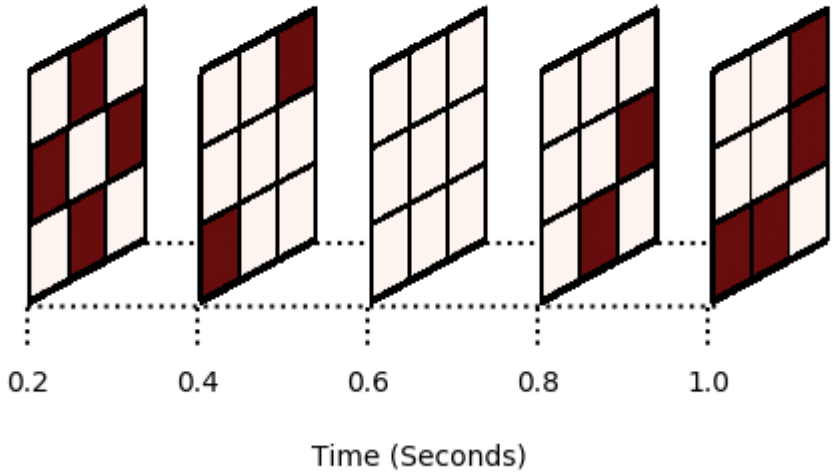
teneto Documentation

Release 0.5.3

William Hedley Thompson

Feb 17, 2021

Contents


0.2
0.4
0.6
0.8
1.0
Time (Seconds)

CHAPTER 1

What are temporal networks?

Temporal networks are, quite simply, network representations that flow through time. They are useful for analysing how a connected system develops, changes or evolves through time. This change in time can depict how information spreads along with a social network or how different brain areas cooperate to perform a task.

This page introduces some of the basic concepts of temporal network theory.

1.1 Node and edges: the basics of a network

A network is a representation of *something* using a graph from mathematics. This *something* can be a representation of an empirical phenomenon or a simulation. A graph contains nodes (sometimes called vertices) and edges (sometimes called links).

Nodes and edges can represent a vast amount of different things in the world. For example, nodes can be friends, cities, or brain regions. Edges between could represent trust relationships, train lines, and neuronal communication.

The flexibility in what nodes are is one of the reasons network theory is very interdisciplinary. The benefits of having network representation are that similar analysis methods can be applied, regardless of what the underlying node or edge represents. This abstractness means that network theory is a very inter-disciplinary subject. However, it also entails that certain concepts have multiple names (e.g. nodes and vertices).

With a network, you can analyse for example, if there is any "hub" node. In transportation networks, there are often hubs which connect many different areas where passengers usually have to change at (e.g. airports like Frankfurt, Heathrow or Denver). In social networks, you can quantify how many steps it is to another person in that network (see the famous six steps to Kevin Bacon).

Mathematically, A network if often referenced as G or (G); *i* and *j* are indices of nodes; a tuple *(i,j)* reference an edge between nodes *i* and *j*. *G* is often expressed in the form of a connectivity matrix (or adjacency matrix) $A_{ij} = 1$ if a connection is present and $A_{ij} = 0$ if a connection is not present. The number of nodes if often referenced to as *N*. Thus, *A* is a *N x N* matrix.

1.2 Different network types

There are a few different versions of networks. Two key distinctions are:

1. Are the connections *binary* or *weighted*.
2. Are the connections *undirected* or *directed*.

If a connection is binary, then (as in the section above) an edge is either present or not. When adding a weight-value, an edge becomes a 3-tuple *(i,j,w)* where *w* is the magnitude of the weight. And in the connectivity matrix, $A_{ij} = w$. Often the weight is between 0 and 1 or -1 and 1, but this does not have to be the case.

When connections are undirected, it means that both nodes share the connection. Examples of such networks can be if two cities are connected by train lines. For such networks $A_{ij} = A_{ji}$. With directed edges, it means that the connection goes from *i* to *j*. Examples of these types of networks can be citation networks. If a scientific article *i* cites another article j, it is not common for *j* to also cite *i*. So in such cases, A_{ij} does not need to equal A_{ji}. It is the common notation for the source node (sending the information) to be first and the target node (receiving the information) to be second.

