
som-pbc

Release v1.0.1

Alex Müller

Nov 27, 2022

TABLE OF CONTENTS:

1	som-pbc README	3
1.1	Installation	3
1.2	Usage	3
1.3	Methods / Functions	4
1.4	References:	9
1.5	Documentation:	9
2	Documentation for the som module	11
3	Example Scripts	17
4	LICENSE	21
5	TODOs	23
6	Indices and tables	25
	Python Module Index	27
	Index	29

som-pbc

Authors: Alex Müller

Copyright: (c) 2017 - 2022; Alex Müller

This package contains a simple self-organizing map implementation in Python with periodic boundary conditions.

Self-organizing maps are also called Kohonen maps and were invented by Teuvo Kohonen.[1] They are an unsupervised machine learning technique to efficiently create spatially organized internal representations of various types of data. For example, SOMs are well-suited for the visualization of high-dimensional data.

This is a simple implementation of SOMs in Python. This SOM has periodic boundary conditions and therefore can be imagined as a “donut”. The implementation uses `numpy`, `scipy`, `scikit-learn` and `matplotlib`.

The project’s GitHub page can be found here: <http://github.com/alexarnimueller/som>

SOM-PBC README

A simple self-organizing map implementation in Python with periodic boundary conditions.

Self-organizing maps are also called Kohonen maps and were invented by Teuvo Kohonen.[1] They are an unsupervised machine learning technique to efficiently create spatially organized internal representations of various types of data. For example, SOMs are well-suited for the visualization of high-dimensional data.

This is a simple implementation of SOMs in Python. This SOM has periodic boundary conditions and therefore can be imagined as a “donut”. The implementation uses `numpy`, `scipy`, `scikit-learn` and `matplotlib`.

1.1 Installation

`som-pbc` can be installed from pypi using `pip`:

```
pip install som-pbc
```

To upgrade `som-pbc` to the latest version, run:

```
pip install --upgrade som-pbc
```

1.2 Usage

Then you can import and use the `SOM` class as follows:

```
import numpy as np
from som import SOM

# generate some random data with 36 features
data1 = np.random.normal(loc=-.25, scale=0.5, size=(500, 36))
data2 = np.random.normal(loc=.25, scale=0.5, size=(500, 36))
data = np.vstack((data1, data2))

som = SOM(10, 10) # initialize a 10 by 10 SOM
som.fit(data, 10000, save_e=True, interval=100) # fit the SOM for 10000 epochs, save
↪ the error every 100 steps
som.plot_error_history(filename='images/som_error.png') # plot the training error
```

(continues on next page)

(continued from previous page)

```

↪ history

targets = np.array(500 * [0] + 500 * [1]) # create some dummy target values

# now visualize the learned representation with the class labels
som.plot_point_map(data, targets, ['Class 0', 'Class 1'], filename='images/som.png')
som.plot_class_density(data, targets, t=0, name='Class 0', colormap='Greens', filename=
↪ 'images/class_0.png')
som.plot_distance_map(colormap='Blues', filename='images/distance_map.png') # plot the
↪ distance map after training

# predicting the class of a new, unknown datapoint
datapoint = np.random.normal(loc=.25, scale=0.5, size=(1, 36))
print("Labels of neighboring datapoints: ", som.get_neighbors(datapoint, data, targets,
↪ d=0))

# transform data into the SOM space
newdata = np.random.normal(loc=.25, scale=0.5, size=(10, 36))
transformed = som.transform(newdata)
print("Old shape of the data:", newdata.shape)
print("New shape of the data:", transformed.shape)

