parameters: Tensor*) → None

(Partial) update of the free parameters of the distribution family.

Parameters

validate
[bool, optional] If True, the provided parameters are validated before being updated. Default=False.

****parameters**
[torch.Tensor] The new parameters.

update_parameters_from_predictions(*data: Dataset*, *predictions: Tensor*) → None

Updates noise-model parameters in-place (nothing done by default).

Parameters

data
[Dataset] The dataset related to the computation of the log likelihood.

predictions
[torch.Tensor] The model's predictions from which to update the parameters.

update_parameters_from_sufficient_statistics(*data: Dataset*, *sufficient_statistics: Dict[str, Tensor]*) → None

Updates noise-model parameters in-place (nothing done by default).

Parameters

data
[Dataset] The dataset related to the computation of the log likelihood.

sufficient_statistics
[DictParamsTorch] The sufficient statistics to use for parameter update.

validate(***params: Any*) → Dict[str, Tensor]

Validation function for parameters (based on 'validate_xxx' methods).

Parameters

****params**
[Any] The parameters to validate.

Returns

DictParamsTorch
The validated parameters.

leaspy.models.noise_models.BaseNoiseModel

class BaseNoiseModel(*parameters: Dict[str, Tensor] | None = None*)

Bases: ABC, DistributionFamily

Base class for valid noise models that may be used in probabilistic models.

The negative log-likelihood (nll, to be minimized) always corresponds to attachment term in models.

Parameters

parameters
[dict [str, torch.Tensor] or None] Values for all the free parameters of the distribution family. All of them must have values before using the sampling methods.

Attributes

parameters
[dict [str, torch.Tensor] or None] All values for the free parameters of the distribution family. None is to be used if and only if there are no *free_parameters* at all.

canonical_loss_properties
[tuple [str, str]] Tuple of strings characterizing the canonical loss of the noise model.

Methods

compute_canonical_loss(data, predictions)	Compute a human-friendly overall loss (independent from instance parameters), useful as a measure of goodness-of-fit after personalization (nll by default - assuming no free parameters).
compute_nll(data, predictions, *[, ...])	Compute negative log-likelihood of data given model predictions (no summation), and its gradient w.r.t.
compute_sufficient_statistics(data, predictions)	Computes the set of noise-related sufficient statistics and metrics (to be extended in child class).
move_to_device(device)	Move all torch tensors stored in this instance to the provided device (parameters & hyperparameters).
raise_if_partially_defined()	Raise an error if some of the free parameters are not defined.
raise_if_unknown_parameters(params)	Raise an error if the provided parameters are not part of the free parameters.
rv_around(loc)	Return the torch distribution centred around values (only if noise is not None).
sample_around(loc)	Realization around *loc* with respect to partially defined distribution.
sampler_around(loc)	Return the sampling function around input values.
to_dict()	Serialize instance as dictionary.
update_parameters(*[, validate])	(Partial) update of the free parameters of the distribution family.
update_parameters_from_predictions(data, ...)	Updates noise-model parameters in-place (nothing done by default).
update_parameters_from_sufficient_statist	Updates noise-model parameters in-place (nothing done by default).
validate(**params)	Validation function for parameters (based on 'validate_xxx' methods).

compute_canonical_loss(*data: Dataset*, *predictions: Tensor*) → Tensor

Compute a human-friendly overall loss (independent from instance parameters), useful as a measure of goodness-of-fit after personalization (nll by default - assuming no free parameters).

Parameters

data
[Dataset] The dataset related to the computation of the log likelihood.

predictions
[torch.Tensor] The model's predictions from which to compute the canonical loss.

Returns

torch.Tensor
The computed loss.

abstract compute_nll(*data: Dataset*, *predictions: Tensor*, *, *with_gradient: bool = False*) → Tensor | Tuple[Tensor, Tensor]

Compute negative log-likelihood of data given model predictions (no summation), and its gradient w.r.t. predictions if requested.

Parameters

data
[Dataset] The dataset related to the computation of the log likelihood.

predictions
[torch.Tensor] The model's predictions from which to compute the log likelihood.

with_gradient
[bool, optional] If True, returns also the gradient of the negative log likelihood wrt the predictions. If False, only returns the negative log likelihood. Default=False.

Returns

torch.Tensor or tuple of torch.Tensor
The negative log likelihood (and its jacobian if requested).

compute_sufficient_statistics(*data: Dataset*, *predictions: Tensor*) → Dict[str, Tensor]

Computes the set of noise-related sufficient statistics and metrics (to be extended in child class).

Parameters

data
[Dataset] The dataset related to the computation of the sufficient statistics.

predictions
[torch.Tensor] The model's predictions from which to compute the sufficient statistics.

Returns

DictParamsTorch
The sufficient statistics.

move_to_device(*device: device*) → None

Move all torch tensors stored in this instance to the provided device (parameters & hyperparameters).

Parameters

device
[torch.device] Torch device on which to move the tensors.

raise_if_partially_defined() → None

Raise an error if some of the free parameters are not defined.

classmethod raise_if_unknown_parameters(*params: Iterable | None*) → None

Raise an error if the provided parameters are not part of the free parameters.

Parameters

params
[Iterable, optional] The list of parameters to analyze.

rv_around(*loc: Tensor*) → Distribution

Return the torch distribution centred around values (only if noise is not None).

Parameters

loc
[torch.Tensor] The loc around which to sample.

Returns

torch.distributions.distribution.Distribution
The torch distribution centered around the loc.

sample_around(*loc: Tensor*) → Tensor

Realization around *loc* with respect to partially defined distribution.

Parameters

loc
[torch.Tensor] The loc around which to sample.

Returns

torch.Tensor
The requested sample.

sampler_around(*loc: Tensor*) → Callable[[], Tensor]

Return the sampling function around input values.

Parameters

loc
[torch.Tensor] The loc around which to sample.

Returns

Callable
The sampler.

to_dict() → Dict[str, Any]

Serialize instance as dictionary.

Returns

KwargsType
The instance serialized as a dictionary.

update_parameters(*, *validate: bool = False*, ***parameters: Tensor*) → None

(Partial) update of the free parameters of the distribution family.

Parameters

validate
[bool, optional] If True, the provided parameters are validated before being updated. Default=False.

****parameters**
[torch.Tensor] The new parameters.

update_parameters_from_predictions(*data: Dataset*, *predictions: Tensor*) → None

Updates noise-model parameters in-place (nothing done by default).

Parameters

data
[Dataset] The dataset related to the computation of the log likelihood.

predictions
[torch.Tensor] The model's predictions from which to update the parameters.

update_parameters_from_sufficient_statistics(*data: Dataset*, *sufficient_statistics: Dict[str, Tensor]*) → None

Updates noise-model parameters in-place (nothing done by default).

Parameters

data
[Dataset] The dataset related to the computation of the log likelihood.

sufficient_statistics
[DictParamsTorch] The sufficient statistics to use for parameter update.

`validate`(***params: Any*) → Dict[str, Tensor]

Validation function for parameters (based on 'validate_xxx' methods).

Parameters

****params**
[Any] The parameters to validate.

Returns

`DictParamsTorch`
The validated parameters.

leaspy.models.noise_models.BernoulliNoiseModel

`class BernoulliNoiseModel`(*parameters: Dict[str, Tensor] | None = None*)

Bases: `BernoulliFamily`, `BaseNoiseModel`

Class implementing Bernoulli noise models.

Parameters

parameters
[dict [str, torch.Tensor] or None] Values for all the free parameters of the distribution family. All of them must have values before using the sampling methods.

Attributes

parameters

Methods

compute_canonical_loss(data, predictions)	Compute a human-friendly overall loss (independent from instance parameters), useful as a measure of goodness-of-fit after personalization (nll by default - assuming no free parameters).
compute_nll(data, predictions, *[, ...])	Compute the negative log-likelihood and its gradient wrt predictions.
compute_sufficient_statistics(data, predictions)	Computes the set of noise-related sufficient statistics and metrics (to be extended in child class).
factory	alias of Bernoulli
move_to_device(device)	Move all torch tensors stored in this instance to the provided device (parameters & hyperparameters).
raise_if_partially_defined()	Raise an error if some of the free parameters are not defined.
raise_if_unknown_parameters(params)	Raise an error if the provided parameters are not part of the free parameters.
rv_around(loc)	Return the torch distribution centred around values (only if noise is not None).
sample_around(loc)	Realization around *loc* with respect to partially defined distribution.
sampler_around(loc)	Return the sampling function around input values.
to_dict()	Serialize instance as dictionary.
update_parameters(*[, validate])	(Partial) update of the free parameters of the distribution family.
update_parameters_from_predictions(data, ...)	Updates noise-model parameters in-place (nothing done by default).
update_parameters_from_sufficient_statist	Updates noise-model parameters in-place (nothing done by default).
validate(**params)	Validation function for parameters (based on 'validate_xxx' methods).

compute_canonical_loss(*data: Dataset*, *predictions: Tensor*) → Tensor

Compute a human-friendly overall loss (independent from instance parameters), useful as a measure of goodness-of-fit after personalization (nll by default - assuming no free parameters).

Parameters

data
[Dataset] The dataset related to the computation of the log likelihood.

predictions
[torch.Tensor] The model's predictions from which to compute the canonical loss.

Returns

torch.Tensor
The computed loss.

compute_nll(*data: Dataset*, *predictions: Tensor*, *, *with_gradient: bool = False*) → Tensor | Tuple[Tensor, Tensor]

Compute the negative log-likelihood and its gradient wrt predictions.

Parameters

data
[Dataset] The dataset related to the computation of the log likelihood.

predictions
[torch.Tensor] The model's predictions from which to compute the log likelihood.

with_gradient
[bool, optional] If True, returns also the gradient of the negative log likelihood wrt the predictions. If False, only returns the negative log likelihood. Default=False.

Returns

torch.Tensor or tuple of torch.Tensor
The negative log likelihood (and its jacobian if requested).

compute_sufficient_statistics(*data: Dataset*, *predictions: Tensor*) → Dict[str, Tensor]

Computes the set of noise-related sufficient statistics and metrics (to be extended in child class).

Parameters

data
[Dataset] The dataset related to the computation of the sufficient statistics.

predictions
[torch.Tensor] The model's predictions from which to compute the sufficient statistics.

Returns

DictParamsTorch
The sufficient statistics.

factory

alias of Bernoulli

move_to_device(*device: device*) → None

Move all torch tensors stored in this instance to the provided device (parameters & hyperparameters).

Parameters

device
[torch.device] Torch device on which to move the tensors.

raise_if_partially_defined() → None

Raise an error if some of the free parameters are not defined.

classmethod raise_if_unknown_parameters(*params: Iterable | None*) → None

Raise an error if the provided parameters are not part of the free parameters.

Parameters

params
[Iterable, optional] The list of parameters to analyze.

rv_around(*loc: Tensor*) → Distribution

Return the torch distribution centred around values (only if noise is not None).

Parameters

loc
[torch.Tensor] The loc around which to sample.

Returns

torch.distributions.distribution.Distribution
The torch distribution centered around the loc.

sample_around(*loc: Tensor*) → Tensor

Realization around *loc* with respect to partially defined distribution.

Parameters

loc
[torch.Tensor] The loc around which to sample.

Returns

torch.Tensor
The requested sample.

sampler_around(*loc: Tensor*) → Callable[[], Tensor]

Return the sampling function around input values.

Parameters

loc
[torch.Tensor] The loc around which to sample.

Returns

Callable
The sampler.

to_dict() → Dict[str, Any]

Serialize instance as dictionary.

Returns

KwargsType
The instance serialized as a dictionary.

update_parameters(*, *validate: bool = False*, ***parameters: Tensor*) → None

(Partial) update of the free parameters of the distribution family.

Parameters

validate
[bool, optional] If True, the provided parameters are validated before being updated. Default=False.

****parameters**
[torch.Tensor] The new parameters.

update_parameters_from_predictions(*data: Dataset*, *predictions: Tensor*) → None

Updates noise-model parameters in-place (nothing done by default).

Parameters

data
[Dataset] The dataset related to the computation of the log likelihood.

predictions
[torch.Tensor] The model's predictions from which to update the parameters.

update_parameters_from_sufficient_statistics(*data: Dataset*, *sufficient_statistics: Dict[str, Tensor]*) → None

Updates noise-model parameters in-place (nothing done by default).

Parameters

data
[Dataset] The dataset related to the computation of the log likelihood.

sufficient_statistics
[DictParamsTorch] The sufficient statistics to use for parameter update.

`validate`(***params: Any*) → Dict[str, Tensor]

Validation function for parameters (based on 'validate_xxx' methods).

Parameters

****params**
[Any] The parameters to validate.

Returns

`DictParamsTorch`
The validated parameters.

leaspy.models.noise_models.GaussianDiagonalNoiseModel

`class GaussianDiagonalNoiseModel`(*parameters: Dict[str, Tensor] | None = None*, *scale_dimension: int | None = None*)

Bases: `AbstractGaussianNoiseModel`

Class implementing diagonal Gaussian noise models.

Parameters

parameters
[dict [str, torch.Tensor] or None] Values for all the free parameters of the distribution family. All of them must have values before using the sampling methods.

scale_dimension
[int, optional] The scale dimension.

Attributes

scale_dimension
[int, optional] The scale dimension.

parameters
[dict] Contains the parameters relative to the noise model.

Methods

compute_canonical_loss(data, predictions)	Compute a human-friendly overall loss (RMSE).
compute_l2_residuals(data, predictions)	Compute the squared residuals of the given predictions.
compute_nll(data, predictions, *[, ...])	Negative log-likelihood without summation (and its gradient w.r.t.
compute_residuals(data, predictions)	Compute the residuals of the given predictions.
compute_rmse(data, predictions)	Computes root mean squared error of provided data vs.
compute_sufficient_statistics(data, predictions)	Compute the specific sufficient statistics and metrics for this noise-model.
factory	alias of Normal
move_to_device(device)	Move all torch tensors stored in this instance to the provided device (parameters & hyperparameters).
raise_if_partially_defined()	Raise an error if some of the free parameters are not defined.
raise_if_unknown_parameters(params)	Raise an error if the provided parameters are not part of the free parameters.
rv_around(loc)	Return the torch distribution centred around values (only if noise is not None).
sample_around(loc)	Realization around *loc* with respect to partially defined distribution.
sampler_around(loc)	Return the sampling function around input values.
to_dict()	Serialize instance as dictionary.
update_parameters(*[, validate])	(Partial) update of the free parameters of the distribution family.
update_parameters_from_predictions(data, ...)	In-place update of free parameters from provided predictions.
update_parameters_from_sufficient_statist	In-place update of free parameters from provided sufficient statistics.
validate(**params)	Validation function for parameters (based on 'validate_xxx' methods).
validate_scale(scale)	Ensure the scale is valid.

classmethod compute_canonical_loss(*data: Dataset*, *predictions: Tensor*) → Tensor

Compute a human-friendly overall loss (RMSE).

Parameters

data
[Dataset] The dataset related to the computation of the log likelihood.

predictions
[torch.Tensor] The model's predictions from which to compute the canonical loss.

Returns

torch.Tensor
The computed loss.

classmethod compute_l2_residuals(*data: Dataset*, *predictions: Tensor*) → Tensor

Compute the squared residuals of the given predictions.

Parameters

data
[Dataset] The dataset related to the computation of the squared residuals.

predictions
[torch.Tensor] The model's predictions from which to compute the squared residuals.

Returns

torch.Tensor
The squared residuals.

compute_nll(*data: Dataset*, *predictions: Tensor*, *, *with_gradient: bool = False*) → Tensor | Tuple[Tensor, Tensor]

Negative log-likelihood without summation (and its gradient w.r.t. predictions if requested).

Parameters

data
[Dataset] The dataset related to the computation of the log likelihood.

predictions
[torch.Tensor] The model's predictions from which to compute the log likelihood.

with_gradient
[bool, optional] If True, returns also the gradient of the negative log likelihood wrt the predictions. If False, only returns the negative log likelihood. Default=False.

Returns

torch.Tensor or tuple of torch.Tensor
The negative log likelihood (and its jacobian if requested).

static compute_residuals(*data: Dataset*, *predictions: Tensor*) → Tensor

Compute the residuals of the given predictions.

Parameters

data
[Dataset] The dataset related to the computation of the residuals.

predictions
[torch.Tensor] The model's predictions from which to compute the residuals.

Returns

torch.Tensor
The residuals.

classmethod compute_rmse(*data: Dataset*, *predictions: Tensor*) → Tensor

Computes root mean squared error of provided data vs. predictions.

Parameters

data
[Dataset] The dataset related to the computation of the root mean squared error.

predictions
[torch.Tensor] The model's predictions from which to compute the root mean squared error.

Returns

torch.Tensor
The root mean squared error.

compute_sufficient_statistics(*data: Dataset*, *predictions: Tensor*) → Dict[str, Tensor]

Compute the specific sufficient statistics and metrics for this noise-model.

Parameters

data
[Dataset] The dataset related to the computation of the sufficient statistics.

predictions
[torch.Tensor] The model's predictions from which to compute the sufficient statistics.

Returns

DictParamsTorch
The sufficient statistics.

factory

alias of Normal

move_to_device(*device: device*) → None

Move all torch tensors stored in this instance to the provided device (parameters & hyperparameters).

Parameters

device
[torch.device] Torch device on which to move the tensors.

raise_if_partially_defined() → None

Raise an error if some of the free parameters are not defined.

classmethod raise_if_unknown_parameters(*params: Iterable | None*) → None

Raise an error if the provided parameters are not part of the free parameters.

Parameters

params
[Iterable, optional] The list of parameters to analyze.

rv_around(*loc: Tensor*) → Distribution

Return the torch distribution centred around values (only if noise is not None).

