SuSi Documentation

Release 1.2.2

Felix M. Riese

Dec 11, 2021

CONTENTS

We present the SuSi package for Python. It includes a fully functional SOM for unsupervised, supervised and semi-supervised tasks.

License 3-Clause BSD license

Author Felix M. Riese

Citation see Citation and in the bibtex file

Paper F. M. Riese, S. Keller and S. Hinz in Remote Sensing, 2020

CHAPTER

ONE

CHANGE LOG

1.1 [1.2.2] - 2021-12-11

- [ADDED] Official support for Python 3.10.
- [REMOVED] Official support for Python 3.6 (will be deprecated end of 2021 anyways). It might still work, but will not be maintained on this version.

1.2 [1.2.1] - 2021-10-19

- [ADDED] Quantization error *get_quantization_error()*

1.3 [1.2] - 2021-04-04

- [ADDED] Landing page with vuepress.
- [ADDED] Conda-forge recipe.
- [ADDED] Function *SOMClassifier.predict_proba()*
- [ADDED] Example notebook for multi-output regression
- [CHANGED] Code formatting to black.
- [CHANGED] CI from travis to GitHub actions.
- [FIXED] Requirements in setup.py

1.4 [1.1.2] - 2021-02-18

- [ADDED] Python 3.9 support. Python 3.6 support will be removed soon.
- [CHANGED] Function names for private use now start with an underscore.

1.5 [1.1.1] - 2020-11-18

- [ADDED] New distance metric "spectralangle".
- [ADDED] FAQs.
- [ADDED] Separate between positional and keyword parameters.
- [ADDED] Plot script for neighborhood distance weight matrix.
- [FIXED] Added inherited members to code documentation.

1.6 [1.1.0] - 2020-08-31

- [ADDED] Logo.
- [ADDED] SOMPlots documentation.
- [REMOVED] Python 3.5 support. Now, only 3.6-3.8 are supported.
- [FIXED] Scikit-learn warnings regarding validation of positional arguments.
- [FIXED] Sphinx documentation warnings.

1.7 [1.0.10] - 2020-04-21

- [ADDED] Support for Python 3.8.x.
- [ADDED] Test coverage and MultiOutput test.
- [CHANGED] Function *setPlaceholder* to *_set_placeholder*.
- [FIXED] Documentation links

1.8 [1.0.9] - 2020-04-07

- [ADDED] Documentation of the hyperparameters.
- [ADDED] Plot scripts.
- [CHANGED] Structure of the module files.

1.9 [1.0.8] - 2020-01-20

- [FIXED] Replaced scikit-learn *sklearn.utils.fixes.parallel_helper*, see #12.

1.10 [1.0.7] - 2019-11-28

- [ADDED] Optional tqdm visualization of the SOM training
- [ADDED] New *init_mode_supervised* called *random_minmax*.
- [CHANGED] Official name of package changes from *SUSI* to *SuSi*.
- [CHANGED] Docstrings for functions are now according to guidelines.
- [FIXED] Semi-supervised classification handling, sample weights
- [FIXED] Supervised classification SOM initalization of *n_iter_supervised*
- [FIXED] Code refactored according to prospector
- [FIXED] Resolved bug in *get_datapoints_from_node()* for unsupervised SOM.

1.11 [1.0.6] - 2019-09-11

- [ADDED] Semi-supervised abilities for classifier and regressor
- [ADDED] Example notebooks for semi-supervised applications
- [ADDED] Tests for example notebooks
- [CHANGED] Requirements for the SuSi package
- [REMOVED] Support for Python 3.4
- [FIXED] Code looks better in documentation with sphinx.ext.napoleon

1.12 [1.0.5] - 2019-04-23

- [ADDED] PCA initialization of the SOM weights with 2 principal components
- [ADDED] Variable variance
- [CHANGED] Moved installation guidelines and examples to documentation

1.13 [1.0.4] - 2019-04-21

- [ADDED] Batch algorithm for unsupervised and supervised SOM
- [ADDED] Calculation of the unified distance matrix (u-matrix)
- [FIXED] Added estimator_check of scikit-learn and fixed recognized issues

1.14 [1.0.3] - 2019-04-09

- [ADDED] Link to arXiv paper
- [ADDED] Mexican-hat neighborhood distance weight
- [ADDED] Possibility for different initialization modes
- [CHANGED] Simplified initialization of estimators
- [FIXED] URLs and styles in documentation
- [FIXED] Colormap in Salinas example

1.15 [1.0.2] - 2019-03-27

- [ADDED] Codecov, Codacy
- [CHANGED] Moved decreasing_rate() out of SOM classes
- [FIXED] Removed duplicate constructor for SOMRegressor, fixed fit() params

1.16 [1.0.1] - 2019-03-26

- [ADDED] Config file for Travis
- [ADDED] Requirements for read-the-docs documentation

1.17 [1.0.0] - 2019-03-26

- Initial release

CHAPTER

TWO

CITATION

If you are publishing results generated with the `susi` package, we are first of all very pround and happy about that! :-)

Please cite our paper, and, if you want, our code. How? Just copy the citation below or download our bibtex file which we provide here.

2.1 Paper citation

F. M. Riese, S. Keller and S. Hinz, "Supervised and Semi-Supervised Self-Organizing Maps for Regression and Classification Focusing on Hyperspectral Data", *Remote Sensing*, vol. 12, no. 1, 2020.

```
@article{riese2020supervised,
    author = {Riese, Felix~M. and Keller, Sina and Hinz, Stefan},
    title = {{Supervised and Semi-Supervised Self-Organizing Maps for
              Regression and Classification Focusing on Hyperspectral Data}},
    journal = {Remote Sensing},
    year = {2020},
    volume = {12},
    number = {1},
    article-number = {7},
    URL = {https://www.mdpi.com/2072-4292/12/1/7},
    ISSN = {2072-4292},
    DOI = {10.3390/rs12010007}
}
```

2.2 Code citation

Felix M. Riese, "SuSi: SUpervised Self-organIzing maps in Python", 10.5281/zenodo.2609130, 2019.

```
@misc{riese2019susicode,
    author = {Riese, Felix~M.},
    title = {{SuSi: SUpervised Self-organIzing maps in Python}},
    year = {2019},
    DOI = {10.5281/zenodo.2609130},
    publisher = {Zenodo},
```

(continues on next page)

(continued from previous page)

```
    howpublished = {\href{https://doi.org/10.5281/zenodo.2609130}{doi.org/10.5281/zenodo.
↪2609130}}
}
```

CHAPTER

THREE

INSTALLATION

3.1 Dependencies

Install Python 3, e.g. via Anaconda.