```

1.2.1 Training Error:

1.2.2 Point Map:

1.2.3 Class Density:

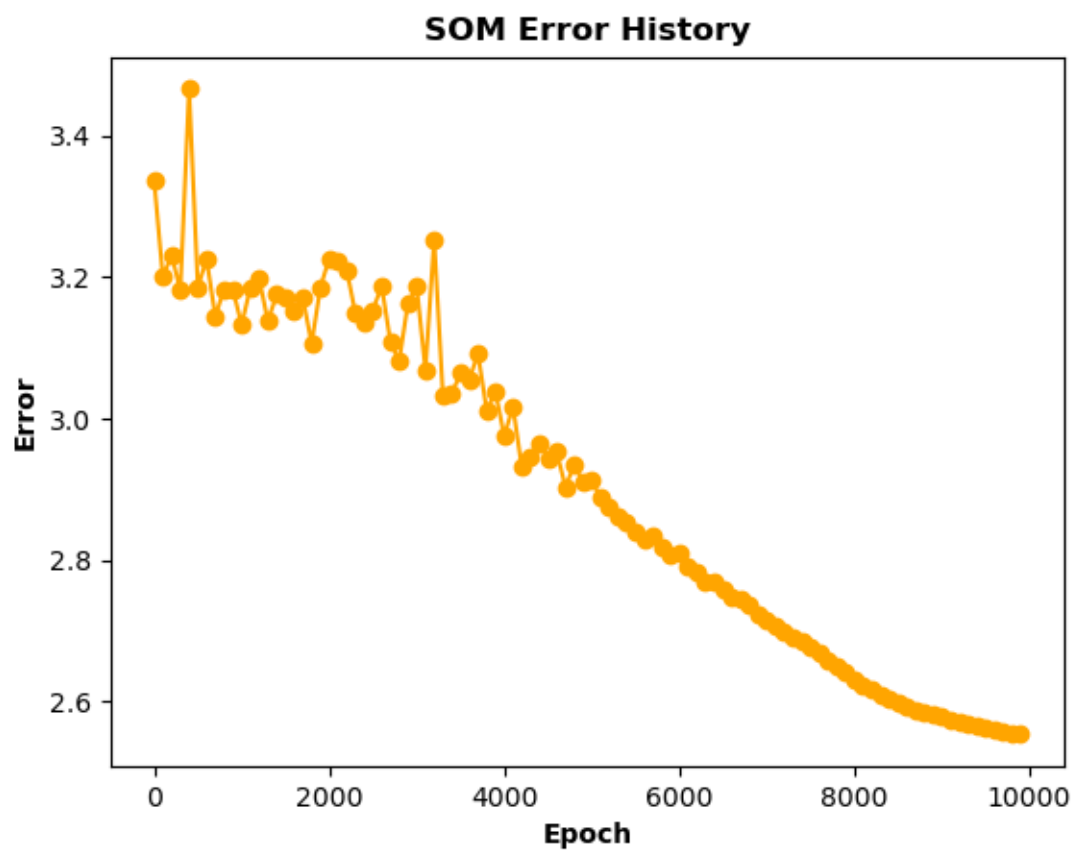
1.2.4 Distance Map:

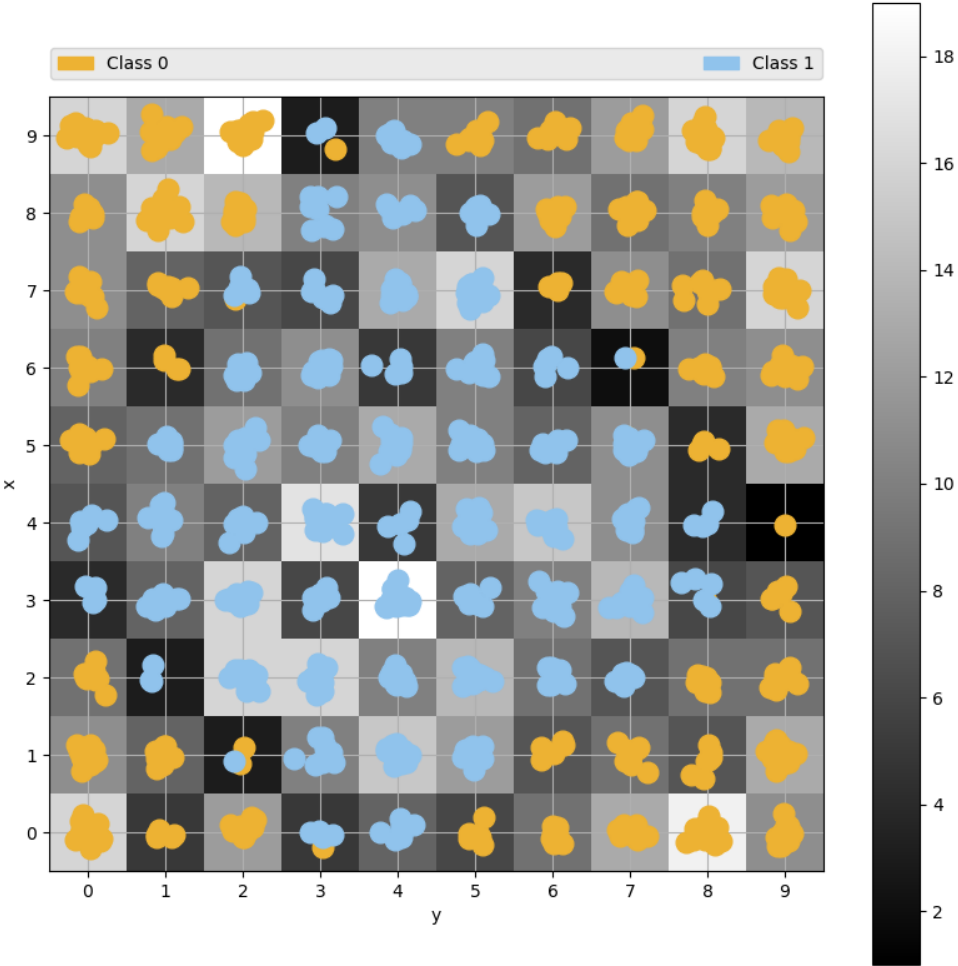
The same way you can handle your own data.