Parameters

loc
[torch.Tensor] The loc around which to sample.

Returns

torch.distributions.distribution.Distribution
The torch distribution centered around the loc.

sample_around(*loc: Tensor*) → Tensor

Realization around *loc* with respect to partially defined distribution.

Parameters

loc
[torch.Tensor] The loc around which to sample.

Returns

torch.Tensor
The requested sample.

sampler_around(*loc: Tensor*) → Callable[[], Tensor]

Return the sampling function around input values.

Parameters

loc
[torch.Tensor] The loc around which to sample.

Returns

Callable
The sampler.

to_dict() → Dict[str, Any]

Serialize instance as dictionary.

Returns

KwargsType
The instance serialized as a dictionary.

update_parameters(*, *validate: bool = False*, ***parameters: Tensor*) → None

(Partial) update of the free parameters of the distribution family.

Parameters

validate
[bool, optional] If True, the provided parameters are validated before being updated. Default=False.

****parameters**
[torch.Tensor] The new parameters.

update_parameters_from_predictions(*data: Dataset*, *predictions: Tensor*) → None

In-place update of free parameters from provided predictions.

Parameters

data
[Dataset] The dataset related to the computation of the log likelihood.

predictions
[torch.Tensor] The model's predictions from which to update the parameters.

update_parameters_from_sufficient_statistics(*data: Dataset*, *sufficient_statistics: Dict[str, Tensor]*) → None

In-place update of free parameters from provided sufficient statistics.

Parameters

data
[Dataset] The dataset related to the computation of the log likelihood.

sufficient_statistics
[DictParamsTorch] The sufficient statistics to use for parameter update.

validate(***params: Any*) → Dict[str, Tensor]

Validation function for parameters (based on 'validate_xxx' methods).

Parameters

****params**
[Any] The parameters to validate.

Returns

DictParamsTorch
The validated parameters.

validate_scale(*scale: Tensor*) → Tensor

Ensure the scale is valid.

Parameters

scale
[torch.Tensor] The scale to validate.

Returns

torch.Tensor
The validated scale.

leaspy.models.noise_models.GaussianScalarNoiseModel

class GaussianScalarNoiseModel(*parameters: Dict[str, Tensor] | None = None*, *scale_dimension: int | None = None*)

Bases: AbstractGaussianNoiseModel

Class implementing scalar Gaussian noise models.

Parameters

parameters
[dict [:obj`str`, torch.Tensor] or None] Values for all the free parameters of the distribution family. All of them must have values before using the sampling methods.

scale_dimension
[int, optional] The scale dimension.

Attributes

scale_dimension
[int, optional] The scale dimension.

parameters
[dict] Contains the parameters relative to the noise model.

Methods

compute_canonical_loss(data, predictions)	Compute a human-friendly overall loss (RMSE).
compute_l2_residuals(data, predictions)	Compute the squared residuals of the given predictions.
compute_nll(data, predictions, *[, ...])	Negative log-likelihood without summation (and its gradient w.r.t.
compute_residuals(data, predictions)	Compute the residuals of the given predictions.
compute_rmse(data, predictions)	Computes root mean squared error of provided data vs.
compute_sufficient_statistics(data, predictions)	Compute the specific sufficient statistics and metrics for this noise-model.
factory	alias of Normal
move_to_device(device)	Move all torch tensors stored in this instance to the provided device (parameters & hyperparameters).
raise_if_partially_defined()	Raise an error if some of the free parameters are not defined.
raise_if_unknown_parameters(params)	Raise an error if the provided parameters are not part of the free parameters.
rv_around(loc)	Return the torch distribution centred around values (only if noise is not None).
sample_around(loc)	Realization around *loc* with respect to partially defined distribution.
sampler_around(loc)	Return the sampling function around input values.
to_dict()	Serialize instance as dictionary.
update_parameters(*[, validate])	(Partial) update of the free parameters of the distribution family.
update_parameters_from_predictions(data, ...)	In-place update of free parameters from provided predictions.
update_parameters_from_sufficient_statist	In-place update of free parameters from provided sufficient statistics.
validate(**params)	Validation function for parameters (based on 'validate_xxx' methods).
validate_scale(scale)	Ensure the scale is valid.

classmethod compute_canonical_loss(*data: Dataset*, *predictions: Tensor*) → Tensor

Compute a human-friendly overall loss (RMSE).

Parameters

data
[Dataset] The dataset related to the computation of the log likelihood.

predictions
[torch.Tensor] The model's predictions from which to compute the canonical loss.

Returns

torch.Tensor
The computed loss.

classmethod compute_l2_residuals(*data: Dataset*, *predictions: Tensor*) → Tensor

Compute the squared residuals of the given predictions.

Parameters

data
[Dataset] The dataset related to the computation of the squared residuals.

predictions
[torch.Tensor] The model's predictions from which to compute the squared residuals.

Returns

torch.Tensor
The squared residuals.

compute_nll(*data: Dataset*, *predictions: Tensor*, *, *with_gradient: bool = False*) → Tensor | Tuple[Tensor, Tensor]

Negative log-likelihood without summation (and its gradient w.r.t. predictions if requested).

Parameters

data
[Dataset] The dataset related to the computation of the log likelihood.

predictions
[torch.Tensor] The model's predictions from which to compute the log likelihood.

with_gradient
[bool, optional] If True, returns also the gradient of the negative log likelihood wrt the predictions. If False, only returns the negative log likelihood. Default=False.

Returns

torch.Tensor or tuple of torch.Tensor
The negative log likelihood (and its jacobian if requested).

static compute_residuals(*data: Dataset*, *predictions: Tensor*) → Tensor

Compute the residuals of the given predictions.

Parameters

data
[Dataset] The dataset related to the computation of the residuals.

predictions
[torch.Tensor] The model's predictions from which to compute the residuals.

Returns

torch.Tensor
The residuals.

classmethod compute_rmse(*data: Dataset*, *predictions: Tensor*) → Tensor

Computes root mean squared error of provided data vs. predictions.

Parameters

data
[Dataset] The dataset related to the computation of the root mean squared error.

predictions
[torch.Tensor] The model's predictions from which to compute the root mean squared error.

Returns

torch.Tensor
The root mean squared error.

compute_sufficient_statistics(*data: Dataset*, *predictions: Tensor*) → Dict[str, Tensor]

Compute the specific sufficient statistics and metrics for this noise-model.

Parameters

data
[Dataset] The dataset related to the computation of the sufficient statistics.

predictions
[torch.Tensor] The model's predictions from which to compute the sufficient statistics.

Returns

DictParamsTorch
The sufficient statistics.

factory
alias of Normal

move_to_device(*device: device*) → None

Move all torch tensors stored in this instance to the provided device (parameters & hyperparameters).

Parameters

device
[torch.device] Torch device on which to move the tensors.

raise_if_partially_defined() → None

Raise an error if some of the free parameters are not defined.

classmethod raise_if_unknown_parameters(*params: Iterable | None*) → None

Raise an error if the provided parameters are not part of the free parameters.

Parameters

params
[Iterable, optional] The list of parameters to analyze.

rv_around(*loc: Tensor*) → Distribution

Return the torch distribution centred around values (only if noise is not None).

Parameters

loc
[torch.Tensor] The loc around which to sample.

Returns

torch.distributions.distribution.Distribution
The torch distribution centered around the loc.

sample_around(*loc: Tensor*) → Tensor

Realization around *loc* with respect to partially defined distribution.

Parameters

loc
[torch.Tensor] The loc around which to sample.

Returns

torch.Tensor
The requested sample.

sampler_around(*loc: Tensor*) → Callable[[], Tensor]

Return the sampling function around input values.

Parameters

loc
[torch.Tensor] The loc around which to sample.

Returns

Callable
The sampler.

to_dict() → Dict[str, Any]

Serialize instance as dictionary.

Returns

KwargsType
The instance serialized as a dictionary.

update_parameters(*, *validate: bool = False*, ***parameters: Tensor*) → None

(Partial) update of the free parameters of the distribution family.

Parameters

validate
[bool, optional] If True, the provided parameters are validated before being updated. Default=False.

****parameters**
[torch.Tensor] The new parameters.

update_parameters_from_predictions(*data: Dataset*, *predictions: Tensor*) → None

In-place update of free parameters from provided predictions.

Parameters

data
[Dataset] The dataset related to the computation of the log likelihood.

predictions
[torch.Tensor] The model's predictions from which to update the parameters.

update_parameters_from_sufficient_statistics(*data: Dataset*, *sufficient_statistics: Dict[str, Tensor]*) → None

In-place update of free parameters from provided sufficient statistics.

Parameters

data
[Dataset] The dataset related to the computation of the log likelihood.

sufficient_statistics
[DictParamsTorch] The sufficient statistics to use for parameter update.

validate(***params: Any*) → Dict[str, Tensor]

Validation function for parameters (based on 'validate_xxx' methods).

Parameters

****params**
[Any] The parameters to validate.

Returns

DictParamsTorch
The validated parameters.

validate_scale(*scale: Tensor*) → Tensor

Ensure the scale is valid.

Parameters

scale
[torch.Tensor] The scale to validate.

Returns

torch.Tensor
The validated scale.

leaspy.models.noise_models.OrdinalNoiseModel

class OrdinalNoiseModel(*parameters: DictParamsTorch | None = None*, *max_levels: Dict[FeatureType, int] | None = None*)

Bases: OrdinalFamily, AbstractOrdinalNoiseModel

Class implementing ordinal noise models (likelihood is based on PDF).

Parameters

parameters
[dict [str, torch.Tensor] or None] Values for all the free parameters of the distribution family. All of them must have values before using the sampling methods.

max_levels
[dict, optional] Maximum levels for ordinal noise.

Attributes

mask
max_level
max_levels
ordinal_infos
parameters

Methods

compute_canonical_loss(data, predictions)	Compute a human-friendly overall loss (independent from instance parameters), useful as a measure of goodness-of-fit after personalization (nll by default - assuming no free parameters).
compute_nll(data, predictions, *[, ...])	Compute the negative log-likelihood and its gradient wrt predictions.
compute_sufficient_statistics(data, predictions)	Computes the set of noise-related sufficient statistics and metrics (to be extended in child class).
factory(pdf, **kws)	Generate a new MultinomialDistribution from its probability density function instead of its survival function.
move_to_device(device)	Move all torch tensors stored in this instance to the provided device (parameters & hyperparameters).
raise_if_partially_defined()	Raise an error if some of the free parameters are not defined.
raise_if_unknown_parameters(params)	Raise an error if the provided parameters are not part of the free parameters.
rv_around(loc)	Return the torch distribution centred around values (only if noise is not None).
sample_around(loc)	Realization around *loc* with respect to partially defined distribution.
sampler_around(loc)	Return the sampling function around input values.
to_dict()	Serialize instance as dictionary.
update_parameters(*[, validate])	(Partial) update of the free parameters of the distribution family.
update_parameters_from_predictions(data, ...)	Updates noise-model parameters in-place (nothing done by default).
update_parameters_from_sufficient_statist	Updates noise-model parameters in-place (nothing done by default).
validate(**params)	Validation function for parameters (based on 'validate_xxx' methods).

compute_canonical_loss(*data: Dataset*, *predictions: Tensor*) → Tensor

Compute a human-friendly overall loss (independent from instance parameters), useful as a measure of goodness-of-fit after personalization (nll by default - assuming no free parameters).

Parameters

data
[Dataset] The dataset related to the computation of the log likelihood.

predictions
[torch.Tensor] The model's predictions from which to compute the canonical loss.

Returns

torch.Tensor
The computed loss.

compute_nll(*data: Dataset*, *predictions: Tensor*, *, *with_gradient: bool = False*) → Tensor | Tuple[Tensor, Tensor]

Compute the negative log-likelihood and its gradient wrt predictions.

Parameters

data
[Dataset] The dataset related to the computation of the log likelihood.

predictions
[torch.Tensor] The model's predictions from which to compute the log likelihood.

with_gradient
[bool, optional] If True, returns also the gradient of the negative log likelihood wrt the predictions. If False, only returns the negative log likelihood. Default=False.

Returns

torch.Tensor or tuple of torch.Tensor
The negative log likelihood (and its jacobian if requested).

compute_sufficient_statistics(*data: Dataset*, *predictions: Tensor*) → Dict[str, Tensor]

Computes the set of noise-related sufficient statistics and metrics (to be extended in child class).

Parameters

data
[Dataset] The dataset related to the computation of the sufficient statistics.

predictions
[torch.Tensor] The model's predictions from which to compute the sufficient statistics.

Returns

DictParamsTorch
The sufficient statistics.

classmethod factory(*pdf: Tensor*, ***kws*)

Generate a new MultinomialDistribution from its probability density function instead of its survival function.

Parameters

pdf
[torch.Tensor] The input probability density function.

****kws**
Additional keyword arguments to be passed for instance initialization.

move_to_device(*device: device*) → None

Move all torch tensors stored in this instance to the provided device (parameters & hyperparameters).

Parameters

device
[torch.device] Torch device on which to move the tensors.

raise_if_partially_defined() → None

Raise an error if some of the free parameters are not defined.

classmethod raise_if_unknown_parameters(*params: Iterable | None*) → None

Raise an error if the provided parameters are not part of the free parameters.

Parameters

params
[Iterable, optional] The list of parameters to analyze.

rv_around(*loc: Tensor*) → Distribution

Return the torch distribution centred around values (only if noise is not None).

Parameters

loc
[torch.Tensor] The loc around which to sample.

Returns

torch.distributions.distribution.Distribution
The torch distribution centered around the loc.

sample_around(*loc: Tensor*) → Tensor

Realization around *loc* with respect to partially defined distribution.

Parameters

loc
[torch.Tensor] The loc around which to sample.

Returns

torch.Tensor
The requested sample.

sampler_around(*loc: Tensor*) → Callable[[], Tensor]

Return the sampling function around input values.

Parameters

loc
[torch.Tensor] The loc around which to sample.

Returns

Callable
The sampler.

to_dict() → Dict[str, Any]

Serialize instance as dictionary.

> **Warning:** Do NOT export hyper-parameters that are derived (error-prone and boring checks when re-creating).

Returns

KwargsType
The instance serialized as a dictionary.

update_parameters(*, *validate: bool = False*, ***parameters: Tensor*) → None

(Partial) update of the free parameters of the distribution family.

Parameters

validate
[bool, optional] If True, the provided parameters are validated before being updated. Default=False.

****parameters**
[torch.Tensor] The new parameters.

`update_parameters_from_predictions`(*data: Dataset*, *predictions: Tensor*) → None

Updates noise-model parameters in-place (nothing done by default).

Parameters

data
[`Dataset`] The dataset related to the computation of the log likelihood.

predictions
[`torch.Tensor`] The model's predictions from which to update the parameters.

`update_parameters_from_sufficient_statistics`(*data: Dataset*, *sufficient_statistics: Dict[str, Tensor]*) → None

Updates noise-model parameters in-place (nothing done by default).

Parameters

data
[`Dataset`] The dataset related to the computation of the log likelihood.

sufficient_statistics
[`DictParamsTorch`] The sufficient statistics to use for parameter update.

`validate`(***params: Any*) → Dict[str, Tensor]

Validation function for parameters (based on 'validate_xxx' methods).

Parameters

****params**
[Any] The parameters to validate.

Returns

`DictParamsTorch`
The validated parameters.

`leaspy.models.noise_models.OrdinalRankingNoiseModel`

`class OrdinalRankingNoiseModel`(*parameters: DictParamsTorch | None = None*, *max_levels: Dict[FeatureType, int] | None = None*)

Bases: `OrdinalRankingFamily`, `AbstractOrdinalNoiseModel`

Class implementing `OrdinalRankingNoiseModel` (likelihood is based on SF).

Parameters

parameters
[`dict` [`str`, `torch.Tensor`] or None] Values for all the free parameters of the distribution family. All of them must have values before using the sampling methods.

max_levels
[`dict`, optional] Maximum levels for ordinal noise.

Attributes

mask
max_level
max_levels
ordinal_infos
parameters

Methods

compute_canonical_loss(data, predictions)	Compute a human-friendly overall loss (independent from instance parameters), useful as a measure of goodness-of-fit after personalization (nll by default - assuming no free parameters).
compute_nll(data, predictions, *[, ...])	Compute the negative log-likelihood and its gradient wrt predictions.
compute_sufficient_statistics(data, predictions)	Computes the set of noise-related sufficient statistics and metrics (to be extended in child class).
factory	alias of MultinomialDistribution
move_to_device(device)	Move all torch tensors stored in this instance to the provided device (parameters & hyperparameters).
raise_if_partially_defined()	Raise an error if some of the free parameters are not defined.
raise_if_unknown_parameters(params)	Raise an error if the provided parameters are not part of the free parameters.
rv_around(loc)	Return the torch distribution centred around values (only if noise is not None).
sample_around(loc)	Realization around *loc* with respect to partially defined distribution.
sampler_around(loc)	Return the sampling function around input values.
to_dict()	Serialize instance as dictionary.
update_parameters(*[, validate])	(Partial) update of the free parameters of the distribution family.
update_parameters_from_predictions(data, ...)	Updates noise-model parameters in-place (nothing done by default).
update_parameters_from_sufficient_statist	Updates noise-model parameters in-place (nothing done by default).
validate(**params)	Validation function for parameters (based on 'validate_xxx' methods).

compute_canonical_loss(*data: Dataset*, *predictions: Tensor*) → Tensor

Compute a human-friendly overall loss (independent from instance parameters), useful as a measure of goodness-of-fit after personalization (nll by default - assuming no free parameters).

Parameters

data
[Dataset] The dataset related to the computation of the log likelihood.

predictions
[torch.Tensor] The model's predictions from which to compute the canonical loss.