Install the required packages:

conda install –file requirements.txt

3.2 Install via PyPi

```
pip3 install susi
```

3.3 Install via Conda

```
conda install -c conda-forge susi
```

3.4 Install manually

1. Download the package on PyPi
2. Navigate to the install folder in your terminal with `cd`
3. Install with `python setup.py install`

3.5 Install latest development version

```
git clone https://github.com/felixriese/susi.git
cd susi/
python setup.py install
```

CHAPTER

FOUR

EXAMPLES

4.1 Regression example

In python3:

```
import susi

som = susi.SOMRegressor()
som.fit(X_train, y_train)
print(som.score(X_test, y_test))
```

4.2 Classification example

In python3:

```
import susi

som = susi.SOMClassifier()
som.fit(X_train, y_train)
print(som.score(X_test, y_test))
```

4.3 More examples

- examples/SOMClustering
- examples/SOMRegressor
- examples/SOMRegressor_semisupervised
- examples/SOMRegressor_multioutput
- examples/SOMClassifier
- examples/SOMClassifier_semisupervised

CHAPTER

FIVE

HYPERPARAMETERS

In the following, the most important hyperparameters of the SuSi package are described. The default hyperparameter settings are a good start, but can always be optimized. You can do that yourself or through an optimization. The commonly used hyperparameter settings are taken from [RieseEtAl2020].

5.1 Grid Size (n_rows, n_columns)

The grid size of a SOM is defined with the parameters `n_rows` and `n_columns`, the numbers of rows and columns. The choice of the grid size depends on several trade-offs.

Characteristics of a larger grid:

- Better adaption on complex problems (good!)
- Better/smoother visualization capabilities (good!)
- More possible overtraining (possibly bad)
- Larger training time (bad if very limited resources)

Our Recommendation: We suggest to start with a small grid, meaning `5 x 5`, and extending this grid size while tracking the test and training error metrics. We consider SOM grids as "large" with a grid size of about `100 x 100` and more. Non-square SOM grids can also be helpful for specific problems. Commonly used grid sizes are `50 x 50` to `80 x 80`.

5.2 Number of iterations and training mode

The number of iterations (`n_iter_unsupervised` and `n_iter_supervised`) depends on the training mode (`train_mode_unsupervised` and `train_mode_supervised`).

Our Recommendation (Online Mode) Use the *online* mode. If your dataset is small (< 1000 datapoints), use 10 000 iterations for the unsupervised SOM and 5000 iterations for the supervised SOM as start values. If your dataset is significantly larger, use significantly more iterations. Commonly used value ranges are for the unsupervised SOM 10 000 to 60 000 and for the (semi-)supervised SOM about 20 000 to 70 000 in the *online* mode.

Our Recommendation (Batch Mode) To be evaluated.

5.3 Neighborhood Distance Weight, Neighborhood Function, and Learning Rate

The hyperparameters around the neighborhood mode (`neighborhood_mode_unsupervised` + `neighborhood_mode_supervised`) and the learning rate (`learn_mode_unsupervised`, `learn_mode_supervised`, `learning_rate_start`, and `learning_rate_end`) depend on the neighborhood distance weight formula `nbh_dist_weight_mode`. Two different modes are implemented so far: `pseudo-gaussian` and `mexican-hat`.

Our Recommendation (Pseudo-Gaussian): Use the `pseudo-gaussian` neighborhood distance weight with the default formulas for the neighborhood mode and the learning rate. The most influence, from our experiences, comes from the start (and end) value of the learning rate (`learning_rate_start`, and `learning_rate_end`). They should be optimized. Commonly used formula are `linear` and `min` for the neighborhood mode, `min` and `exp` for the learning rate mode, start values from 0.3 to 0.8 and end values from 0.1 to 0.005.

Our Recommendation (Mexican-Hat): To be evaluated.

5.4 Distance Metric

In the following, we give recommendations for the different distance metrics. Implemented into the SuSi package are the following metrics:

- Euclidean Distance, see Wikipedia "Euclidean Distance"
- Manhattan Distance, see Wikipedia "Taxicab geometry"
- Mahalanobis Distance, see Wikipedia "Mahalanobis distance"
- Tanimoto Distance, see Wikipedia "Jaccard index - Tanimoto similarity and distance
- Spectral Angle Distance, see e.g. [YuhasEtAl1992]

Our Recommendation: Depending on the bandwidth, number of channels, and overlap of the spectral channels, a distance metric can have a significant impact on the training. While we have solely relied on the Euclidean distance in [RieseEtAl2020], we have seen in other, not SOM-related articles, that the Mahalanobis and Spectral Angle distance were helpful in the spectral separation of classes.

5.5 Hyperparameter optimization

Possible ways to find optimal hyperparameters for a problem are a grid search or randomized search. Because the SuSi package is developed according to several scikit-learn guidelines, it can be used with:

- scikit-learn.model_selection.GridSearchCV
- scikit-learn.model_selection.RandomizedSearchCV

For example, the randomized search can be applied as follows in `Python3`:

```
import susi
from sklearn.datasets import load_iris
from sklearn.model_selection import RandomizedSearchCV

iris = load_iris()
param_grid = {
```

(continues on next page)

(continued from previous page)

```
    "n_rows": [5, 10, 20],
    "n_columns": [5, 20, 40],
    "learning_rate_start": [0.5, 0.7, 0.9],
    "learning_rate_end": [0.1, 0.05, 0.005],
}
som = susi.SOMRegressor()
clf = RandomizedSearchCV(som, param_grid, random_state=1)
clf.fit(iris.data, iris.target)
print(clf.best_params_)
```

5.6 References

CHAPTER

SIX

FREQUENTLY ASKED QUESTIONS (FAQ)

Can I use SuSi for my article / poster / website / blog post . . . ? Yes! We are very proud and happy if SuSi helps you in any way. Please cite us, as described in documentation/citation.

How should I set the initial hyperparameters of a SOM? For more details on the hyperparameters, see in documentation/hyperparameters.