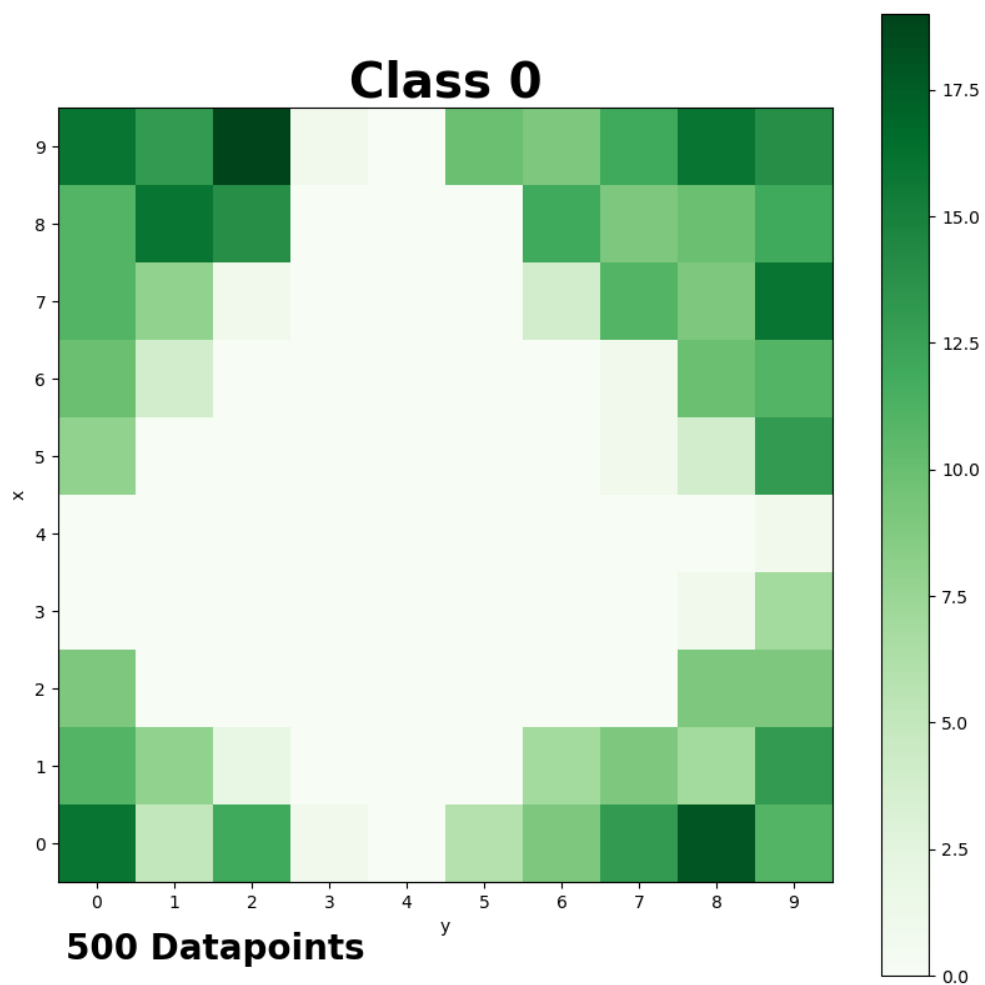
1.3 Methods / Functions

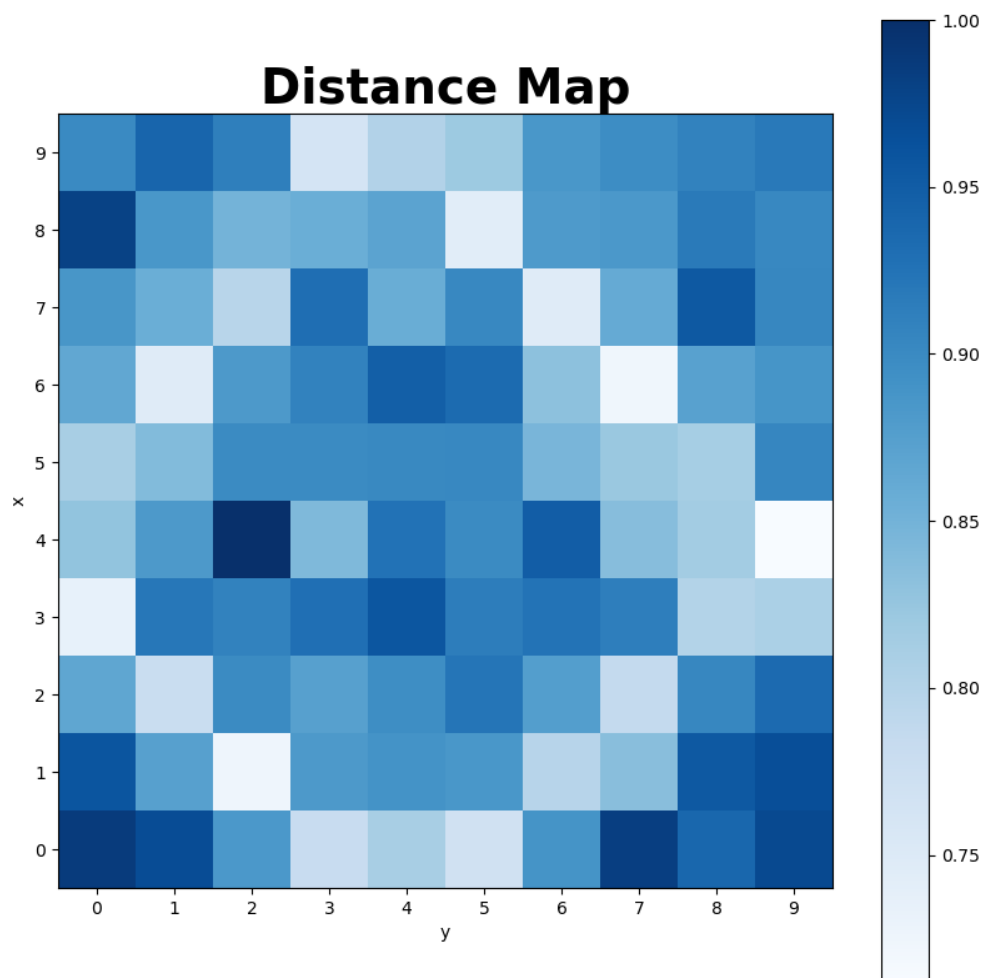
The SOM class has the following methods:

- `initialize(data, how='pca')`: initialize the SOM, either via Eigenvalues (pca) or randomly (random)
- `winner(vector)`: compute the winner neuron closest to a given data point in vector (Euclidean distance)
- `cycle(vector)`: perform one iteration in adapting the SOM towards the chosen data point in vector
- `fit(data, epochs=0, save_e=False, interval=1000, decay='hill')`: train the SOM on the given data for several epochs
- `transform(data)`: transform given data in to the SOM space
- `distance_map(metric='euclidean')`: get a map of every neuron and its distances to all neighbors based on the neuron weights
- `winner_map(data)`: get the number of times, a certain neuron in the trained SOM is winner for the given data
- `winner_neurons(data)`: for every data point, get the winner neuron coordinates