Returns

torch.Tensor
The computed loss.

compute_nll(*data: Dataset*, *predictions: Tensor*, *, *with_gradient: bool = False*) → Tensor | Tuple[Tensor, Tensor]

Compute the negative log-likelihood and its gradient wrt predictions.

Parameters

data
[Dataset] The dataset related to the computation of the log likelihood.

predictions
[torch.Tensor] The model's predictions from which to compute the log likelihood.

with_gradient
[bool, optional] If True, returns also the gradient of the negative log likelihood wrt the predictions. If False, only returns the negative log likelihood. Default=False.

Returns

torch.Tensor or tuple of torch.Tensor
The negative log likelihood (and its jacobian if requested).

compute_sufficient_statistics(*data: Dataset*, *predictions: Tensor*) → Dict[str, Tensor]

Computes the set of noise-related sufficient statistics and metrics (to be extended in child class).

Parameters

data
[Dataset] The dataset related to the computation of the sufficient statistics.

predictions
[torch.Tensor] The model's predictions from which to compute the sufficient statistics.

Returns

DictParamsTorch
The sufficient statistics.

factory

alias of MultinomialDistribution

move_to_device(*device: device*) → None

Move all torch tensors stored in this instance to the provided device (parameters & hyperparameters).

Parameters

device
[torch.device] Torch device on which to move the tensors.

raise_if_partially_defined() → None

Raise an error if some of the free parameters are not defined.

classmethod raise_if_unknown_parameters(*params: Iterable | None*) → None

Raise an error if the provided parameters are not part of the free parameters.

Parameters

params
[Iterable, optional] The list of parameters to analyze.

rv_around(*loc: Tensor*) → Distribution

Return the torch distribution centred around values (only if noise is not None).

Parameters

loc
[torch.Tensor] The loc around which to sample.

Returns

torch.distributions.distribution.Distribution
The torch distribution centered around the loc.

sample_around(*loc: Tensor*) → Tensor

Realization around *loc* with respect to partially defined distribution.

Parameters

loc
[torch.Tensor] The loc around which to sample.

Returns

torch.Tensor
The requested sample.

sampler_around(*loc: Tensor*) → Callable[[], Tensor]

Return the sampling function around input values.

Parameters

loc
[torch.Tensor] The loc around which to sample.

Returns

Callable
The sampler.

to_dict() → Dict[str, Any]

Serialize instance as dictionary.

> **Warning:** Do NOT export hyper-parameters that are derived (error-prone and boring checks when re-creating).

Returns

KwargsType
The instance serialized as a dictionary.

update_parameters(*, *validate: bool = False*, ***parameters: Tensor*) → None

(Partial) update of the free parameters of the distribution family.

Parameters

validate
[bool, optional] If True, the provided parameters are validated before being updated. Default=False.

****parameters**
[torch.Tensor] The new parameters.

update_parameters_from_predictions(*data: Dataset*, *predictions: Tensor*) → None

Updates noise-model parameters in-place (nothing done by default).

Parameters

data
[Dataset] The dataset related to the computation of the log likelihood.

predictions
[torch.Tensor] The model's predictions from which to update the parameters.

update_parameters_from_sufficient_statistics(*data: Dataset*, *sufficient_statistics: Dict[str, Tensor]*) → None

Updates noise-model parameters in-place (nothing done by default).

Parameters

data
[Dataset] The dataset related to the computation of the log likelihood.

sufficient_statistics
[DictParamsTorch] The sufficient statistics to use for parameter update.

validate(***params: Any*) → Dict[str, Tensor]

Validation function for parameters (based on 'validate_xxx' methods).

Parameters

****params**
[Any] The parameters to validate.

Returns

DictParamsTorch
The validated parameters.

noise_model_factory(noise_model, **kws)	Factory for noise models.
export_noise_model(noise_model)	Serialize a given BaseNoiseModel as a dict.

leaspy.models.noise_models.noise_model_factory

noise_model_factory(*noise_model: str | BaseNoiseModel | Dict[str, Any]*, ***kws*) → BaseNoiseModel

Factory for noise models.

Parameters

noise_model
[str or BaseNoiseModel or dict [str, ...]]

- If an instance of a subclass of BaseNoiseModel, returns the instance.
- If a string, then returns a new instance of the appropriate class (with optional parameters *kws*).
- If a dictionary, it must contain the 'name' key and other initialization parameters.

****kws**
Optional parameters for initializing the requested noise-model when a string.

Returns

BaseNoiseModel
The desired noise model.

Raises

LeaspyModelInputError
If *noise_model* is not supported.

leaspy.models.noise_models.export_noise_model

export_noise_model(*noise_model: BaseNoiseModel*) → Dict[str, Any]

Serialize a given BaseNoiseModel as a dict.

Parameters

noise_model
[BaseNoiseModel] The noise model to serialize.

Returns

KwargsType
The noise model serialized as a dict.

3.2.12 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.
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 AttributesFactory

Bases: object

Return an *Attributes* class object based on the given parameters.

Methods

attributes(name, dimension[, ...])	Class method to build correct model attributes depending on model *name*.

classmethod attributes(*name: str*, *dimension: int*, *source_dimension: int | None = None*, *ordinal_infos=None*) → AbstractAttributes

Class method to build correct model attributes depending on model *name*.

Parameters

name
[str]

dimension
[int]

source_dimension
[int, optional (default None)]

ordinal_infos
[dict, optional] Only for models with ordinal noise. Cf ordinal_infos attribute of MultivariateModel

Returns

`AbstractAttributes`

Raises

`LeaspyModelInputError`
if any inconsistent parameter.

leaspy.models.utils.attributes.abstract_attributes.AbstractAttributes

`class AbstractAttributes(name: str, dimension: int, source_dimension: int)`

Bases: `ABC`

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)]

Raises

`LeaspyModelInputError`
if any inconsistent parameter.

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
[set[str] (default empty)] Contains the available parameters to update. Different models have different parameters.

Methods

`get_attributes`()	Returns the attributes of the model, which is a tuple of three torch tensors.
`move_to_device`(device)	Move the tensor attributes of this class to the specified device.
`update`(names_of_changed_values, values)	Update model group average parameter(s).

abstract get_attributes() → Tuple[FloatTensor, FloatTensor, FloatTensor]

Returns the attributes of the model, which is a tuple of three torch tensors.

For the precise definitions of those attributes please refer to the exact attributes class associated to your model.

Returns

positions: *torch.Tensor*
velocities: *torch.Tensor*
mixing_matrix: *torch.Tensor*

move_to_device(*device: device*)

Move the tensor attributes of this class to the specified device.

Parameters

device
[torch.device]

abstract update(*names_of_changed_values: Set[str]*, *values: Dict[str, Tensor]*) → None

Update model group average parameter(s).

Parameters

names_of_changed_values
[set[str]] Values to be updated

values
[dict [str, *torch.Tensor*]] New values used to update the model's group average parameters

Raises

LeaspyModelInputError
If *names_of_changed_values* contains unknown values to update.

leaspy.models.utils.attributes.abstract_manifold_model_attributes.AbstractManifoldModelAttributes

class AbstractManifoldModelAttributes(*name: str*, *dimension: int*, *source_dimension: int*)

Bases: 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

name
[str]

dimension
[int]

source_dimension
[int (default None)]

Raises

LeaspyModelInputError
if any inconsistent parameter.

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
[set[str]] Contains the available parameters to update. Different models have different parameters.

positions
[torch.Tensor [dimension] (default None)] <!> Depending on the model it does not correspond to the same thing.

velocities
[torch.Tensor [dimension] (default None)] Vector of velocities for each feature (positive components). For multivariate models only (except for parallel model as it is useless).

orthonormal_basis
[torch.Tensor [dimension, dimension - 1] (default None)] For multivariate and multivariate parallel models, with source_dimension >= 1.

betas
[torch.Tensor [dimension - 1, source_dimension] (default None)] For multivariate and multivariate parallel models, with source_dimension >= 1.

mixing_matrix
[torch.Tensor [dimension, source_dimension] (default None)] Matrix A such that w_i = A * s_i. For multivariate and multivariate parallel models, with source_dimension >= 1.

Methods

get_attributes()	Returns the attributes of the model, which is a tuple of three torch tensors.
move_to_device(device)	Move the tensor attributes of this class to the specified device.
update(names_of_changed_values, values)	Update model group average parameter(s).

get_attributes() → Tuple[FloatTensor, FloatTensor, FloatTensor]

Returns the attributes of the model, which is a tuple of three torch tensors.

For the precise definitions of those attributes please refer to the exact attributes class associated to your model.

Returns

positions: *torch.Tensor*
velocities: *torch.Tensor*
mixing_matrix: *torch.Tensor*

move_to_device(*device: device*)

Move the tensor attributes of this class to the specified device.

Parameters

device
[torch.device]

abstract update(*names_of_changed_values: Set[str]*, *values: Dict[str, Tensor]*) → None

Update model group average parameter(s).

Parameters

names_of_changed_values
[set[str]] Values to be updated

values
[dict [str, *torch.Tensor*]] New values used to update the model's group average parameters

Raises

LeaspyModelInputError
If *names_of_changed_values* contains unknown values to update.

leaspy.models.utils.attributes.linear_attributes.LinearAttributes

class LinearAttributes(*name*, *dimension*, *source_dimension*)

Bases: AbstractManifoldModelAttributes

Attributes of leaspy linear models.

Contains the common attributes & methods to update the linear model's attributes.

Parameters

name
[str]

dimension
[int]

source_dimension
[int]

See also:

UnivariateModel
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
[set[str] (default {'all', 'g', 'v0', 'betas'})] Contains the available parameters to update. Different models have different parameters.

positions
[torch.Tensor [dimension] (default None)] positions = realizations['g'] such that "p0" = positions

velocities
[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 attributes of the model, which is a tuple of three torch tensors.
`move_to_device`(device)	Move the tensor attributes of this class to the specified device.
`update`(names_of_changed_values, values)	Update model group average parameter(s).

`get_attributes`() → Tuple[FloatTensor, FloatTensor, FloatTensor]

Returns the attributes of the model, which is a tuple of three torch tensors.

For the precise definitions of those attributes please refer to the exact attributes class associated to your model.

Returns

positions: ***torch.Tensor***
velocities: ***torch.Tensor***
mixing_matrix: ***torch.Tensor***

`move_to_device`(*device: device*)

Move the tensor attributes of this class to the specified device.

Parameters

device
[torch.device]

`update`(*names_of_changed_values*, *values*)

Update model group average parameter(s).

Parameters

names_of_changed_values
[set[str]]

Elements of set must be either:

- `all` (update everything)
- `g` correspond to the attribute `positions`.
- `v0` (only for multivariate models) correspond to the attribute `velocities`. When we are sure that the v0 change is only a scalar multiplication (in particular, when we reparametrize log(v0) <- log(v0) + mean(xi)), we may update velocities using `v0_collinear`, otherwise we always assume v0 is NOT collinear to previous value (no need to perform the verification it is - would not be really efficient)
- `betas` correspond to the linear combination 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

`LeaspyModelInputError`
If *names_of_changed_values* contains unknown parameters.

leaspy.models.utils.attributes.logistic_attributes.LogisticAttributes

class LogisticAttributes(*name*, *dimension*, *source_dimension*)

Bases: AbstractManifoldModelAttributes

Attributes of leaspy logistic models.

Contains the common attributes & methods to update the logistic model's attributes.

Parameters

name
[str]

dimension
[int]

source_dimension
[int]

See also:

UnivariateModel
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
[set[str] (default {'all', 'g', 'v0', 'betas'})] Contains the available parameters to update. Different models have different parameters.

positions
[torch.Tensor [dimension] (default None)] positions = exp(realizations['g']) such that "p0" = 1 / (1 + positions)

velocities
[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 attributes of the model, which is a tuple of three torch tensors.
`move_to_device`(device)	Move the tensor attributes of this class to the specified device.
`update`(names_of_changed_values, values)	Update model group average parameter(s).

`get_attributes`() → Tuple[FloatTensor, FloatTensor, FloatTensor]

Returns the attributes of the model, which is a tuple of three torch tensors.

For the precise definitions of those attributes please refer to the exact attributes class associated to your model.

Returns

positions: ***torch.Tensor***
velocities: ***torch.Tensor***
mixing_matrix: ***torch.Tensor***

`move_to_device`(*device: device*)

Move the tensor attributes of this class to the specified device.

Parameters

device
[torch.device]

`update`(*names_of_changed_values*, *values*)

Update model group average parameter(s).

Parameters

names_of_changed_values
[set[str]]

Elements of set must be either:

- `all` (update everything)
- `g` correspond to the attribute `positions`.
- `v0` (only for multivariate models) correspond to the attribute `velocities`. When we are sure that the v0 change is only a scalar multiplication (in particular, when we reparametrize log(v0) <- log(v0) + mean(xi)), we may update velocities using `v0_collinear`, otherwise we always assume v0 is NOT collinear to previous value (no need to perform the verification it is - would not be really efficient)
- `betas` correspond to the linear combination 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

`LeaspyModelInputError`
If *names_of_changed_values* contains unknown parameters.

leaspy.models.utils.attributes.logistic_parallel_attributes.LogisticParallelAttributes

class LogisticParallelAttributes(*name*, *dimension*, *source_dimension*)

Bases: AbstractManifoldModelAttributes

Attributes of leaspy logistic parallel models.

Contains the common attributes & methods of the logistic parallel models' attributes.

Parameters

name
[str]

dimension
[int]

source_dimension
[int]

Raises

LeaspyModelInputError
if any inconsistent parameters for the model.

See also:

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
[set[str] (default {'all', 'g', 'deltas', 'betas'})] Contains the available parameters to update. Different models have different parameters.

positions
[torch.Tensor (scalar) (default None)] positions = exp(realizations['g']) such that "p0" = 1 / (1 + positions * exp(-deltas))

deltas
[torch.Tensor [dimension] (default None)] deltas = [0, delta_2_realization, ..., delta_n_realization]

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.
move_to_device(device)	Move the tensor attributes of this class to the specified device.
update(names_of_changed_values, values)	Update model group average parameter(s).

get_attributes()

Returns the following attributes: positions, deltas & mixing_matrix.

Returns

positions: *torch.Tensor*
deltas: *torch.Tensor*
mixing_matrix: *torch.Tensor*

move_to_device(*device: device*)

Move the tensor attributes of this class to the specified device.

Parameters

device
[torch.device]

update(*names_of_changed_values*, *values*)

Update model group average parameter(s).

Parameters

names_of_changed_values
[set[str]]

Elements of set must be either:

- all (update everything)
- g correspond to the attribute positions.
- deltas correspond to the attribute deltas.
- betas correspond to the linear combination 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

LeaspyModelInputError
If *names_of_changed_values* contains unknown parameters.

3.2.13 leaspy.models.utils.initialization: Initialization methods

Available methods to initialize model parameters before a fit.

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

initialize_parameters(*model*, *dataset*, *method='default'*) → tuple

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.

Raises

LeaspyInputError
If no initialization method is known for model type / method

3.3 leaspy.algo: Algorithms

Contains all algorithms used in the package.

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

3.3.1 leaspy.algo.abstract_algo.AbstractAlgo

class AbstractAlgo(*settings: AlgorithmSettings*)

Bases: ABC

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

Parameters

settings
[AlgorithmSettings] The specifications of the algorithm as a AlgorithmSettings instance.

Attributes

name
[str] Name of the algorithm.

family
[str]

Family of the algorithm. For now, valid families are:

- 'fit'`
- 'personalize'`
- 'simulate'

deterministic
[bool] True, if and only if algorithm does not involve in randomness. Setting a seed and such algorithms will be useless.

algo_parameters
[dict] Contains the algorithm's parameters. Those are controlled by the AlgorithmSettings.parameters class attribute.

seed
[int, optional] Seed used by numpy and torch.

output_manager
[FitOutputManager] Optional output manager of the algorithm

Methods

load_parameters(parameters)	Update the algorithm's parameters by the ones in the given dictionary.
run(model, *args[, return_loss])	Main method, run the algorithm.
run_impl(model, *args, **extra_kwargs)	Run the algorithm (actual implementation), to be implemented in children classes.
set_output_manager(output_settings)	Set a FitOutputManager object for the run of the algorithm

load_parameters(*parameters: dict*)

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: AbstractModel*, **args*, *return_loss: bool = False*, ***extra_kwargs*) → Any

Main method, run the algorithm.

TODO fix proper abstract class method: input depends on algorithm… (esp. simulate != from others…)

Parameters

model
[`AbstractModel`] The used model.

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

return_loss
[bool (default False), keyword only] Should the algorithm return main output and optional loss output as a 2-tuple?

Returns

Depends on algorithm class: TODO change?

See also:

AbstractFitAlgo
AbstractPersonalizeAlgo
SimulationAlgorithm

abstract run_impl(*model: AbstractModel*, **args*, ***extra_kwargs*) → Tuple[Any, torch.FloatTensor | None]

Run the algorithm (actual implementation), to be implemented in children classes.