How can I optimize the hyperparameters? The SuSi hyperparameters can be optimized, for example, with scikit-learn.model_selection.GridSearchCV, since the SuSi package is developed according to several scikit-learn guidelines.

CHAPTER

SEVEN

SOMCLUSTERING

class susi.SOMClustering(*n_rows: int = 10*, *n_columns: int = 10*, ***, *init_mode_unsupervised: str = 'random'*, *n_iter_unsupervised: int = 1000*, *train_mode_unsupervised: str = 'online'*, *neighborhood_mode_unsupervised: str = 'linear'*, *learn_mode_unsupervised: str = 'min'*, *distance_metric: str = 'euclidean'*, *learning_rate_start: float = 0.5*, *learning_rate_end: float = 0.05*, *nbh_dist_weight_mode: str = 'pseudo-gaussian'*, *n_jobs: Optional[int] = None*, *random_state=None*, *verbose: Optional[int] = 0*)

Bases: object

Unsupervised self-organizing map for clustering.

Parameters

- **n_rows** (*int, optional (default=10)*) – Number of rows for the SOM grid
- **n_columns** (*int, optional (default=10)*) – Number of columns for the SOM grid
- **init_mode_unsupervised** (*str, optional (default="random")*) – Initialization mode of the unsupervised SOM
- **n_iter_unsupervised** (*int, optional (default=1000)*) – Number of iterations for the unsupervised SOM
- **train_mode_unsupervised** (*str, optional (default="online")*) – Training mode of the unsupervised SOM
- **neighborhood_mode_unsupervised** (*str, optional (default="linear")*) – Neighborhood mode of the unsupervised SOM
- **learn_mode_unsupervised** (*str, optional (default="min")*) – Learning mode of the unsupervised SOM
- **distance_metric** (*str, optional (default="euclidean")*) – Distance metric to compare on feature level (not SOM grid). Possible metrics: {"euclidean", "manhattan", "mahalanobis", "tanimoto", "spectralangle"}. Note that "tanimoto" tends to be slow.

  New in version 1.1.1: Spectral angle metric.
- **learning_rate_start** (*float, optional (default=0.5)*) – Learning rate start value
- **learning_rate_end** (*float, optional (default=0.05)*) – Learning rate end value (only needed for some lr definitions)
- **nbh_dist_weight_mode** (*str, optional (default="pseudo-gaussian")*) – Formula of the neighborhood distance weight. Possible formulas are: {"pseudo-gaussian", "mexican-hat"}.
- **n_jobs** (*int or None, optional (default=None)*) – The number of jobs to run in parallel.

- **random_state** (*int, RandomState instance or None, optional (default=None)*) – If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by *np.random*.
- **verbose** (*int, optional (default=0)*) – Controls the verbosity.

Variables

- **node_list_** (*np.ndarray of (int, int) tuples*) – List of 2-dimensional coordinates of SOM nodes
- **radius_max_** (*float, int*) – Maximum radius of the neighborhood function
- **radius_min_** (*float, int*) – Minimum radius of the neighborhood function
- **unsuper_som_** (*np.ndarray*) – Weight vectors of the unsupervised SOM shape = (self.n_rows, self.n_columns, X.shape[1])
- **X_** (*np.ndarray*) – Input data
- **fitted_** (*boolean*) – States if estimator is fitted to X
- **max_iterations_** (*int*) – Maximum number of iterations for the current training
- **bmus_** (*list of (int, int) tuples*) – List of best matching units (BMUs) of the dataset X
- **variances_** (*array of float*) – Standard deviations of every feature

fit(*X: Sequence*, *y: Optional[Sequence] = None*)

Fit unsupervised SOM to input data.

Parameters

- **X** (*array-like matrix of shape = [n_samples, n_features]*) – The training input samples.
- **y** (*None*) – Not used in this class.

Returns **self**

Return type object

Examples

Load the SOM and fit it to your input data *X* with:

```
>>> import susi
>>> som = susi.SOMClustering()
>>> som.fit(X)
```

fit_transform(*X: Sequence*, *y: Optional[Sequence] = None*) → numpy.ndarray

Fit to the input data and transform it.

Parameters

- **X** (*array-like matrix of shape = [n_samples, n_features]*) – The training and prediction input samples.
- **y** (*None, optional*) – Ignored.

Returns Predictions including the BMUs of each datapoint

Return type np.array of tuples (int, int)

Examples

Load the SOM, fit it to your input data *X* and transform your input data with:

```
>>> import susi
>>> som = susi.SOMClustering()
>>> X_transformed = som.fit_transform(X)
```

get_bmu(*datapoint: numpy.ndarray*, *som_array: numpy.ndarray*) → Tuple[int, int]
Get best matching unit (BMU) for datapoint.

Parameters

- **datapoint** (*np.ndarray, shape=shape[1]*) – Datapoint = one row of the dataset X
- **som_array** (*np.ndarray*) – Weight vectors of the SOM shape = (self.n_rows, self.n_columns, X.shape[1])

Returns Position of best matching unit (row, column)

Return type tuple, shape = (int, int)

get_bmus(*X: numpy.ndarray*, *som_array: Optional[numpy.array] = None*) → Optional[List[Tuple[int, int]]]
Get Best Matching Units for big datalist.

Parameters

- **X** (*np.ndarray*) – List of datapoints
- **som_array** (np.ndarray, optional (default=`None`)) – Weight vectors of the SOM shape = (self.n_rows, self.n_columns, X.shape[1])

Returns **bmus** – Position of best matching units (row, column) for each datapoint

Return type list of (int, int) tuples

Examples

Load the SOM, fit it to your input data *X* and transform your input data with:

```
>>> import susi
>>> import matplotlib.pyplot as plt
>>> som = susi.SOMClustering()
>>> som.fit(X)
>>> bmu_list = som.get_bmus(X)
>>> plt.hist2d([x[0] for x in bmu_list], [x[1] for x in bmu_list]
```

get_clusters(*X: numpy.ndarray*) → Optional[List[Tuple[int, int]]]
Calculate the SOM nodes on the unsupervised SOM grid per datapoint.

Parameters **X** (*np.ndarray*) – Input data

Returns List of SOM nodes, one for each input datapoint

Return type list of tuples (int, int)

get_datapoints_from_node(*node: Tuple[int, int]*) → List[int]
Get all datapoints of one node.

