apollon

Release 0.1.3

Michael Blaß

Apr 20, 2021

CONTENTS

apollon is a feature extraction and modeling framework for music data analysis. It handles low-level audio feature extraction, their aggreagation using Hidden Markov models, and comparison by means of the self-organizing map. See the *Framework* chapter for gentle introduction to the mentioned concepts.

CHAPTER

ONE

CONTENTS

1.1 Download

You can either download the source code from the apollon GitHub repository or clone it directly with

```
git clone https://github.com/teagum/apollon.git
```

1.2 Installation

apollon can be installed on GNU/Linux, macOS, and Windows. Installation process is similar on each of these plaforms. Note, however, that apollon contains CPython extension modules, which have to be compiled locally for GNU/Linux and Windows users. If you work on those platforms, please make shure that there is a C compiler set up on your machine; otherwise the installation will fail. In the case of macOS, a precompiled wheel is provided for the latest version only.

1.2.1 Install using pip

The Python packager manager can automatically download and install apollon from Pypi. Simply run the following command from your terminal:

```
python3 -m pip install apollon
```

1.2.2 Install from source

You can also install and compile apollon directly from its sources in three steps:

- Download the apollon source code
- Open a terminal and navigate to the apollon root directory
- Install and compile with the following command

```
python3 -m pip install .
```

1.3 Framework

1.3.1 Audio Feature Extraction

Extract some of the most common low-level audio feauters.

1.3.2 Hidden Markov Model

Estimate Poisson-distributed Hidden Markov Models.

1.3.3 Self-Organizing Map

Train some Self-organizing maps.

1.4 apollon

1.4.1 apollon package

Apollon feature extraction framework.

Subpackages

apollon.hmm package

Submodules

apollon.hmm.poisson module

poisson_hmm.py – HMM with Poisson-distributed state dependent process. Copyright (C) 2018 Michael Blaß <mblass@posteo.net>

Functions: to_txt Serializes model to text file. to_json JSON serialization.

is_tpm Check weter array is stochastic matrix. _check_poisson_intput Check wheter input is suitable for PoissonHMM.

Classes: PoissonHMM HMM with univariat Poisson-distributed states.

class apollon.hmm.poisson.**Params**(*lambda_*, *gamma_*, *delta_*)
Bases: object

Easy access to estimated HMM parameters and quality measures.

class apollon.hmm.poisson.**PoissonHmm**(*X: numpy.ndarray*, *m_states: int*, *init_lambda: Union[numpy.ndarray, str] = 'quantile'*, *init_gamma: Union[numpy.ndarray, str] = 'uniform'*, *init_delta: Union[numpy.ndarray, str] = 'stationary'*, *g_dirichlet: Optional[Iterable] = None*, *d_dirichlet: Optional[Iterable] = None*, *fill_diag: float = 0.8*, *verbose: bool = True*)
Bases: object

Hidden-Markov Model with univariate Poisson-distributed states.

decoding

fit (*X: numpy.ndarray*) → bool
Fit the initialized PoissonHMM to the input data set.

Parameters **X** (*np.ndarray*) –

Returns (int) True on success else False.

hyper_params

init_params

params

quality

score (*X: numpy.ndarray*)
Compute the log-likelihood of *X* under this HMM.

success

to_dict ()
Returns HMM parameters as dict.

training_date

verbose

version

class apollon.hmm.poisson.**QualityMeasures** (*aic*, *bic*, *nll*, *n_iter*)
Bases: object

apollon.hmm.poisson.**assert_poisson_input_data** (*X: numpy.ndarray*)
Raise if X is not a array of integers.

Parameters **X** (*np.ndarray*) –

Raises **ValueError** –

apollon.hmm.utilities module

Functions: assert_poisson_input Raise if array does not conform restrictions. assert_st_matrix Raise if array is not a stochastic matrix. assert_st_vector Raise if array is not a stochastic vector.

init_lambda_linear Init linearly between min and max. init_lambda_quantile Init regarding data quantiles. init_lambda_random Init with random samples from data range.

init_gamma_dirichlet Init using Dirichlet distribution. init_gamma_softmax Init with softmax of random floats. init_gamma_uniform Init with uniform distr over the main diagonal.

init_delta_dirichlet Init using Dirichlet distribution. init_delta_softmax Init with softmax of random floats. init_delta_stationary Init with stationary distribution. init_delta_uniform Init with uniform distribution.

stationary_distr Compute stationary distribution of tpm.

get_off_diag Return off-diagonal elements of square array. set_off_diag Set off-diagonal elements of square array. logit_gamma Transform tpm to logit space. expit_gamma Transform tpm back from logit space. sort_param Sort messed up gamma.

class apollon.hmm.utilities.**StartDistributionInitializer**
Bases: object

Initializes the start distribution of HMM.

static dirichlet(*m: int*, *alpha: tuple*) → numpy.ndarray
Initialize the initial distribution with a Dirichlet random sample.

Parameters

- **m** (*int*) –
- **alpha** (*iterable*) –

Returns (np.ndarray) Stochastic vector of shape (m,).

methods = ('dirichlet', 'softmax', 'stationary', 'uniform')

static softmax(*m: int*) → numpy.ndarray
Initialize the initial distribution by applying softmax to a sample of random floats.

Parameters **m** (*int*) –

Returns (np.ndarray) Stochastic vector of shape (m,).

static stationary(*gamma_: numpy.ndarray*) → numpy.ndarray
Initialize the initial distribution with the stationary distribution of *init_gamma*.

Parameters **gamma** (*np.ndarray*) –

Returns (np.ndarray) Stochastic vector of shape (m,).

static uniform(*m: int*) → numpy.ndarray
Initialize the initial distribution uniformly. The initial values are set to the inverse of the number of states.

Parameters **m** (*int*) –

Returns (np.ndarray) Stochastic vector of shape (m,).

class apollon.hmm.utilities.**StateDependentMeansInitializer**
Bases: object

Initializer methods for state-dependent vector of means.

static hist(*data: numpy.ndarray*, *m_states: int*) → numpy.ndarray
Initialize state-dependent means based on a histogram of `data`.

The histogram is calculated with ten bins. The centers of the `m_states` most frequent bins are returned as estimates of lambda.

Parameters

- **data** – Input data.
- **m_states** – Number of states.

Returns Lambda estimates.

static linear(*X: numpy.ndarray*, *m: int*) → numpy.ndarray
Initialize state-dependent means with *m* linearily spaced values from [min(data), max(data)].

Args: X (np.ndarray) Input data. m (int) Number of states.

Returns: (np.ndarray) Initial state-dependent means of shape (m,).

methods = ('hist', 'linear', 'quantile', 'random')

static quantile(*X: numpy.ndarray*, *m: int*) → numpy.ndarray
Initialize state-dependent means with *m* equally spaced percentiles from data.