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

A 2-tuple containing:

- the result to send back to user
- optional float tensor representing loss (to be printed)

See also:

AbstractFitAlgo
AbstractPersonalizeAlgo
SimulationAlgorithm

set_output_manager(*output_settings: OutputsSettings*) → None

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))
```

3.3.2 leaspy.algo.algo_factory.AlgoFactory

class AlgoFactory

Bases: object

Return the wanted algorithm given its name.

Notes

For developers: add your new algorithm in corresponding category of _algos dictionary.

Methods

algo(algorithm_family, settings)	Return the wanted algorithm given its name.
get_class(name)	Get the class of the algorithm identified as *name*.

classmethod algo(*algorithm_family: str*, *settings*) → AbstractAlgo

Return the wanted algorithm given its name.

Parameters

algorithm_family
[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 family.

Raises

LeaspyAlgoInputError

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

classmethod get_class(*name: str*) → Type[AbstractAlgo]

Get the class of the algorithm identified as *name*.

3.3.3 leaspy.algo.fit: Fit algorithms

Algorithms used to calibrate (fit) a model.

abstract_fit_algo.AbstractFitAlgo(settings)	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 AbstractFitAlgo(*settings*)

Bases: AlgoWithDeviceMixin, AbstractAlgo

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

Parameters

settings
[AlgorithmSettings] The specifications of the algorithm as a AlgorithmSettings instance.

See also:

Leaspy.fit()

Attributes

algorithm_device
[str] Valid torch device

current_iteration
[int, default 0] The number of the current iteration. The first iteration will be 1 and the last one *n_iter*.

sufficient_statistics
[dict[str, *torch.FloatTensor*] or None] The previous step sufficient statistics. It is None during all the burn-in phase.

Inherited attributes
From AbstractAlgo

Methods

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, *args[, return_loss])	Main method, run the algorithm.
run_impl(model, dataset)	Main method, run the algorithm.
set_output_manager(output_settings)	Set a FitOutputManager object for the run of the algorithm

abstract iteration(*dataset: Dataset*, *model: AbstractModel*, *realizations: CollectionRealization*)

Update the parameters (abstract method).

Parameters

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

model
[AbstractModel] The used model.

realizations
[CollectionRealization] The parameters.

load_parameters(*parameters: dict*)

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: AbstractModel*, **args*, *return_loss: bool = False*, ***extra_kwargs*) → Any

Main method, run the algorithm.

TODO fix proper abstract class method: input depends on algorithm... (esp. simulate != from others...)

Parameters

model
[AbstractModel] The used model.

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

return_loss
[bool (default False), keyword only] Should the algorithm return main output and optional loss output as a 2-tuple?

Returns

Depends on algorithm class: TODO change?

See also:

AbstractFitAlgo
AbstractPersonalizeAlgo
SimulationAlgorithm

run_impl(*model: AbstractModel*, *dataset: 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

2-tuple:

- **realizations**
  [CollectionRealization] The optimized parameters.
- None : placeholder for noise-std

set_output_manager(*output_settings: OutputsSettings*) → None

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 AbstractFitMCMC(*settings*)

Bases: AlgoWithAnnealingMixin, AlgoWithSamplersMixin, 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:

leaspy.algo.utils.samplers

Attributes

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

random_order_variables
[bool (default True)] This attribute controls whether we randomize the order of variables at each iteration. Article https://proceedings.neurips.cc/paper/2016/hash/e4da3b7fbbce2345d7772b0674a318d5-Abstract.html gives a rationale on why we should activate this flag.

temperature
[float]

temperature_inv
[float] Temperature and its inverse (modified during algorithm when using annealing)

Methods

`iteration`(dataset, model, realizations)	MCMC-SAEM iteration.
`load_parameters`(parameters)	Update the algorithm's parameters by the ones in the given dictionary.
`run`(model, *args[, return_loss])	Main method, run the algorithm.
`run_impl`(model, dataset)	Main method, run the algorithm.
`set_output_manager`(output_settings)	Set a `FitOutputManager` object for the run of the algorithm

iteration(*dataset: Dataset*, *model: AbstractModel*, *realizations: CollectionRealization*) → None

MCMC-SAEM iteration.

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

Parameters

dataset
[Dataset]

model
[AbstractModel]

realizations
[CollectionRealization]

load_parameters(*parameters: dict*)

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}
```

(continues on next page)

(continued from previous page)

```
>>> 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: AbstractModel*, **args*, *return_loss: bool = False*, ***extra_kwargs*) → Any

Main method, run the algorithm.

TODO fix proper abstract class method: input depends on algorithm... (esp. simulate != from others...)

Parameters

model
[`AbstractModel`] The used model.

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

return_loss
[bool (default False), keyword only] Should the algorithm return main output and optional loss output as a 2-tuple?

Returns

Depends on algorithm class: TODO change?

See also:

`AbstractFitAlgo`
`AbstractPersonalizeAlgo`
`SimulationAlgorithm`

run_impl(*model: AbstractModel*, *dataset: 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

2-tuple:

- **realizations**
  [CollectionRealization] The optimized parameters.
- None : placeholder for noise-std

set_output_manager(*output_settings: OutputsSettings*) → None

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 TensorMCMCSAEM(*settings*)

Bases: AbstractFitMCMC

Main algorithm for MCMC-SAEM.

Parameters

settings
[AlgorithmSettings] MCMC fit algorithm settings

See also:

AbstractFitMCMC

Methods

`iteration`(dataset, model, realizations)	MCMC-SAEM iteration.
`load_parameters`(parameters)	Update the algorithm's parameters by the ones in the given dictionary.
`run`(model, *args[, return_loss])	Main method, run the algorithm.
`run_impl`(model, dataset)	Main method, run the algorithm.
`set_output_manager`(output_settings)	Set a `FitOutputManager` object for the run of the algorithm

`iteration`(*dataset: Dataset*, *model: AbstractModel*, *realizations: CollectionRealization*) → None

MCMC-SAEM iteration.

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

Parameters

dataset
[Dataset]

model
[AbstractModel]

realizations
[CollectionRealization]

`load_parameters`(*parameters: dict*)

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}
```

(continues on next page)

(continued from previous page)

```
>>> 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: AbstractModel*, **args*, *return_loss: bool = False*, ***extra_kwargs*) → Any

Main method, run the algorithm.

TODO fix proper abstract class method: input depends on algorithm... (esp. simulate != from others...)

Parameters

model
[`AbstractModel`] The used model.

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

return_loss
[bool (default False), keyword only] Should the algorithm return main output and optional loss output as a 2-tuple?

Returns

Depends on algorithm class: TODO change?

See also:

`AbstractFitAlgo`
`AbstractPersonalizeAlgo`
`SimulationAlgorithm`

run_impl(*model: AbstractModel*, *dataset: 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

2-tuple:

- **realizations**
  [CollectionRealization] The optimized parameters.
- None : placeholder for noise-std

set_output_manager(*output_settings: OutputsSettings*) → None

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))
```

3.3.4 leaspy.algo.personalize: Personalization algorithms

Algorithms used to personalize a model to given 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 AbstractPersonalizeAlgo(*settings: AlgorithmSettings*)

Bases: 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

name
[str] Algorithm's name.

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

algo_parameters
[dict] Algorithm's parameters.

Methods

load_parameters(parameters)	Update the algorithm's parameters by the ones in the given dictionary.
run(model, *args[, return_loss])	Main method, run the algorithm.
run_impl(model, dataset)	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

load_parameters(*parameters: dict*)

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,
```

(continues on next page)

(continued from previous page)

```
'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: AbstractModel*, **args*, *return_loss: bool = False*, ***extra_kwargs*) → Any

Main method, run the algorithm.

TODO fix proper abstract class method: input depends on algorithm… (esp. simulate != from others…)

Parameters

model
[`AbstractModel`] The used model.

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

return_loss
[bool (default False), keyword only] Should the algorithm return main output and optional loss output as a 2-tuple?

Returns

Depends on algorithm class: TODO change?

See also:

`AbstractFitAlgo`
`AbstractPersonalizeAlgo`
`SimulationAlgorithm`

run_impl(*model: AbstractModel*, *dataset: Dataset*) → Tuple[IndividualParameters, Tensor]

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

Parameters

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

dataset
[`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 *model.noise_model*

is 'gaussian_diagonal')

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

where $\varepsilon_{i,j} = (f(\theta, (z_{i,j}), (t_{i,j})) - (y_{i,j}))^2$, 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: OutputsSettings*) → None

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 ScipyMinimize(*settings*)

Bases: AbstractPersonalizeAlgo

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

Parameters

settings
[AlgorithmSettings] Settings of the algorithm. In particular the parameter *custom_scipy_minimize_params* may contain keyword arguments passed to scipy.optimize.minimize().

Attributes

scipy_minimize_params
[dict] Keyword arguments to be passed to scipy.optimize.minimize(). A default setting depending on whether using jacobian or not is applied (cf. *ScipyMinimize.DEFAULT_SCIPY_MINIMIZE_PARAMS_WITH_JACOBIAN*

and *ScipyMinimize.DEFAULT_SCIPY_MINIMIZE_PARAMS_WITHOUT_JACOBIAN*).

You may customize it by setting the *custom_scipy_minimize_params* algorithm parameter.

format_convergence_issues
[str] Formatting of convergence issues. It should be a formattable string using any of those variables:

- patient_id: str
- optimization_result_pformat: str
- (optimization_result_obj: dict-like)

cf. *ScipyMinimize.DEFAULT_FORMAT_CONVERGENCE_ISSUES* for the default format. You may customize it by setting the *custom_format_convergence_issues* algorithm parameter.

logger
[None or callable str -> None] The function used to display convergence issues returned by `scipy.optimize.minimize()`. By default we print the convergences issues if and only if we do not use BFGS optimization method. You can customize it at initialization by defining a *logger* attribute to your *AlgorithmSettings* instance.

Methods

`is_jacobian_implemented`(model)	Check that the jacobian of model is implemented.
`load_parameters`(parameters)	Update the algorithm's parameters by the ones in the given dictionary.
`obj`(x, model, dataset, with_gradient)	Objective loss function to minimize in order to get patient's individual parameters
`run`(model, *args[, return_loss])	Main method, run the algorithm.
`run_impl`(model, dataset)	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

`is_jacobian_implemented`(*model: AbstractModel*) → bool

Check that the jacobian of model is implemented.

`load_parameters`(*parameters: dict*)

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}}
```

obj(*x: list*, *model: AbstractModel*, *dataset: Dataset*, *with_gradient: bool*)

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

Parameters

x
: [list[torch.tensors]] Individual **standardized** parameters At initialization `x = [xi_mean/xi_std, tau_mean/tau_std] (+ [0.] * n_sources if multivariate model)`

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

dataset
: [`Dataset`] A dataset instance for the single patient being optimized.

with_gradient
: [bool] If True: return (objective, gradient_objective) Else: simply return objective

Returns

objective
: [float] Value of the loss function (negative 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

Raises

LeaspyAlgoInputError
if noise model is not currently supported by algorithm.

run(*model: AbstractModel*, **args*, *return_loss: bool = False*, ***extra_kwargs*) → Any

Main method, run the algorithm.

TODO fix proper abstract class method: input depends on algorithm… (esp. simulate != from others…)

Parameters

model
[AbstractModel] The used model.

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

return_loss
[bool (default False), keyword only] Should the algorithm return main output and optional loss output as a 2-tuple?

Returns

Depends on algorithm class: TODO change?

See also:

AbstractFitAlgo
AbstractPersonalizeAlgo
SimulationAlgorithm

run_impl(*model: AbstractModel*, *dataset: Dataset*) → Tuple[IndividualParameters, Tensor]

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

Parameters

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

dataset
[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 *model.noise_model* is 'gaussian_diagonal')

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

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: OutputsSettings*) → None

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))
```

3.3.5 `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 SimulationAlgorithm(*settings*)

Bases: `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(s) and the cofactor(s) 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.

Parameters

settings
[`AlgorithmSettings`] The algorithm settings. They may include the following parameters, described in __Attributes__ section:

- *noise*
- *bandwidth_method*
- *cofactor*

- *cofactor_state*
- *number_of_subjects*
- *mean_number_of_visits*, *std_number_of_visits*, *min_number_of_visits*, *max_number_of_visits*
- *delay_btw_visits*
- *reparametrized_age_bounds*
- *sources_method*
- *prefix*
- *features_bounds*
- *features_bounds_nb_subjects_factor*

Raises

`LeaspyAlgoInputError`
If algorithm parameters are of bad type or do not comply to detailed requirements.

Notes

The baseline ages are no more jointly learnt with individual parameters. Instead, we jointly learn the _reparametrized_ baseline ages, together with individual parameters. The baseline ages are then reconstructed from the simulated reparametrized baseline ages and individual parameters.

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 derived by the *model* object.

One can restrict the interval of the baseline reparametrized age to be _learnt_ in kernel, by setting bounds in *reparametrized_age_bounds*. Note that the simulated reparametrized baseline ages are unconstrained and thus could, theoretically (but very unlikely), be out of these prescribed bounds.

Attributes

name
['simulation'] Algorithm's name.

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

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
[list[str], optional (default = None)] The list of cofactors included used to select the wanted group of patients (ex - ['genetic']). All of them must correspond to an existing cofactor in the attribute *Data* of the input *result* of the `run()` method. TODO? should we allow to learn joint distribution of individual parameters and numeric/categorical cofactors (not fixed)?

cofactor_state
[list[str], optional (default None)] The cofactors states used to select the wanted group of patients (ex - ['APOE4']). There is exactly one state per cofactor in *cofactor* (same order). It must correspond to an existing cofactor state in the attribute *Data* of the input *result* of the `run()` method. TODO? it could be replaced by methods to easily sub-select individual having certain cofactors PRIOR to running this algorithm + the functionality described just above (included varying cofactors as part of the distribution to estimate).

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 baseline data (TODO: "baseline" instead?).
- *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), ...}`

features_bounds_nb_subjects_factor
[float > 1 (default 10)] Only used if *features_bounds* is not False. The ratio of simulated subjects (> 1) so that there is at least *number_of_subjects* that comply to features bounds constraint.

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

std_number_of_visits
[int or float > 0, or None (default 3)] Standard deviation used into the generation of the number of visits per simulated patient. If <= 0 or None: number of visits will be deterministic

min_number_of_visits, max_number_of_visits
[int (optional for max)] Minimum (resp. maximum) number of visits. Only used when *std_number_of_visits* > 0. *min_number_of_visits* should be >= 1 (default), *max_number_of_visits* can be None (no limit, default).

delay_btw_visits

Control by how many years consecutive visits of a patient are delayed. Multiple options are possible:

- float > 0 : regular spacing between all visits
- dictionary : {'min': float > 0, 'mean': float >= min, 'std': float > 0 [, 'max': float >= mean]}

Specify a Gaussian random spacing (truncated between min, and max if given) * function : n (int >= 1) => 1D numpy.ndarray[float > 0] of length *n* giving delay between visits (e.g.: 3 => [0.5, 1.5, 1.])

noise
[None or str in {'model', 'inherit_struct'} or DistributionFamily or dict or float or array-like[float]]

Wanted noise-model for the generated observations:

- Set noise to `None` will lead to patients follow the model exactly (no noise added).

- Set to `'inherit_struct'`, the noise added will follow the model noise structure and for Gaussian noise it will be computed from reconstruction errors on data & individual parameters provided.
- Set noise to `'model'`, the noise added will follow the model noise structure as well as its parameters.
- Set to a valid input for *noise_model_factory* to get the corresponding noise-model, e.g. set to `'bernoulli'`, to simulate Bernoulli realizations.
- Set a float will add for each feature's scores a noise of standard deviation the given float ('gaussian_scalar' noise).
- Set an array-like[float] (1D of length *n_features*) will add for the feature *j* a noise of standard deviation `noise[j]` ('gaussian_diagonal' noise).

<!> When you simulate data from an ordinal model, you HAVE to keep the default noise='inherit_struct' (default)
(or use 'model', which is the same in this case since there are no scaling parameter for ordinal noise)

number_of_subjects
[int > 0] Number of subject to simulate.

reparametrized_age_bounds
[tuple[float, float], optional (default None)] Define the minimum and maximum reparametrized ages of subjects included in the kernel estimation. See Notes section. Example: reparametrized_age_bounds = (65, 70)

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.

prefix
[str] Prefix appended to simulated patients' identifiers

Methods

`load_parameters`(parameters)	Update the algorithm's parameters by the ones in the given dictionary.
`run`(model, *args[, return_loss])	Main method, run the algorithm.
`run_impl`(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

`load_parameters`(*parameters: dict*)

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: AbstractModel*, **args*, *return_loss: bool = False*, ***extra_kwargs*) → Any

Main method, run the algorithm.

TODO fix proper abstract class method: input depends on algorithm… (esp. simulate != from others…)

Parameters

model
[`AbstractModel`] The used model.

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

return_loss
[bool (default False), keyword only] Should the algorithm return main output and optional loss output as a 2-tuple?

Returns

Depends on algorithm class: TODO change?

See also:

AbstractFitAlgo
AbstractPersonalizeAlgo
SimulationAlgorithm

run_impl(*model: AbstractModel*, *individual_parameters: IndividualParameters*, *data: Data*) → Tuple[Result, torch.FloatTensor | None]

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

<!> The *AbstractAlgo.run* signature is not respected for simulation algorithm… TODO: respect it… at least use (model, dataset, individual_parameters) signature…

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.

data
[Data] The data object.

