Community Characterization in Temporal Networks

Release 0.0.1

Bengier Ülgen Kılıç

Dec 16, 2022

CONTENTS:

1 Introduction **3**
- 1.1 Installation/Usage . . . 3

2 Generating time series of spiking neurons **5**
- 2.1 Planting dynamic communities into synthetic time series data . . . 5
- 2.2 Constructing chronologically ordered set of network states . . . 8

3 Dynamic Community Detection (DCD) **11**
- 3.1 Skeleton coupling . . . 11
- 3.2 Computing partition quality . . . 12

4 The `temporal_network` class **15**

5 Indices and tables **31**

Index **33**

Community Characterization in Temporal Networks is our python wrap-up for applying dynamic community detection on temporal networks obtained from spike-train data. We generate spiking neuronal activity with varying community events and compare the performances of 5 different community detection algorithms: MMM, Infomap, Tensor Factorization, DSBM and DPPM where DPPM is not included in this package since it is available in MATLAB.

CHAPTER

ONE

INTRODUCTION

1.1 Installation/Usage

As the package has not been published on PyPi yet, it CANNOT be installed using `pip`.

For now, the suggested method is to put the file `Temporal_Community_Detection.py` in the same directory as your source files and call:

```
from Temporal_Community_Detection import temporal_network
```

If you additionally play with synthetically generated data, refer to the `helpers.py` file and call:

```
from helpers import *
```

CHAPTER

TWO

GENERATING TIME SERIES OF SPIKING NEURONS

One can use the below framework to create time series of spiking neurons which are synchronized in various assymblies. This behavior is exhibited as planted community structure. Note that this package allows simulation of different community events which can be found in The `Temporal Network` class.

2.1 Planting dynamic communities into synthetic time series data

We are going to simulate a growing community evolution scnerio in which one community keeps expanding over time.

```
#### Inputs
fixed_size = int(abs(np.random.normal(30,10))) # choose a fixed number for size of each␣
↪community
layers = 6
num_neurons = fixed_size*layers
comm_sizes = [fixed_size for i in range(layers)]
spike_rates = [int(abs(np.random.normal(20,8))) for i in range(layers)]
display_truth(comm_sizes, community_operation = 'grow')
```

We then create associated time series by planting in dynamic communities determined by the above parameters. We jitter a master spike randomly in order to create communities. We also choose a window size at which a community event will happen.

```
window_size = 1000 # size, in frames, each adjacency matrix correspond to. better to be␣
↪equal to bin_size
k = 5 #parameter for jittering the spikes

spike_rates = [int(abs(np.random.normal(20,8))) for i in range(layers)]

spikes = create_time_series('grow', comm_sizes, spike_rates, windowsize = window_size, k␣
↪= k)
```

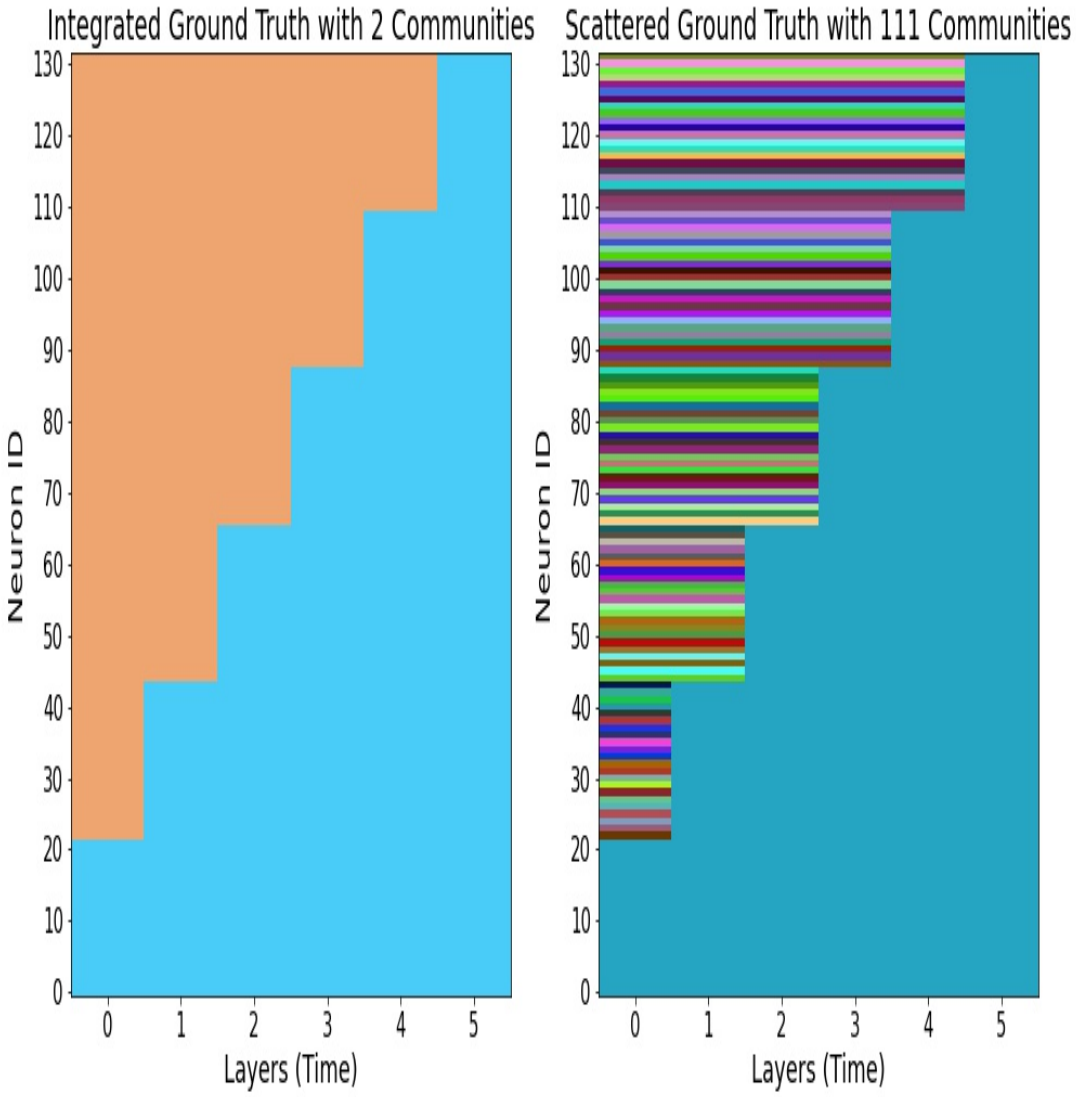

Fig. 1: Colors indicate different community labels in a community expansion scnerio. In the left panel, we consider neurons that aren't part of any communities as a one big community which is lumped together, whereas in the right panel, we assign a unique community label for each neuron that they keep belong until they join the expanding community over time.