1.3 Adding a time dimension

In the above formulation of networks A_{ij} only has one edge. In a temporal network, a time-stamp is also included in the edge's tuple. Thus, binary edges are not expressed as 3-tuples (i,j,t) and weighted networks as 4 tuples (i,j,t,w). Connectivity matrices are now three dimensional: $A_{ijt} = 1$ in binary and $A_{ijt} = w$ in weighted networks.

The time indices are an ordered sequence. This ordering can now reveal information about what is occurring in the network through time.

For example, using friends' lists from social network profiles can be used to create a static network about who is friends with who. However, imagine one person enters a group of friends, by seeing when everyone become friends, this gives the network more explanatory power.

Compare the following two figures representing meetings between friends:

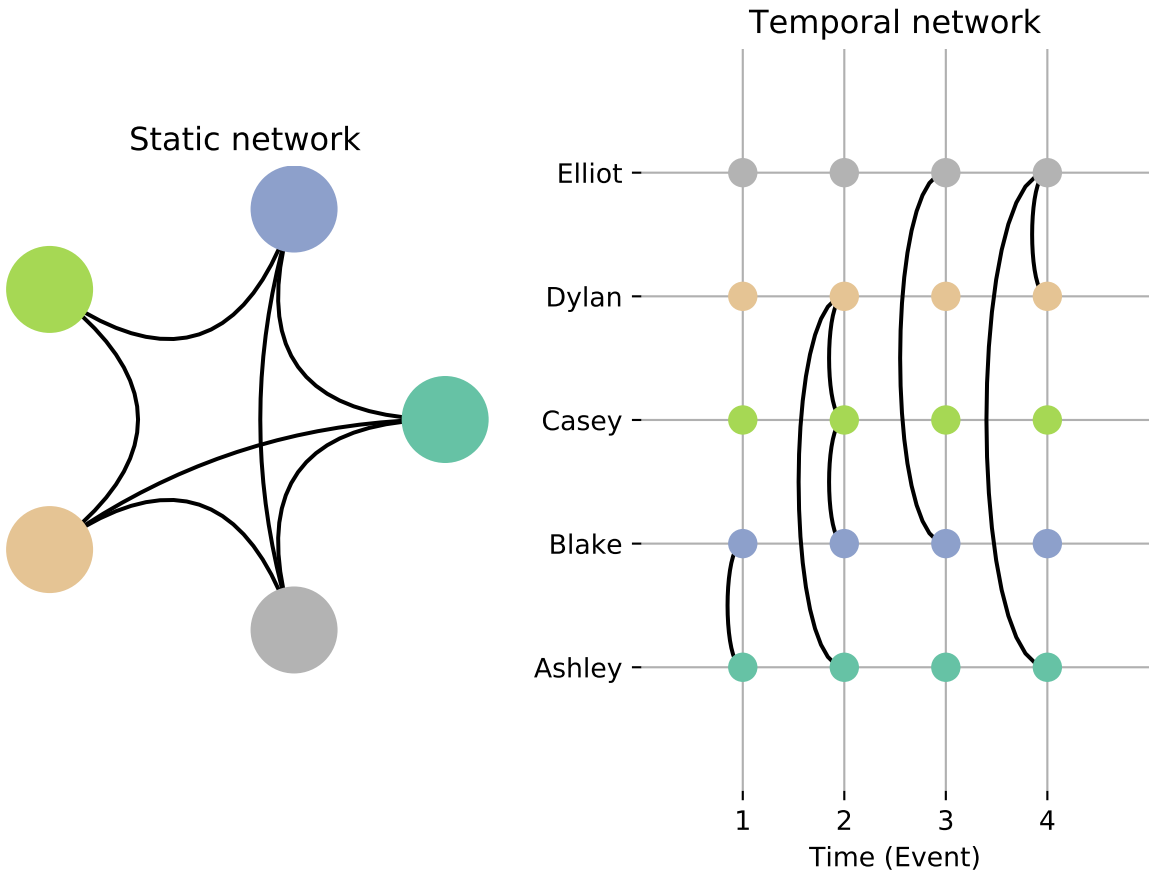
In the static network, on the left, each person (node) is a circle, and each black line connecting the rings is an edge. In this figure, we can see that everyone has met everyone except Dylan (orange) and Casey (light green).

The slice_plot on the left shows nodes (circles) at multiple "slices" (time-points). Each column represents of nodes represents one time-point. The black line connecting two nodes at a time-point signifies that they met at that time-point.