Parameters **node** (*tuple, shape (int, int)*) – Node for which the linked datapoints are calculated

Returns **datapoints** – List of indices of the datapoints that are linked to *node*

Return type list of int

get_quantization_error(*X: Optional[Sequence] = None*) → float
Get quantization error for *X* (or the training data).

Parameters **X** (*array-like matrix, optional (default=True)*) – Samples of shape = [n_samples, n_features]. If *None*, the training data is used for the calculation.

Returns Mean quantization error over all datapoints.

Return type float

Raises **RuntimeError** – Raised if the SOM is not fitted yet.

get_u_matrix(*mode: str = 'mean'*) → numpy.ndarray
Calculate unified distance matrix (u-matrix).

Parameters **mode** (*str, optional (default="mean)*) – Choice of the averaging algorithm

Returns **u_matrix** – U-matrix containing the distances between all nodes of the unsupervised SOM. Shape = (n_rows*2-1, n_columns*2-1)

Return type np.ndarray

Examples

Fit your SOM to input data *X* and then calculate the u-matrix with *get_u_matrix()*. You can plot the u-matrix then with e.g. *pyplot.imshow()*.

```
>>> import susi
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> som = susi.SOMClustering()
>>> som.fit(X)
>>> umat = som.get_u_matrix()
>>> plt.imshow(np.squeeze(umat))
```

transform(*X: Sequence*, *y: Optional[Sequence] = None*) → numpy.ndarray
Transform input data.

Parameters

- **X** (*array-like matrix of shape = [n_samples, n_features]*) – The prediction input samples.
- **y** (*None, optional*) – Ignored.

Returns Predictions including the BMUs of each datapoint

Return type np.array of tuples (int, int)

Examples

Load the SOM, fit it to your input data *X* and transform your input data with:

```
>>> import susi
>>> som = susi.SOMClustering()
>>> som.fit(X)
>>> X_transformed = som.transform(X)
```

CHAPTER

EIGHT

SOMREGRESSOR

class susi.SOMRegressor(*n_rows: int = 10*, *n_columns: int = 10*, ***, *init_mode_unsupervised: str = 'random'*, *init_mode_supervised: str = 'random'*, *n_iter_unsupervised: int = 1000*, *n_iter_supervised: int = 1000*, *train_mode_unsupervised: str = 'online'*, *train_mode_supervised: str = 'online'*, *neighborhood_mode_unsupervised: str = 'linear'*, *neighborhood_mode_supervised: str = 'linear'*, *learn_mode_unsupervised: str = 'min'*, *learn_mode_supervised: str = 'min'*, *distance_metric: str = 'euclidean'*, *learning_rate_start: float = 0.5*, *learning_rate_end: float = 0.05*, *nbh_dist_weight_mode: str = 'pseudo-gaussian'*, *missing_label_placeholder: Optional[Union[int, str]] = None*, *n_jobs: Optional[int] = None*, *random_state=None*, *verbose: Optional[int] = 0*)

Bases: susi.SOMEstimator.SOMEstimator, sklearn.base.RegressorMixin

Supervised SOM for estimating continuous variables (= regression).

Parameters

- **n_rows** (*int, optional (default=10)*) – Number of rows for the SOM grid
- **n_columns** (*int, optional (default=10)*) – Number of columns for the SOM grid
- **init_mode_unsupervised** (*str, optional (default="random")*) – Initialization mode of the unsupervised SOM
- **init_mode_supervised** (*str, optional (default="random")*) – Initialization mode of the supervised SOM
- **n_iter_unsupervised** (*int, optional (default=1000)*) – Number of iterations for the unsupervised SOM
- **n_iter_supervised** (*int, optional (default=1000)*) – Number of iterations for the supervised SOM
- **train_mode_unsupervised** (*str, optional (default="online")*) – Training mode of the unsupervised SOM
- **train_mode_supervised** (*str, optional (default="online")*) – Training mode of the supervised SOM
- **neighborhood_mode_unsupervised** (*str, optional (default="linear")*) – Neighborhood mode of the unsupervised SOM
- **neighborhood_mode_supervised** (*str, optional (default="linear")*) – Neighborhood mode of the supervised SOM
- **learn_mode_unsupervised** (*str, optional (default="min")*) – Learning mode of the unsupervised SOM

- **learn_mode_supervised** (*str, optional (default="min")*) – Learning mode of the supervised SOM
- **distance_metric** (*str, optional (default="euclidean")*) – Distance metric to compare on feature level (not SOM grid). Possible metrics: {"euclidean", "manhattan", "mahalanobis", "tanimoto", "spectralangle"}. Note that "tanimoto" tends to be slow.

  New in version 1.1.1: Spectral angle metric.
- **learning_rate_start** (*float, optional (default=0.5)*) – Learning rate start value
- **learning_rate_end** (*float, optional (default=0.05)*) – Learning rate end value (only needed for some lr definitions)
- **nbh_dist_weight_mode** (*str, optional (default="pseudo-gaussian")*) – Formula of the neighborhood distance weight. Possible formulas are: {"pseudo-gaussian", "mexican-hat"}.
- **missing_label_placeholder** (*int or str or None, optional (default=None)*) – Label placeholder for datapoints with no label. This is needed for semi-supervised learning.
- **n_jobs** (*int or None, optional (default=None)*) – The number of jobs to run in parallel.
- **random_state** (*int, RandomState instance or None, optional (default=None)*) – If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by *np.random*.
- **verbose** (*int, optional (default=0)*) – Controls the verbosity.

Variables

- **node_list_** (*np.ndarray of (int, int) tuples*) – List of 2-dimensional coordinates of SOM nodes
- **radius_max_** (*float, int*) – Maximum radius of the neighborhood function
- **radius_min_** (*float, int*) – Minimum radius of the neighborhood function
- **unsuper_som_** (*np.ndarray*) – Weight vectors of the unsupervised SOM shape = (self.n_rows, self.n_columns, X.shape[1])
- **X_** (*np.ndarray*) – Input data
- **fitted_** (*bool*) – States if estimator is fitted to X
- **max_iterations_** (*int*) – Maximum number of iterations for the current training
- **bmus_** (*list of (int, int) tuples*) – List of best matching units (BMUs) of the dataset X
- **sample_weights_** (*np.ndarray*) – Sample weights.
- **n_regression_vars_** (*int*) – Number of regression variables. In most examples, this equals one.
- **n_features_in_** (*int*) – Number of input features

fit(*X: Sequence*, *y: Optional[Sequence] = None*)

Fit supervised SOM to the input data.