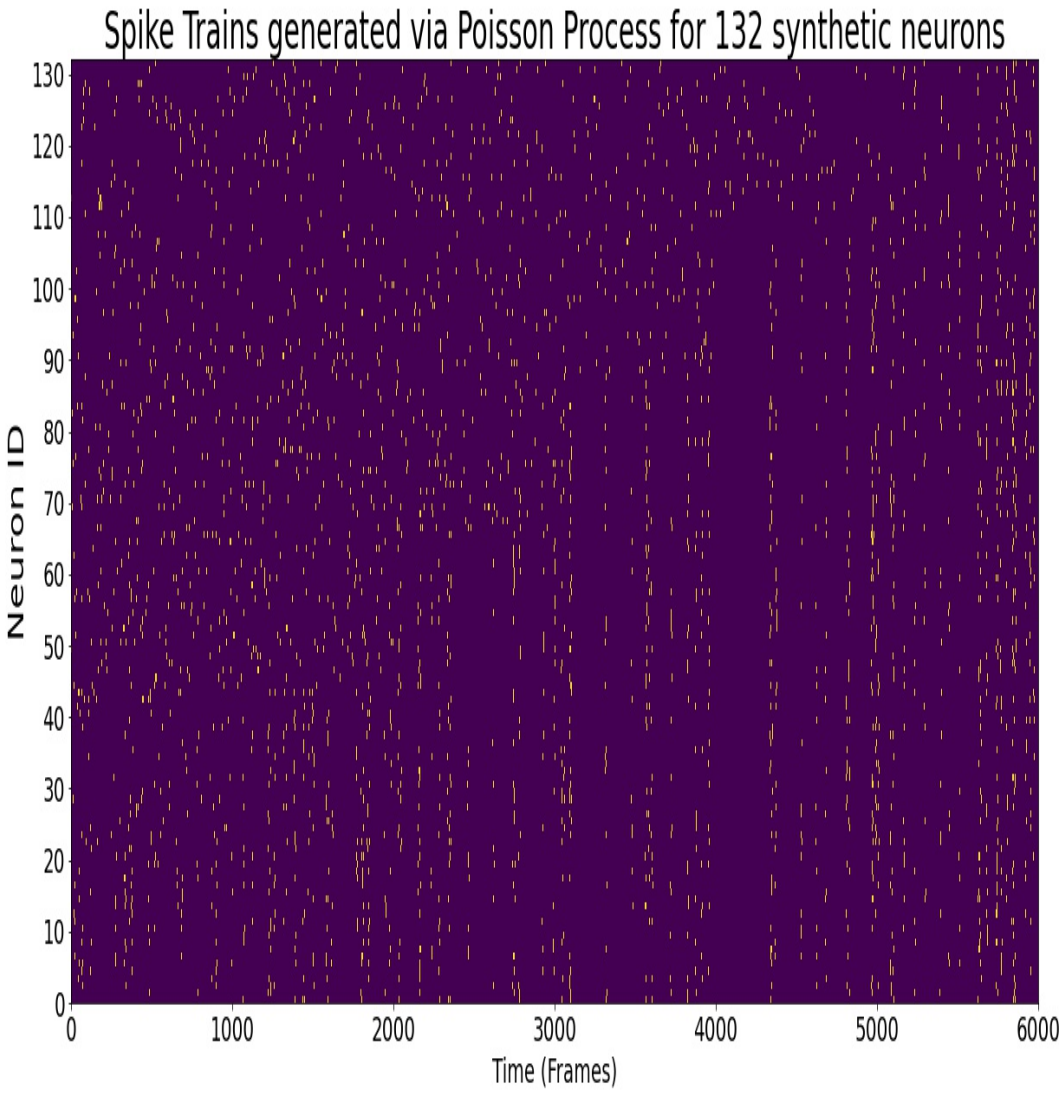

Fig. 2: Spike train generated by Poisson process with planted community structure.

2.2 Constructing chronologically ordered set of network states

We bin the spikes into time-windows and compute positive maximum cross-correlation. We choose our bin size equal to window size to capture community events properly. We also multiply the spike trains with a gaussian kernel to maximize the correlation.

```
adjacency_matrices = []
standard_dev = 1.2 # for gaussian kernel
binned_spikes = bin_time_series(spikes, window_size, gaussian = True, sigma = standard_
↪dev)

for i in range(layers):
    adjacency_matrices.append(cross_correlation_matrix(binned_spikes[i])[0])
```

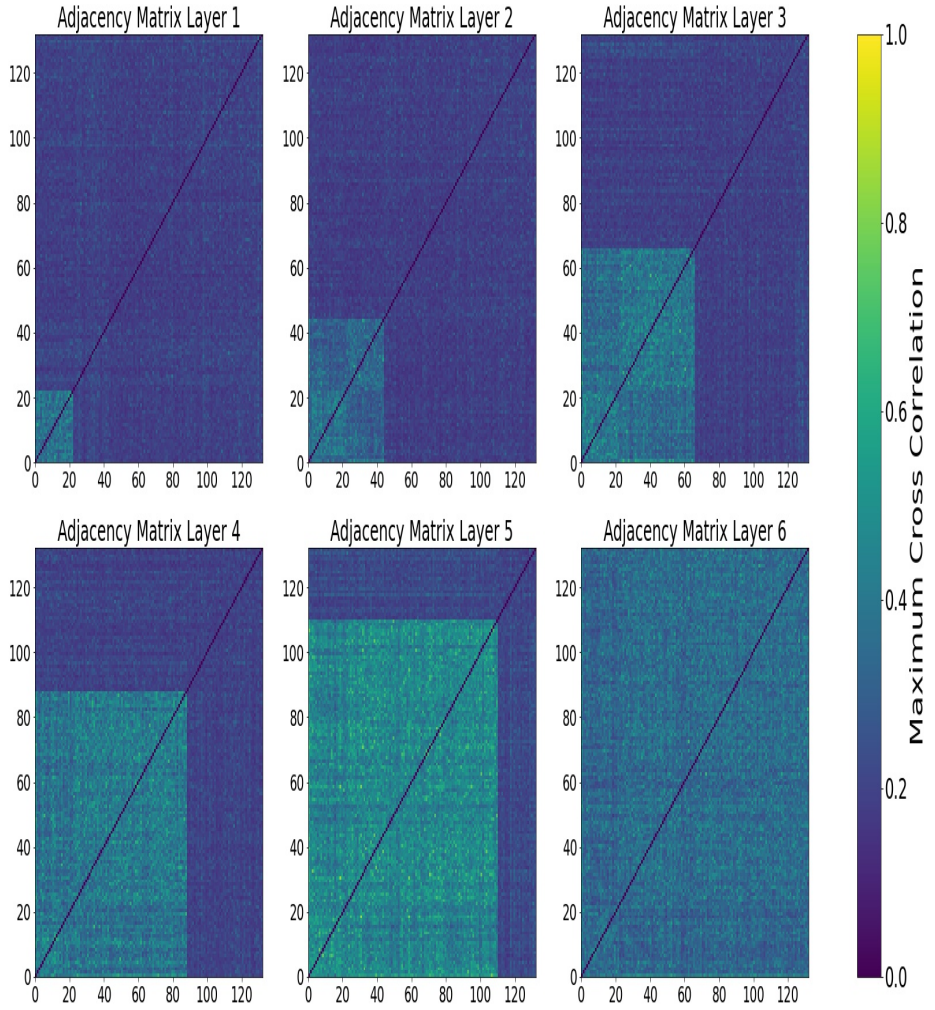

Fig. 3: Resulting adjacency matrices for each snapshot. Observe the planted community structure in each matrix,

CHAPTER

THREE

DYNAMIC COMMUNITY DETECTION (DCD)

Once we have a set of n by n adjacency matrices for a temporal network with T many snapshots, we can create a `temporal_network` object. One can input either a list of adjacency matrices, an edge list or, a supra-adjacency matrix in order to construct the temporal network.