In the temporal network, we can see a progression of who met who and when. At event 1, Ashley and Blake met. Then A-D all met together at event 2. At event 3, Blake met Dylan. And at event 4, Elliot met Dylan and Ashley (but those two themselves did not attend). This depiction allows for new properties to be quantified that missed in a static network.

1.4 What is time-varying connectivity?

Another concept that is often used within fields such as cognitive neuroscience is *time-varying connectivity*. Time-varying connectivity is a larger domain of methods that analyse distributed patterns over time where temporal network theory is one set of analysis methods within it. Temporal network theory analyses time-varying connectivity representations that consist of time-stamped edges between nodes. There are other alternatives to analyse such representations and other time-varying connectivity representations as well (e.g. temporal ICA).


Static network
Temporal network
Elliot
Dylan
Casey
Blake
Ashley
1
2
3
4
Time (Event)

1.5 What is teneto?

Teneto is a python package that can several quantify temporal network measures (more are being added). It can also use methods from time-varying connectivity to derive connectivity estimate from time-series data.

1.6 Further reading

Holme, P., & Saramäki, J. (2012). Temporal networks. Physics reports, 519(3), 97-125. [Arxiv link] - Comprehensive introduction about core concepts of temporal networks.

Kivelä, M., Arenas, A., Barthelemy, M., Gleeson, J. P., Moreno, Y., & Porter, M. A. (2014). Multilayer networks. Journal of complex networks, 2(3), 203-271. [Link] - General overview of multilayer networks.

Lurie, D., Kessler, D., Bassett, D., Betzel, R. F., Breakspear, M., Keilholz, S., ... & Calhoun, V. (2018). On the nature of resting fMRI and time-varying functional connectivity. [Psyarxiv link] - Review of time-varying connectivity in human neuroimaging.

Masuda, N., & Lambiotte, R. (2016). A Guidance to Temporal Networks. [Link to book's publisher] - Book that covers a lot of the mathematics of temporal networks.

Nicosia, V., Tang, J., Mascolo, C., Musolesi, M., Russo, G., & Latora, V. (2013). Graph metrics for temporal networks. In Temporal networks (pp. 15-40). Springer, Berlin, Heidelberg. [Arxiv link] - Review of some temporal network metrics.

Thompson, W. H., Brantefors, P., & Fransson, P. (2017). From static to temporal network theory: Applications to functional brain connectivity. Network Neuroscience, 1(2), 69-99. [Link] - Article introducing temporal network's in cognitive neuroscience context.

CHAPTER 2

Tutorial

2.1 Tutorial: Network representation in Teneto

There are three ways that network's are represented in Teneto:

1. A TemporalNetwork object
2. Numpy array/snapshot
3. Dictionary/contact representation

This tutorial goes through what these different representations. Teneto is migrating towards the TemporalNetwork object. However, it is possible to still use with the other two representations.

2.1.1 TemporalNetwork object

TemporalNetwork is a class in teneto.

```
>>> from teneto import TemporalNetwork
>>> tnet = TemporalNetwork()
...
```

As an input, you can pass it a 3D numpy array, a contact representation (see below), a list of edges or a pandas df (see below).

A feature of the TemporalNetwork class is that the different functions such as plotting and networkmeasures can be accessed within the object.

For example, the code below calls the function *teneto.generatenetwork.rand_binomial* with all subsequent arguments being arguments for the *rand_binomial* function:

```
>>> import numpy as np
>>> np.random.seed(2019) # Set random seed for replication
>>> tnet.generatenetwork('rand_binomial',size=(5,3), prob=0.5)
```

The data this creates is found in *tnet.network* which is a pandas data frame. To have a peak at the top of the data frame, we can call:

```
>>> tnet.network.head()
   i  j  t
0  0  1  0
1  0  1  1
2  0  2  0
3  0  2  1
4  0  2  2
```

Each line in the data frame represents one edge. *i* and *j* are both node indexes and *t* is a temporal index. These column names are always present in data frames made by Teneto. There is no *weight* column here which indicates this is a binary network.