- `som_error(data)`: calculates the overall error as the average difference between the winning neurons and the data
- `get_neighbors(datapoint, data, labels, d=0)`: get the labels of all data examples that are `d` neurons away from `datapoint` on the map
- `save(filename)`: save the whole SOM instance into a pickle file
- `load(filename)`: load a SOM instance from a pickle file
- `plot_point_map(data, targets, targetnames, filename=None, colors=None, markers=None, density=True)`: visualize the som with all data as points around the neurons
- `plot_density_map(data, filename=None, internal=False)`: visualize the data density in different areas of the SOM.
- `plot_class_density(data, targets, t, name, colormap='Oranges', filename=None)`: plot a density map only for the given class
- `plot_distance_map(colormap='Oranges', filename=None)`: visualize the distance of the neurons in the trained SOM
- `plot_error_history(color='orange', filename=None)`: visualize the training error history after training (fit with `save_e=True`)

1.4 References:

[1] Kohonen, T. Self-Organized Formation of Topologically Correct Feature Maps. Biol. Cybern. 1982, 43 (1), 59–69.

This work was partially inspired by [ramalina's som implementation](#) and [JustGlowing's minisom](#).

1.5 Documentation:

Documentation for som-pbc is hosted on readthedocs.io.

DOCUMENTATION FOR THE SOM MODULE

class som.SOM(*x: int, y: int, alpha_start: float = 0.6, sigma_start: Optional[float] = None, seed: Optional[int] = None*)

Class implementing a self-organizing map with periodic boundary conditions. It has the following methods:

cycle(*vector: ndarray, verbose: bool = True*)

Perform one iteration in adapting the SOM towards a chosen data point

Parameters

- **vector** (*np.ndarray*) – current data point
- **verbose** (*bool*) – verbosity control

distance_map(*metric: str = 'euclidean'*)

Get the distance map of the neuron weights. Every cell is the normalised average of all distances between the neuron and all other neurons.

Parameters

metric (*str*) – distance metric to be used (see `scipy.spatial.distance.cdist`)

Returns

normalized sum of distances for every neuron to its neighbors, stored in `SOM.distmap`

fit(*data: ndarray, epochs: int = 0, save_e: bool = False, interval: int = 1000, decay: str = 'hill', verbose: bool = True*)

Train the SOM on the given data for several iterations

Parameters

- **data** (*np.ndarray*) – data to train on
- **epochs** (*int, optional*) – number of iterations to train; if 0, `epochs=len(data)` and every data point is used once
- **save_e** (*bool, optional*) – whether to save the error history
- **interval** (*int, optional*) – interval of epochs to use for saving training errors
- **decay** (*str, optional*) – type of decay for alpha and sigma. Choose from 'hill' (Hill function) and 'linear', with 'hill' having the form $y = 1 / (1 + (x / 0.5) ** 4)$
- **verbose** (*bool*) – verbosity control

get_neighbors(*datapoint: ndarray, data: ndarray, labels: ndarray, d: int = 0*) → ndarray

return the labels of the neighboring data instances at distance *d* for a given data point of interest

Parameters

- **datapoint** (*np.ndarray*) – descriptor vector of the data point of interest to check for neighbors
- **data** (*np.ndarray*) – reference data to compare *datapoint* to
- **labels** (*np.ndarray*) – array of labels describing the target classes for every data point in *data*
- **d** (*int*) – length of Manhattan distance to explore the neighborhood (0: same neuron as data point)

Returns

found neighbors (labels)

Return type

np.ndarray

initialize(*data: ndarray, how: str = 'pca'*)

Initialize the SOM neurons

Parameters

- **data** (*numpy.ndarray*) – data to use for initialization
- **how** (*str*) – how to initialize the map, available: *pca* (via 4 first eigenvalues) or *random* (via random values normally distributed in the shape of *data*)

Returns

initialized map in *SOM.map*

load(*filename: str*)

Load a SOM instance from a pickle file.