3.1 Skeleton coupling

The API is designed to perform a GridSearch on the parameter spaces of algorithms. For example, if one is utilizing skeleton coupling Multilayer Modularity Maximization, then you can provide a range of values for resolutions parameter and interlayer edge weights. Note that by default, interlayer coupling will be uniform diagonal.

```
interlayer_edge_weights = np.linspace(0, 1, 6)
resolutions_parameters = np.linspace(0.9, 1.1, 6)

partitions, C = TN.run_community_detection(method = 'MMM', ## modularity maximization
                                           update_method = 'skeleton', ## skeleton coupling
                                           interlayers = interlayer_edge_weights,
↪#gridsearch parameters1
                                           resolutions = resolution_parameters, #gridsearch␣
↪parameters2
                                           spikes = spikes) # Spike train
```

Alternatively, if one wants to use skeleton coupling with Infomap, below is an example code. Note that, if one wants to test a single parameter, they need to pass an array-like for the GridSearch to run properly.

```
interlayer_edge_weights = [0.2]
edge_thresholds = [0.5]

thresholded_adjacency_matrices = []
#we need to build a new temporal network object in which edge weights are thresholded
for i in range(layers):
    thresholded_adjacency_matrices.append(threshold(adjacency_matrices[i],edge_
↪thresholds[0]))

TN_thresholded = temporal_network(size, length, window_size, data = 'list__adjacency',␣
↪list_adjacency = thresholded_adjacency_matrices, omega = 1, kind = 'ordinal')

pred_partitions, C = TN_thresholded.run_community_detection(method = 'Infomap', ##␣
↪modularity maximization
                                           update_method = 'skeleton', ## skeleton coupling
```

(continues on next page)

(continued from previous page)

```
                                        interlayers = interlayer_edge_weights,
↪#gridsearch parameters1
                                        thresholds = edge_thresholds, #gridsearch␣
↪parameters2
                                        spikes = spikes) # Spike train
```

In general, below code returns a dictionary whose values are accessed by `'%d,%d'%(t,node)` where $0 <= t <= T_{max} - 1$ is the layer id in which a `node` is belong to and $node$ is the node id. Each value is a list of nodes (or an empty list) indicating the skeleton coupling assignment of the `node` in snaphot `t`.

```
membership_static = TN.infomap_static(adjacency_matrices)
bridge_links = TN.find_skeleton(membership_static)

membership_static = TN.MMM_static()
bridge_links = TN.find_skeleton(membership_static)
```

Refer to the supplementary material of the paper for the psedocode computing skeleton coupling edges.

3.2 Computing partition quality

Once we found predicted partitions, we can compare them with the ground truth to compute accuracy of the algorithms on a grid of parameters. For example, one can compute normalized mutual information (NMI), adjusted rand index (ARI), accuracy and F1-scores.

```
NMI_mmm = np.zeros((len(interlayers), len(resolutions)))
ARI_mmm = np.zeros((len(interlayers), len(resolutions)))
ACC_mmm = np.zeros((len(interlayers), len(resolutions)))
F1S_mmm = np.zeros((len(interlayers), len(resolutions)))

true_labels = generate_ground_truth(comm_sizes, community_operation = 'grow')

for i, e in enumerate(interlayers):
    for j, f in enumerate(resolutions):
        NMI_mmm[i][j] = normalized_mutual_info_score(true_labels,␣
↪list(C[i*len(resolutions)+j].astype(int)))
        ARI_mmm[i][j] = adjusted_rand_score(true_labels, list(C[i*len(resolutions)+j].
↪astype(int)))
        F1S_mmm[i][j] = f1_score(true_labels, list(C[i*len(resolutions)+j].astype(int)),␣
↪average = 'weighted')
        ACC_mmm[i][j] = accuracy_score(true_labels, list(C[i*len(resolutions)+j].
↪astype(int)), normalize = True)
```