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 subjects provided are used to estimate the joined distribution.

set_output_manager(*output_settings: OutputsSettings*) → None

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
```

(continues on next page)

(continued from previous page)

```
                }
>>> my_algo.set_output_manager(OutputsSettings(settings))
```

3.3.6 leaspy.algo.others: Other algorithms

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

constant_prediction_algo.ConstantPredictionAlgorithm(...)	ConstantPredictionAlgorithm is the algorithm that outputs a constant prediction
lme_fit.LMEFitAlgorithm(settings)	Calibration algorithm associated to LMEModel
lme_personalize.LMEPersonalizeAlgorithm(settin	Personalization algorithm associated to LMEModel

leaspy.algo.others.constant_prediction_algo.ConstantPredictionAlgorithm

class ConstantPredictionAlgorithm(*settings*)

Bases: AbstractAlgo

ConstantPredictionAlgorithm is the algorithm that outputs a constant prediction

It is associated to ConstantModel

TODO: it should be a child of *AbstractPersonalizeAlgorithm* (refactoring needed)

Parameters

settings

[AlgorithmSettings] The settings of constant prediction algorithm. It may define *prediction_type* (str):

- 'last': last value seen during calibration (even if NaN) [default],
- 'last_known': last non NaN value seen during calibration*§,
- '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.

Raises

LeaspyAlgoInputError

If any invalid setting for the algorithm

Methods

load_parameters(parameters)	Update the algorithm's parameters by the ones in the given dictionary.
run(model, *args[, return_loss])	Main method, run the algorithm.
run_impl(model, dataset)	Main method, refer to abstract definition in run().
set_output_manager(output_settings)	Not implemented.

load_parameters(*parameters: dict*)

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: AbstractModel*, **args*, *return_loss: bool = False*, ***extra_kwargs*) → Any

Main method, run the algorithm.

TODO fix proper abstract class method: input depends on algorithm… (esp. simulate != from others…)

Parameters

model
[AbstractModel] The used model.

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

return_loss
[bool (default False), keyword only] Should the algorithm return main output and optional loss output as a 2-tuple?

Returns

Depends on algorithm class: TODO change?

See also:

AbstractFitAlgo
AbstractPersonalizeAlgo
SimulationAlgorithm

run_impl(*model: ConstantModel*, *dataset: Dataset*)
Main method, refer to abstract definition in run().

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(*output_settings*)
Not implemented.

leaspy.algo.others.lme_fit.LMEFitAlgorithm

class LMEFitAlgorithm(*settings*)
Bases: 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

  Deprecated since version 1.2.

  You should rather define this directly as an hyperparameter of LME model.

- **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

`load_parameters`(parameters)	Update the algorithm's parameters by the ones in the given dictionary.
`run`(model, *args[, return_loss])	Main method, run the algorithm.
`run_impl`(model, dataset)	Main method, refer to abstract definition in `run()`.
`set_output_manager`(output_settings)	Not implemented.

load_parameters(*parameters: dict*)

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: AbstractModel*, **args*, *return_loss: bool = False*, ***extra_kwargs*) → Any

Main method, run the algorithm.

TODO fix proper abstract class method: input depends on algorithm… (esp. simulate != from others…)

Parameters

model
[AbstractModel] The used model.

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

return_loss
[bool (default False), keyword only] Should the algorithm return main output and optional loss output as a 2-tuple?

Returns

Depends on algorithm class: TODO change?

See also:

AbstractFitAlgo
AbstractPersonalizeAlgo
SimulationAlgorithm

run_impl(*model: LMEModel*, *dataset: 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

2-tuple:

- None
- noise scale (std-dev), scalar

set_output_manager(*output_settings*)

Not implemented.

leaspy.algo.others.lme_personalize.LMEPersonalizeAlgorithm

class LMEPersonalizeAlgorithm(*settings: AlgorithmSettings*)

Bases: AbstractAlgo

Personalization algorithm associated to LMEModel

TODO: it should be a child of *AbstractPersonalizeAlgorithm* (refactoring needed)

Parameters

settings
[AlgorithmSettings] Algorithm settings (none yet). Most LME parameters are defined within LME model and LME fit algorithm.

Attributes

name
['lme_personalize']

Methods

load_parameters(parameters)	Update the algorithm's parameters by the ones in the given dictionary.
run(model, *args[, return_loss])	Main method, run the algorithm.
run_impl(model, dataset)	Main method, refer to abstract definition in run().
set_output_manager(output_settings)	Not implemented.

load_parameters(*parameters: dict*)

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}}
```

(continues on next page)