Parameters

- **X** (*array-like matrix of shape = [n_samples, n_features]*) – The prediction input samples.
- **y** (*array-like matrix of shape = [n_samples, 1]*) – The labels (ground truth) of the input samples

Returns self

Return type object

Examples

Load the SOM and fit it to your input data *X* and the labels *y* with:

```
>>> import susi
>>> som = susi.SOMRegressor()
>>> som.fit(X, y)
```

fit_transform(*X: Sequence*, *y: Optional[Sequence] = None*) → numpy.ndarray
Fit to the input data and transform it.

Parameters

- **X** (*array-like matrix of shape = [n_samples, n_features]*) – The training and prediction input samples.
- **y** (*array-like matrix of shape = [n_samples, 1]*) – The labels (ground truth) of the input samples

Returns Predictions including the BMUs of each datapoint

Return type np.array of tuples (int, int)

Examples

Load the SOM, fit it to your input data *X* and transform your input data with:

```
>>> import susi
>>> som = susi.SOMClassifier()
>>> tuples = som.fit_transform(X, y)
```

get_bmu(*datapoint: numpy.ndarray*, *som_array: numpy.ndarray*) → Tuple[int, int]
Get best matching unit (BMU) for datapoint.

Parameters

- **datapoint** (*np.ndarray, shape=shape[1]*) – Datapoint = one row of the dataset X
- **som_array** (*np.ndarray*) – Weight vectors of the SOM shape = (self.n_rows, self.n_columns, X.shape[1])

Returns Position of best matching unit (row, column)

Return type tuple, shape = (int, int)

get_bmus(*X: numpy.ndarray*, *som_array: Optional[numpy.array] = None*) → Optional[List[Tuple[int, int]]]
Get Best Matching Units for big datalist.

Parameters

- **X** (*np.ndarray*) – List of datapoints
- **som_array** (np.ndarray, optional (default=`None`)) – Weight vectors of the SOM shape = (self.n_rows, self.n_columns, X.shape[1])

Returns **bmus** – Position of best matching units (row, column) for each datapoint

Return type list of (int, int) tuples

Examples

Load the SOM, fit it to your input data *X* and transform your input data with:

```
>>> import susi
>>> import matplotlib.pyplot as plt
>>> som = susi.SOMClustering()
>>> som.fit(X)
>>> bmu_list = som.get_bmus(X)
>>> plt.hist2d([x[0] for x in bmu_list], [x[1] for x in bmu_list]
```

get_clusters(*X: numpy.ndarray*) → Optional[List[Tuple[int, int]]]
Calculate the SOM nodes on the unsupervised SOM grid per datapoint.

Parameters **X** (*np.ndarray*) – Input data

Returns List of SOM nodes, one for each input datapoint

Return type list of tuples (int, int)

get_datapoints_from_node(*node: Tuple[int, int]*) → List[int]
Get all datapoints of one node.

Parameters **node** (*tuple, shape (int, int)*) – Node for which the linked datapoints are calculated

Returns **datapoints** – List of indices of the datapoints that are linked to *node*

Return type list of int

get_estimation_map() → numpy.ndarray
Return SOM grid with the estimated value on each node.

Returns **super_som_** – Supervised SOM grid with estimated value on each node.

Return type np.ndarray

Examples

Fit the SOM on your data *X, y*:

```
>>> import susi
>>> import matplotlib.pyplot as plt
>>> som = susi.SOMClassifier()
>>> som.fit(X, y)
>>> estimation_map = som.get_estimation_map()
>>> plt.imshow(np.squeeze(estimation_map,) cmap="viridis_r")
```

get_params(*deep=True*)
Get parameters for this estimator.

Parameters **deep** (*bool, default=True*) – If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns **params** – Parameter names mapped to their values.

Return type dict

get_quantization_error(*X: Optional[Sequence] = None*) → float
Get quantization error for *X* (or the training data).

Parameters **X** (*array-like matrix, optional (default=True)*) – Samples of shape = [n_samples, n_features]. If *None*, the training data is used for the calculation.

Returns Mean quantization error over all datapoints.

Return type float

Raises RuntimeError – Raised if the SOM is not fitted yet.

get_u_matrix(*mode: str = 'mean'*) → numpy.ndarray
Calculate unified distance matrix (u-matrix).

Parameters **mode** (*str, optional (default="mean)*) – Choice of the averaging algorithm

Returns **u_matrix** – U-matrix containing the distances between all nodes of the unsupervised SOM. Shape = (n_rows*2-1, n_columns*2-1)

Return type np.ndarray

Examples

Fit your SOM to input data *X* and then calculate the u-matrix with *get_u_matrix()*. You can plot the u-matrix then with e.g. *pyplot.imshow()*.

```
>>> import susi
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> som = susi.SOMClustering()
>>> som.fit(X)
>>> umat = som.get_u_matrix()
>>> plt.imshow(np.squeeze(umat))
```

predict(*X: Sequence*, *y: Optional[Sequence] = None*) → List[float]
Predict output of data X.

Parameters

- **X** (*array-like matrix of shape = [n_samples, n_features]*) – The prediction input samples.
- **y** (*None, optional*) – Ignored.

Returns **y_pred** – List of predicted values.

Return type list of float

Examples

Fit the SOM on your data *X, y*:

```
>>> import susi
>>> som = susi.SOMClassifier()
>>> som.fit(X, y)
>>> y_pred = som.predict(X)
```

score(*X*, *y*, *sample_weight=None*)
Return the coefficient of determination of the prediction.

The coefficient of determination R^2 is defined as $(1 - \frac{u}{v})$, where u is the residual sum of squares `((y_true - y_pred)** 2).sum()` and v is the total sum of squares `((y_true - y_true.mean()) ** 2).sum()`. The best possible score is 1.0 and it can be negative (because the model can be arbitrarily worse). A constant model that always predicts the expected value of *y*, disregarding the input features, would get a R^2 score of 0.0.

Parameters

- **X** (*array-like of shape (n_samples, n_features)*) – Test samples. For some estimators this may be a precomputed kernel matrix or a list of generic objects instead with shape `(n_samples, n_samples_fitted)`, where `n_samples_fitted` is the number of samples used in the fitting for the estimator.
- **y** (*array-like of shape (n_samples,) or (n_samples, n_outputs)*) – True values for *X*.
- **sample_weight** (*array-like of shape (n_samples,), default=None*) – Sample weights.