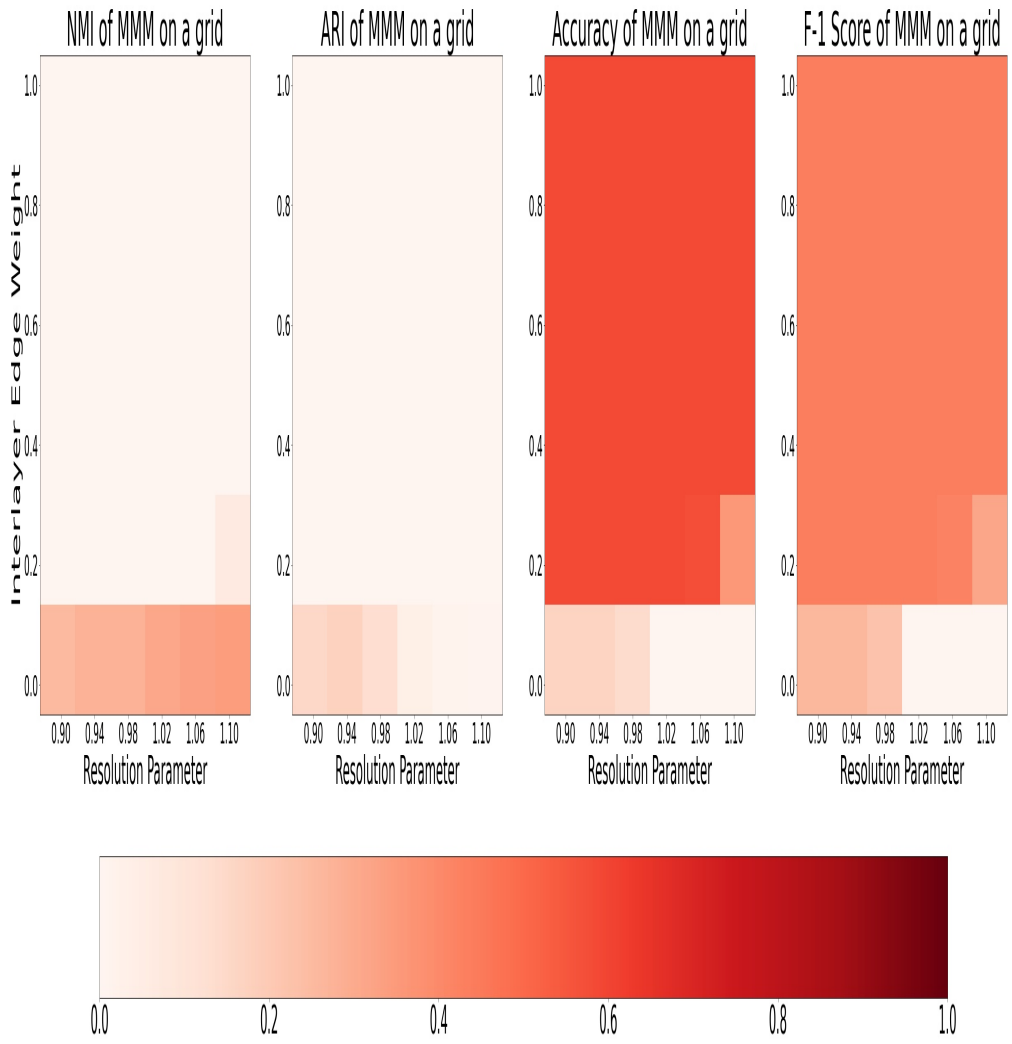

Fig. 1: Shade of the color represents different partition quality metrics in each panel.

CHAPTER

FOUR

THE TEMPORAL_NETWORK CLASS

class temporal_network.**temporal_network**(*size*, *length*, *window_size*, *data*, ***kwargs*)

Bases: `object`

Temporal network object to run dynamic community detection and other multilayer network diagnostics on. Temporal network is a memoryless multiplex network where every node exists in every layer.

size

Number of nodes in any given layer.

Type
int

length

Total number of layers.

Type
int

nodes

A list of node ids starting from 0 to `size-1`.

Type
list

windowsize

Assuming that temporal network is created from a continous time-series data, windowsize is the size of the windows we are splitting the time-series into.

Type
int

supra_adjacency

The supra adjacency matrix to encode the connectivity information of the multilayer network.

Type
array, `size*length x size*length`

list_adjacency

A list of arrays of length `length` where each array is `size x size` encoding the connectivity information of each layer.

Type
list, [array1, array2, . . .]

`edge_list`

A list of length `length` of lists where each element of the sublist is a 4-tuple (i,j,w,t) indicating there is an edge from node i to node j of nonzero weight w in the layer t. So, every quadruplet in the t'th sublist in `edge_list` has 4th entry t.

Type
list, [list1, list2, ...]

Parameters

- **`size`** (*int*) – Number of nodes in any given layer.
- **`length`** (*int*) – Total number of layers.
- **`window_size`** (*int*) – Size of the windows the time series will be divided into.
- **`data`** (*str*) – `temporal_network` accepts three types of connectivity input, `supra_adjacency`, `list_adjacency` and `edge_list` (see the attributes). So, we must specify which one of these types we are submitting the connectivity information to the `temporal_network`. Accordingly, this parameter can be one of the `supra__adjacency`, `list__adjacency` and `edge__list`, respectively.

  Once the data type is understood, object converts the given input into the other two data types so that if it needs to use one of the other types(it is easier to work with `list_adjacency` for example, but some helper functions from different libraries such as `igraph`, processes `edge_list` better), it can switch back and forth quicker.

- **`**kwargs`** –

  supra_adjacency: array, `size*length x size*length`
  The supra adjacency matrix to encode the connectivity information of the multilayer network. Should be provided if `data = supra__adjacency`.

- **`**kwargs`** –

  list_adjacency: list, [array1, array2, ...]
  A list of arrays of length `length` where each array is `size x size` encoding the connectivity information of each layer. Should be provided if `data = list__adjacency`.

- **`**kwargs`** –

  edge_list: list, [list1, list2, ...]
  A list of length `length` of lists where each element of the sublist is a 4-tuple (i,j,w,t) indicating there is an edge from node i to node j of nonzero weight w in the layer t. So, every quadruplet in the t'th sublist in `edge_list` has 4th entry t. Should be provided if `data = edge__list`.

- **`**kwargs`** –

  omega: int
  Interlayer edge coupling strength. Should be provided if data is `list__adjacency` or `edge__list`. For now, we will assume all the coupling is going to be diagonal with a constant strength.

  TODO: extend omega to a vector(for differing interlayer diagonal coupling strengths) and to a matrix(for non-diagonal coupling).

- **`**kwargs`** –

  kind:
  Interlayer coupling type. Can be either `ordinal` where only the adjacent layers are cou-

pled or `cardinal` where all layers are pairwise coupled with strength `omega`. Should be provided if data is `list__adjacency` or `edge__list`.

`MMM_static()`

Running leiden algorithm on the individual layers of temporal network for skeleton coupling.

Returns
inter_membership – List that contain layer, membership and node information respectively.

Return type
triple list

`aggragate(normalized=True)`

Helper function to aggragate layers of the temporal network.

Parameters
`normalized` (*`Bool`*) – divides the total edge weight of each edge by the number of layers(`self.length`).

Return type
`n x n` aggragated adjacecy array.

`bin_time_series(array, gaussian=True, **kwargs)`

Helper function for windowing a given time series of spikes into a desired size matrices.

Parameters

- **`array`** (*`np.array`*) – `n x t` array where n is the number of neurons and t is the length of the time series.
- **`gaussain`** (*`bool (Default: True)`*) – If True, every spike in the time series is multiplied by a 1d-gaussian of size `sigma`.
- **`**kwargs`** – sigma: size of the gaussian (See gaussian_filter).

Returns
A – Matrix of size `l x n x windowsize` where l is the number of layers (`= t/self.windowsize`), n is the number of neurons.

Return type
np.array

`binarize(array, thresh=None)`

Helper function to binarize the network edges.

Parameters

- **`array`** (*`np.array`*) – Input array corresponding to one layer of the temporal network.
- **`thresh`** (*`float (Default: None)`*) – if provided, edges with weight less than `thresh` is going to be set to 0 and 1 otherwise. If not provided, thresh = 0.

Returns
binary_spikes – `n x n` binary adjacency matrix.

Return type
np.array

`community(membership, ax)`

Helper function to visualize the community assignment of the nodes. At every run, a random set of colors are generated to indicate community assignment.

Parameters

- **membership** (*list*) – A list of length `number of communities` where each list contains (node,time) pairs indicating the possesion of that `node` at the `time` to that community.
- **ax** (*matplotlib.axis object*) – An axis for plotting of the communites.

Returns

- **comms** (array of shape `n x t`) – array to be visualized.
- **color** (*list*) – list of colors to be plotted for future use.

`community_consensus_iterative`(*C*)

Function finding the consensus on the given set of partitions. See the paper:

'Robust detection of dynamic community structure in networks', Danielle S. Bassett, Mason A. Porter, Nicholas F. Wymbs, Scott T. Grafton, Jean M. Carlson et al.

We apply Leiden algorithm to maximize modularity.