Exploring the network

You can inspect different parts of the network by calling *tnet.get_network_when()* and specifying an i, j or t argument.

```
>>> tnet.get_network_when(i=1)
   i  j  t
6  1  2  0
7  1  2  2
8  1  3  0
9  1  4  1
```

The different argument can also be combined.

```
>>> tnet.get_network_when(i=1, t=0)
   i  j  t
6  1  2  0
8  1  3  0
```

Lists can also be specified as arguments:

```
>>> tnet.get_network_when(i=[0, 1], t=1)
   i  j  t
1  0  1  1
3  0  2  1
5  0  4  1
9  1  4  1
```

The logic within each argument is OR (i.e. about get all where i == 1 OR i == 0). The logic between the different arguments, defaults to AND. (i.e. get when i == [0 or 1] AND t == 1). In some cases, you may might the between argument logic to be OR:

```
>>> tnet.get_network_when(i=1, j=1, logic='or')
   i  j  t
0  0  1  0
1  0  1  1
6  1  2  0
7  1  2  2
8  1  3  0
9  1  4  1
```

In the above case we select all edges where i == 1 OR j == 1.

Weighted networks

When a network is weighted, the weight appears in its own column in the pandas data frame.

```
>>> np.random.seed(2019) # For reproducibility
>>> G = np.random.beta(1, 1, [5,5,3]) # Creates 5 nodes and 3 time-points
>>> tnet = TemporalNetwork(from_array=G, nettype='wd', diagonal=True)
>>> tnet.network.head()
   i  j  t    weight
0  0  0  0  0.628820
1  0  0  1  0.059084
2  0  0  2  0.833974
3  0  1  0  0.856509
4  0  1  1  0.518670
```

Self edges get deleted unless the argument *diagonal=True* is passed. Above we can see that there are edges when both i and j are 0.

Dense and sparse networks

The example we saw previously was of a *sparse* network representation. This means that only the active connections are encoded in the representation and all other edges can be assumed to be zero/absent.

There are many weighted networks all edges have a value. These networks are called *dense*.

In denser networks, tnet.network will be a numpy array with node,node,time dimensions. The reason for this is simply speed. If you do not want a dense network to be created, you can pass a forcesparse=True argument when creating the TemporalNetwork.

If teneto is slow, it could be that creating the sparse network is taking too much time. So one way to ensure the dense representation is forced is to set the parameter *dense_threshold*. The default value is 0.1 (i.e. 10%), which means that if 10% of the network's connections are present, teneto will make the network dense. But you can set this to any value.

The TemporalNetwork functions such as get_network_when() still function with the dense representation.

Exporting to a numpy array

You can export the network to a numpy array from the pandas data frame by calling to array:

```
>>> np.random.seed(2019) # For reproducibility
>>> G = np.random.beta(1, 1, [5,5,3]) # Creates 5 nodes and 3 time-points
>>> tnet = TemporalNetwork(from_array=G, nettype='wd', diagonal=True)
>>> G2 = tnet.to_array()
>>> G == G2
True
```

Here G2 is a 3D numpy array which is equal to the input G (a numpy array).

Meta-information

Within the object there are multiple bits of information about the network. We, for example, check that the above network create below is binary:

```
>>> tnet = TemporalNetwork()
>>> tnet.generatenetwork('rand_binomial',size=(3,5), prob=0.5)
>>> tnet.nettype
'bu'
```

There are 4 different nettypes: bu, wu, wd and bd.

where b is for binary, w is for weighted, u means undirected and d means directed. Teneto tries to estimate the nettype, but specifying it is good practice.

You can also get the size of the network by using:

```
>>> tnet.netshape
(3, 5)
```

Which means there are 3 nodes and 5 time-points.

Certain metainformation is automatically used in the plotting tools. For example, you can add some meta information using the *nodelabels* (give names to the nodes), *timelabels* (give names to the time points), and *timeunit* arguments.

```
>>> import matplotlib.pyplot as plt
>>> tlabs = ['2014','2015','2016','2017','2018']
>>> tunit = 'years'
>>> nlabs = ['Ashley', 'Blake', 'Casey']
>>> tnet = TemporalNetwork(nodelabels=nlabs, timeunit=tunit, timelabels=tlabs,␣
↪nettype='bu')
>>> tnet.generatenetwork('rand_binomial',size=(3,5), prob=0.5)
>>> tnet.plot('slice_plot', cmap='Set2')
>>> plt.show()
```

Importing data to TemporalNetwork

There are multiple ways to add data to the TemporalNetwork object. These include:

1. A 3D numpy array
2. Contact representation
3. Pandas data frame
4. List of edges

Numpy Arrays

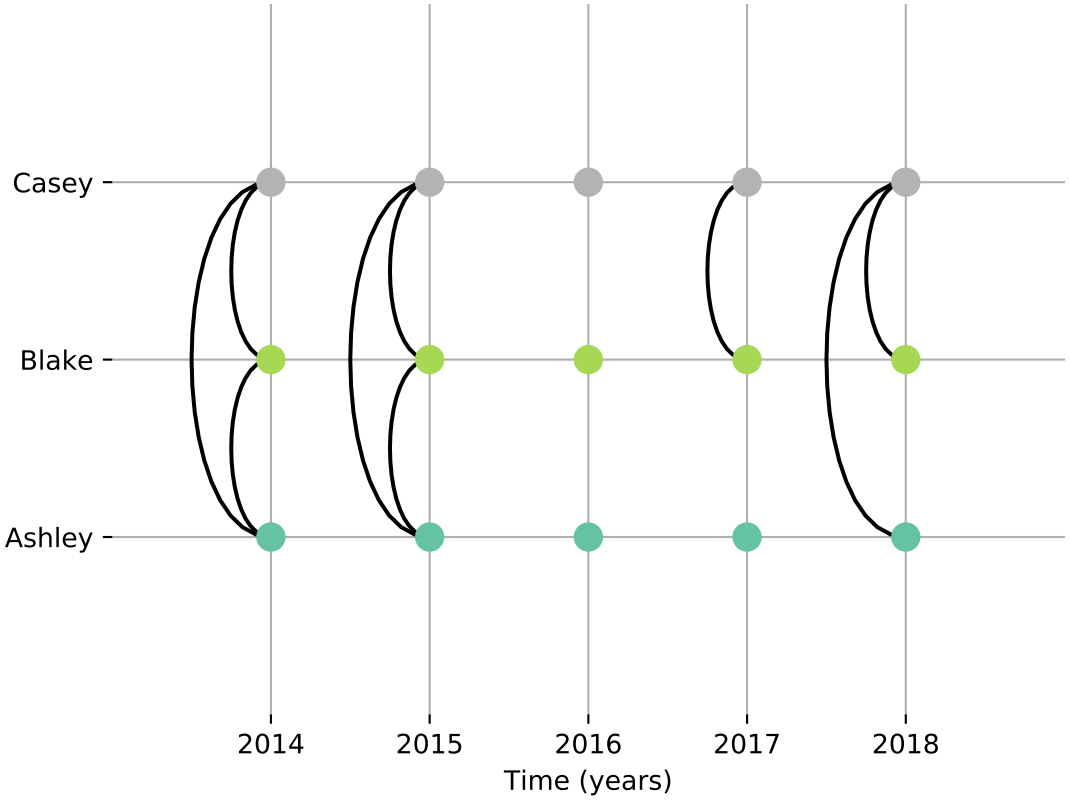
For example, here we create a random network based on a beta distribution.

```
>>> np.random.seed(2019)
>>> G = np.random.beta(1, 1, [5,5,3])
>>> G.shape
(5, 5, 3)
```

Numpy arrays can get added by using the from_array argument

```
>>> tnet = TemporalNetwork(from_array=G)
```

Or for an already defined object:


Casey
Blake
Ashley
2014
2015
2016
2017
2018
Time (years)

```
>>> tnet.network_from_array(G)
```

Contact representation

The contact representation (see below) is a dictionary which a key called *contacts* includes a contact list of lists and some additional metadata. Here the argument is *from_dict* should be called.

```
>>> C = {'contacts': [[0,1,2],[1,0,0]],
        'nettype': 'bu',
        'netshape': (2,2,3),
        't0': 0,
        'nodelabels': ['A', 'B'],
        'timeunit': 'seconds'}
>>> tnet = TemporalNetwork(from_dict=C)
```

Or alternatively:

```
>>> tnet = TemporalNetwork()
>>> tnet.network_from_dict(C)
```