(continued from previous page)

```
>>> 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: AbstractModel*, **args*, *return_loss: bool = False*, ***extra_kwargs*) → Any

Main method, run the algorithm.

TODO fix proper abstract class method: input depends on algorithm... (esp. simulate != from others...)

Parameters

model
[`AbstractModel`] The used model.

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

return_loss
[bool (default False), keyword only] Should the algorithm return main output and optional loss output as a 2-tuple?

Returns

Depends on algorithm class: TODO change?

See also:

AbstractFitAlgo
AbstractPersonalizeAlgo
SimulationAlgorithm

run_impl(*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`(*output_settings*)
Not implemented.

3.4 leaspy.samplers: Samplers

Samplers used by the *MCMC* algorithms.

`AbstractSampler`(name, shape, *[, ...])	Abstract sampler class.
`AbstractPopulationSampler`(name, shape, *[, ...])	Abstract class for samplers of population random variables.
`AbstractIndividualSampler`(name, shape, *, ...)	Abstract class for samplers of individual random variables.
`IndividualGibbsSampler`(name, shape, *, ...)	Gibbs sampler for individual variables.
`PopulationGibbsSampler`(name, shape, *, scale)	Gibbs sampler for population variables.
`PopulationFastGibbsSampler`(name, shape, *, scale)	Fast Gibbs sampler for population variables.
`PopulationMetropolisHastingsSampler`(name, ...)	Metropolis-Hastings sampler for population variables.

3.4.1 leaspy.samplers.AbstractSampler

class AbstractSampler(*name: str*, *shape: Tuple[int, ...]*, *, *acceptation_history_length: int = 25*)

Bases: `ABC`

Abstract sampler class.

Parameters

name
[`str`] The name of the random variable to sample.

shape
[`tuple` of `int`] The shape of the random variable to sample.

acceptation_history_length
[`int` > 0 (default 25)] Deepness (= number of iterations) of the history kept for computing the mean acceptation rate. (It is the same for population or individual variables.)

Raises

`LeaspyModelInputError`

Attributes

name
[`str`] Name of variable

shape
[`tuple` of `int`] Shape of variable

acceptation_history_length
[`int`] Deepness (= number of iterations) of the history kept for computing the mean acceptation rate. (Same for population or individual variables by default.)

acceptation_history
[torch.Tensor] History of binary acceptations to compute mean acceptation rate for the sampler in MCMC-SAEM algorithm. It keeps the history of the last *acceptation_history_length* steps.

Methods

sample(dataset, model, realizations, ...)	Sample new realization (either population or individual) for a given CollectionRealization state, Dataset, AbstractModel, and temperature.

abstract sample(*dataset: Dataset*, *model: AbstractModel*, *realizations: CollectionRealization*, *temperature_inv: float*, ***attachment_computation_kws*) → Tuple[Tensor, Tensor]

Sample new realization (either population or individual) for a given CollectionRealization state, Dataset, AbstractModel, and temperature.

<!> Modifies in-place the realizations object, <!> as well as the model through its *update_MCMC_toolbox* for population variables.

Parameters

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

model
[AbstractModel] Model for loss computations and updates

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

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

****attachment_computation_kws**
Optional keyword arguments for attachment computations. As of now, we only use it for individual variables, and only *attribute_type*. It is used to know whether to compute attachments from the MCMC toolbox (esp. during fit) or to compute it from regular model parameters (esp. during personalization in mean/mode realization)

Returns

attachment, regularity_var
[torch.Tensor] The attachment and regularity tensors (only for the current variable) at the end of this sampling step (globally or per individual, depending on variable type). The tensors are 0D for population variables, or 1D for individual variables (with length *n_individuals*).

abstract property shape_acceptation: Tuple[int, ...]

Return the shape of acceptation tensor for a single iteration.

Returns

tuple of int
The shape of the acceptation history.

3.4.2 leaspy.samplers.AbstractPopulationSampler

class AbstractPopulationSampler(*name: str*, *shape: Tuple[int, ...]*, ***, *acceptation_history_length: int = 25*, *mask: Tensor | None = None*)

Bases: AbstractSampler

Abstract class for samplers of population random variables.

Parameters

name
[str] The name of the random variable to sample.

shape
[tuple of int] The shape of the random variable to sample.

acceptation_history_length
[int > 0 (default 25)] Deepness (= number of iterations) of the history kept for computing the mean acceptation rate. (It is the same for population or individual variables.)

mask
[torch.Tensor, optional] If not None, mask should be 0/1 tensor indicating the sampling variable to adapt variance from 1 indices are kept for sampling while 0 are excluded.

Attributes

name
[str] Name of variable

shape
[tuple of int] Shape of variable

acceptation_history_length
[int] Deepness (= number of iterations) of the history kept for computing the mean acceptation rate. (It is the same for population or individual variables.)

acceptation_history
[torch.Tensor] History of binary acceptations to compute mean acceptation rate for the sampler in MCMC-SAEM algorithm. It keeps the history of the last *acceptation_history_length* steps.

mask
[torch.Tensor of *obj*:bool, optional] If not None, mask should be 0/1 tensor indicating the sampling variable to adapt variance from 1 (True) indices are kept for sampling while 0 (False) are excluded.

Methods

sample(dataset, model, realizations, ...)	Sample new realization (either population or individual) for a given CollectionRealization state, Dataset, AbstractModel, and temperature.

abstract sample(*dataset: Dataset*, *model: AbstractModel*, *realizations: CollectionRealization*, *temperature_inv: float*, ***attachment_computation_kws*) → Tuple[Tensor, Tensor]

Sample new realization (either population or individual) for a given CollectionRealization state, Dataset, AbstractModel, and temperature.

<!> Modifies in-place the realizations object, <!> as well as the model through its *update_MCMC_toolbox* for population variables.

Parameters

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

model
[AbstractModel] Model for loss computations and updates

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

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

****attachment_computation_kws**
Optional keyword arguments for attachment computations. As of now, we only use it for individual variables, and only *attribute_type*. It is used to know whether to compute attachments from the MCMC toolbox (esp. during fit) or to compute it from regular model parameters (esp. during personalization in mean/mode realization)

Returns

attachment, regularity_var
[torch.Tensor] The attachment and regularity tensors (only for the current variable) at the end of this sampling step (globally or per individual, depending on variable type). The tensors are 0D for population variables, or 1D for individual variables (with length *n_individuals*).

abstract property shape_acceptation: Tuple[int, ...]

Return the shape of acceptation tensor for a single iteration.

Returns

tuple of int
The shape of the acceptation history.

3.4.3 leaspy.samplers.AbstractIndividualSampler

class AbstractIndividualSampler(*name: str*, *shape: Tuple[int, ...]*, ***, *n_patients: int*, *acceptation_history_length: int = 25*)

Bases: AbstractSampler

Abstract class for samplers of individual random variables.

Parameters

name
[str] The name of the random variable to sample.

shape
[tuple of int] The shape of the random variable to sample.

n_patients
[int] Number of patients.

acceptation_history_length
[int > 0 (default 25)] Deepness (= number of iterations) of the history kept for computing the mean acceptation rate. (It is the same for population or individual variables.)

Attributes

name
[str] Name of variable

shape
[tuple of int] Shape of variable

n_patients
[int] Number of patients.

acceptation_history_length
[int] Deepness (= number of iterations) of the history kept for computing the mean acceptation rate. (It is the same for population or individual variables.)

acceptation_history
[torch.Tensor] History of binary acceptations to compute mean acceptation rate for the sampler in MCMC-SAEM algorithm. It keeps the history of the last *acceptation_history_length* steps.

Methods

sample(dataset, model, realizations, ...)	Sample new realization (either population or individual) for a given CollectionRealization state, Dataset, AbstractModel, and temperature.

abstract sample(*dataset: Dataset*, *model: AbstractModel*, *realizations: CollectionRealization*, *temperature_inv: float*, ***attachment_computation_kws*) → Tuple[Tensor, Tensor]

Sample new realization (either population or individual) for a given CollectionRealization state, Dataset, AbstractModel, and temperature.

<!> Modifies in-place the realizations object, <!> as well as the model through its *update_MCMC_toolbox* for population variables.

Parameters

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

model
[AbstractModel] Model for loss computations and updates

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

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

****attachment_computation_kws**
Optional keyword arguments for attachment computations. As of now, we only use it for individual variables, and only *attribute_type*. It is used to know whether to compute attachments from the MCMC toolbox (esp. during fit) or to compute it from regular model parameters (esp. during personalization in mean/mode realization)

Returns

attachment, regularity_var
[torch.Tensor] The attachment and regularity tensors (only for the current variable) at the end of this sampling step (globally or per individual, depending on variable type). The tensors are 0D for population variables, or 1D for individual variables (with length *n_individuals*).

abstract property shape_acceptation: Tuple[int, ...]

Return the shape of acceptation tensor for a single iteration.

Returns

tuple of int
The shape of the acceptation history.

3.4.4 leaspy.samplers.IndividualGibbsSampler

class IndividualGibbsSampler(*name: str*, *shape: tuple*, ***, *n_patients: int*, *scale: float | FloatTensor*, *random_order_dimension: bool = True*, *mean_acceptation_rate_target_bounds: Tuple[float, float] = (0.2, 0.4)*, *adaptive_std_factor: float = 0.1*, ***base_sampler_kws*)

Bases: GibbsSamplerMixin, AbstractIndividualSampler

Gibbs sampler for individual variables.

Individual variables are handled with a grouped Gibbs sampler. There is currently no other sampler available for individual variables.

Parameters

name
[str] The name of the random variable to sample.

shape
[tuple of int] The shape of the random variable to sample.

n_patients
[int] Number of patients.

scale
[float > 0 or torch.FloatTensor > 0] An approximate scale for the variable. It will be used to scale the initial adaptive std-dev used in sampler. An extra 1% factor will be applied on top of this scale (STD_SCALE_FACTOR) Note that if you pass a torch tensor, its shape should be compatible with shape of the variable.

random_order_dimension
[bool (default True)] This parameter controls whether we randomize the order of indices during the sampling loop. Article https://proceedings.neurips.cc/paper/2016/hash/e4da3b7fbbce2345d7772b0674a318d5-Abstract.html gives a rationale on why we should activate this flag.

mean_acceptation_rate_target_bounds
[:obj:`tuple`[lower_bound: float, upper_bound: float] with 0 < lower_bound < upper_bound < 1] Bounds on mean acceptation rate. Outside this range, the adaptation of the std-dev of sampler is triggered so to maintain a target acceptation rate in between these too bounds (e.g: ~30%).

adaptive_std_factor
[float in]0, 1[] Factor by which we increase or decrease the std-dev of sampler when we are out of the custom bounds for the mean acceptation rate. We decrease it by *1 - factor* if too low, and increase it with *1 + factor* if too high.

****base_sampler_kws**
Keyword arguments passed to *AbstractSampler* init method. In particular, you may pass the *acceptation_history_length* hyperparameter.

Attributes

shape_acceptation
shape_adapted_std
Shape of adaptative variance.

Methods

sample(data, model, realizations, ...)	For each individual variable, compute current patient-batched attachment and regularity.
validate_scale(scale)	Validate user provided scale in float or torch.Tensor form.

sample(*data: Dataset*, *model: AbstractModel*, *realizations: CollectionRealization*, *temperature_inv: float*, ***attachment_computation_kws*) → Tuple[Tensor, Tensor]

For each individual 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]

****attachment_computation_kws**
Optional keyword arguments for attachment computations. As of now, we only use it for individual variables, and only *attribute_type*. It is used to know whether to compute attachments from the MCMC toolbox (esp. during fit) or to compute it from regular model parameters (esp. during personalization in mean/mode realization)

Returns

attachment, regularity_var
[torch.Tensor 1D (n_individuals,)] The attachment and regularity (only for the current variable) at the end of this sampling step, per individual.

property shape_acceptation: Tuple[int, ...]

Return the shape of acceptation tensor for a single iteration.

Returns

tuple of int
The shape of the acceptation history.

property shape_adapted_std: tuple

Shape of adaptative variance.

validate_scale(*scale: float | Tensor*) → Tensor

Validate user provided scale in `float` or `torch.Tensor` form.

Scale of variable should always be positive (component-wise if multidimensional).

Parameters

scale
[`float` or `torch.Tensor`] The scale to be validated.

Returns

torch.Tensor
Valid scale.

3.4.5 leaspy.samplers.PopulationGibbsSampler

class PopulationGibbsSampler(*name: str*, *shape: tuple*, *, *scale: float | FloatTensor*, *random_order_dimension: bool = True*, *mean_acceptation_rate_target_bounds: Tuple[float, float] = (0.2, 0.4)*, *adaptive_std_factor: float = 0.1*, ***base_sampler_kws*)

Bases: `AbstractPopulationGibbsSampler`

Gibbs sampler for population variables.

The sampling is done iteratively for all coordinate values.

Parameters

name
[`str`] The name of the random variable to sample.

shape
[`tuple` of `int`] The shape of the random variable to sample.

scale
[`float` > 0 or `torch.FloatTensor` > 0] An approximate scale for the variable. It will be used to scale the initial adaptive std-dev used in sampler. An extra 1% factor will be applied on top of this scale (STD_SCALE_FACTOR) Note that if you pass a torch tensor, its shape should be compatible with shape of the variable.

random_order_dimension
[`bool` (default True)] This parameter controls whether we randomize the order of indices during the sampling loop. Article https://proceedings.neurips.cc/paper/2016/hash/e4da3b7fbbce2345d7772b0674a318d5-Abstract.html gives a rationale on why we should activate this flag.

mean_acceptation_rate_target_bounds
[:obj:`tuple`[lower_bound: float, upper_bound: float] with 0 < lower_bound < upper_bound < 1] Bounds on mean acceptation rate. Outside this range, the adaptation

of the std-dev of sampler is triggered so to maintain a target acceptation rate in between these too bounds (e.g: ~30%).

adaptive_std_factor
[float in]0, 1[] Factor by which we increase or decrease the std-dev of sampler when we are out of the custom bounds for the mean acceptation rate. We decrease it by *1 - factor* if too low, and increase it with *1 + factor* if too high.

****base_sampler_kws**
Keyword arguments passed to `AbstractSampler.__init__()` method. In particular, you may pass the *acceptation_history_length* hyperparameter.

Attributes

shape_acceptation
shape_adapted_std
Shape of adaptative variance.

Methods

sample(data, model, realizations, ...)	For each dimension (1D or 2D) of the population variable, compute current attachment and regularity.
validate_scale(scale)	Validate user provided scale in float or torch.Tensor form.

sample(*data: Dataset*, *model: AbstractModel*, *realizations: CollectionRealization*, *temperature_inv: float*, ***attachment_computation_kws*) → Tuple[Tensor, Tensor]

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] Dataset used for sampling.

model
[AbstractModel] Model for which to sample a random variable.

realizations
[CollectionRealization] Realization state.

temperature_inv
[float > 0] The temperature to use.

****attachment_computation_kws**
Currently not used for population parameters.

Returns

attachment, regularity_var
[torch.Tensor 0D (scalars)] The attachment and regularity (only for the current variable) at the end of this sampling step (summed on all individuals).

property shape_acceptation: `Tuple[int, ...]`

Return the shape of acceptation tensor for a single iteration.

Returns

`tuple` **of** `int`
The shape of the acceptation history.

property shape_adapted_std: `tuple`

Shape of adaptative variance.

validate_scale(*scale: float | Tensor*) → Tensor

Validate user provided scale in `float` or `torch.Tensor` form.

If necessary, scale is casted to a `torch.Tensor`.

Parameters

scale
[`float` or `torch.Tensor`] The scale to be validated.

Returns

`torch.Tensor`
Valid scale.

3.4.6 leaspy.samplers.PopulationFastGibbsSampler

class PopulationFastGibbsSampler(*name: str*, *shape: tuple*, *, *scale: float | FloatTensor*, *random_order_dimension: bool = True*, *mean_acceptation_rate_target_bounds: Tuple[float, float] = (0.2, 0.4)*, *adaptive_std_factor: float = 0.1*, ***base_sampler_kws*)

Bases: `AbstractPopulationGibbsSampler`

Fast Gibbs sampler for population variables.

Note: The sampling batches along the dimensions except the first one. This speeds up sampling process for 2 dimensional parameters.

Parameters

name
[`str`] The name of the random variable to sample.

shape
[`tuple` of `int`] The shape of the random variable to sample.

scale
[`float` > 0 or `torch.FloatTensor` > 0] An approximate scale for the variable. It will be used to scale the initial adaptive std-dev used in sampler. An extra 1% factor will be applied on top of this scale (STD_SCALE_FACTOR) Note that if you pass a torch tensor, its shape should be compatible with shape of the variable.

random_order_dimension
[`bool` (default True)] This parameter controls whether we randomize the order of indices during the sampling loop. Article https://proceedings.neurips.cc/paper/2016/hash/e4da3b7fbbce2345d7772b0674a318d5-Abstract.html gives a rationale on why we should activate this flag.

mean_acceptation_rate_target_bounds
[:obj:`tuple`[lower_bound: float, upper_bound: float] with 0 < lower_bound < upper_bound < 1] Bounds on mean acceptation rate. Outside this range, the adaptation of the std-dev of sampler is triggered so to maintain a target acceptation rate in between these too bounds (e.g: ~30%).

adaptive_std_factor
[`float` in]0, 1[] Factor by which we increase or decrease the std-dev of sampler when we are out of the custom bounds for the mean acceptation rate. We decrease it by *1 - factor* if too low, and increase it with *1 + factor* if too high.

****base_sampler_kws**
Keyword arguments passed to *AbstractSampler* init method. In particular, you may pass the *acceptation_history_length* hyperparameter.

Attributes

shape_acceptation
shape_adapted_std
Shape of adaptative variance.

Methods

`sample`(data, model, realizations, ...)	For each dimension (1D or 2D) of the population variable, compute current attachment and regularity.
`validate_scale`(scale)	Validate user provided scale in `float` or `torch.Tensor` form.

sample(*data: Dataset*, *model: AbstractModel*, *realizations: CollectionRealization*, *temperature_inv: float*, ***attachment_computation_kws*) → Tuple[Tensor, Tensor]

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`] Dataset used for sampling.

model
[`AbstractModel`] Model for which to sample a random variable.

realizations
[`CollectionRealization`] Realization state.

temperature_inv
[`float` > 0] The temperature to use.

****attachment_computation_kws**
Currently not used for population parameters.

Returns

attachment, regularity_var
[`torch.Tensor` 0D (scalars)] The attachment and regularity (only for the current variable) at the end of this sampling step (summed on all individuals).

property shape_acceptation: Tuple[int, ...]

Return the shape of acceptation tensor for a single iteration.

Returns

tuple of int
The shape of the acceptation history.

property shape_adapted_std: tuple

Shape of adaptative variance.

validate_scale(*scale: float | Tensor*) → Tensor

Validate user provided scale in `float` or `torch.Tensor` form.

If necessary, scale is casted to a `torch.Tensor`.

Parameters

scale
[`float` or `torch.Tensor`] The scale to be validated.

Returns

torch.Tensor
Valid scale.

3.4.7 leaspy.samplers.PopulationMetropolisHastingsSampler

class PopulationMetropolisHastingsSampler(*name: str*, *shape: tuple*, *, *scale: float | FloatTensor*, *random_order_dimension: bool = True*, *mean_acceptation_rate_target_bounds: Tuple[float, float] = (0.2, 0.4)*, *adaptive_std_factor: float = 0.1*, ***base_sampler_kws*)

Bases: `AbstractPopulationGibbsSampler`

Metropolis-Hastings sampler for population variables.

Note: The sampling is done for all values at once. This speeds up considerably sampling but usually requires more iterations.

Parameters

name
[`str`] The name of the random variable to sample.

shape
[`tuple` of `int`] The shape of the random variable to sample.

scale
[`float` > 0 or `torch.FloatTensor` > 0] An approximate scale for the variable. It will be used to scale the initial adaptive std-dev used in sampler. An extra 1% factor will be applied on top of this scale (STD_SCALE_FACTOR) Note that if you pass a torch tensor, its shape should be compatible with shape of the variable.

random_order_dimension
[`bool` (default True)] This parameter controls whether we randomize the order of indices during the sampling loop. Article https://proceedings.neurips.cc/paper/2016/hash/

e4da3b7fbbce2345d7772b0674a318d5-Abstract.html gives a rationale on why we should activate this flag.

mean_acceptation_rate_target_bounds
[:obj:`tuple`[lower_bound: float, upper_bound: float] with 0 < lower_bound < upper_bound < 1] Bounds on mean acceptation rate. Outside this range, the adaptation of the std-dev of sampler is triggered so to maintain a target acceptation rate in between these too bounds (e.g: ~30%).

adaptive_std_factor
[`float` in]0, 1[] Factor by which we increase or decrease the std-dev of sampler when we are out of the custom bounds for the mean acceptation rate. We decrease it by *1 - factor* if too low, and increase it with *1 + factor* if too high.

****base_sampler_kws**
Keyword arguments passed to *AbstractSampler* init method. In particular, you may pass the *acceptation_history_length* hyperparameter.

Attributes

shape_acceptation
shape_adapted_std
Shape of adaptative variance.

Methods

`sample`(data, model, realizations, ...)	For each dimension (1D or 2D) of the population variable, compute current attachment and regularity.
`validate_scale`(scale)	Validate user provided scale in `float` or `torch.Tensor` form.

sample(*data: Dataset*, *model: AbstractModel*, *realizations: CollectionRealization*, *temperature_inv: float*, ***attachment_computation_kws*) → Tuple[Tensor, Tensor]

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`] Dataset used for sampling.

model
[`AbstractModel`] Model for which to sample a random variable.

realizations
[`CollectionRealization`] Realization state.

temperature_inv
[`float` > 0] The temperature to use.

****attachment_computation_kws**
Currently not used for population parameters.

Returns

attachment, regularity_var
[torch.Tensor 0D (scalars)] The attachment and regularity (only for the current variable) at the end of this sampling step (summed on all individuals).

property shape_acceptation: Tuple[int, ...]

Return the shape of acceptation tensor for a single iteration.

Returns

tuple of int
The shape of the acceptation history.

property shape_adapted_std: tuple

Shape of adaptative variance.

validate_scale(*scale: float | Tensor*) → Tensor

Validate user provided scale in `float` or `torch.Tensor` form.

If necessary, scale is casted to a `torch.Tensor`.

Parameters

scale
[`float` or `torch.Tensor`] The scale to be validated.

Returns

torch.Tensor
Valid scale.

sampler_factory(sampler, variable_type, **kwargs)	Factory for Samplers.

3.4.8 leaspy.samplers.sampler_factory

sampler_factory(*sampler: str | AbstractSampler*, *variable_type: VariableType*, ***kwargs*) → AbstractSampler

Factory for Samplers.

Parameters

sampler
[`AbstractSampler` or `str`] If an instance of a subclass of `AbstractSampler`, returns the instance (no copy). If a string, returns a new instance of the appropriate class (with optional parameters *kwargs*).

variable_type
[`VariableType`] The type of random variable that the sampler is supposed to sample.

****kwargs**
Optional parameters for initializing the requested Sampler (not used if input is a subclass of `AbstractSampler`).

Returns

AbstractSampler
The desired sampler.

Raises

LeaspyAlgoInputError:
If the sampler provided is not supported.

3.5 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.

3.5.1 leaspy.datasets.loader.Loader

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

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-putamen-train_and_test'}] 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 differentiate *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 already calibrated on the synthetic dataset corresponding to the name of the instance.

Parameters

ip_name
[{'alzheimer-multivariate', 'parkinson-multivariate', 'parkinson-putamen-train'}] Name of the individual parameters.

Returns

IndividualParameters
Leaspy instance with a model already calibrated.

static load_leaspy_instance(*instance_name*)

Load a Leaspy instance with a model already calibrated on the synthetic dataset corresponding to the name of the instance.

Parameters

instance_name
[{'alzheimer-multivariate', 'parkinson-multivariate', 'parkinson-putamen-train'}] Name of the instance.

Returns

Leaspy
Leaspy instance with a model already calibrated.

3.6 leaspy.io: Inputs / Outputs

Containers classes used as input / outputs in the *Leaspy* package.

3.6.1 leaspy.io.data: Data containers

data.Data()	Main data container for a collection of individuals
dataset.Dataset(data, *[, no_warning])	Data container based on torch.Tensor, used to run algorithms.

leaspy.io.data.data.Data

class Data

Bases: Iterable

Main data container for a collection of individuals

It can be iterated over and sliced, both of these operations being applied to the underlying *individuals* attribute.

Attributes

individuals
[Dict[IDType, IndividualData]] Included individuals and their associated data

iter_to_idx
[Dict[int, IDType]] Maps an integer index to the associated individual ID

headers
[List[FeatureType]] Feature names

dimension
[int] Number of features

n_individuals
[int] Number of individuals

n_visits
[int] Total number of visits

cofactors
[List[FeatureType]] Feature names corresponding to cofactors

Methods

from_csv_file(path, **kws)	Create a *Data* object from a CSV file.
from_dataframe(df, **kws)	Create a *Data* object from a pandas.DataFrame.
from_individual_values(indices, timepoints, ...)	Construct *Data* from a collection of individual data points
from_individuals(individuals, headers)	Construct *Data* from a list of individuals
load_cofactors(df, *[, cofactors])	Load cofactors from a *pandas.DataFrame* to the *Data* object
to_dataframe(*[, cofactors, reset_index])	Convert the Data object to a pandas.DataFrame

property cofactors: List[str]

Feature names corresponding to cofactors

property dimension: int | None

Number of features

static from_csv_file(*path: str*, ***kws*) → Data

Create a *Data* object from a CSV file.

Parameters

path
[str] Path to the CSV file to load (with extension)

****kws**
Keyword arguments that are sent to `CSVDataReader`

Returns

`Data`

static from_dataframe(*df: DataFrame*, ***kws*) → Data

Create a *Data* object from a `pandas.DataFrame`.

Parameters

df
[`pandas.DataFrame`] Dataframe containing ID, TIME and features.

****kws**
Keyword arguments that are sent to `DataframeDataReader`

Returns

`Data`

static from_individual_values(*indices: List[str]*, *timepoints: List[List[float]]*, *values: List[List[List[float]]]*, *headers: List[str]*) → Data

Construct *Data* from a collection of individual data points

Parameters

indices
[List[IDType]] List of the individuals' unique ID

timepoints
[List[List[float]]] For each individual `i`, list of timepoints associated with the observations. The number of such timepoints is noted `n_timepoints_i`

values
[List[array-like[float, 2D]]] For each individual `i`, two-dimensional array-like object containing observed data points. Its expected shape is `(n_timepoints_i, n_features)`

headers
[List[FeatureType]] Feature names. The number of features is noted `n_features`

Returns

`Data`

static from_individuals(*individuals: List[IndividualData]*, *headers: List[str]*) → Data

Construct *Data* from a list of individuals

Parameters

individuals
[List[IndividualData]] List of individuals

headers
[List[FeatureType]] List of feature names

Returns

Data

load_cofactors(*df: DataFrame*, *, *cofactors: List[str] | None = None*) → None

Load cofactors from a *pandas.DataFrame* to the *Data* object

Parameters

df
[pandas.DataFrame] The dataframe where the cofactors are stored. Its index should be ID, the identifier of subjects and it should uniquely index the dataframe (i.e. one row per individual).

cofactors
[List[FeatureType] or None (default)] Names of the column(s) of df which shall be loaded as cofactors. If None, all the columns from the input dataframe will be loaded as cofactors.

Raises

LeaspyDataInputError

property n_individuals: int

Number of individuals

property n_visits: int

Total number of visits

to_dataframe(*, *cofactors: List[str] | str | None = None*, *reset_index: bool = True*) → DataFrame

Convert the Data object to a pandas.DataFrame

Parameters

cofactors
[List[FeatureType], 'all', or None (default None)] Cofactors to include in the DataFrame. If None (default), no cofactors are included. If "all", all the available cofactors are included.

reset_index
[bool (default True)] Whether to reset index levels in output.

Returns

pandas.DataFrame
A DataFrame containing the individuals' ID, timepoints and associated observations (optional - and cofactors).

Raises

LeaspyDataInputError
LeaspyTypeError

leaspy.io.data.dataset.Dataset

class Dataset(*data: Data*, *, *no_warning: bool = False*)

Bases: object

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

Parameters

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

no_warning
[bool (default False)] Whether to deactivate warnings that are emitted by methods of this dataset instance. We may want to deactivate them because we rebuild a dataset per individual in scipy minimize. Indeed, all relevant warnings certainly occurred for the overall dataset.

Raises

LeaspyInputError
if data, model or algo are not compatible together.

Attributes

headers
[list[str]] Features names

dimension
[int] Number of features

n_individuals
[int] Number of individuals

indices
[list[ID]] Order of patients

n_visits_per_individual
[list[int]] Number of visits per individual

n_visits_max
[int] Maximum number of visits for one individual

n_visits
[int] Total number of visits

n_observations_per_ind_per_ft
[torch.LongTensor, shape (n_individuals, dimension)] Number of observations (not taking into account missing values) per individual per feature

n_observations_per_ft
[torch.LongTensor, shape (dimension,)] Total number of observations per feature

n_observations
[int] Total number of observations

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

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

mask
[`torch.FloatTensor`, shape (n_individuals, n_visits_max, 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.FloatTensor`, shape (dimension,)] Sum of all non-nan squared values, feature per feature

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

no_warning
[bool (default False)] Whether to deactivate warnings that are emitted by methods of this dataset instance. We may want to deactivate them because we rebuild a dataset per individual in scipy minimize. Indeed, all relevant warnings certainly occurred for the overall dataset.

_one_hot_encoding
[Dict[sf: bool, `torch.LongTensor`]] Values of patients for each visit for each feature, but tensorized into a one-hot encoding (pdf or sf) Shapes of tensors are (n_individuals, n_visits_max, dimension, max_ordinal_level [-1 when *sf=True*])

Methods

`get_one_hot_encoding`(*, sf, ordinal_infos)	Builds the one-hot encoding of ordinal data once and for all and returns it.
`get_times_patient`(i)	Get ages for patient number `i`
`get_values_patient`(i, *[, adapt_for_model])	Get values for patient number `i`, with nans.
`move_to_device`(device)	Moves the dataset to the specified device.
`to_pandas`()	Convert dataset to a *DataFrame* with ['ID', 'TIME'] index.

`get_one_hot_encoding`(*, *sf: bool*, *ordinal_infos: Dict[str, Any]*)

Builds the one-hot encoding of ordinal data once and for all and returns it.

Parameters

sf
[bool] Whether the vector should be the survival function [1(X > l), l=0..max_level-1] instead of the probability density function [1(X=l), l=0..max_level]

ordinal_infos
[dict[str, Any]] All the hyperparameters concerning ordinal modelling (in particular maximum level per features)

Returns

One-hot encoding of data values.

`get_times_patient`(*i: int*) → FloatTensor

Get ages for patient number `i`

Parameters

i
[int] The index of the patient (<!> not its identifier)

Returns

`torch.Tensor`, **shape (n_obs_of_patient,)**
Contains float

`get_values_patient`(*i: int*, *, *adapt_for_model=None*) → FloatTensor

Get values for patient number `i`, with nans.

Parameters

i
[int] The index of the patient (<!> not its identifier)

adapt_for_model
[None (default) or AbstractModel] The values returned are suited for this model. In particular:

- For model with *noise_model='ordinal'* will return one-hot-encoded values [P(X = l), l=0..ordinal_max_level]
- For model with *noise_model='ordinal_ranking'* will return survival function values [P(X > l), l=0..ordinal_max_level-1]

If None, we return the raw values, whatever the model is.

Returns

`torch.Tensor`, **shape (n_obs_of_patient, dimension [, extra_dimension_for_ordinal_models])**
Contains float or nans

`move_to_device`(*device: device*) → None

Moves the dataset to the specified device.

Parameters

device
[torch.device]

`to_pandas`() → DataFrame

Convert dataset to a *DataFrame* with ['ID', 'TIME'] index.

Returns

`pandas.DataFrame`

`class Data`

Main data container for a collection of individuals

It can be iterated over and sliced, both of these operations being applied to the underlying *individuals* attribute.

Attributes

individuals
[Dict[IDType, IndividualData]] Included individuals and their associated data

iter_to_idx
[Dict[int, IDType]] Maps an integer index to the associated individual ID

headers
[List[FeatureType]] Feature names

`dimension`
[int] Number of features

n_individuals
[int] Number of individuals

n_visits
[int] Total number of visits

cofactors
[List[FeatureType]] Feature names corresponding to cofactors

Methods

from_csv_file(path, **kws)	Create a *Data* object from a CSV file.
from_dataframe(df, **kws)	Create a *Data* object from a pandas.DataFrame.
from_individual_values(indices, timepoints, ...)	Construct *Data* from a collection of individual data points
from_individuals(individuals, headers)	Construct *Data* from a list of individuals
load_cofactors(df, *[, cofactors])	Load cofactors from a *pandas.DataFrame* to the *Data* object
to_dataframe(*[, cofactors, reset_index])	Convert the Data object to a pandas.DataFrame

3.6.2 leaspy.io.settings: Settings classes

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 ModelSettings(*path_to_model_settings_or_dict: str | dict*)

Bases: object

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

Parameters

path_to_model_settings_or_dict
[dict or str]

- If a str: path to a json file containing model settings
- If a dict: content of model settings

Raises

LeaspyModelInputError

leaspy.io.settings.algorithm_settings.AlgorithmSettings

class AlgorithmSettings(*name: str*, ***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 LME model only)
- **For *personalize* algorithms:**
  - 'scipy_minimize'
  - 'mean_real'
  - 'mode_real'
  - 'constant_prediction' (for constant model only)
  - 'lme_personalize' (for LME model only)
- **For *simulate* algorithms:**
  - 'simulation'

****kwargs**
[any]

Depending on the algorithm you are setting up, various parameters are possible (not exhaustive):

- **seed**
  [int, optional, default None] Used for stochastic algorithms.
- **model_initialization_method**
  [str, optional] For **fit** algorithms only, 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 leaspy.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.

- **use_jacobian**
  [bool, optional, default True] Used in `scipy_minimize` algorithm to perform a *L-BFGS* instead of a *Powell* algorithm.
- **n_jobs**
  [int, optional, default 1] Used in `scipy_minimize` algorithm to accelerate calculation with parallel derivation using joblib.
- **progress_bar**
  [bool, optional, default True] Used to display a progress bar during computation.
- **device: str or torch.device, optional**
  Specifies on which device the algorithm will run. Only 'cpu' and 'cuda' are supported for this argument. Only `'mcmc_saem'`, `'mean_real'` and `'mode_real'` algorithms support this setting.

For the complete list of the available parameters for a given algorithm, please directly refer to its documentation.

Raises

`LeaspyAlgoInputError`

See also:

`leaspy.algo`

Notes

For developers: use `_dynamic_default_parameters` to dynamically set some default parameters, depending on other parameters that were set, while these *dynamic* parameters were not set.
Example:
you could want to set burn in iterations or annealing iterations as fractions of non-default number of iterations given.
Format:

```
{algo_name: [
    (functional_condition_to_trigger_dynamic_setting(kwargs),
    {
        nested_keys_of_dynamic_setting: dynamic_value(kwargs)
    })
]}
```

Attributes

name
[str] 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 `leaspy.algo`).

seed
[int, optional, default None] Used for stochastic algorithms.

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

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

device
[str (or torch.device), optional, default 'cpu'] Used to specify on which device the algorithm will run. This should either be: 'cpu' or 'cuda' and is only supported in specific algorithms (inheriting *AlgoWithDeviceMixin*). Note that specifying an indexed CUDA device (such as 'cuda:1') is not supported. In order to specify the precise cuda device index, one should use the *CUDA_VISIBLE_DEVICES* environment variable.

Methods

`check_consistency`()	Check internal consistency of algorithm settings and warn or raise a *LeaspyAlgoInputError* if not.
`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])	Use this method to monitor the convergence of a model calibration.

property algo_class

Class of the algorithm derived from its name (shorthand).

check_consistency() → None

Check internal consistency of algorithm settings and warn or raise a *LeaspyAlgoInputError* if not.

classmethod load(*path_to_algorithm_settings: str*)

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.

Raises

LeaspyAlgoInputError
if anything is invalid in algo settings

Examples

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

save(*path: str*, ***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. Default: dict(indent=2)

Examples

```
>>> from leaspy import AlgorithmSettings
>>> settings = AlgorithmSettings('scipy_minimize', seed=42)
>>> settings.save('outputs/scipy_minimize-settings.json')
```

set_logs(*path: str | None = None*, ***kwargs*)

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

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

Parameters

path
[str, optional] The path of the folder to store the graphs and csv files. No data will be saved if it is None, as well as save_periodicity and plot_periodicity.

****kwargs**

- **console_print_periodicity: int, optional, default 100**
  Display logs in the console/terminal every N iterations.
- **save_periodicity: int, optional, default 50**
  Saves the values in csv files every N iterations.
- **plot_periodicity: int, optional, default 1000**
  Generates plots from saved values every N iterations. Note that:
  - it should be a multiple of save_periodicity
  - setting a too low value (frequent) we seriously slow down you calibration
- **overwrite_logs_folder: bool, optional, default False**
  Set it to `True` to overwrite the content of the folder in `path`.

Raises

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

Notes

By default, if the folder given in `path` already 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 OutputsSettings(*settings*)

Bases: object

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

Parameters

settings
[dict[str, Any]]

Parameters of the object. It may be in:

- **path**
  [str or None] Where to store logs (relative or absolute path) If None, nothing will be saved (only console prints), unless save_periodicity is not None (default relative path './_outputs/' will be used).
- **console_print_periodicity**
  [int >= 1 or None] Flag to log into console convergence data every N iterations If None, no console prints.
- **save_periodicity**
  [int >= 1 or None] Flag to save convergence data every N iterations If None, no data will be saved.
- **plot_periodicity**
  [int >= 1 or None] Flag to plot convergence data every N iterations If None, no plots will be saved. Note that you can not plot convergence data without saving data (and not more frequently than these saves!)
- **overwrite_logs_folder**
  [bool] Flag to remove all previous logs if existing (default False)

Raises

LeaspyAlgoInputError

class AlgorithmSettings(*name: str*, ***kwargs*)

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 LME model only)
- **For *personalize* algorithms:**

- `'scipy_minimize'`
- `'mean_real'`
- `'mode_real'`
- `'constant_prediction'` (for constant model only)
- `'lme_personalize'` (for LME model only)

- **For *simulate* algorithms:**
  - `'simulation'`

****kwargs**
[any]

Depending on the algorithm you are setting up, various parameters are possible (not exhaustive):

- **seed**
  [int, optional, default None] Used for stochastic algorithms.
- **model_initialization_method**
  [str, optional] For **fit** algorithms only, 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 `leaspy.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.
- **use_jacobian**
  [bool, optional, default True] Used in `scipy_minimize` algorithm to perform a *L-BFGS* instead of a *Powell* algorithm.
- **n_jobs**
  [int, optional, default 1] Used in `scipy_minimize` algorithm to accelerate calculation with parallel derivation using joblib.
- **progress_bar**
  [bool, optional, default True] Used to display a progress bar during computation.
- **device: str or torch.device, optional**
  Specifies on which device the algorithm will run. Only 'cpu' and 'cuda' are supported for this argument. Only `'mcmc_saem'`, `'mean_real'` and `'mode_real'` algorithms support this setting.

For the complete list of the available parameters for a given algorithm, please directly refer to its documentation.

Raises

`LeaspyAlgoInputError`

See also:

`leaspy.algo`

Notes

For developers: use `_dynamic_default_parameters` to dynamically set some default parameters, depending on other parameters that were set, while these *dynamic* parameters were not set.

Example:

you could want to set burn in iterations or annealing iterations as fractions of non-default number of iterations given.

Format:

```
{algo_name: [
    (functional_condition_to_trigger_dynamic_setting(kwargs),
    {
        nested_keys_of_dynamic_setting: dynamic_value(kwargs)
    })
]}
```

Attributes

name
[str] 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 `leaspy.algo`).

seed
[int, optional, default None] Used for stochastic algorithms.

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

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

device
[str (or torch.device), optional, default 'cpu'] Used to specify on which device the algorithm will run. This should either be: 'cpu' or 'cuda' and is only supported in specific algorithms (inheriting *AlgoWithDeviceMixin*). Note that specifying an indexed CUDA device (such as 'cuda:1') is not supported. In order to specify the precise cuda device index, one should use the *CUDA_VISIBLE_DEVICES* environment variable.

Methods

check_consistency()	Check internal consistency of algorithm settings and warn or raise a *LeaspyAlgoInputError* if not.
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])	Use this method to monitor the convergence of a model calibration.

3.6.3 leaspy.io.outputs: Outputs classes

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

leaspy.io.outputs.individual_parameters.IndividualParameters

class IndividualParameters

Bases: object

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

There are used as output of the *personalization algorithms* and as input/output 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_aggregate(parameter, function)	Returns the result of aggregation by *function* of parameter values across all patients
get_mean(parameter)	Returns the mean value of a parameter across all patients
get_std(parameter)	Returns the standard deviation of a parameter across all patients
items()	Get items of dict _individual_parameters.
load(path)	Static method that loads the individual parameters (json or csv) existing at the path location
save(path, **kwargs)	Saves the individual parameters (json or csv) at the path location
subset(indices, *[, copy])	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

add_individual_parameters(*index: str*, *individual_parameters: Dict[str, Any]*)

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

LeaspyIndividualParamsInputError

- If the index is not a string or has already been added
- Or if the individual parameters is not a dict.
- Or if individual parameters are not self-consistent.

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: DataFrame*)

Static method that returns an IndividualParameters object from the dataframe

Parameters

df
[pandas.DataFrame] Dataframe of the individual 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: List[str]*, *dict_pytorch: Dict[str, Tensor]*)

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

Raises

LeaspyIndividualParamsInputError

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_aggregate(*parameter: str*, *function: Callable*) → List

Returns the result of aggregation by *function* of parameter values across all patients

Parameters

parameter
[str] Name of the parameter

function
[callable] A function operating on iterables and supporting axis keyword, and outputing an iterable supporting the *tolist* method.

Returns

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

Raises

LeaspyIndividualParamsInputError

- If individual parameters are empty,
- or if the parameter is not in the IndividualParameters.

Examples

```
>>> ip = IndividualParameters.load("path/to/individual_parameters")
>>> tau_median = ip.get_aggregate("tau", np.median)
```

get_mean(*parameter: str*)

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

LeaspyIndividualParamsInputError

- If individual parameters are empty,
- or 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: str*)

Returns the standard deviation of a parameter across all patients

Parameters

parameter
[str] Name of the parameter

Returns

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

Raises

LeaspyIndividualParamsInputError

- If individual parameters are empty,
- or if the parameter is not in the IndividualParameters.

Examples

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

items()

Get items of dict _individual_parameters.

classmethod load(*path: str*)

Static method that loads the individual parameters (json or csv) existing at the path location

Parameters

path
[str] Path and file name of the individual parameters.

Returns

IndividualParameters
Individual parameters object load from the file

Raises

LeaspyIndividualParamsInputError
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: str*, ***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 to pass to either: * pandas.DataFrame.to_csv() * json.dump() depending on saving format requested

Raises

LeaspyIndividualParamsInputError

- If extension not supported for saving
- If individual parameters are empty

subset(*indices: Iterable[str]*, ***, *copy: bool = True*)

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

copy
[bool, optional (default True)] Should we copy underlying parameters or not?

Returns

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

Raises

LeaspyIndividualParamsInputError
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() → DataFrame

Returns the dataframe of individual parameters

Returns

pandas.DataFrame
Each row corresponds to one individual. The index corresponds to the individual index ('ID'). 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() → Tuple[List[str], Dict[str, Tensor]]

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()
```

3.6.4 `leaspy.io.realizations`: Realizations classes

Internal classes used for random variables in *MCMC* algorithms.

`AbstractRealization`(name, shape, *[, ...])	Abstract class for Realization.
`IndividualRealization`(name, shape, *, ...)	Class for realizations of individual variables.
`PopulationRealization`(name, shape, *[, ...])	Class for realizations of population variables.
`DictRealizations`([realizations])	Dictionary of abstract realizations providing an easy-to-use interface.
`CollectionRealization`(*[, population, ...])	Realizations of population and individual variables, stratified per variable type.
`VariableType`(value)	Possible types for variables.

`leaspy.io.realizations.AbstractRealization`

class `AbstractRealization`(*name: str*, *shape: Tuple[int, ...]*, *, *tensor: Tensor | None = None*, *tensor_copy: bool = True*, ***kwargs*)

Bases: `object`

Abstract class for Realization.

Parameters

name
[ParamType] The name of the variable associated with the realization.

shape
[Tuple[int, ...]] The shape of the tensor realization.

tensor
[torch.Tensor, optional] If not None, the tensor realization to be stored.

tensor_copy
[bool (default True)] Whether the *tensor* provided is copied or not.

****kwargs**
[dict] Additional parameters.

Attributes

name
[ParamType] The name of the variable associated with the realization.

shape
[Tuple[int, ...]] The shape of the tensor realization.

tensor
[torch.Tensor] The tensor realization.

Methods

`initialize`(model, **kwargs)	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*.
`to_dict`()	Return a serialized dictionary of realization attributes.
`unset_autograd`()	Unset autograd for tensor of realizations

abstract initialize(*model: AbstractModel*, ***kwargs: KwargsType*)

Initialize realization from a given model.

Parameters

model
[`AbstractModel`] The model you want realizations for.

****kwargs**
[KwargsType] Additional parameters for initialization.

Raises

LeaspyModelInputError
if unknown variable type

set_autograd() → None

Set autograd for tensor of realizations.

TODO remove? only in legacy code

Raises

ValueError
if inconsistent internal request

See also:

`torch.Tensor.requires_grad_`

set_tensor_realizations_element(*element: Tensor*, *dim: tuple[int, ...]*) → None

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

Parameters

element
[torch.Tensor] The element to put in the tensor realization.

dim
[Tuple[int, ...]] The dimension where to put the element.

`to_dict()` → Dict[str, Any]

Return a serialized dictionary of realization attributes.

`unset_autograd()` → None

Unset autograd for tensor of realizations

TODO remove? only in legacy code

Raises

`ValueError`
if inconsistent internal request

See also:

`torch.Tensor.requires_grad_`

leaspy.io.realizations.IndividualRealization

`class IndividualRealization(name: str, shape: Tuple[int, ...], *, n_individuals: int, **kwargs)`

Bases: `AbstractRealization`

Class for realizations of individual variables.

Parameters

name
[ParamType] The name of the variable associated with the realization.

shape
[Tuple[int, ...]] The shape of the tensor realization.

n_individuals
[int] The number of individuals related to this realization.

****kwargs**
[dict] Additional parameters (including *tensor* and *tensor_copy*).

Attributes

tensor

Methods

`initialize`(model, *[, init_at_mean])	Initialize the realization from a model instance.
`initialize_around_mean`(mean, std)	Initialize the realization around the provided mean value.
`initialize_at_mean`(mean)	Initialize the realization at provided mean value.
`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*.
`to_dict`()	Return a serialized dictionary of realization attributes.
`unset_autograd`()	Unset autograd for tensor of realizations

initialize(*model: AbstractModel*, *, *init_at_mean: bool = False*, ***kwargs: KwargsType*)

Initialize the realization from a model instance.

Parameters

model
[AbstractModel] The model from which to initialize the realization.

init_at_mean
[bool, optional] If True, the realization is initialized at the corresponding variable mean value, otherwise it the initial value is sampled around its mean value with a normal distribution.

****kwargs**
[KwargsType] Additional parameters for initialization.

initialize_around_mean(*mean: Tensor*, *std: Tensor*) → None

Initialize the realization around the provided mean value.

The initial value is sampled according to a normal distribution with provided mean and std parameters.

Parameters

mean
[torch.Tensor] Mean value around which to sample the initial value.

std
[torch.Tensor] Standard deviation for the normal distribution used to sample the initial value.

initialize_at_mean(*mean: Tensor*) → None

Initialize the realization at provided mean value.

Parameters

mean
[torch.Tensor] The mean at which to initialize the realization.

set_autograd() → None

Set autograd for tensor of realizations.

TODO remove? only in legacy code

Raises

ValueError
if inconsistent internal request

See also:

torch.Tensor.requires_grad_

set_tensor_realizations_element(*element: Tensor*, *dim: tuple[int, ...]*) → None

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

Parameters

element
[torch.Tensor] The element to put in the tensor realization.

dim
[Tuple[int, ...]] The dimension where to put the element.

`to_dict()`

Return a serialized dictionary of realization attributes.

`unset_autograd()` → None

Unset autograd for tensor of realizations

TODO remove? only in legacy code

Raises

`ValueError`
if inconsistent internal request

See also:

`torch.Tensor.requires_grad_`

leaspy.io.realizations.PopulationRealization

`class PopulationRealization(name: str, shape: Tuple[int, ...], *, tensor: Tensor | None = None, tensor_copy: bool = True, **kwargs)`

Bases: `AbstractRealization`

Class for realizations of population variables.