Returns **score** – R^2 of `self.predict(X)` wrt. *y*.

Return type float

Notes

The R^2 score used when calling `score` on a regressor uses `multioutput='uniform_average'` from version 0.23 to keep consistent with default value of `r2_score()`. This influences the `score` method of all the multioutput regressors (except for `MultiOutputRegressor`).

`set_params`(***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters ****params** (*dict*) – Estimator parameters.

Returns **self** – Estimator instance.

Return type estimator instance

`transform`(*X: Sequence*, *y: Optional[Sequence] = None*) → numpy.ndarray
Transform input data.

Parameters

- **X** (*array-like matrix of shape = [n_samples, n_features]*) – The prediction input samples.
- **y** (*None, optional*) – Ignored.

Returns Predictions including the BMUs of each datapoint

Return type np.array of tuples (int, int)

Examples

Load the SOM, fit it to your input data *X* and transform your input data with:

```
>>> import susi
>>> som = susi.SOMClustering()
>>> som.fit(X)
>>> X_transformed = som.transform(X)
```

CHAPTER

NINE

SOMCLASSIFIER

class susi.SOMClassifier(*n_rows: int = 10*, *n_columns: int = 10*, ***, *init_mode_unsupervised: str = 'random'*, *init_mode_supervised: str = 'majority'*, *n_iter_unsupervised: int = 1000*, *n_iter_supervised: int = 1000*, *train_mode_unsupervised: str = 'online'*, *train_mode_supervised: str = 'online'*, *neighborhood_mode_unsupervised: str = 'linear'*, *neighborhood_mode_supervised: str = 'linear'*, *learn_mode_unsupervised: str = 'min'*, *learn_mode_supervised: str = 'min'*, *distance_metric: str = 'euclidean'*, *learning_rate_start: float = 0.5*, *learning_rate_end: float = 0.05*, *nbh_dist_weight_mode: str = 'pseudo-gaussian'*, *missing_label_placeholder: Optional[Union[int, str]] = None*, *do_class_weighting: bool = True*, *n_jobs: Optional[int] = None*, *random_state=None*, *verbose: Optional[int] = 0*)

Bases: susi.SOMEstimator.SOMEstimator, sklearn.base.ClassifierMixin

Supervised SOM for estimating discrete variables (= classification).

Parameters

- **n_rows** (*int, optional (default=10)*) – Number of rows for the SOM grid
- **n_columns** (*int, optional (default=10)*) – Number of columns for the SOM grid
- **init_mode_unsupervised** (*str, optional (default="random")*) – Initialization mode of the unsupervised SOM
- **init_mode_supervised** (*str, optional (default="majority")*) – Initialization mode of the classification SOM
- **n_iter_unsupervised** (*int, optional (default=1000)*) – Number of iterations for the unsupervised SOM
- **n_iter_supervised** (*int, optional (default=1000)*) – Number of iterations for the classification SOM
- **train_mode_unsupervised** (*str, optional (default="online")*) – Training mode of the unsupervised SOM
- **train_mode_supervised** (*str, optional (default="online")*) – Training mode of the classification SOM
- **neighborhood_mode_unsupervised** (*str, optional (default="linear")*) – Neighborhood mode of the unsupervised SOM
- **neighborhood_mode_supervised** (*str, optional (default="linear")*) – Neighborhood mode of the classification SOM
- **learn_mode_unsupervised** (*str, optional (default="min")*) – Learning mode of the unsupervised SOM

- **learn_mode_supervised** (*str, optional (default="min")*) – Learning mode of the classification SOM
- **distance_metric** (*str, optional (default="euclidean")*) – Distance metric to compare on feature level (not SOM grid). Possible metrics: {"euclidean", "manhattan", "mahalanobis", "tanimoto", "spectralangle"}. Note that "tanimoto" tends to be slow.

  New in version 1.1.1: Spectral angle metric.
- **learning_rate_start** (*float, optional (default=0.5)*) – Learning rate start value
- **learning_rate_end** (*float, optional (default=0.05)*) – Learning rate end value (only needed for some lr definitions)
- **nbh_dist_weight_mode** (*str, optional (default="pseudo-gaussian")*) – Formula of the neighborhood distance weight. Possible formulas are: {"pseudo-gaussian", "mexican-hat"}.
- **missing_label_placeholder** (*int or str or None, optional (default=None)*) – Label placeholder for datapoints with no label. This is needed for semi-supervised learning.
- **do_class_weighting** (*bool, optional (default=True)*) – If true, classes are weighted.
- **n_jobs** (*int or None, optional (default=None)*) – The number of jobs to run in parallel.
- **random_state** (*int, RandomState instance or None, optional (default=None)*) – If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by *np.random*.
- **verbose** (*int, optional (default=0)*) – Controls the verbosity.

Variables

- **node_list_** (*np.ndarray of (int, int) tuples*) – List of 2-dimensional coordinates of SOM nodes
- **radius_max_** (*float, int*) – Maximum radius of the neighborhood function
- **radius_min_** (*float, int*) – Minimum radius of the neighborhood function
- **unsuper_som_** (*np.ndarray*) – Weight vectors of the unsupervised SOM shape = (self.n_rows, self.n_columns, X.shape[1])
- **X_** (*np.ndarray*) – Input data
- **fitted_** (*bool*) – States if estimator is fitted to X
- **max_iterations_** (*int*) – Maximum number of iterations for the current training
- **bmus_** (*list of (int, int) tuples*) – List of best matching units (BMUs) of the dataset *X*.
- **placeholder_dict_** (*dict*) – Dict of placeholders for initializing nodes without mapped class.
- **n_features_in_** (*int*) – Number of input features in *X*.
- **classes_** (*np.ndarray*) – Unique classes in the dataset labels *y*.
- **class_counts_** (*np.ndarray*) – Number of datapoints per unique class in *y*.
- **class_dtype_** (*type*) – Type of a label in *y*.

fit(*X: Sequence*, *y: Optional[Sequence] = None*)

Fit classification SOM to the input data.