Pandas data frame

Using a pandas data frame the data can also be imported. Here the required columns are: i, j and t (the first two are nodes, the latter is time index). The column weight is also needed for weighted networks.

```
>>> import pandas as pd
>>> netin = {'i': [0,0,1,1], 'j': [1,2,2,2], 't': [0,0,0,1], 'weight': [0.5,0.75,0.25,
↪1]}
>>> df = pd.Data Frame(data=netin)
>>> tnet = TemporalNetwork(from_df=df)
>>> tnet.network
   i  j  t  weight
0  0  1  0    0.50
1  0  2  0    0.75
2  1  2  0    0.25
3  1  2  1    1.00
```

List of edges

Alternatively a list of lists can be given to *TemporalNetwork*, in such cases each sublist should follow the order [i,j,t,[weight]]. For example:

```
>>> edgelist = [[0,1,0,0.5], [0,1,1,0.75]]
>>> tnet = TemporalNetwork(from_edgelist=edgelist)
>>> tnet.network
   i  j  t  weight
0  0  1  0    0.50
1  0  1  1    0.75
```

This creates two edges between nodes 0 and 1 at two different time-points with two weights.

2.1.2 Array/snapshot representation

The array/snapshort representation is a three dimensional numpy array. The dimensions are (node,node,time).

The positives of arrays are that they is easy to understand and manipulate. The downside is that any meta-information about the network is lost and, when the networks are big, can use a lot of memory.

2.1.3 Contact representation

Note, the contact representation is going to be phased out in favour for the TemporalNetwork object with time.

The contact representations is a dictionary that can includes more information about the network than an array.

The keys in the dictionary include 'contact' (node,node,timestamp) which define all the edges in te network. A weights key is present in weighted networks containing the weights. Other keys for meta-information include: 'dimord' (dimension order), 'Fs' (sampling rate), 'timeunit', 'nettype' (if network is weighted/binary, undirected/directed), 'timetype', *nodelabels* (node labels), *t0* (the first time point).

2.1.4 Converting between contact and graphlet representations

Converting between the two different network representations is quite easy. Let us generate a random network that consists of 3 nodes and 5 time points.

```
import teneto
import numpy as np

# For reproducibility
np.random.seed(2018)
# Number of nodes
N = 3
# Number of time-points
T = 5
# Probability of edge activation
p0to1 = 0.2
p1to1 = .9
G = teneto.generatenetwork.rand_binomial([N,N,T],[p0to1, p1to1],'graphlet','bu')
# Show shape of network
print(G.shape)
```

You can convert a graphlet representation to contact representation with: teneto.utils.graphlet2contact

```
C = teneto.utils.graphlet2contact(G)
print(C.keys)
```

To convert the opposite direction, type teneto.utils.contact2graphlet:

```
G2 = teneto.utils.contact2graphlet(C)
G==G2
```

2.2 Tutorial: Temporal network measures

The module *teneto.networkmeasures* includes several functions to quantify different properties of temporal networks. Below are four different types of properties which can be calculated for each node. For all these properties you can generally derive a time-averaged version or one value per time-point.

Many of the functions use a calc argument to specify what type of measure you want to quantify. For example *calc='global'* will return the global version of a measure and *calc='communities'* will return the community version of the function.

2.2.1 Centrality measures

Centrality measures quantify a value per node. These can be useful for finding important nodes in the network.

- `temporal_degree_centrality()`
- *`temporal_betweenness_centrality()`*
- `temporal_closeness_centrality()`
- `topological_overlap()`
- `bursty_coeff()`

2.2.2 Community dependent measures

Community measure quantify a value per community or a value for community interactions. Communities are an important part of network theory, where nodes are grouped into groups.

- `sid()`, when *calc='community_avg'* or *calc='community_pairs'*
- `bursty_coeff()`, when *calc='communities'*
- `volatility()`, when *calc='communities'*

Node measures that are dependent on community vector

- *`temporal_participation_coeff()`*
- `temporal_degree_centrality()`, when *calc='module_degree_zscore'*

2.2.3 Global measures

Global measures try and calculate one value to reflect the entire network. Examples of global measures:

- `temporal_efficiency()`
- `reachability_latency()`
- `fluctuability()`
- `volatility()`, when *calc='global'*
- `topological_overlap()`, when *calc='global'*
- `sid()`, when *calc='global'*

2.2.4 Edge measures

Edge measures quantify a property for each edge.