Parameters

- **X** (*np.ndarray*) –
- **m** (*int*) –

Returns (np.ndarray) Initial state-dependent means of shape (m,).

static random(*X: numpy.ndarray*, *m: int*) → numpy.ndarray
Initialize state-dependent means with random integers from [min(x), max(x)[.

Parameters

- **X** (*np.ndarray*) –
- **m** (*int*) –

Retruns: (np.ndarray) Initial state-dependent means of shape (m,).

class apollon.hmm.utilities.**TpmInitializer**
Bases: object

Initializes transition probability matrix.

static dirichlet(*m: int*, *alpha: tuple*) → numpy.ndarray

Parameters

- **m** (*int*) –
- **alpha** (*iterable*) – Iterable of size m. Each entry controls the probability mass that is put on the respective transition.

Returns (np.ndarray) Transition probability matrix of shape (m, m).

methods = ('dirichlet', 'softmax', 'uniform')

static softmax(*m: int*) → numpy.ndarray
Initialize *init_gamma* by applying softmax to a sample of random floats.

Parameters **m** (*int*) –

Returns (np.ndarray) Transition probability matrix of shape (m, m).

static uniform(*m: int*, *diag: float*) → numpy.ndarray
Fill the main diagonal of *init_gamma* with *diag*. Set the off-diagoanl elements to the proportion of the remaining probability mass and the remaining number of elements per row.

Args: m (int) Number of states. diag (float) Value on main diagonal in [0, 1].

Returns: (np.ndarray) Transition probability matrix of shape (m, m).

apollon.hmm.utilities.**assert_poisson_input**(*X: numpy.ndarray*)
Check wether data is a one-dimensional array of integer values. Otherwise raise an exception.

Parameters **X** (*np.ndarray*) –

Raises

- **TypeError** –
- **ValueError** –

apollon.hmm.utilities.**assert_st_matrix**(*arr: numpy.ndarray*)
Raise if *arr* is not a valid two-dimensional stochastic matrix.

A stochastic matrix is a (1) two-dimensional, (2) quadratic matrix, with (3) elements from [0.0, 1.0] and (4) rows sums of exactly exactly 1.0.

Parameters **arr** (*np.ndarray*) –

Raises **ValueError** –

apollon.hmm.utilities.**assert_st_val**(*val: float*)
Check wheter *val* is suitable as element of stochastic matrix.

Parameters **val** (*float*) –

Raises

- **TypeError** –
- **ValueError** –

apollon.hmm.utilities.**assert_st_vector**(*vect: numpy.ndarray*)
Raise if *vect* is not a valid one-dimensional stochastic vector.

Parameters **vect** (*np.ndarray*) –

Raises **ValueError** –

apollon.hmm.utilities.**expit_gamma**(*lg_tpm: numpy.ndarray*, *m: int*) → numpy.ndarray
Transform *lg_tpm* back from logit space.

Parameters

- **lg_tpm** (*np.ndarray*) –
- **m** (*int*) –

Returns (np.ndarray) Transition probability matrix.

apollon.hmm.utilities.**get_off_diag**(*mat: numpy.ndarray*) → numpy.ndarray
Return the off-diagonal elements of square array.

Parameters **mat** (*np.ndarray*) –

Returns (np.ndarray) mat filled with values

Raises **ValueError** –

apollon.hmm.utilities.**logit_tpm**(*tpm: numpy.ndarray*) → numpy.ndarray
Transform tpm to logit space for unconstrained optimization.

Note: There must be no zeros on the main diagonal.

Parameters **tpm** (*np.ndarray*) –

Returns (np.nadarray) lg_tpm of shape (1, m**2-m).

apollon.hmm.utilities.**set_offdiag**(*mat: numpy.ndarray*, *vals: numpy.ndarray*)
Set all off-diagonal elements of square array to elements of *values*.

Parameters **mat** (*np.ndarray*) –

Returns vals (np.ndarray) values

Raises **ValueError** –

apollon.hmm.utilities.**sort_param**(*m_key: numpy.ndarray*, *m_param: numpy.ndarray*)
Sort one- or two-dimensional parameter array according to a unsorted 1-d array of distribution parameters.

In some cases the estimated distribution parameters are not in order. The transition probability matrix and the distribution parameters have then to be reorganized according to the array of sorted values.

Parameters

- **m_key** (*np.ndarray*) –
- **m_parma** (*np.ndarray*) –

Returns (np.ndarray) Reordered parameter.

apollon.hmm.utilities.**stationary_distr**(*tpm: numpy.ndarray*) → numpy.ndarray
Calculate the stationary distribution of the transition probability matrix *tpm*.

Parameters **tpm** (*np.ndarray*) –

Returns (np.ndarray) Stationary distribution of shape (m,).

apollon.io package

Submodules

apollon.io.io module

apollon/io.py – General I/O functionallity.

Licensed under the terms of the BSD-3-Clause license. Copyright (C) 2019 Michael Blaß, mblass@posteo.net

Classes: FileAccessControl Descriptor for file name attributes.

Functions: array_print_opt Set format for printing numpy arrays. files_in_folder Iterate over all files in given folder. generate_outpath Compute path for feature output. load_from_pickle Load pickled data. repath Change path but keep file name. save_to_pickle Pickle some data.

class apollon.io.io.**PoissonHmmEncoder**(*, *skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *sort_keys=False*, *indent=None*, *separators=None*, *default=None*)
Bases: *apollon.io.json.ArrayEncoder*

JSON encoder for PoissonHmm.

default(*o*)
Custon default JSON encoder. Properly handles <class 'PoissonHMM'>.

Note: Falls back to `ArrayEncoder` for all types that do not implement a `to_dict()` method.

Params: o (any) Object to encode.

Returns (dict)

class apollon.io.io.**WavFileAccessControl**
Bases: `object`

Control initialization and access to the `file` attribute of class:`AudioData`.

This assures that the path indeed points to a file, which has to be a .wav file. Otherwise an error is raised. The path to the file is saved as absolute path and the attribute is read-only.

apollon.io.io.**array_print_opt**(**args*, ***kwargs*)
Set print format for numpy arrays.

Thanks to unutbu: https://stackoverflow.com/questions/2891790/how-to-pretty-print-a- numpy-array-without-scientific-notation-and-with-given-pre

apollon.io.io.**generate_outpath**(*in_path: Union[str, pathlib.Path]*, *out_path: Optional[Union[str, pathlib.Path]]*, *suffix: Optional[str] = None*) → Union[str, pathlib.Path]
Generates file paths for feature und HMM output files.

If `out_path` is `None`, the basename of `in_path` is taken with the extension replaced by `suffix`.

Parameters

- **in_path** – Path to file under analysis.
- **out_path** – Commandline argument.
- **suffix** – File extension.

Returns Valid output path.

apollon.io.io.**load_from_npy**(*path: Union[str, pathlib.Path]*) → numpy.ndarray
Load data from numpy's binary format.