Parameters

name
[ParamType] The name of the variable associated with the realization.

shape
[Tuple[int, ...]] The shape of the tensor realization.

tensor
[torch.Tensor, optional] If not None, the tensor realization to be stored.

tensor_copy
[bool (default True)] Whether the *tensor* provided is copied or not.

****kwargs**
[dict] Additional parameters.

Attributes

tensor

Methods

`initialize`(model, **kwargs)	Initialize the realization from a model instance.
`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*.
`to_dict`()	Return a serialized dictionary of realization attributes.
`unset_autograd`()	Unset autograd for tensor of realizations

initialize(*model: AbstractModel*, ***kwargs: KwargsType*) → None

Initialize the realization from a model instance.

Parameters

model
[AbstractModel] The model from which to initialize the realization.

****kwargs**
[KwargsType] Additional parameters for initialization.

set_autograd() → None

Set autograd for tensor of realizations.

TODO remove? only in legacy code

Raises

ValueError
if inconsistent internal request

See also:

torch.Tensor.requires_grad_

set_tensor_realizations_element(*element: Tensor*, *dim: tuple[int, ...]*) → None

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

Parameters

element
[torch.Tensor] The element to put in the tensor realization.

dim
[Tuple[int, ...]] The dimension where to put the element.

to_dict() → Dict[str, Any]

Return a serialized dictionary of realization attributes.

unset_autograd() → None

Unset autograd for tensor of realizations

TODO remove? only in legacy code

Raises

ValueError
if inconsistent internal request

See also:

torch.Tensor.requires_grad_

leaspy.io.realizations.DictRealizations

class DictRealizations(*realizations: Dict[str, AbstractRealization] | None = None*)

Bases: object

Dictionary of abstract realizations providing an easy-to-use interface.

Parameters

realizations
[Dict[str, AbstractRealization], optional] The dictionary of realizations (empty if None).

Attributes

realizations_dict
[Dict[str, AbstractRealization]]

Methods

clone()	Deep-copy of the CollectionRealization instance.

clone()

Deep-copy of the CollectionRealization instance.

In particular the underlying realizations are cloned and detached.

Returns

CollectionRealization
The cloned collection of realizations.

property names: List[str]

Return the list of variable names.

leaspy.io.realizations.CollectionRealization

class CollectionRealization(*, *population: DictRealizations | Dict[str, AbstractRealization] | None = None*, *individual: DictRealizations | Dict[str, AbstractRealization] | None = None*)

Bases: DictRealizations

Realizations of population and individual variables, stratified per variable type.

Parameters

population
[DictRealizationsType, optional]

individual
[DictRealizationsType, optional]

Attributes

individual
names
Return the list of variable names.

population
realizations
realizations_dict
Pooled dictionary of realizations, to (almost) provide the common *DictRealizations* interface.

tensors
tensors_dict

Methods

clone()	Deep-copy of the CollectionRealization instance.
initialize(model, *, n_individuals[, ...])	Initialize the CollectionRealization instance from a Model instance.
initialize_individuals(model, *, n_individuals)	Initialize the individual part of the CollectionRealization instance from a Model instance.
initialize_population(model, *[, skip_variable])	Initialize the population part of the CollectionRealization instance from a Model instance.

clone() → CollectionRealization

Deep-copy of the CollectionRealization instance.

In particular the underlying realizations are cloned and detached.

Returns

CollectionRealization
The cloned collection of realizations.

initialize(*model: AbstractModel*, *, *n_individuals: int*, *skip_variable: Callable[[dict], bool] | None = None*, ***realization_init_kws*) → None

Initialize the CollectionRealization instance from a Model instance.

Parameters

model
[AbstractModel] The model from which to initialize the collection of realizations.

n_individuals
[int] The number of individuals in the data.

skip_variable
[Callable or bool, optional] Whether some variables should be skipped or not.

****realization_init_kws**
[dict] Kwargs for initializing the Realizations.

initialize_individuals(*model: AbstractModel*, *, *n_individuals: int*, *skip_variable: Callable[[dict], bool] | None = None*, ***realization_init_kws*) → None

Initialize the individual part of the CollectionRealization instance from a Model instance.

Parameters

model
[AbstractModel] The model from which to initialize the collection of realizations.

n_individuals
[int] The number of individuals in the data.

skip_variable
[Callable or bool, optional] Whether some variables should be skipped or not.

****realization_init_kws**
[dict] Kwargs for initializing the Realizations.

initialize_population(*model: AbstractModel*, *, *skip_variable: Callable[[dict], bool] | None = None*, ***realization_init_kws*) → None

Initialize the population part of the CollectionRealization instance from a Model instance.

Parameters

model
[AbstractModel] The model from which to initialize the collection of realizations.

skip_variable
[Callable or bool, optional] Whether some variables should be skipped or not.

****realization_init_kws**
[dict] Kwargs for initializing the Realizations.

property names: List[str]

Return the list of variable names.

property realizations_dict: Dict[str, AbstractRealization]

Pooled dictionary of realizations, to (almost) provide the common *DictRealizations* interface.

leaspy.io.realizations.VariableType

class VariableType(*value*)

Bases: Enum

Possible types for variables.

realization_factory(...)	Factory for Realizations.

leaspy.io.realizations.realization_factory

realization_factory(*realization_or_variable_type: AbstractRealization | VariableType | str*, ***kws*) → AbstractRealization

Factory for Realizations.

Parameters

realization_or_variable_type
[AbstractRealization or VariableType or str] If an instance of a subclass of AbstractRealization, returns the instance (no copy). If a VariableType variant (or a valid string keyword for this variant), then returns a new instance of the appropriate class (with optional parameters *kws*).

****kws**
Optional parameters for initializing the requested Realization (not used if input is a subclass of AbstractRealization).

Returns

AbstractRealization
The desired realization.

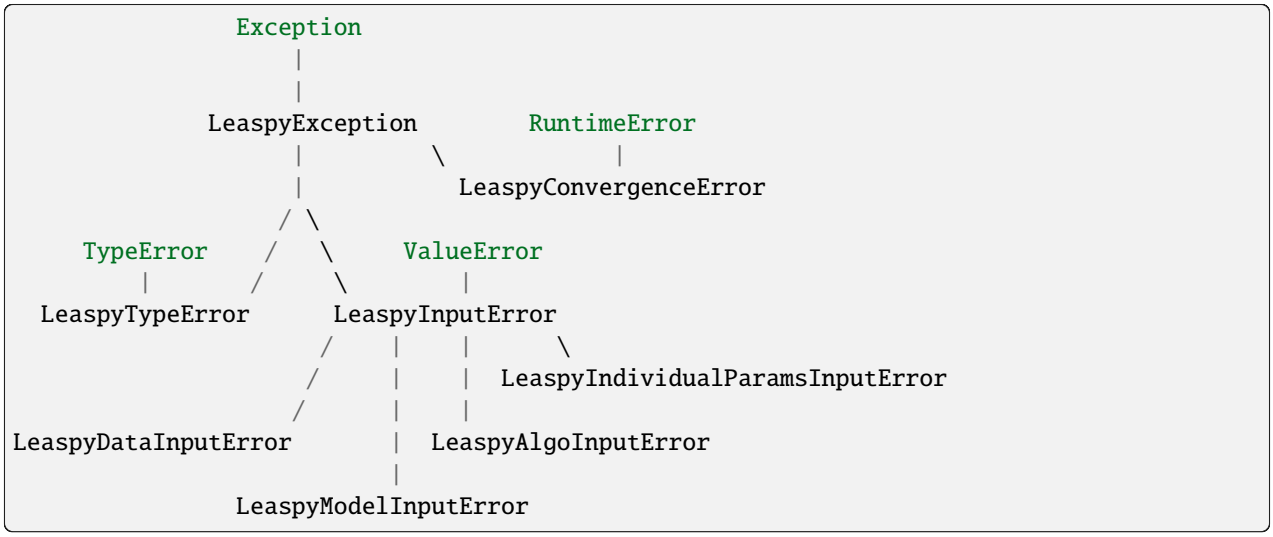
Raises

LeaspyInputError:
If the variable type provided is not supported.

3.7 leaspy.exceptions: Exceptions

Define custom Leaspy exceptions for better downstream handling.

Exceptions classes are nested so to handle in the most convenient way for users:



For I/O operations, non-Leaspy specific errors may be raised, in particular:

- FileNotFoundError
- NotADirectoryError

LeaspyException	Base of all Leaspy exceptions.
LeaspyTypeError	Leaspy Exception, deriving from *TypeError*.
LeaspyInputError	Leaspy Exception, deriving from *ValueError*.
LeaspyDataInputError	Leaspy Input Error for data related issues.
LeaspyModelInputError	Leaspy Input Error for model related issues.
LeaspyAlgoInputError	Leaspy Input Error for algorithm related issues.
LeaspyIndividualParamsInputError	Leaspy Input Error for individual parameters related issues.
LeaspyConvergenceError	Leaspy Exception for errors relative to convergence.

3.7.1 leaspy.exceptions.LeaspyException

exception LeaspyException

Bases: Exception

Base of all Leaspy exceptions.

3.7.2 leaspy.exceptions.LeaspyTypeError

exception LeaspyTypeError

Bases: LeaspyException, TypeError

Leaspy Exception, deriving from *TypeError*.

3.7.3 leaspy.exceptions.LeaspyInputError

exception LeaspyInputError

Bases: LeaspyException, ValueError

Leaspy Exception, deriving from *ValueError*.

3.7.4 leaspy.exceptions.LeaspyDataInputError

exception LeaspyDataInputError

Bases: LeaspyInputError

Leaspy Input Error for data related issues.

3.7.5 leaspy.exceptions.LeaspyModelInputError

exception LeaspyModelInputError

Bases: LeaspyInputError

Leaspy Input Error for model related issues.

3.7.6 leaspy.exceptions.LeaspyAlgoInputError

exception LeaspyAlgoInputError

Bases: LeaspyInputError

Leaspy Input Error for algorithm related issues.

3.7.7 leaspy.exceptions.LeaspyIndividualParamsInputError

exception LeaspyIndividualParamsInputError

Bases: LeaspyInputError

Leaspy Input Error for individual parameters related issues.

3.7.8 leaspy.exceptions.LeaspyConvergenceError

exception LeaspyConvergenceError

Bases: LeaspyException, RuntimeError

Leaspy Exception for errors relative to convergence.

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

TODO

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

INDEX

CHAPTER

SIX

GLOSSARY

The Glossary provides short definitions of concepts as well as Leaspy specific vocabulary.

If you wish to add a missing term, please create an issue or open a Merge Request.

API
: An Application Programming Interface is a way for two or more computer programs to communicate with each other. It is a type of software interface, offering a service to other pieces of software. In the case of Leaspy, the API describes the public classes and functions that are exposed to client code.

attachment
: The attachment is a diminutive for attachment to data. It a term of the *log-likelihood* that describes how close to the real data the model is.

biomarker
: In biomedical contexts, a biomarker, or biological marker, is a measurable indicator of some biological state or condition. Biomarkers are often measured and evaluated using blood, urine, or soft tissues to examine normal biological processes, pathogenic processes, or pharmacologic responses to a therapeutic intervention.

calibration
: Is the process that computes the population parameters, the *fixed effects* of the model. It is done by a *likelihood* maximisation using an *MCMC-SAEM* algorithm.

estimation
: The estimation consists in computing the trajectory of a given patient thanks to population parameters (computed during *calibration* step) and individual parameters (computed during *personalization* step).

fixed effects model
: In statistics, a fixed effects model is a statistical model in which the model parameters are fixed or non-random quantities. This is in contrast to *random effects models* and *mixed models* in which all or some of the model parameters are random variables.

JSON
: JavaScript Object Notation (JSON) is an open standard file format and data interchange format that uses human-readable text to store and transmit data objects consisting of attribute–value pairs and arrays (or other serializable values). JSON is a language-independent data format. It was derived from JavaScript, but many modern programming languages include code to generate and parse JSON-format data. JSON filenames use the extension `.json`.

likelihood
: Is the probability of the data knowing the model parameters. The log-likelihood could be separated into two parts:

  - the data *attachment* term which describes how close the model is to the data.
  - the *regularity* term which corresponds to how far from the priors the model is.

MCMC
: In statistics, Markov chain Monte Carlo (MCMC) methods comprise a class of algorithms for sampling from a probability distribution. By constructing a Markov chain that has the desired distribution as its equilibrium distribution, one can obtain a sample of the desired distribution by recording states from the chain. The more steps that are included, the more closely the distribution of the sample matches the actual desired distribution. Various algorithms exist for constructing chains, including the Metropolis–Hastings algorithm.

MCMC-SAEM
: The MCMC-SAEM is a powerful algorithm used to estimate maximum *likelihood* in the wide class of exponential non-linear mixed effects models.

mixed model
: A mixed model, "mixed-effects model" or "mixed error-component model" is a statistical model containing both *fixed effects* and *random effects*.

mixing matrix
: The matrix involved in the dimensionality reduction of the *space shifts* through an Independant Component Analysis (ICA). This matrix is used to construct the *space shifts* by applying it to a vector of *sources*.

MMSE
: The Mini Mental State Examination (or Folstein test) is a 30-point questionnaire that is used extensively in clinical and research settings to measure cognitive impairment. It is commonly used in medicine and allied health to screen for dementia.

overfitting
: In mathematical modeling, overfitting is the production of an analysis that corresponds too closely or exactly to a particular set of data, and may therefore fail to fit to additional data or predict future observations reliably. An overfitted model is a mathematical model that contains more parameters than can be justified by the data.

personalization
: Once the population parameters (*fixed effects*) have been computed, the user might want to know the individual parameters (*random effects*) of a given patient that are necessary in order to compute his/her individual trajectory (*estimation* step).

random effects model
: In statistics, a random effects model, also called a "variance components model", is a statistical model where the model parameters are random variables. A random effects model is a special case of a *mixed model*.

regularity
: Is a term of the *log-likelihood* that describes to how far from the priors the parameters of the model are. It is sort of a bayesian version of the machine learning *regularization*.

regularization
: In mathematics, statistics, finance, computer science, particularly in machine learning and inverse problems, regularization is a process that changes the result answer to be "simpler". It is often used to obtain results for ill-posed problems or to prevent *overfitting*.

sources
: Vector stemming from the *dimensionality reduction* of the *space shifts*. The **`sources`_** are one of the *random effects* that are estimated by the model to fit the reference disease trajectory to each patient.

space shifts
: Also referred to as the *spatial effects* of our model, the **`space shifts`_** capture the variability in the disease that is orthogonal to the temporal reparametrization. That is to say that two subjects could be modelled at the same stage of the disease and with the same progression rate but still exhibit different patterns (for instance in the ordering of the abnormal features). They are usually denoted w_i.

spatial effects
: See *space shifts*.

sufficient statistics

In statistics, a statistic is sufficient with respect to a statistical model and its associated unknown parameter if "no other statistic that can be calculated from the same sample provides any additional information as to the value of the parameter". In particular, a statistic is sufficient for a family of probability distributions if the sample from which it is calculated gives no additional information than the statistic, as to which of those probability distributions is the sampling distribution.

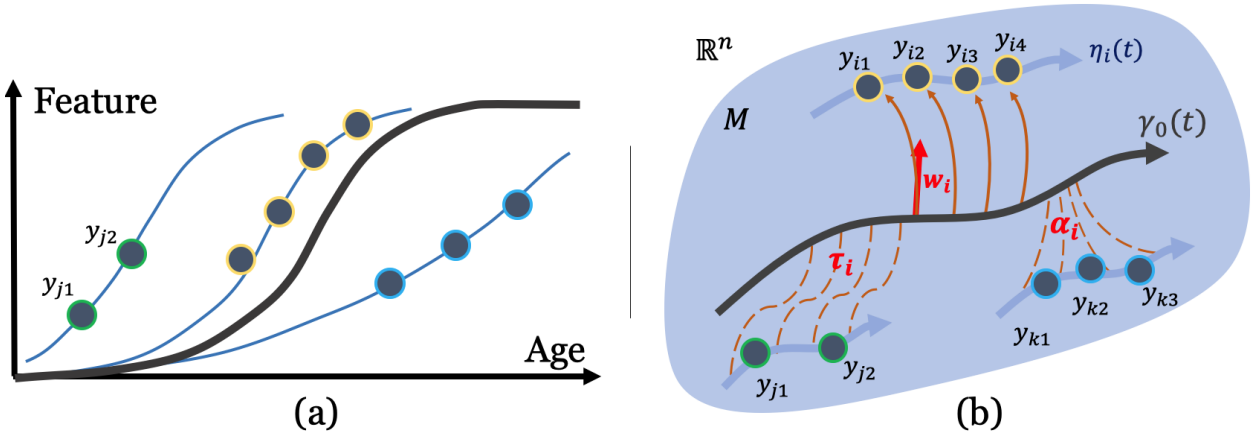
CHAPTER

SEVEN

LEARNING SPATIOTEMPORAL PATTERNS IN PYTHON

7.1 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 (different 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 un-bias the initial cohort or mimic its characteristics

The software package can be used with scalar multivariate data whose progression can be modelled 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.

7.2 Getting started

Information to install, test, and contribute to the package.

7.3 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.

7.4 User Guide

The main documentation. This contains an in-depth description of all algorithms and how to apply them.

7.5 License

The package is distributed under the BSD 3-Clause license.

7.6 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, Scientific Reports, 2021. 11(1):1-16 Open Access
- www.digital-brain.org : Website related to the application of the model for Alzheimer's disease.

- Disease Course Mapping webpage by Igor Koval

INDEX

D

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

V