Parameters

- **X** (*array-like matrix of shape = [n_samples, n_features]*) – The prediction input samples.
- **y** (*array-like matrix of shape = [n_samples, 1], optional*) – The labels (ground truth) of the input samples

Returns self

Return type object

Examples

Load the SOM and fit it to your input data *X* and the labels *y* with:

```
>>> import susi
>>> som = susi.SOMClassifier()
>>> som.fit(X, y)
```

fit_transform(*X: Sequence*, *y: Optional[Sequence] = None*) → numpy.ndarray
Fit to the input data and transform it.

Parameters

- **X** (*array-like matrix of shape = [n_samples, n_features]*) – The training and prediction input samples.
- **y** (*array-like matrix of shape = [n_samples, 1]*) – The labels (ground truth) of the input samples

Returns Predictions including the BMUs of each datapoint

Return type np.array of tuples (int, int)

Examples

Load the SOM, fit it to your input data *X* and transform your input data with:

```
>>> import susi
>>> som = susi.SOMClassifier()
>>> tuples = som.fit_transform(X, y)
```

get_bmu(*datapoint: numpy.ndarray*, *som_array: numpy.ndarray*) → Tuple[int, int]
Get best matching unit (BMU) for datapoint.

Parameters

- **datapoint** (*np.ndarray, shape=shape[1]*) – Datapoint = one row of the dataset X
- **som_array** (*np.ndarray*) – Weight vectors of the SOM shape = (self.n_rows, self.n_columns, X.shape[1])

Returns Position of best matching unit (row, column)

Return type tuple, shape = (int, int)

get_bmus(*X: numpy.ndarray*, *som_array: Optional[numpy.array] = None*) → Optional[List[Tuple[int, int]]]
Get Best Matching Units for big datalist.

Parameters

- **X** (*np.ndarray*) – List of datapoints

- **som_array** (np.ndarray, optional (default=`None`)) – Weight vectors of the SOM shape = (self.n_rows, self.n_columns, X.shape[1])

Returns **bmus** – Position of best matching units (row, column) for each datapoint

Return type list of (int, int) tuples

Examples

Load the SOM, fit it to your input data *X* and transform your input data with:

```
>>> import susi
>>> import matplotlib.pyplot as plt
>>> som = susi.SOMClustering()
>>> som.fit(X)
>>> bmu_list = som.get_bmus(X)
>>> plt.hist2d([x[0] for x in bmu_list], [x[1] for x in bmu_list]
```

get_clusters(*X: numpy.ndarray*) → Optional[List[Tuple[int, int]]]
Calculate the SOM nodes on the unsupervised SOM grid per datapoint.

Parameters **X** (*np.ndarray*) – Input data

Returns List of SOM nodes, one for each input datapoint

Return type list of tuples (int, int)

get_datapoints_from_node(*node: Tuple[int, int]*) → List[int]
Get all datapoints of one node.

Parameters **node** (*tuple, shape (int, int)*) – Node for which the linked datapoints are calculated

Returns **datapoints** – List of indices of the datapoints that are linked to *node*

Return type list of int

get_estimation_map() → numpy.ndarray
Return SOM grid with the estimated value on each node.

Returns **super_som_** – Supervised SOM grid with estimated value on each node.

Return type np.ndarray

Examples

Fit the SOM on your data *X, y*:

```
>>> import susi
>>> import matplotlib.pyplot as plt
>>> som = susi.SOMClassifier()
>>> som.fit(X, y)
>>> estimation_map = som.get_estimation_map()
>>> plt.imshow(np.squeeze(estimation_map,) cmap="viridis_r")
```

get_params(*deep=True*)
Get parameters for this estimator.

Parameters **deep** (*bool, default=True*) – If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns **params** – Parameter names mapped to their values.

Return type dict

get_quantization_error(*X: Optional[Sequence] = None*) → float
Get quantization error for *X* (or the training data).

Parameters **X** (*array-like matrix, optional (default=True)*) – Samples of shape = [n_samples, n_features]. If *None*, the training data is used for the calculation.

Returns Mean quantization error over all datapoints.

Return type float

Raises **RuntimeError** – Raised if the SOM is not fitted yet.

get_u_matrix(*mode: str = 'mean'*) → numpy.ndarray
Calculate unified distance matrix (u-matrix).

Parameters **mode** (*str, optional (default="mean)*) – Choice of the averaging algorithm

Returns **u_matrix** – U-matrix containing the distances between all nodes of the unsupervised SOM. Shape = (n_rows*2-1, n_columns*2-1)

Return type np.ndarray

Examples

Fit your SOM to input data *X* and then calculate the u-matrix with *get_u_matrix()*. You can plot the u-matrix then with e.g. *pyplot.imshow()*.

```
>>> import susi
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> som = susi.SOMClustering()
>>> som.fit(X)
>>> umat = som.get_u_matrix()
>>> plt.imshow(np.squeeze(umat))
```

predict(*X: Sequence*, *y: Optional[Sequence] = None*) → List[float]
Predict output of data X.

Parameters

- **X** (*array-like matrix of shape = [n_samples, n_features]*) – The prediction input samples.
- **y** (*None, optional*) – Ignored.

Returns **y_pred** – List of predicted values.

Return type list of float

Examples

Fit the SOM on your data *X, y*:

```
>>> import susi
>>> som = susi.SOMClassifier()
>>> som.fit(X, y)
>>> y_pred = som.predict(X)
```

predict_proba(*X: Sequence*, *y: Optional[Sequence] = None*) → numpy.ndarray
Predict class probabilities for *X*.

New in version 1.1.3.

Parameters

- **X** (*array-like matrix of shape = [n_samples, n_features]*) – The prediction input samples.
- **y** (*array-like matrix of shape = [n_samples, 1], optional*) – The labels (ground truth) of the input samples

Returns List of probabilities of shape (n_samples, n_classes)

Return type np.ndarray

score(*X*, *y*, *sample_weight=None*)
Return the mean accuracy on the given test data and labels.

In multi-label classification, this is the subset accuracy which is a harsh metric since you require for each sample that each label set be correctly predicted.

Parameters

- **X** (*array-like of shape (n_samples, n_features)*) – Test samples.
- **y** (*array-like of shape (n_samples,) or (n_samples, n_outputs)*) – True labels for *X*.
- **sample_weight** (*array-like of shape (n_samples,), default=None*) – Sample weights.

Returns **score** – Mean accuracy of `self.predict(X)` wrt. *y*.