Parameters **path** – File path.

Returns Data as numpy array.

apollon.io.io.**load_from_pickle**(*path: Union[str, pathlib.Path]*) → Any
Load a pickled file.

Parameters **path** – Path to file.

Returns Unpickled object

apollon.io.io.**repath**(*current_path: Union[str, pathlib.Path]*, *new_path: Union[str, pathlib.Path]*, *ext: Optional[str] = None*) → Union[str, pathlib.Path]
Change the path and keep the file name. Optinally change the extension, too.

Parameters

- **current_path** – The path to change.
- **new_path** – The new path.
- **ext** – Change file extension if `ext` is not None.

Returns New path.

apollon.io.io.**save_to_npy**(*data: numpy.ndarray*, *path: Union[str, pathlib.Path]*) → None
Save an array to numpy binary format without using pickle.

Parameters

- **data** – Numpy array.
- **path** – Path to save the file.

apollon.io.io.**save_to_pickle**(*data: Any*, *path: Union[str, pathlib.Path]*) → None
Pickles data to path.

Parameters

- **data** – Pickleable object.
- **path** – Path to save the file.

apollon.io.json module

apollon/io/json.py – General JSON IO.

Classes: ArrayEncoder

Functions: dump decode_ndarray encode_ndarray load validate_ndarray

class apollon.io.json.**ArrayEncoder**(*, *skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *sort_keys=False*, *indent=None*, *separators=None*, *default=None*)

Bases: json.encoder.JSONEncoder

Encode np.ndarrays to JSON.

Simply set the cls parameter of the dump method to this class.

default(*inp: Any*) → Any

Custon SON encoder for numpy arrays. Other types are passed on to JSONEncoder.default.

Parameters **inp** – Object to encode.

Returns JSON-serializable dictionary.

apollon.io.json.**decode_ndarray**(*instance: dict*) → numpy.ndarray

Decode numerical numpy arrays from a JSON data stream.

Parameters **instance** – Instance of ndarray.schema.json.

Returns Numpy array.

apollon.io.json.**dump**(*obj: Any*, *path: Union[str, pathlib.Path]*) → None

Write obj to JSON file.

This function can handel numpy arrays.

If path is None, this fucntion writes to stdout. Otherwise, encoded object is written to path.

Parameters

- **obj** – Object to be encoded.
- **path** – Output file path.

apollon.io.json.**encode_ndarray**(*arr: numpy.ndarray*) → dict

Transform an numpy array to a JSON-serializable dict.

Array must have a numerical dtype. Datetime objects are currently not supported.

Parameters **arr** – Numpy ndarray.

Returns JSON-serializable dict adhering ndarray.schema.json.

apollon.io.json.**load**(*path: Union[str, pathlib.Path]*)

Load JSON file.

Parameters **path** – Path to file.

Returns JSON file as FeatureSpace.

apollon.io.json.**load_schema**(*schema_name: str*) → dict

Load a JSON schema.

This function searches within apollon's own schema repository. If a schema is found it is additionally validated agains Draft 7.

Parameters **schema_name** – Name of schema. Must be file name without extension.

Returns Schema instance.

Raises **IOError** –

apollon.io.json.**validate_ndarray**(*encoded_arr: dict*) → bool
Check whether encoded_arr is a valid instance of ndarray.schema.json.

Parameters **encoded_arr** – Instance to validate.

Returns True, if instance is valid.

apollon.signal package

Signal processing tools

Audio features

—

Submodules

apollon.signal.container module

apollon.signal.critical_bands module

apollon.signal.critical_bands.**filter_bank**(*frqs: numpy.ndarray*) → numpy.ndarray
Return a critical band rate scaled filter bank.

Each filter is triangular, which lower and upper cuttoff frequencies set to lower and upper bound of the given critical band rate.

Parameters **frqs** – Frequency axis in Hz.

Returns Bark scaled filter bank.

apollon.signal.critical_bands.**frq2cbr**(*frq: numpy.ndarray*) → numpy.ndarray
Transform frquencies in Hz to critical band rates in Bark.

Parameters **Frequency in Hz.** (*frq*) –

Returns Critical band rate.

apollon.signal.critical_bands.**level**(*cbi: numpy.ndarray*)
Compute the critical band level L_G from critical band intensities I_G.

Parameters **cbi** – Critical band intensities.

Returns Critical band levels.

apollon.signal.critical_bands.**sharpness**(*cbr_spctrm: numpy.ndarray*) → numpy.ndarray
Calculate a measure for the perception of auditory sharpness from a spectrogram of critical band levels.

Parameters **cbr_spctrm** (*ndarray*) –

Returns (ndarray) Sharpness for each time instant of the cbr_spctrm

apollon.signal.critical_bands.**specific_loudness**(*cbr: numpy.ndarray*)
Compute the specific loudness of a critical band rate spectra.

The specific loudness is the loudness per critical band rate. The spectra should be scaled in critical band levels.

Parameters **cbr** – Critical band rate spectrum.

Returns Specific loudness.

apollon.signal.critical_bands.**total_loudness**(*cbr: numpy.ndarray*) → numpy.ndarray
Compute the totals loudness of critical band rate spectra.

The total loudness is the sum of the specific loudnesses. The spectra should be scaled to critical band levels.

Parameters **cbr_spctr** (*ndarray*) –

Returns (ndarray) Total loudness.

apollon.signal.critical_bands.**weight_factor**(*z*)
Return weighting factor per critical band rate for sharpness calculation.

This is an improved version of Peeters (2004), section 8.1.3.

Parameters **z** – Critical band rate.

Returns Weighting factor.

apollon.signal.features module

apollon.signal.filter module

apollon.signal.filter.**bandpass_filter**(*x: numpy.ndarray*, *fs: int*, *low: int*, *high: int*, *order: int = 4*) → numpy.ndarray
Apply a Butterworth bandpass filter to input signal *x*.

Parameters

- **x** (*np.ndarray*) –
- **fs** (*int*) –
- **low** (*int*) –
- **high** (*int*) –
- **order** (*int*) –

Returns (np.ndarray) Filtered input signal.

apollon.signal.filter.**coef_bw_bandpass**(*low: int*, *high: int*, *fs: int*, *order: int = 4*) → tuple
Return coefficients for a Butterworth bandpass filter.

Parameters

- **low** (*int*) –
- **high** (*int*) –
- **fs** (*int*) –
- **order** (*int*) –

Returns (tuple) (b, a) Filter coefficients.

apollon.signal.spectral module

apollon.signal.tools module

apollon/signal/tools.py

Functions: acf Normalized autocorrelation. acf_pearson Normalized Pearson acf. corr_coef_pearson Correlation coefficient after Pearson. c_weighting C-weighting for SPL. freq2mel Transform frequency to mel. limit Limit dynamic range. mel2freq Transform mel to frequency. frq2bark Transform frequency to Bark scale. maxamp Maximal amplitude of signal. minamp Minimal amplitude of signal. normalize Scale data betwee -1.0 and 1.0. noise Generate white noise. sinusoid Generate sinusoidal signal. zero_padding Append array with zeros. trim_spectrogram Trim spectrogram to a frequency range.