Parameters

filename (*str*) – filename (best to end with .p)

Returns

updated instance with data from *filename*

plot_class_density(*data: ndarray, targets: Union[list, ndarray], t: int = 1, name: str = 'actives', colormap: str = 'gray', example_dict: Optional[dict] = None, filename: Optional[str] = None*)

Plot a density map only for the given class

Parameters

- **data** (*np.ndarray*) – data to visualize the SOM density (number of times a neuron was winner)
- **targets** (*list, np.ndarray*) – array of target classes (0 to len(targetnames)) corresponding to data
- **t** (*int*) – target class to plot the density map for
- **name** (*str*) – target name corresponding to target given in *t*
- **colormap** (*str*) – colormap to use, select from matplotlib sequential colormaps
- **example_dict** (*dict*) – dictionary containing names of examples as keys and corresponding descriptor values as values. These examples will be mapped onto the density map and marked
- **filename** (*str*) – optional, if given, the plot is saved to this location

Returns

plot shown or saved if a filename is given

plot_density_map(*data: ndarray, colormap: str = 'gray', filename: Optional[str] = None, example_dict: Optional[dict] = None, internal: bool = False*)

Visualize the data density in different areas of the SOM.

Parameters

- **data** (*np.ndarray*) – data to visualize the SOM density (number of times a neuron was winner)
- **colormap** (*str*) – colormap to use, select from matplotlib sequential colormaps
- **filename** (*str*) – optional, if given, the plot is saved to this location
- **example_dict** (*dict*) – dictionary containing names of examples as keys and corresponding descriptor values as values. These examples will be mapped onto the density map and marked
- **internal** (*bool*) – if True, the current plot will stay open to be used for other plot functions

Returns

plot shown or saved if a filename is given

plot_distance_map(*colormap: str = 'gray', filename: Optional[str] = None*)

Plot the distance map after training.

Parameters

- **colormap** (*str*) – colormap to use, select from matplotlib sequential colormaps
- **filename** (*str*) – optional, if given, the plot is saved to this location

Returns

plot shown or saved if a filename is given

plot_error_history(*color: str = 'orange', filename: Optional[str] = None*)

plot the training reconstruction error history that was recorded during the fit

Parameters

- **color** (*str*) – color of the line
- **filename** (*str*) – optional, if given, the plot is saved to this location

Returns

plot shown or saved if a filename is given

plot_point_map(*data: ndarray, targets: Union[list, ndarray], targetnames: Union[list, ndarray], filename: Optional[str] = None, colors: Optional[Union[list, ndarray]] = None, markers: Optional[Union[list, ndarray]] = None, colormap: str = 'gray', example_dict: Optional[dict] = None, density: bool = True, activities: Optional[Union[list, ndarray]] = None*)

Visualize the som with all data as points around the neurons

Parameters

- **data** (*np.ndarray*) – data to visualize with the SOM
- **targets** (*list, np.ndarray*) – array of target classes (0 to len(targetnames)) corresponding to data
- **targetnames** (*list, np.ndarray*) – names describing the target classes given in targets

- **filename** (*str*, *optional*) – if provided, the plot is saved to this location
- **colors** (*list*, *np.ndarray*, *None*; *optional*) – if provided, different classes are colored in these colors
- **markers** (*list*, *np.ndarray*, *None*; *optional*) – if provided, different classes are visualized with these markers
- **colormap** (*str*) – colormap to use, select from matplotlib sequential colormaps
- **example_dict** (*dict*) – dictionary containing names of examples as keys and corresponding descriptor values as values. These examples will be mapped onto the density map and marked
- **density** (*bool*) – whether to plot the density map with winner neuron counts in the background
- **activities** (*list*, *np.ndarray*, *None*; *optional*) – list of activities (e.g. IC50 values) to use for coloring the points accordingly; high values will appear in blue, low values in green

Returns

plot shown or saved if a filename is given

save(*filename: str*)

Save the SOM instance to a pickle file.

Parameters

filename (*str*) – filename (best to end with .p)

Returns

saved instance in file with name *filename*

som_error(*data: ndarray*) → float

Calculates the overall error as the average difference between the winning neurons and the data points

Parameters

data (*np.ndarray*) – data to calculate the overall error for

Returns

normalized error

Return type

float

transform(*data: ndarray*) → ndarray

Transform data in to the SOM space

Parameters

data (*np.ndarray*) – data to be transformed

Returns

transformed data in the SOM space

Return type

np.ndarray

winner(*vector: ndarray*) → ndarray

Compute the winner neuron closest to the vector (Euclidean distance)

Parameters

vector (*np.ndarray*) – vector of current data point(s)

Returns

indices of winning neuron

Return type

np.ndarray

winner_map(*data: ndarray*) → ndarray

Get the number of times, a certain neuron in the trained SOM is the winner for the given data.