Parameters
C (*array*) – Matrix of size `parameter_space x (length x size)` where each row is the community assignment of the corresponding parameters.

Returns
partition – See https://leidenalg.readthedocs.io/en/stable/

Return type
Leidenalg object

`create_igraph`()

Helper function that creates igraphs for modularity maximization.

`disjoint_union_attrs`(*graphs*)

Helper function to take the disjoint union of igraph objects. See `slices_to_layers`.

`dsbm_via_graphtool`(*edge_list*, *deg*)

Running DSBM using https://graph-tool.skewed.de

Overlap is True by default according to https://journals.aps.org/pre/abstract/10.1103/PhysRevE.92.042807

Parameters

- **edge_list** (*list ([list1,list2,...])*) – List of lists of length `length``where each list contains the edge list of the corresponding layer. Output of ``process_matrices`.
- **deg** (*Bool*) – If True degree_corrected model will be used and vice versa.

Returns

- **membership** (*list([list1,list2,...])*) – List of lists of length `number_of_communities` where each list contains the community assignment of the nodes it contains.
- **labels** (*list*) – List of length `length x size` containing all the community assignments.

`edgelist2edges`()

Helper function for creating edge lists for iGraph construction.

Returns

- **all_edges** (*list ([list1,list2,...])*) – A list of length *length* lists where each list contains node pairs (i,j) in the corresponding layer.
- **all_weights** (*list ([list1, list2,...])*) – A list of length *length* lists where each list contains floats in the corresponding layer indicating the edge weight between the node pair.

find_comm_size(*n*, *list_of_lists*)

Helper function for finding the comunities in the next layer of a given node.

Parameters

- **n** (*int*) – Node id to be found whose communities of.
- **list_of_lists** (*list*) – First dimension of output of infomap_static.

Returns

- **comm** (*list*) – Community membership of the given node in the next time step.
- **len** (*int*) – Size of that community.

find_skeleton(*static_memberships*)

Function that finds links of skeleton coupling.

Parameters
static_membership (*list*) – Output of infomap_static.

Returns
bridge_links – Dictionary of skeleton links.

Return type
dict

get_attrs_or_nones(*seq*, *attr_name*)

Helper method.

get_normalized_outlinks(*thresholded_adjacency*, *interlayer*)

Helper function for neighborhood coupling that finds the interlayer neighbors of a every node in the next and previous layers and normalizes edge weights.

Parameters

- **thresholded_adjacency** (*list*) – List of adjacency matrices corresponding to every layer of the temporal network.
- **interlayer** (*float*) – The node itselves edge weight that is connected to its future(or past) self that is the maximal among other interlayer neighbors.

Returns

- **interlayer_indices** (*dict (dict['t,i'])*) – Dictionary of interlayer neighbors of a node i in layer t.
- **interlayer_weights** (*dict (dict['t,i'])*) – Dictionary of interlayer weights corresponding to indices of node i in layer t.

infomap(*inter_edge*, *threshold*, *update_method=None*, ***kwargs*)

Function that runs Infomap algorithm on the temporal network.

https://www.mapequation.org

Parameters

- **inter_edge** (*float*) – Interlayer edge weight.
- **threshold** (*float*) – Value for thresholding the network edges. Functional networks obtained by correlation is going to need thresholding with infomap.
- **update_method** (None, local, global, neighborhood or skeleton. Default None.) – Updating the interlayer edges according to either of these methods.

- ****kwargs** –

  spikes: array
  if `local` or `global` update method is being used, initial `spikes` that is used to obtain the correlation matrices needs to be provided.

infomap_static(*thresholded_adjacency*)

Helper function for running infomap on the individual layers of temporal network.

Parameters
thresholded_adjacency (*list*) – List of adjacency matrices.

Returns
inter_membership – List that contain layer, membership and node information respectively.

Return type
triple list

leiden(*G*, *interslice*, *resolution*)

Function that runs Multilayer Modularity Maximization using Leiden solver.

Traag, V.A., Waltman, L. & van Eck, N.J. From Louvain to Leiden: guaranteeing well-connected communities. Sci Rep 9, 5233 (2019). https://doi.org/10.1038/s41598-019-41695-z

Parameters

- **G** (*list ([g1,g2,...])*) – A list of igraph objects corresponding to different layers of the temporal network.
- **interslice** (*float*) – Leidenalg package automatically utilizes diagonal coupling of layers. If a float is provided as `interslice` a uniform interlayer coupling weight is going to be applied for all nodes in all layers. If a list of length `size` is provided, every node will be coupled with themselves with given weight. If a list of, length `length -1`, lists is provided, then you can tune individual interlayer weights as well.
- **resolution** (*float*) – Resolution parameter.

Returns

- **partitions** (*leidenalg object. See https://leidenalg.readthedocs.io/en/stable/*)
- **interslice_partitions** (*leidenalg object. See https://leidenalg.readthedocs.io/en/stable/*)

make_tensor(*rank*, *threshold*, *update_method=None*, ***kwargs*)

Helper function to utilize Tensor Factorization Approach described in:

https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0086028

Parameters

- **rank** (*int*) – Input for predetermined number of communites to be found.
- **threshold** (*float*) – Edge threshold for adjacency matrices.
- **update_method** (`local`, `global` or ``neighborhood``(Default: None)) – Updating the edges according to one of these methods although this is not an applied technique in the literature. Go with None unless you know what you are doing.
- ****kwargs** –

  spikes: array
  Initial spike train matrix of size `n x t`

Returns

- **weights_parafac** (*array*) – See the paper.
- **factors_parafac** (*array*) – See the paper.

membership(*interslice_partition*)

Returns the community assignments from the Leiden algorithm as tuple (n,t) where `n` is the node id `t` is the layer that node belongs to.

neighborhood_flow(*layer*, *node*, *interlayer_indices*, *interlayer_weights*, *thresh*)

Helper function to evaluate the weights of the individual non-diagonal interlinks using jensenshannon entropy. We also threshold weaker interlinks and keep only the ones that have maximal interlayer edge weight for computational purposes. In this sense, we are coupling a maximal neighborhood around a node with previous and future layer.

Parameters

- **layer** (*int*) – Layer that node belongs to.
- **node** (*int*) – Node ID
- **interlayer_indices** (*dict*) – First output of the `get_normalized_outlinks`.
- **interlayer_weights** (*dict*) – Second output of the `get_normalized_outlinks`.
- **thresh** (*float*) – Value for thresholding the weakest `thresh` percentage of interlinks that this node has.
- **Return** –
- **---------** –
- **w** (*float*) – Neighborhood coupling weight.
- **nbr** (*dict*) – Thresholded list of maximal interlinks

neighbors(*node_id*, *layer*)

Helper function for finding the neighbors of a given node.

Parameters

- **node_id** (*int*) – ID of the node to be found the neighbors of.
- **layer** (*int*) – Layer ID of the node that it belongs to.

Returns
neighbors – list of node IDs of the neighbors of `node_id` in layer `layer`.