`apollon.signal.tools.`**`acf`**(*inp: numpy.ndarray*) → numpy.ndarray
Normalized estimate of the autocorrelation function of `inp` by means of cross correlation.

Parameters **`inp`** – One-dimensional input array.

Returns Autocorrelation function for all positive lags.

`apollon.signal.tools.`**`acf_pearson`**(*inp_sig*)
Normalized estimate of the autocorrelation function of *inp_sig* by means of pearson correlation coefficient.

`apollon.signal.tools.`**`amp`**(*spl: Union[numpy.ndarray, int, float]*, *ref: float = 2e-05*) → Union[numpy.ndarray, float]
Computes amplitudes form sound pressure level.

The reference pressure defaults to the human hearing treshold of 20 Pa.

Parameters **`spl`** – Sound pressure level.

Returns DFT magnituds.

`apollon.signal.tools.`**`c_weighting`**(*frqs: numpy.ndarray*) → numpy.ndarray
C-weighhting for SPL.

Parameters **`frq`** – Frequencies.

Returns Weight for DFT bin with center frequency `frq`.

`apollon.signal.tools.`**`corr_coef_pearson`**(*x_sig: numpy.ndarray*, *y_sig: numpy.ndarray*) → float
Fast perason correlation coefficient.

`apollon.signal.tools.`**`freq2mel`**(*frqs*)
Transforms Hz to Mel-Frequencies.

Params: frqs: Frequency in Hz.

Returns Frequency transformed to Mel scale.

`apollon.signal.tools.`**`limit`**(*inp: numpy.ndarray*, *ldb: Optional[float] = None*, *udb: Optional[float] = None*) → numpy.ndarray
Limit the dynamic range of `inp` to [`ldb`, `udb`].

Boundaries are given in dB SPL.

Parameters

- **`inp`** – DFT bin magnitudes.
- **`ldb`** – Lower clip boundary in deci Bel.

- **udb** – Upper clip boundary in deci Bel.

Returns Copy of `inp` with values clipped.

`apollon.signal.tools.`**`maxamp`**(*sig*)
Maximal absolute elongation within the signal.

Params: sig (array-like) Input signal.

Returns (scalar) Maximal amplitude.

`apollon.signal.tools.`**`mel2freq`**(*zfrq*)
Transforms Mel-Frequencies to Hzfrq.

Parameters **zfrq** – Mel-Frequency.

Returns Frequency in Hz.

`apollon.signal.tools.`**`minamp`**(*sig*)
Minimal absolute elongation within the signal.

Params sig (array-like) Input signal.

Returns (scalar) Maximal amplitude.

`apollon.signal.tools.`**`noise`**(*level*, *n=9000*)
Generate withe noise.

Params: level (float) Noise level as standard deviation of a gaussian. n (int) Length of noise signal in samples.

Returns (ndarray) White noise signal.

`apollon.signal.tools.`**`normalize`**(*sig*)
Normlize a signal to [-1.0, 1.0].

Params: sig (np.nadarray) Signal to normalize.

Returns (np.ndarray) Normalized signal.

`apollon.signal.tools.`**`sinusoid`**(*frqs: Union[Sequence, numpy.ndarray, int, float]*, *amps: Union[Sequence, numpy.ndarray, int, float] = 1*, *fps: int = 9000*, *length: float = 1.0*, *noise: Optional[float] = None*, *comps: bool = False*) → numpy.ndarray
Generate sinusoidal signal.

Parameters

- **frqs** – Component frequencies.
- **amps** – Amplitude of each component in `frqs`. If `amps` is an integer, each component of `frqs` is scaled according to `amps`. If `amps` iis an iterable each frequency is scaled by the respective amplitude.
- **fps** – Sample rate.
- **length** – Length of signal in seconds.
- **noise** – Add gaussian noise with standard deviation `noise` to each sinusodial component.
- **comps** – If True, return the components of the signal, else return the sum.

Returns Array of signals.

apollon.signal.tools.**zero_padding**(*sig: numpy.ndarray*, *n_pad: int*, *dtype: Optional[Union[str, numpy.dtype]] = None*) → numpy.ndarray

Append n zeros to signal. *sig* must be 1D array.

Parameters

- **sig** – Array to be padded.
- **n_pad** – Number of zeros to be appended.

Returns Zero-padded input signal.

apollon.som package

apollon/som/__init__.py

Submodules

apollon.som.datasets module

apollon/som/datasets.py

Licensed under the terms of the BSD-3-Clause license. Copyright (C) 2019 Michael Blaß mblass@posteo.net

Function for generating test and illustration data sets.

apollon.som.datasets.**norm_circle**(*n_classes: int*, *n_per_class: int*, *class_std: int*, *center: Tuple[int, int] = (0, 0)*, *radius: int = 5*, *seed: Optional[int] = None*)

Generate `n_per_class` samples from `n_classes` bivariate normal distributions, each with standard deviation `class_std`. The means are equidistantly placed on a circle with radius `radius`.

Parameters

- **n_classes** – Number of classes.
- **n_per_class** – Number of samples in each class.
- **class_std** – Standard deviation for every class.
- **center** – Center of ther circle.
- **radius** – Radius of the circle on which the means are placed.
- **seed** – Set the random seed.

Returns Data set and target vector.

apollon.som.defaults module

apollon.som.neighbors module

apollon/som/neighbors.py

Neighborhood computations

Functions: gaussian N-Dimensional Gausian neighborhood.

apollon.som.neighbors.**check_bounds**(*shape: Tuple[int, int]*, *point: Tuple[int, int]*) → bool

Return `True` if `point` is valid index in `shape`.

Parameters

- **shape** – Shape of two-dimensional array.
- **point** – Two-dimensional coordinate.

Returns True if `point` is within `shape` else `False`.

`apollon.som.neighbors.`**direct_rect_nb**(*shape: Tuple[int, int]*, *point: Tuple[int, int]*) → Tuple[List[int], List[int]]

Return the set of direct neighbours of `point` given rectangular topology.

Parameters

- **shape** – Shape of two-dimensional array.
- **point** – Two-dimensional coordinate.

Returns Advanced index of points in neighbourhood set.

`apollon.som.neighbors.`**gauss_kern**(*nhb*, *r*)

`apollon.som.neighbors.`**gaussian**(*grid*, *center*, *radius*)

Compute n-dimensional Gaussian neighbourhood.

Gaussian neighborhood smoothes the array.

Params: grid Array of n-dimensional indices. center Index of the neighborhood center. radius Size of neighborhood.