- `shortest_temporal_paths()`
- `intercontacttimes()`

- `local_variation()`

2.2.5 Community measures

Community measures quantify the community partition instead of the underlying network. These are found in the module *teneto.communitymeasures*

- *`allegiance()`*
- *`flexibility()`*
- *`integration()`*
- *`persistence()`*
- *`promiscuity()`*
- *`recruitment()`*

2.3 Workflows

Many analyses can be constructed as a graph to depict all the steps that are made during the analysis. This graph of an analysis is called a workflow. There are many benefits to creating a workflow:

- Construct entire analysis workflow and view it before running.
- Carefully records every step, so you know exactly what you did.
- Can share the entire analysis with someone else (good for reproducibility).

TenetoWorkflow allows you to define a workflow object and then run it. A workflow consists of a directed graph. The nodes of this graph are different Teneto functions. The directed edges of the graph is the sequence the pipeline is run in.

The workflows function around the TenetoBIDS or TemporalNetwork classes. Any analysis made using those classes can be made into a workflow.

There are three different types of nodes in this graph:

root nodes: These are nodes that do not depend on any other nodes in the analysis. These are calls to create a _TenetoBIDS_ or _TemporalNetwork_ object.

non-terminal nodes: These are nodes that are intermediate steps in the analysis.

terminal nodes: These are the final nodes in the analysis. These nodes will include the output of the analysis.

Understanding the concept of root and terminal nodes are useful to understand how the input and output of TenetoWorkflow.

2.3.1 Creating a workflow

We are going to create a workflow that does the following three steps:

1. Creates a temporal network object (root node)
2. Generates random data (non-terminal node)
3. Calculates the temporal degree centrality of each node (terminal node)

We start by creating a workflow object, and defining the first node:

```
>>> from teneto import TenetoWorkflow
>>> twf = TenetoWorkflow()
>>> nodename = 'create_temporalnetwork'
>>> func = 'TemporalNetwork'
>>> twf.add_node(nodename=nodename, func=func)
```

Each node in the workflow graph needs a unique name (argument: nodename). If you create two different TemporalNetwork objects in the workflow, these need different names to differentiate them.

The func argument specifies the class that is initiated or the function that is run.

There are two more optional arguments that can be passed to add_node: depends_on and params. We will look at those later though.

By adding a node, this creates an attribute in the workflow object which can be viewed as:

```
>>> twf.nodes
{'create_temporalnetwork': {'func': 'TemporalNetwork', 'params': {}}}
```

It also creates a graph (pandas dataframe) which is found in TenetoWorkflow.graph.

```
>>> twf.graph
    i   j
0   isroot  create_temporalnetwork
```

Since this is the first node in the workflow, _isroot_ is placed in the _i_ column to signify that _create_temporalnetwork_ is the root node.

Now let us add the next two nodes and we will see the params argument add_node:

```
>>> # Generate network node
>>> nodename = 'generatenetwork'
>>> func = 'generatenetwork'
>>> params = {
    'networktype': 'rand_binomial',
    'size': (10,5),
    'prob': (0.5,0.25),
    'randomseed': 2019
    }
>>> twf.add_node(nodename, func, params=params)
>>> # Calc temporal degree centrality node
>>> nodename = 'degree'
>>> func = 'calc_networkmeasure'
>>> params = {
    'networkmeasure': 'temporal_degree_centrality',
    'calc': 'time'
    }
>>> twf.add_node(nodename, func, params=params)
```

Here we see that the params argument is a dictionary of _*kwargs_ for the _TemporalNetwork.generatenetwork_ and _TemporalNetwork.calc_networkmeasure_ functions.

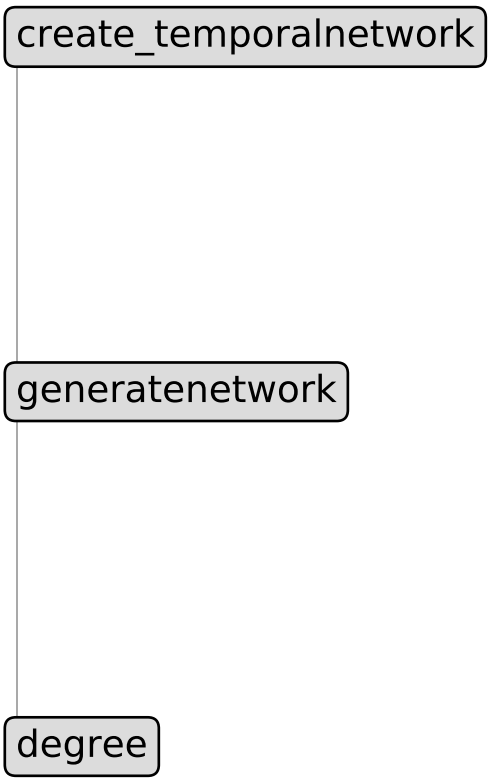
Now we have three nodes defined, so we can look at the TenetoWorkflow.graph:

```
>>> twf.graph
    i   j
0   isroot  create_temporalnetwork
1   create_temporalnetwork generatenetwork
2   generatenetwork    degree
```