Return type
list

process_matrices(*threshs*)

Helper function preparing adjacency matrices into the pipeline for DSBM converting the matrix into an edge_list.

Parameters

- **threshs** (*1-D array*) – Set of threshold values.
- **Returns** –
- **-----------** –
- **processed_matrices** (*dict*) – Dictionary of edge list values corresponding to each given threshold value.

process_tensor(*factors*, *rank*)

Helper function for converting the output of make_tensor as in the function membership.

Parameters

- **factors** (*array*) – First output of make_tensor.
- **rank** (*int*) – Number of communities to be found which is an ad-hoc parameter in this algorithm.

Returns

- **membership** (*list ([list1,list2,...])*) – List of length rank of lists where each list contains the membership information of the nodes belonging to corresponding community.
- **comms** (*list*) – List of length length x size for community assignment.

raster_plot(*spikes*, *ax*, *color=None*, ***kwargs*)

Plots the raster plot of the spike activity on a given axis. if color provided, raster includes the community assignments.

Parameters

- **spikes** (array n x t) – Initial spike train array for n nodes of length t.
- **ax** (*matplotlib.axis object*) – axis to be plotted.
- **color** (*list*) – Second output of the self.community. List of colors of length number of communities.
- ****kwargs** –

  comm_assignment: array
  First output of the self.community. If not provided raster is going to be plotted blue.

run_community_detection(*method*, *update_method=None*, *consensus=False*, ***kwargs*)

Wrap-up function to run community detection using one of the 4 methods:

1) Multilayer Modularity Maximization (MMM): https://leidenalg.readthedocs.io/en/stable/

P. J. Mucha, T. Richardson, K. Macon, M. A. Porter and J.-P. Onnela, Science 328, 876-878 (2010).

2) Infomap: https://www.mapequation.org

Mapping higher-order network flows in memory and multilayer networks with Infomap, Daniel Edler, Ludvig Bohlin, and Martin Rosvall, arXiv:1706.04792v2.

3) Non-negative tensor factorization using PARAFAC: http://tensorly.org/stable/index.html

Detecting the Community Structure and Activity Patterns of Temporal Networks: A Non-Negative Tensor Factorization Approach, Laetitia Gauvin , André Panisson, Ciro Cattuto.

4) Dynamical Stochastic Block Model (DSBM): https://graph-tool.skewed.de

Inferring the mesoscale structure of layered, edge-valued, and time-varying networks, Tiago P. Peixoto, Phys. Rev. E, 2015.

Parameters

- **method** (*str*) – Either MMM, Infomap, PARA_FACT``(Tensor Factorization) or ``DSBM indicating the community detection method.

- **update_method** (*str (Default: None)*) – Interlayer edges will be processed based on one of the three methods, either 'local', 'global', 'neigborhood' and 'skeleton'. Available only for `MMM` and `Infomap`.
- **consensus** (*bool*) – Statistically significant partitions will be found from a given set of parameters. See `community_consensus_iterative`.
- ****kwargs** –

  interlayers: 1-D array like
  A range of values for setting the interlayer edges of the network. Pass this argument if you are using `MMM` or `Infomap`.
- ****kwargs** –

  resolutions: 1-D array like
  A range of values for the resolution parameters. Pass this argument if you are using `MMM`.
- ****kwargs** –

  thresholds: 1-D array like
  A range of values to threshold the network. Pass this argumment if you are using `Infomap`, `PARA_FACT` or `DSBM`.
- ****kwargs** –

  ranks: 1-D array like
  A range of integers for ad-hoc number of communities. Pass this argument if you are using `PARA_FACT`.
- ****kwargs** –

  degree_correction: list
  A list of boolean values(either True or False) for degree correction. Pass this argument if you are using `DSBM`.
- ****kwargs** –

  spikes: 2-D array
  Initial spike train array containing the spikes of size `n x t`. Pass this argument if your `update_method` is `local` or `global`.

Returns

- **membership_partitions** (*dict*) – Dictionary with keys as first set of parameters lists and second set of parameters list indices indicating the community assignment of each node.
- **C** (*array*) – Matrix of size `parameter_space x(length x size)`. This is the input for `community_consensus_iterative`.

`slices_to_layers`(*G_coupling*, *interlayer_indices*, *interlayer_weights*, *update_method*, *slice_attr='slice'*, *vertex_id_attr='id'*, *edge_type_attr='type'*, *weight_attr='weight'*)

Actual function implementing non-diagonal coupling with Modularity Maximization. Leiden algorithm's python package inherently only allows diagonal coupling. So, this function is needed for non-diagonal coupling.

`threshold`(*array*, *thresh*)

Helper function to threshold the network edges.

Parameters

- **array** (*np.array*) – Input array corresponding to one layer of the temporal network.

- **thresh** (*float*) – Threshold to keep the edges stronger than this value where weaker edges are going to be set to 0.

Returns
thresholded_array – `n x n` thresholded adjacency matrix.

Return type
np.array

time_slices_to_layers(*graphs*, *interlayer_indices*, *interlayer_weights*, *update_method*, *interslice_weight=1*, *slice_attr='slice'*, *vertex_id_attr='id'*, *edge_type_attr='type'*, *weight_attr='weight'*)

Helper function for implementing non-diagonal coupling with Modularity Maximization. See `slices_to_layers`.

trajectories(*thresh=0.9*, *node_id=None*, *community=None*, *edge_color=True*, *pv=None*)

Function graphing the edge trajcetories of the temporal network.

Parameters

- **thresh** (*float*) – Threshold for keeping filtering the edge weights.
- **node_id** (*int (Default: None)*) – If None, function is going to graph all of the nodes's trajectories.
- **community** (*array (Default: None)*) – First output of `self.community` indicating the community assignment of the nodes if exists.
- **edge_color** (*bool*) – Different colors on each layer if True, black otherwise.
- **pv** (*list*) – Pass a list of pv cell indices or None –dashes the pv cells.

update_interlayer(*spikes*, *X*, *omega_global*, *percentage*, *method*)

Function for local and global updates. This function assumes diagonal coupling and evaluates the interlink weights according to the `local` or `global` change in some nodal property, spike rates in our case.

Parameters

- **spikes** (*array*) – Initial spike train array.
- **X** (*float*) – Value for determining if the nodal property between consecutive layers(local), or compared to global average, is less than `X` standard deviation.
- **omega_global** (*float*) – Initial interlayer value for all diagonal links.
- **percentage** (*float*) – If the nodal property is less than `X` standard deviation, for a given node, interlayer edge weight is adjusted so that new weight is equal to `omega_global x percentage`.
- **method** (*'local' or 'global'*) – Method for updating the interlayer edges. If local a comparison between consecutive layers is made and if global, overall average of the spike rates are hold as a basis.
- **Returns** –
- `----------` –
- **interlayers** (*list*) – A list of length `(length-1) x size` indicating interlayer edge weights of every node.

helpers.**bin_time_series**(*array*, *binsize*, *gaussian=True*, ***kwargs*)

Helper function for windowing the time series into smaller chunks.