`apollon.som.neighbors.`**is_neighbour**(*cra: numpy.ndarray*, *crb: numpy.ndarray*, *grid: numpy.ndarray*, *metric: str*) → numpy.ndarray

Compute neighbourship between each coordinate in `units_a` abd `units_b` on `grid`.

Parameters

- **cra** – (n x 2) array of grid coordinates.
- **crb** – (n x 2) array of grid coordinates.
- **grid** – SOM grid array.
- **metric** – Name of distance metric function.

Returns One-dimensional boolean array. `True` in position n means that the points `cra[n]` and `crb[n]` are direct neighbours on `grid` regarding `metric`.

`apollon.som.neighbors.`**mexican**(*grid*, *center*, *radius*)

Compute n-dimensional Mexcican hat neighbourhood.

Mexican hat neighborhood smoothes the array.

Params: grid Array of n-dimensional indices. center Index of the neighborhood center. radius Size of neighborhood.

`apollon.som.neighbors.`**neighborhood**(*grid*, *metric='sqeuclidean'*)

Compute n-dimensional cityblock neighborhood.

The cityblock neighborhood is a star-shaped area around `center`.

Params: grid: Array of n-dimensional indices. metric: Distance metric.

Returns Pairwise distances of map units.

apollon.som.neighbors.**rect**(*grid*, *center*, *radius*)
Compute n-dimensional Chebychev neighborhood.

The Chebychev neighborhood is a square-shaped area around `center`.

Params: grid Array of n-dimensional indices. center Index of the neighborhood center. radius Size of neighborhood.

Returns Two-dimensional array of in

apollon.som.neighbors.**star**(*grid*, *center*, *radius*)
Compute n-dimensional cityblock neighborhood.

The cityblock neighborhood is a star-shaped area around `center`.

Params: grid Array of n-dimensional indices. center Index of the neighborhood center. radius Size of neighborhood.

Returns:

apollon.som.plot module

apollon/som/plot.py

Plotting functions for SOMs.

Licensed under the terms of the BSD-3-Clause license. Copyright (C) 2019 Michael Blaß mblass@posteo.net

apollon.som.plot.**cluster_by**(*ax: matplotlib.axes._axes.Axes*, *som*, *data: numpy.ndarray*, *target: numpy.ndarray*, ***kwargs*) → None
Plot bmu colored by `traget`.

Parameters

- **ax** – Axis subplot.
- **som** – SOM instance.
- **data** – Input data.
- **target** – Target labels.

apollon.som.plot.**component**(*ax: matplotlib.axes._axes.Axes*, *som*, *comp: int*, *outline: bool = False*, ***kwargs*) → None
Plot a component plane.

Parameters

- **ax** – Axis subplot.
- **som** – SOM instance.
- **comp** – Component number.

apollon.som.plot.**hit_counts**(*ax: matplotlib.axes._axes.Axes*, *som*, *transform: Optional[Callable] = None*, ***kwargs*) → None
Plot the winner histogram.

Each unit is colored according to the number of times it was bmu.

Parameters

- **ax** – Axis subplot.
- **som** – SOM instance.

- **mode** – Choose either 'linear', or 'log'.

apollon.som.plot.**label_target**(*ax: matplotlib.axes._axes.Axes*, *som*, *data: numpy.ndarray*, *target: numpy.ndarray*, ***kwargs*) → None

Add target labels for each bmu.

Parameters

- **ax** – Axis subplot.
- **som** – SOM instance.
- **data** – Input data.
- **target** – Target labels.

apollon.som.plot.**qerror**(*ax: matplotlib.axes._axes.Axes*, *som*, ***kwargs*) → None

Plot quantization error.

apollon.som.plot.**umatrix**(*ax: matplotlib.axes._axes.Axes*, *som*, *outline: bool = False*, ***kwargs*) → None

Plot the U-matrix.

Parameters

- **ax** – Axis subplot.
- **som** – SOM instance.

Note: Figure aspect is set to 'eqaul'.

---

apollon.som.plot.**wire**(*ax: matplotlib.axes._axes.Axes*, *som*, *unit_size: Union[int, float, numpy.ndarray] = 100.0*, *line_width: Union[int, float] = 1.0*, *highlight: Optional[numpy.ndarray] = None*, *labels: bool = False*, ***kwargs*) → None

Plot the weight vectors of a SOM with two-dimensional feature space.

Neighbourhood relations are indicate by connecting lines.

Parameters

- **ax** – The axis subplot.
- **som** – SOM instance.
- **unit_size** – Size for each unit.
- **line_width** – Width of the wire lines.
- **highlight** – Index of units to be marked in different color.
- **labels** – If `True`, attach a box with coordinates to each unit.

Returns vlines, hlines, bgmarker, umarker

apollon.som.som module

class apollon.som.som.**BatchMap**(*dims: Tuple[int, int, int]*, *n_iter: int*, *eta: float*, *nhr: float*, *nh_shape: str = 'gaussian'*, *init_weights: Union[Callable[[numpy.ndarray, Tuple[int, int]], numpy.ndarray], str] = 'rnd'*, *metric: Union[Callable[[numpy.ndarray, numpy.ndarray], float], str] = 'euclidean'*, *seed: Optional[int] = None*)

Bases: *apollon.som.som.SomBase*

class apollon.som.som.**IncrementalKDTReeMap**(*dims: tuple*, *n_iter: int*, *eta: float*, *nhr: float*, *nh_shape: str = 'star2'*, *init_distr: str = 'uniform'*, *metric: str = 'euclidean'*, *seed: Optional[int] = None*)

Bases: *apollon.som.som.SomBase*

fit(*train_data*, *verbose=False*)

Fit SOM to input data.

class apollon.som.som.**IncrementalMap**(*dims: Tuple[int, int, int]*, *n_iter: int*, *eta: float*, *nhr: float*, *nh_shape: str = 'gaussian'*, *init_weights: Union[Callable[[numpy.ndarray, Tuple[int, int]], numpy.ndarray], str] = 'rnd'*, *metric: Union[Callable[[numpy.ndarray, numpy.ndarray], float], str] = 'euclidean'*, *seed: Optional[int] = None*)

Bases: *apollon.som.som.SomBase*

fit(*train_data*, *verbose=False*, *output_weights=False*)

class apollon.som.som.**SomBase**(*dims: Tuple[int, int, int]*, *n_iter: int*, *eta: float*, *nhr: float*, *nh_shape: str*, *init_weights: Union[Callable[[numpy.ndarray, Tuple[int, int]], numpy.ndarray], str]*, *metric: Union[Callable[[numpy.ndarray, numpy.ndarray], float], str]*, *seed: Optional[float] = None*)

Bases: `object`

calibrate(*data: numpy.ndarray*, *target: numpy.ndarray*) → numpy.ndarray

Retrieve the target value of the best matching input data vector for each unit weight vector.

Parameters

- **data** – Input data set.
- **target** – Target labels.