Return type float

set_params(***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters ****params** (*dict*) – Estimator parameters.

Returns **self** – Estimator instance.

Return type estimator instance

transform(*X: Sequence*, *y: Optional[Sequence] = None*) → numpy.ndarray
Transform input data.

Parameters

- **X** (*array-like matrix of shape = [n_samples, n_features]*) – The prediction input samples.
- **y** (*None, optional*) – Ignored.

Returns Predictions including the BMUs of each datapoint

Return type np.array of tuples (int, int)

Examples

Load the SOM, fit it to your input data *X* and transform your input data with:

```
>>> import susi
>>> som = susi.SOMClustering()
>>> som.fit(X)
>>> X_transformed = som.transform(X)
```

CHAPTER

TEN

SOMPLOTS

SOMPlots functions.

susi.SOMPlots.**plot_estimation_map**(*estimation_map: numpy.ndarray*, *cbar_label: str = 'Variable in unit'*, *cmap: str = 'viridis'*, *fontsize: int = 20*) → matplotlib.axes._axes.Axes

Plot estimation map.

Parameters

- **estimation_map** (*np.ndarray*) – Estimation map of the size (n_rows, n_columns)
- **cbar_label** (*str, optional*) – Label of the colorbar, by default "Variable in unit"
- **cmap** (*str, optional (default="viridis")*) – Colormap
- **fontsize** (*int, optional (default=20)*) – Fontsize of the labels

Returns **ax** – Plot axis

Return type pyplot.axis

susi.SOMPlots.**plot_nbh_dist_weight_matrix**(*som*, *it_frac: float = 0.1*) → matplotlib.axes._axes.Axes

Plot neighborhood distance weight matrix in 3D.

Parameters

- **som** (*susi.SOMClustering or related*) – Trained (un)supervised SOM
- **it_frac** (*float, optional (default=0.1)*) – Fraction of *som.n_iter_unsupervised* for the plot state.

Returns **ax** – Plot axis

Return type pyplot.axis

susi.SOMPlots.**plot_som_histogram**(*bmu_list: List[Tuple[int, int]]*, *n_rows: int*, *n_columns: int*, *n_datapoints_cbar: int = 5*, *fontsize: int = 22*) → matplotlib.axes._axes.Axes

Plot 2D Histogram of SOM.

Plot 2D Histogram with one bin for each SOM node. The content of one bin is the number of datapoints matched to the specific node.

Parameters

- **bmu_list** (*list of (int, int) tuples*) – Position of best matching units (row, column) for each datapoint
- **n_rows** (*int*) – Number of rows for the SOM grid

- **n_columns** (*int*) – Number of columns for the SOM grid
- **n_datapoints_cbar** (*int, optional (default=5)*) – Maximum number of datapoints shown on the colorbar
- **fontsize** (*int, optional (default=22)*) – Fontsize of the labels

Returns **ax** – Plot axis

Return type pyplot.axis

susi.SOMPlots.**plot_umatrix**(*u_matrix: numpy.ndarray*, *n_rows: int*, *n_colums: int*, *cmap: str = 'Greys'*, *fontsize: int = 18*) → matplotlib.axes._axes.Axes

Plot u-matrix.

Parameters

- **u_matrix** (*np.ndarray*) – U-matrix containing the distances between all nodes of the unsupervised SOM. Shape = (n_rows*2-1, n_columns*2-1)
- **n_rows** (*int*) – Number of rows for the SOM grid
- **n_columns** (*int*) – Number of columns for the SOM grid
- **cmap** (*str, optional (default="Greys)*) – Colormap
- **fontsize** (*int, optional (default=18)*) – Fontsize of the labels

Returns **ax** – Plot axis

Return type pyplot.axis

CHAPTER

ELEVEN

SOMUTILS

SOMUtils functions.

susi.SOMUtils.**check_estimation_input**(*X: Sequence*, *y: Sequence*, ***, *is_classification: bool = False*) → Tuple[numpy.array, numpy.array]

Check input arrays.

This function is adapted from sklearn.utils.validation.

Parameters

- **X** (*nd-array or list*) – Input data.
- **y** (*nd-array, list*) – Labels.
- **is_classification** (boolean (default=`False`)) – Wether the data is used for classification or regression tasks.

Returns

- **X** (*object*) – The converted and validated *X*.
- **y** (*object*) – The converted and validated *y*.

susi.SOMUtils.**decreasing_rate**(*a_1: float*, *a_2: float*, ***, *iteration_max: int*, *iteration: int*, *mode: str*) → float

Return a decreasing rate from collection.

Parameters

- **a_1** (*float*) – Starting value of decreasing rate
- **a_2** (*float*) – End value of decreasing rate
- **iteration_max** (*int*) – Maximum number of iterations
- **iteration** (*int*) – Current number of iterations
- **mode** (*str*) – Mode (= formula) of the decreasing rate

Returns **rate** – Decreasing rate

Return type float

Examples

```
>>> import susi
>>> susi.decreasing_rate(0.8, 0.1, 100, 5, "exp")
```

susi.SOMUtils.**modify_weight_matrix_online**(*som_array: numpy.ndarray*, *, *dist_weight_matrix: numpy.ndarray*, *true_vector: numpy.ndarray*, *learning_rate: float*) → numpy.ndarray

Modify weight matrix of the SOM for the online algorithm.

Parameters

- **som_array** (*np.ndarray*) – Weight vectors of the SOM shape = (self.n_rows, self.n_columns, X.shape[1])
- **dist_weight_matrix** (*np.ndarray of float*) – Current distance weight of the SOM for the specific node
- **true_vector** (*np.ndarray*) – True vector
- **learning_rate** (*float*) – Current learning rate of the SOM

Returns Weight vector of the SOM after the modification

Return type np.array

BIBLIOGRAPHY

[RieseEtAl2020] F. M. Riese, S. Keller and S. Hinz, “Supervised and Semi-Supervised Self-Organizing Maps for Regression and Classification Focusing on Hyperspectral Data”, *Remote Sensing*, vol. 12, no. 1, 2020. MDPI Link

[YuhasEtAl1992] R. H. Yuhas, A. F. Goetz, & J. W. Boardman (1992). Discrimination among semi-arid landscape endmembers using the spectral angle mapper (SAM) algorithm. NASA Link

PYTHON MODULE INDEX

S

INDEX