Parameters

data (*np.ndarray*) – data to compute the winner neurons on

Returns

map with winner counts at corresponding neuron location

Return type

np.ndarray

winner_neurons(*data: ndarray*) → ndarray

For every datapoint, get the winner neuron coordinates.

Parameters

data (*np.ndarray*) – data to compute the winner neurons on

Returns

winner neuron coordinates for every datapoint

Return type

np.ndarray

som.**man_dist_pbc**(*m: ndarray, vector: ndarray, shape: tuple = (10, 10)*) → ndarray

Manhattan distance calculation of coordinates with periodic boundary condition

Parameters

- **m** (*np.ndarray*) – array / matrix (reference)
- **vector** (*np.ndarray*) – array / vector (target)
- **shape** (*tuple, optional*) – shape of the full SOM

Returns

Manhattan distance for v to m

Return type

np.ndarray

EXAMPLE SCRIPTS

Using som-pbc to map to random distributions:

```
import numpy as np
from som import SOM

# generate some random data with 36 features
data1 = np.random.normal(loc=-.25, scale=0.5, size=(500, 36))
data2 = np.random.normal(loc=.25, scale=0.5, size=(500, 36))
data = np.vstack((data1, data2))

som = SOM(10, 10) # initialize the SOM
som.fit(data, 10000, save_e=True, interval=100) # fit the SOM for 10000 epochs, save
↳ the error every 100 steps
som.plot_error_history(filename='../images/som_error.png') # plot the training error
↳ history

targets = np.array(500 * [0] + 500 * [1]) # create some dummy target values

# now visualize the learned representation with the class labels
som.plot_point_map(data, targets, ['Class 0', 'Class 1'], filename='../images/som.png')
som.plot_class_density(data, targets, t=0, name='Class 0', filename='../images/class_0.
↳ png')
som.plot_distance_map(filename='../images/distance_map.png') # plot the distance map
↳ after training
```

Advanced script to train, save and load soms:

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Alex Müller    2021-05-18    Created
"""

import logging
import os
import sys
import time
from argparse import ArgumentParser

import numpy as np
import pandas as pd
```

(continues on next page)

(continued from previous page)

```

from som import SOM

logger = logging.getLogger(__name__)
__version__ = '1.0'
__author__ = 'Alex Müller'

def main(in_file, out_file, x, y, epochs, ref=None, test=False, verbose=0):
    if test:
        df = pd.DataFrame(in_file, columns=range(in_file.shape[1]))
    else:
        df = pd.read_table(in_file, sep='\t', low_memory=True, index_col=0)

    s = df.shape[0]
    df.dropna(axis=0, how='any', inplace=True)
    sn = df.shape[0]
    if s != sn:
        logger.warning('%d rows dropped due to missing values' % (s - sn))

    s = df.shape[1]
    df = df.select_dtypes(include=[np.number])
    sn = df.shape[1]
    if s != sn:
        logger.warning('%d columns dropped due to non-numeric data type' % (s - sn))

    basedir = os.path.dirname(os.path.abspath(__file__))
    som = SOM(x, y)
    if ref == 'IRCI':
        som = som.load('/SOM.pkl')
        embedding = som.winner_neurons(df.values)
    else:
        som.fit(df.values, epochs, verbose=verbose)
        embedding = som.winner_neurons(df.values)
        if ref == 'Create':
            som.save(basedir + '/SOM.pkl')

    emb_df = pd.DataFrame({'ID': df.index})
    emb_df['X'] = embedding[:, 1]
    emb_df['Y'] = embedding[:, 0]
    if test:
        return emb_df
    else:
        emb_df.to_csv(out_file, index=False, sep='\t')

if __name__ == "__main__":
    description = "Self-Organizing Map\n\n"
    description += "%s [options] -i infile -o outfile\n\n" % os.path.split(__file__)[1]
    description += "%s: version %s - created by %s\n" % (os.path.split(__file__)[1], __
    ↪ version__, __author__)

    parser = ArgumentParser(description=description)

```

(continues on next page)

(continued from previous page)