Returns Array of target values.

property dims

Return the SOM dimensions.

distribute(*data: numpy.ndarray*) → Dict[int, List[int]]

Distribute the vectors of `data` on the SOM.

Indices of vectors n `data` are mapped to the index of their best matching unit.

Parameters **data** – Input data set.

Returns Dictionary with SOM unit indices as keys. Each key maps to a list that holds the indices of rows in `data`, which best match this key.

property dists

Return the distance matrix of the grid points.

property dw
Return the dimension of the weight vectors.

property dx
Return the number of units along the first dimension.

property dy
Return the number of units along the second dimension.

property grid
Return the grid.

property hit_counts
Return total hit counts for each SOM unit.

match(*data: numpy.ndarray*) → numpy.ndarray
Return the multi index of the best matching unit for each vector in `data`.

Caution: This function returns the multi index into the array.

Parameters **data** – Input data set.

Returns Array of SOM unit indices.

match_flat(*data: numpy.ndarray*) → numpy.ndarray
Return the index of the best matching unit for each vector in `data`.

Parameters **data** – Input data set.

Returns Array of SOM unit indices.

property n_units
Return the total number of units on the SOM.

predict(*data: numpy.ndarray*) → numpy.ndarray
Predict the SOM index of the best matching unit for each item in `data`.

Parameters **data** – Input data. Rows are items, columns are features.

Returns One-dimensional array of indices.

property quantization_error
Return quantization error.

save(*path*) → None
Save som object to file using pickle.

Parameters **path** – Save SOM to this path.

save_weights(*path*) → None
Save weights only.

Parameters **path** – File path

property shape
Return the map shape.

property topographic_error
Return topographic error.

transform(*data: numpy.ndarray*) → numpy.ndarray
Transform each item in `data` to feature space.

This, in principle, returns best matching unit's weight vectors.

Parameters **data** – Input data. Rows are items, columns are features.

Returns Position of each data item in the feature space.

umatrix(*radius: int = 1*, *scale: bool = True*, *norm: bool = True*)
Compute U-matrix of SOM instance.

Parameters

- **radius** – Map neighbourhood radius.
- **scale** – If `True`, scale each U-height by the number of the associated unit's neighbours.
- **norm** – Normalize U-matrix if `True`.

Returns Unified distance matrix.

property weights
Return the weight vectors.

class apollon.som.som.**SomGrid**(*shape: Tuple[int, int]*)
Bases: `object`

cr()

nhb(*point: Tuple[int, int]*, *radius: float*) → numpy.ndarray
Compute neighbourhood within `radius` around `pouint`.

Parameters

- **point** – Coordinate in a two-dimensional array.
- **radius** – Lenght of radius.

Returns Array of positions of neighbours.

nhb_idx(*point: Tuple[int, int]*, *radius: float*) → numpy.ndarray
Compute the neighbourhood within `radius` around `point`.

Parameters

- **point** – Coordinate in a two-dimensional array.
- **radius** – Lenght of radius.

Returns Array of indices of neighbours.

rc()

apollon.som.topologies module

apollon/som/topologies.py

(c) Michael Blaß 2016

Topologies for self-organizing maps.

Functions: vn_neighbourhood Return 4-neighbourhood.

apollon.som.topologies.**vn_neighbourhood**(*x*, *y*, *dx*, *dy*, *flat=False*)
Compute Von Neuman Neighbourhood.

Compute the Von Neuman Neighbourhood of index (x, y) given an array with dimension (dx, dy). The Von Neumann Neighbourhood is the 4-neighbourhood, which includes the four direct neighbours of index (x, y) given a rect- angular array.

Params: x (int) x-Index for which to compute the neighbourhood. y (int) y-Index for which to compute the neighbourhood. dx (int) Size of enclosing array's x-axis. dy (int) Size of enclosing array's y-axis. flat (bool) Return flat index if True. Default is False.

Returns 1d-array of ints if flat, 2d-array otherwise.

apollon.som.utilities module

apollon/som/utilites.py

Utilities for self.organizing maps.

`apollon.som.utilities.`**`best_match`**(*weights: numpy.ndarray*, *inp: numpy.ndarray*, *metric: str*)
Compute the best matching unit of `weights` for each element in `inp`.

If several elemets in `weights` have the same distance to the current element of `inp`, the first element of `weights` is choosen to be the best matching unit.

Parameters

- **`weights`** – Two-dimensional array of weights, in which each row represents an unit.
- **`inp`** – Array of test vectors. If two-dimensional, rows are assumed to represent observations.
- **`metric`** – Distance metric to use.

Returns Index and error of best matching units.

`apollon.som.utilities.`**`decrease_expo`**(*start: float*, *step: float*, *stop: float = 1.0*) → Iterator[float]
Exponentially decrease `start` in `step` steps to `stop`.

`apollon.som.utilities.`**`decrease_linear`**(*start: float*, *step: float*, *stop: float = 1.0*) → Iterator[float]
Linearily decrease `start` in `step` steps to `stop`.

`apollon.som.utilities.`**`distribute`**(*bmu_idx: Iterable[int]*, *n_units: int*) → Dict[int, List[int]]
List training data matches per SOM unit.

This method assumes that the ith element of `bmu_idx` corresponds to the ith vetor in a array of input data vectors.

Empty units result in empty list.

Parameters

- **`bmu_idx`** – Indices of best matching units.
- **`n_units`** – Number of units on the SOM.

Returns Dictionary in which the keys represent the flat indices of SOM units. The corresponding value is a list of indices of those training data vectors that have been mapped to this unit.

`apollon.som.utilities.`**`grid`**(*n_rows: int*, *n_cols: int*) → numpy.ndarray
Compute grid indices of a two-dimensional array.

Parameters

- **`n_rows`** – Number of array rows.
- **`n_cols`** – Number of array columns.

Returns Two-dimensional array in which each row represents an multi-index.

apollon.som.utilities.**grid_iter**(*n_rows: int*, *n_cols: int*) → Iterator[Tuple[int, int]]
Compute grid indices of an two-dimensional array.

Parameters

- **n_rows** – Number of array rows.
- **n_cols** – Number of array columns.

Returns Multi-index iterator.

apollon.som.utilities.**sample_hist**(*dims: Tuple[int, int, int]*, *data: Optional[numpy.ndarray] = None*, ***kwargs*) → numpy.ndarray
Sample sum-normalized histograms.

Parameters

- **dims** – Dimensions of SOM.
- **data** – Input data set.

Returns Two-dimensional array in which each row is a historgram.

apollon.som.utilities.**sample_pca**(*dims: Tuple[int, int, int]*, *data: Optional[numpy.ndarray] = None*, ***kwargs*) → numpy.ndarray
Compute initial SOM weights by sampling from the first two principal components of the input data.

Parameters

- **dims** – Dimensions of SOM.
- **data** – Input data set.
- **adapt** – If `True`, the largest value of `shape` is applied to the principal component with the largest sigular value. This orients the map, such that map dimension with the most units coincides with principal component with the largest variance.