Parameters

- **array** (*2_D array*) – n x t matrix where n is the number of neuorons and t is the length of the time series.
- **binsize** (*int*) – Size of each window. This number needs to be smaller than t and a positive divider of t.
- **gaussian** (*bool (Default: True)*) – If True, each spike is going to be multiplied by a 1-D gaussian of length sigma.
- ****kwargs** –

  sigma: float
  Size of the gaussian. See gaussian_filter.

Returns
A – Matrix of size l x n x t where l is the number of windows(=t/binsize), n is number of neurons and t is the length of the time series.

Return type
array

helpers.**binarize**(*array*, *thresh=None*)

Function for binarizing adjacency matrices.

Parameters

- **array** (*array like*) – Cross-correlation matrix.
- **thresh** (*float (Default: None)*) – If None, entries that are non-zero are going to be set to 1. If a value between [0,1] is given, then every entry smaller than thresh will be set to 0 and 1 otherwise.

Returns
binary_spikes – Binarized cross-correlation matrix of same size.

Return type
array like

helpers.**community_consensus_iterative**(*C*)

Function finding the consensus on the given set of partitions. See the paper:

'Robust detection of dynamic community structure in networks', Danielle S. Bassett, Mason A. Porter, Nicholas F. Wymbs, Scott T. Grafton, Jean M. Carlson et al.

We apply Leiden algorithm to maximize modularity.

Parameters
C (*array*) – Matrix of size parameter_space x (length * size) where each row is the community assignment of the corresponding parameters.

Returns
partition – See https://leidenalg.readthedocs.io/en/stable/

Return type
Leidenalg object

helpers.**consensus_display**(*partition*, *n*, *t*)

Helper function to visualize the consensus from community_consensus_iterative.

Parameters

- **partition** (*Leidenalg object*) – See https://leidenalg.readthedocs.io/en/stable/
- **n** (*int*) – Number of neurons.

- **t** (*int*) – Number of layers.

Returns

- **comms** (*array*) – Community membership array of size `n x t`.
- **cmap** (*matplotlib object*) – Colormap used to plot the membership information.
- **color** (*list*) – List of strings encoding the colors of the communities.

helpers.**create_time_series**(*operation*, *community_sizes*, *spiking_rates*, *spy=True*, *windowsize=1000*, *k=5*)

Main function for creating spike trains using Homogeneous Poisson Process.

Parameters

- **operation** (`grow`, `contract`, `merge` or `transient`) – Community operation.
- **community_sizes** (*list or list of lists*) – If the `operation` is `grow` (or `contract`), this should be a list indicating the number of neurons joining (or leaving from) the main community. If the `operation` is `merge` or `transient`, then this should be a list of lists([list1,list2,...]) where each list contains sizes of the communities in that layer.
- **spike_rates** (*list or list of lists*) – This should be of same size and shape as `community_sizes` indicating the spike rates of the corresponding communities.
- **spy** (*bool (Deafult: True)*) – Displays the time series if True.
- **windowsize** (*int (Default: 1000)*) – Length of the window size for a new layer of events to be created.
- **k** (*int (Default: 5)*) – Constant for jittering the spikes when creating new communities.

Returns

spikes – Matrix of size `n x t` where n is the number of neuorons and t is the length of the time series.

Return type

array

helpers.**cross_correlation_matrix**(*data*)

Main function to call for computing cross-correlation matrix of a time series.

Parameters

data (*array*) – `n x t` matrix where n is the number of neuorons and t is the length of the time series.

Returns

- **X_full** (*array*) – `n x n` symmetric cross-correlation matrix.
- **X** (*array*) – `n x n` upper triangular cross-correlation matrix.
- **lag** (*array*) – `n x n` lag matrix.

helpers.**display_truth**(*comm_sizes*, *community_operation*, *ax=None*)

Function for displaying the ground truths.

Parameters

- **comm_sizes** (*list, or list of lists*) – This will be passed to `generate_ground_truth` where pad is True by default.
- **community_operation** (`grow`, `contract`, `merge` or `transient`) – Type of the community event which will also be passed to `generate_ground_truth`.

- **ax** (*matplotlib object (Default: None)*) – If None, a new axis will be created, otherwise the ground truth will be plotted to the provided axis.

helpers.**find_repeated**(*l*)

Helper function for generating transient communities.

helpers.**gaussian_filter**(*array*, *sigma*)

Function that multiplies vectors with a gaussian.

Parameters

- **array** (*1_D array like*) – Input vector.
- **sigma** (*float*) – 1 spike turns into 3 non-zero spikes(one at each side of smaller magnitude) with sigma=0.25. 1 spike turns into 5 non-zero spikes(two at each side of smaller magnitude) with sigma=0.50. 1 spike turns into 9 non-zero spikes(four at each side of smaller magnitude) with sigma=1, and so on..

Returns

array – Gaussian vector.

Return type

1_D array like

helpers.**generate_ground_truth**(*comm_sizes*, *method='scattered'*, *pad=False*, *community_operation='grow'*)

Main function that generates ground truth labels for the experiments. Community labels according to two methods one in which the rest of the network except the planted communities are scattered i.e. they all have their own community or they are all in one community, integrated.

Parameters

- **comm_sizes** (*list, or list of lists*) – If the `community_operation` is `grow` (or `contract`), this should be a list indicating the number of neurons joining (or leaving from) the main community. If the `community_operation` is `merge` or `transient`, then this should be a list of lists([list1,list2,...]) where each list contains sizes of the communities in that layer. For example, [[6,1,1,1,1],[6,4]] indicates a 6 neuron community in the first layer and additional 4 neurons, that are independently firing, merges into 1 community in the second layer.
- **method** (`scattered` or `integrated` (Default: 'scattered')) – If the `community_operation` is `grow` (or `contract`), two types of ground truths can be prepared. Integrated is the one where independently firing neurons are grouped together into one single community and scattered is the one with independently firing neurons.
- **pad** (*bool (Default: False)*) – If True, the truth will be padded from the beginning and the end by the exact same community membership.
- **community_operation** (`grow`, `contract`, `merge` or `transient` (Default: 'grow')) – Type of community events that are available. Community expansion, community contraction, community merge and transient communities.

Returns

truth_labels – List of truth labels, of length `n*t` where n is the number of neuorons and t is the number of layers. If pad, length will be `n*(t+2)`.

Return type

list

helpers.**generate_transient**(*comm_per_layer*)

Helper function for creating the time series for the transient communities.

Parameters

comm_per_layer (*list of lists*) – List of lists of length `number of layers` where each list contains the number of communities at that layer.