```

parser.add_argument('-i', '--infile', dest='file_in', metavar='FILE',
                    help='Specify the input file (TAB format with ID in column 1)',
↪ action='store', default="-")
parser.add_argument('-o', '--outfile', dest='file_out', metavar='FILE',
                    help='Specify the output file (default is STDOUT).', action=
↪ 'store', default="-")
parser.add_argument('-x', '--x', dest='x', action='store', type=int, default=10,
                    help='Size of the SOM in x-coordinate')
parser.add_argument('-y', '--y', dest='y', action='store', type=int, default=10,
                    help='Size of the SOM in y-coordinate')
parser.add_argument('-e', '--epochs', dest='epochs', action='store', type=int,
↪ default=1000,
                    help='Number of epochs to train.')
parser.add_argument('-r', '--ref', dest='ref', choices=['Create', 'IRCI', 'None'],
↪ default='None',
                    help='Use or create a reference PCA / UMAP model. If `None`, a
↪ new one is trained (not saved).')
parser.add_argument('-v', '--verbose', dest='verbose', const=1, default=0, type=int,
↪ nargs="?",
                    help="increase verbosity: 0=warnings, 1=info, 2=debug. No number
↪ means info. Default is 0.")
parser.add_argument('-s', '--test', dest='test', type=bool, default=False, action=
↪ 'store', help='Use for testing.')
args = parser.parse_args()

if args.test:
    import matplotlib.pyplot as plt
    from sklearn.datasets import make_blobs

    X = make_blobs(n_features=512, cluster_std=3.)
    T = main(X[0], None, args.x, args.y, args.epochs, ref=args.ref, test=True)
    plt.scatter(T['X'], T['Y'], c=X[1])
    plt.title('SOM test plot')
    plt.savefig('SOM_test.png')
else:
    infile = sys.stdin if args.file_in == '-' else args.file_in
    outfile = sys.stdout if args.file_out == '-' else args.file_out

    # Start Time Monitoring
    timestart = time.time()

    # Initialisation...
    level = logging.WARNING
    if args.verbose == 1:
        level = logging.INFO
    elif args.verbose == 2:
        level = logging.DEBUG
    logging.basicConfig(level=level, format="%asctime)s %(module)s %(levelname)-7s
↪ %(message)s",
                        datefmt="%Y/%b/%d %H:%M:%S")

    try:

```

(continues on next page)

(continued from previous page)

```
    main(infile, outfile, args.x, args.y, args.epochs, args.ref)
except Exception as err:
    logger.warning('Error occurred: %s' % str(err))

if args.verbose:
    timetotal = time.time() - timestart
    logger.info("%s completed!" % os.path.split(__file__)[1])
    logger.info('Total wall time in seconds was: %f' % timetotal)

# Close properly
logging.shutdown()
sys.stdin.close()
sys.stdout.close()
```


LICENSE**MIT License**

Copyright (c) 2018 Alex Müller

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the “Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

CHAPTER
FIVE

TODOS

INDICES AND TABLES

- `genindex`
- `modindex`
- `search`

PYTHON MODULE INDEX

S

`som`, [11](#)

INDEX

C

`cycle()` (*som.SOM method*), 11

D

`distance_map()` (*som.SOM method*), 11

F

`fit()` (*som.SOM method*), 11

G

`get_neighbors()` (*som.SOM method*), 11

I

`initialize()` (*som.SOM method*), 12

L

`load()` (*som.SOM method*), 12

M

`man_dist_pbc()` (*in module som*), 15

module

`som`, 11

P

`plot_class_density()` (*som.SOM method*), 12

`plot_density_map()` (*som.SOM method*), 13

`plot_distance_map()` (*som.SOM method*), 13

`plot_error_history()` (*som.SOM method*), 13

`plot_point_map()` (*som.SOM method*), 13

S

`save()` (*som.SOM method*), 14

`som`

module, 11

`SOM` (*class in som*), 11

`som_error()` (*som.SOM method*), 14

T

`transform()` (*som.SOM method*), 14

W

`winner()` (*som.SOM method*), 14

`winner_map()` (*som.SOM method*), 15

`winner_neurons()` (*som.SOM method*), 15