Returns Array of SOM weights.

apollon.som.utilities.**sample_rnd**(*dims: Tuple[int, int, int]*, *data: Optional[numpy.ndarray] = None*, ***kwargs*) → numpy.ndarray
Compute initial SOM weights by sampling uniformly from the data space.

Parameters

- **dims** – Dimensions of SOM.
- **data** – Input data set. If `None`, sample from [-10, 10].

Returns Array of SOM weights.

apollon.som.utilities.**sample_stm**(*dims: Tuple[int, int, int]*, *data: Optional[numpy.ndarray] = None*, ***kwargs*) → numpy.ndarray
Compute initial SOM weights by sampling stochastic matrices from Dirichlet distribution.

The rows of each n by n stochastic matrix are sampes drawn from the Dirichlet distribution, where n is the number of rows and cols of the matrix. The diagonal elemets of the matrices are set to twice the probability of the remaining elements. The square root of the weight vectors' size must be a real integer.

Parameters

- **dims** – Dimensions of SOM.
- **data** – Input data set.

Returns Array of SOM weights.

Notes

Each row of the output array is to be considered a flattened stochastic matrix, such that each `N = sqrt(data.shape[1])` values are a discrete probability distribution forming the `N` th row of the matrix.

Submodules

apollon.aplot module

apollon/aplot.py

General plotting routines.

Functions: fourplot Create a four plot of time a signal. marginal_distr Plot the marginal distribution of a PoissonHMM. onsets Plot onsets over a signal. onest_decoding Plot decoded onsets over a signal. signal Plot a time domain signal.

`apollon.aplot.`**`center_spines`**(*axs: Union[matplotlib.axes._axes.Axes, Iterable[matplotlib.axes._axes.Axes]]*, *intersect: Tuple[float, float] = (0.0, 0.0)*) → None

Display axes in crosshair fashion.

Parameters

- **`axs`** – Axis or iterable of axes.
- **`intersect`** – Coordinate of axes' intersection point.

`apollon.aplot.`**`fourplot`**(*data: numpy.ndarray*, *lag: int = 1*) → tuple

Plot time series, lag-plot, histogram, and probability plot.

Parameters

- **`data`** – Input data set.
- **`lag`** – Lag for lag-plot given in number of samples.

Returns Parameters

`apollon.aplot.`**`marginal_distr`**(*train_data: numpy.ndarray*, *state_means: numpy.ndarray*, *stat_dist: numpy.ndarray*, *bins: int = 20*, *legend: bool = True*, ***kwargs*) → tuple

Plot the marginal distribution of a PoissonHMM.

Parameters

- **`train_data`** – Training data set.
- **`state_means`** – State dependend means.
- **`stat_dist`** – Stationary distribution.

Returns Figure and Axes.

`apollon.aplot.`**`onset_decoding`**(*odf: numpy.ndarray*, *onset_index: numpy.ndarray*, *decoding: numpy.ndarray*, *cmap='viridis'*, ***kwargs*) → tuple

Plot sig and and onsetes color coded regarding dec.

Parameters

- **`odf`** – Onset detection function or an arbitrary time series.

- **onset_index** – Onset indices relative to `odf`.
- **decoding** – State codes in [0, . . . , n].
- **cmap** – Colormap for onsets.

Returns Figure and axes.

`apollon.aplot.`**`onsets`**(*sig*, *ons*, ***kwargs*) → tuple
Indicate onsets on a time series.

Parameters

- **sig** – Input to onset detection.
- **ons** – Onset detector instance.

Returns Figure and axes.

`apollon.aplot.`**`outward_spines`**(*axs: Union[matplotlib.axes._axes.Axes, Iterable[matplotlib.axes._axes.Axes]]*, *offset: float = 10.0*) → None
Display only left and bottom spine and displace them.

Parameters

- **axs** – Axis or iterable of axes.
- **offset** – Move the spines `offset` pixels in the negative direction.

Note: Increasing `offset` may breaks the layout. Since the spine is moved, so is the axis label, which is in turn forced out of the figure's bounds.

`apollon.aplot.`**`signal`**(*values: numpy.ndarray*, *fps: Optional[int] = None*, ***kwargs*) → tuple
Plot time series with constant sampling interval.

Parameters

- **values** – Values of the time series.
- **fps** – Sampling frequency in samples.
- **time_scale** – Seconds or samples.

Returns Figure and axes.

apollon.audio module

apollon.container module

apollon/container.py – Container Classes.

Licensed under the terms of the BSD-3-Clause license. Copyright (C) 2019 Michael Blaß mblass@posteo.net

Classes: FeatureSpace Params

class `apollon.container.`**`FeatureSpace`**(***kwargs*)
Bases: *apollon.container.NameSpace*

Container class for feature vectors.

`as_dict`() → Dict[str, Any]
Returns the FeatureSpace converted to a dict.

items() → List[Tuple[str, Any]]
Provides the the FeatureSpace's items.

Returns List of (key, value) pairs.

keys() → List[str]
Provides the FeatureSpaces's keys.

Returns List of keys.

to_csv(*path: Optional[str] = None*) → None
Write FeatureSpace to csv file.

If `path` is `None`, comma separated values are written stdout.

Parameters **path** – Output file path.

Returns FeatureSpace as csv-formatted string if `path` is `None`, else `None`.

to_json(*path: Optional[str] = None*) → Optional[str]
Convert FeaturesSpace to JSON.

If `path` is `None`, this method returns of the data of the `FeatureSpace` in JSON format. Otherwise, data is written to `path`.

Parameters **path** – Output file path.

Returns FeatureSpace as JSON-formatted string if path is not `None`, else `None`.

update(*key: str*, *val: Any*) → None
Update the set of parameters.

Parameters

- **key** – Field name.
- **val** – Field value.

values() → List[Any]
Provides the FeatureSpace's values.

Returns List of values.

class apollon.container.**NameSpace**(***kwargs*)
Bases: `object`

Simple name space object.

class apollon.container.**Params**
Bases: `object`

Parmeter base class.

classmethod from_dict(*instance: dict*) → GenericParams
Construct Params from dictionary

property schema
Returns the serialization schema.

to_dict() → dict
Returns parameters as dictionary.

to_json(*path: Union[str, pathlib.Path]*) → None
Write parameters to JSON file.

Parameters **path** – File path.

apollon.datasets module

datasets.py – Load test data sets.

apollon.datasets.**DataSet**
alias of apollon.datasets.EarthquakeData

apollon.datasets.**load_earthquakes**() → apollon.datasets.EarthquakeData
Load earthquakes dataset.

Returns (namedtuple) EqData(data, N, descr)

apollon.fractal module

apollon/fractal.py

Tools for estimating fractal dimensions.