Returns

- **comm_sizes** (*list of lists*) – Sizes of the communities in the corresponding layers which will be passed to `create_time_series`.
- **spike_rate** (*list of lists*) – Randomly selected corresponding spike rates for the Homogeneous Poisson process that generates spike trains.
- **num_neurons** (*int*) – Number of neurons.

helpers.**getOverlap**(*a*, *b*)

Helper function for generating transient communities. Finds repeated indices.

helpers.**get_repeated_indices**(*l*)

Helper function for generating transient communities.

helpers.**information_recovery**(*pred_labels*, *comm_size*, *truth*, *interlayers*, *other_parameter*, *com_op*)

Function for calculating the quality of the resulting partitions on a parameter plane and visualizes the quality landscape according to NMI, ARI and F1-Score.

Parameters

- **pred_labels** (*list*) – List of truth labels appended in the order of layers. This should be the same length as the output of `generate_ground_truth`.
- **comm_size** (*list, or list of lists*) – This will be passed to `generate_ground_truth` that assumes pad to be True.
- **truth** (`integrated` or `scattered`) – Same as in `generate_ground_truth`.
- **interlayers** (*1_D array like*) – To get the landscape information on a plane of parameters, we pass two array like object. This one is the y-axis one on the result.
- **other_parameter** (*1_D array like*) – This is the x-axis array for quality.
- **com_op** (`grow`, `contract`, `merge` or `transient`) – Same as in `generate_ground_truth`.

Returns

- **fig** (*matplotlib object*) – Figure object for the plots.
- **ax** (*matplotlib object*) – Axis objects for the plots.

helpers.**jitter**(*spike*, *k*)

Function for randomly jittering spikes when generating communities.

Parameters

- **spike** (*array*) – Spike train to be jittered.
- **k** (*int*) – Number of time frames, to the right or to the left, for a spike to be jittered.

Returns

jittered – Jittered spike train.

Return type

array

helpers.**max_norm_cross_corr**(*x1*, *x2*)

Function for computing maximum cross-correlation.

Parameters

- **x1** (*1_D array like*) – First vector.
- **x2** (*1_D array like*) – Second vector.

Returns

- **max_corr** (*int*) – Maximum cross-correlation between the two input vectors.
- **lag** (*int*) – Lag difference where the maximum cross-correlation occurs.

helpers.**normalized_cross_corr**(*x*, *y*)

Function to compute normalized cross-correlation between two vectors.

Parameters

- **x** (*1_D array like*) – First vector.
- **y** (*1_D array like*) – Second vector.

Returns

corr_array – Correlation array between x and y.

Return type

array

helpers.**space_comms**(*comm_size*)

Helper function for spacing the communities randomly for the transient communities.

helpers.**spike_count**(*spikes*, *ax*, *num_bins=None*, *t_min=None*, *t_max=None*)

Helper function to visualize the distribution of the number of spikes in a given spike train.

Parameters

- **spikes** (*array*) – Spike train matrix of size `n x t`.
- **ax** (*matplotlib axis*) – Axis for distribution to be plotted.
- **num_bins** (*int (Default: None)*) – If None, this will be the difference between maximum and minimum number of spikes in a population.
- **t_min** (*int (Default: None)*) – if None, this will be `0`, otherwise spikes will be counted in the given range.
- **t_max** (*int (Default: None)*) – if None, this will be `t`, otherwise spikes will be counted in the given range.

Returns

- **n** (*array*) – The values of the histogram bins.
- **bins** (*array*) – The edges of the bins.

helpers.**threshold**(*array*, *thresh*)

Function for thresholding the adjacency matrices.

Parameters

- **array** (*array like*) – Cross-correlation matrix.
- **thresh** (*float*) – Value in which every entry smaller than `thresh` will be set to 0 and entries greater than `thresh` will stay the same.

Returns
threhsolded_array – Thresholded cross-correlation matrix of same size.

Return type
array like

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

A

aggragate() (*temporal_network.temporal_network method*), 17

B

bin_time_series() (*in module helpers*), 24
bin_time_series() (*temporal_network.temporal_network method*), 17
binarize() (*in module helpers*), 25
binarize() (*temporal_network.temporal_network method*), 17

C

community() (*temporal_network.temporal_network method*), 17
community_consensus_iterative() (*in module helpers*), 25
community_consensus_iterative() (*temporal_network.temporal_network method*), 18
consensus_display() (*in module helpers*), 25
create_igraph() (*temporal_network.temporal_network method*), 18
create_time_series() (*in module helpers*), 26
cross_correlation_matrix() (*in module helpers*), 26

D

disjoint_union_attrs() (*temporal_network.temporal_network method*), 18
display_truth() (*in module helpers*), 26
dsbm_via_graphtool() (*temporal_network.temporal_network method*), 18

E

edge_list (*temporal_network.temporal_network attribute*), 15
edgelist2edges() (*temporal_network.temporal_network method*), 18

F

find_comm_size() (*temporal_network.temporal_network method*), 18
find_repeated() (*in module helpers*), 27
find_skeleton() (*temporal_network.temporal_network method*), 19

G

gaussian_filter() (*in module helpers*), 27
generate_ground_truth() (*in module helpers*), 27
generate_transient() (*in module helpers*), 27
get_attrs_or_nones() (*temporal_network.temporal_network method*), 19
get_normalized_outlinks() (*temporal_network.temporal_network method*), 19
get_repeated_indices() (*in module helpers*), 28
getOverlap() (*in module helpers*), 28

H

helpers
    module, 24

I

infomap() (*temporal_network.temporal_network method*), 19
infomap_static() (*temporal_network.temporal_network method*), 20
information_recovery() (*in module helpers*), 28

J

jitter() (*in module helpers*), 28

L

leiden() (*temporal_network.temporal_network method*), 20
length (*temporal_network.temporal_network attribute*), 15
list_adjacency (*temporal_network.temporal_network attribute*), 15

M

make_tensor() (*temporal_network.temporal_network method*), 20
max_norm_cross_corr() (*in module helpers*), 28
membership() (*temporal_network.temporal_network method*), 21
MMM_static() (*temporal_network.temporal_network method*), 17
module
    helpers, 24

N

neighborhood_flow() (*temporal_network.temporal_network method*), 21
neighbors() (*temporal_network.temporal_network method*), 21
nodes (*temporal_network.temporal_network attribute*), 15
normalized_cross_corr() (*in module helpers*), 29

P

process_matrices() (*temporal_network.temporal_network method*), 21
process_tensor() (*temporal_network.temporal_network method*), 21

R

raster_plot() (*temporal_network.temporal_network method*), 22
run_community_detection() (*temporal_network.temporal_network method*), 22

S

size (*temporal_network.temporal_network attribute*), 15
slices_to_layers() (*temporal_network.temporal_network method*), 23
space_comms() (*in module helpers*), 29
spike_count() (*in module helpers*), 29
supra_adjacency (*temporal_network.temporal_network attribute*), 15

T

temporal_network (*class in temporal_network*), 15
threshold() (*in module helpers*), 29
threshold() (*temporal_network.temporal_network method*), 23
time_slices_to_layers() (*temporal_network.temporal_network method*), 24
trajectories() (*temporal_network.temporal_network method*), 24

U

update_interlayer() (*temporal_network.temporal_network method*), 24

W

windowsize (*temporal_network.temporal_network attribute*), 15