Function: lorenz_attractor Simulate Lorenz system.

apollon.fractal.**delay_embedding**(*inp: numpy.ndarray*, *delay: int*, *m_dim: int*) → numpy.ndarray
Compute a delay embedding of the *inp*.

This method makes a hard cut at the upper bound of *inp* and does not perform zero padding to match the input size.

Params: inp: One-dimensional input vector. delay: Vector delay in samples. m_dim: Number of embedding dimension.

Returns Two-dimensional delay embedding array in which the nth row represents the n * *delay* samples delayed vector.

apollon.fractal.**embedding_dists**(*inp: numpy.ndarray*, *delay: int*, *m_dim: int*, *metric: str = 'euclidean'*) → numpy.ndarray
Perfom a delay embedding and return the pairwaise distances of the delayed vectors.

The returned vector is the flattend upper triangle of the distance matrix.

Params: inp: One-dimensional input vector. delay: Vector delay in samples. m_dim Number of embedding dimension. metric: Metric to use.

Returns Flattened upper triangle of the distance matrix.

apollon.fractal.**embedding_entropy**(*emb: numpy.ndarray*, *n_bins: int*) → numpy.ndarray
Compute the information entropy from an embedding.

Params: emb: Input embedding. bins: Number of bins per dimension.

Returns Entropy of the embedding.

apollon.fractal.**log_histogram_bin_edges**(*dists*, *n_bins: int*, *default: Optional[float] = None*)
Compute histogram bin edges that are equidistant in log space.

apollon.fractal.**lorenz_attractor**(*n*, *sigma=10*, *rho=28*, *beta=2.6666666666666665*, *init_xyz=(0.0, 1.0, 1.05)*, *dt=0.01*)
Simulate a Lorenz system with given parameters.

Params: n (int) Number of data points to generate. sigma (float) System parameter. rho (rho) System parameter. beta (beta) System parameter. init_xyz (tuple) Initial System state. dt (float) Step size.

Returns xyz (array) System states.

apollon.onsets module

apollon.segment module

apollon.tools module

Common tool library. Licensed under the terms of the BSD-3-Clause license.

`apollon.tools.`**`L1_Norm`**(*arr: numpy.ndarray*) → float
Compute the L_1 norm of input vector *x*.

This implementation is generally faster than np.norm(arr, ord=1).

`apollon.tools.`**`assert_and_pass`**(*func: Callable*, *arg: Any*)
Call `func` with `arg` and return `arg`. Additionally allow arg to be `None`.

Parameters

- **`func`** – Test function.
- **`arg`** – Function argument.

Returns Result of `func(arg)`.

`apollon.tools.`**`assert_array`**(*arr: numpy.ndarray*, *ndim: int*, *size: int*, *lower_bound: float = - inf*, *upper_bound: float = inf*, *name: str = 'arr'*)
Raise an error if shape of *arr* does not match given arguments.

Parameters

- **`arr`** – Array to test.
- **`ndim`** – Expected number of dimensions.
- **`size`** – Expected total number of elements.
- **`lower_bound`** – Lower bound for array elements.
- **`upper_bound`** – Upper bound for array elements.

Raises **`ValueError`** –

`apollon.tools.`**`fsum`**(*arr: numpy.ndarray*, *axis: Optional[int] = None*, *keepdims: bool = False*, *dtype: str = 'float64'*) → numpy.ndarray
Return math.fsum along the specifyed axis.

This function supports at most two-dimensional arrays.

Parameters

- **`arr`** – Input array.
- **`axis`** – Reduction axis.
- **`keepdims`** – If `True`, the output will have the same dimensionality as the input.
- **`dtype`** – Numpy data type.

Returns Sums along axis.

apollon.tools.**jsonify**(*inp: Any*)
Returns a representation of `inp` that can be serialized to JSON.

This method passes through Python objects of type dict, list, str, int float, True, False, and None. Tuples will be converted to list by the JSON encoder. Numpy arrays will be converted to list using thier .to_list() method. On all other types, the method will try to call str() and raises on error.

Parameters **inp** – Input to be jsonified.

Returns Jsonified input.

apollon.tools.**normalize**(*arr: numpy.ndarray*, *mode: str = 'array'*)
Normalize an arbitrary array_like.

Parameters

- **arr** – Input signal.
- **mode** – Normalization mode: 'array' -> (default) Normalize whole array. 'rows' -> Normalize each row separately. 'cols' -> Normalize each col separately.

Returns Normalized input.

apollon.tools.**pca**(*data: numpy.ndarray*, *n_comps: int = 2*) → Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray]
Compute a PCA based on `numpy.linalg.svd`.

Interanlly, `data` will be centered but not scaled.

Parameters

- **data** – Data set.
- **n_comps** – Number of principal components.

Returns `n_comps` largest singular values, `n_comps` largest eigen vectors, transformed input data.

apollon.tools.**rowdiag**(*arr: numpy.ndarray*, *k: int = 0*) → numpy.ndarray
Get or set `k` th diagonal of square matrix.

Get the `k` th diagonal of a square matrix sorted by rows or construct a sqare matrix with the elements of v as the main diagonal of the second and third dimension.

Parameters

- **arr** – Square array.
- **k** – Number of diagonal.

Returns Flattened diagonal.

apollon.tools.**scale**(*arr: numpy.ndarray*, *new_min: int = 0*, *new_max: int = 1*, *axis: int = - 1*) → numpy.ndarray
Scale `arr` between `new_min` and `new_max`.

Parameters

- **arr** – Array to be scaled.
- **new_min** – Lower bound.
- **new_max** – Upper bound.

Returns One-dimensional array of transformed values.

apollon.tools.**smooth_stat**(*arr: numpy.ndarray*) → numpy.ndarray
Smooth the signal based on its mean and standard deviation.

Parameters **arr** – Input signal.

Returns smoothed input signal.

apollon.tools.**standardize**(*arr: numpy.ndarray*) → numpy.ndarray
Retrun z-transformed values of arr.

Parameters **arr** – Input array.

Returns z-transformed values

apollon.tools.**time_stamp**(*fmt: Optional[str] = None*) → str
Report call time as UTC time stamp.

If fmt is not given, this function returns time stampes in ISO 8601 format.

Parameters **fmt** – Format specification.

Returns Time stamp according to fmt.

apollon.tools.**within**(*val: float*, *bounds: Tuple[float, float]*) → bool
Return True if x is in window.

Parameters **val** – Value to test.

Returns True, if val is within bounds.

apollon.tools.**within_any**(*val: float*, *windows: numpy.ndarray*) → bool
Return True if x is in any of the given windows.

Parameters

- **val** – Value to test.
- **windows** – Array of bounds.

Returns:

apollon.types module

apollon/types.py – Collection of static type hints. Licensed under the terms of the BSD-3-Clause license. Copyright (C) 2019 Michael Blaß mblass@posteo.net

- genindex
- modindex

PYTHON MODULE INDEX

a

INDEX

T

U

V

W

Z