Each row here shows the new node in the _j_-th column and the step preceding node in the _i_-th column.

The workflow graph can be plotted with:

```
>>> fig, ax = twf.make_workflow_figure()
>>> fig.show()
```



2.3.2 Running a workflow

Now the workflow has been defined, it can be run by typing:

```
>>> tfw.run()
```

And this will run all of steps.

2.3.3 Viewing the output

The output of the final step will be found in TenetoWorkflow.output_[<nodename>].

The nodes included here will be all the terminal nodes. However when defining the TenetoWorkflow, you can set the argument, _remove_nonterminal_output_ to False and all node output will be stored.

The output from the above is found in:

```
>>> tfw.output_['degree']
...
```

2.3.4 More complicated workflows

The previous example consists of only three steps and occurs linearly. In practice analyses are usually more complex. One typical example is where multiple parameters are run (e.g. to demonstrate that a result is dependent on that parameter).

Here we define a more complex network where we generate two different networks. One where there is a high probability of edges in the network and one where there is a low probability.

When adding a node, the node refers to the last node defined unless depends_on is set. This should point to another preset node.

Example:

First define the object.

```
>>> from teneto import TenetoWorkflow
>>> twf = TenetoWorkflow()
>>> nodename = 'create_temporalnetwork'
>>> func = 'TemporalNetwork'
>>> twf.add_node(nodename=nodename, func=func)
```

Then we generate the first network where edges have low probability.

```
>>> nodename = 'generatenetwork_lowprob'
>>> func = 'generatenetwork'
>>> params = {
    'networktype': 'rand_binomial',
    'size': (10,5),
    'prob': (0.25,0.25),
    'randomseed': 2019
    }
>>> twf.add_node(nodename, func, params=params)
```

Then add the calculate degree step.

```
>>> nodename = 'degree_lowprob'
>>> func = 'calc_networkmeasure'
>>> params = {
    'networkmeasure': 'temporal_degree_centrality',
    'calc': 'time'
    }
>>> twf.add_node(nodename, func, params=params)
```

Now we generate a second network where edges have higher probability. Here depends_on is called and refers back to the create_temporalnetwork node.

```
>>> nodename = 'generatenetwork_highprob'
>>> func = 'generatenetwork'
>>> depends_on = 'create_temporalnetwork'
>>> params = {
    'networktype': 'rand_binomial',
    'size': (10,5),
    'prob': (0.75,0.1),
    'randomseed': 2019
    }
>>> twf.add_node(nodename, func, depends_on, params)
```

Now we can calculate temporal degree centrality on this network:

```
>>> nodename = 'degree_highprob'
>>> func = 'calc_networkmeasure'
>>> params = {
    'networkmeasure': 'temporal_degree_centrality',
    'calc': 'time'
    }
>>> twf.add_node(nodename, func, params=params)
```

And this workflow can be plotted like before:

```
>>> fig, ax = twf.make_workflow_figure()
>>> fig.show()
```

2.4 TenetoBIDS

TenetoBIDS allows use of Teneto functions to analyse entire datasets in just a few lines of code. The output from Teneto is then ready to be placed in statistical models, machine learning algorithms and/or plotted.

2.4.1 Prerequisites

To use *TenetoBIDS* you need preprocessied fMRI data in the BIDS format. It is tested and optimized for fMRIPrep but other preprocessing software following BIDS should (in theory) work too. For fMRIPrep V1.4 or later is requiresd. This preprocessed data should be in the ~BIDS_dir/derivatives/ directory. The output from teneto will always be found in .../BIDS_dir/derivatives/ in directories that begin with teneto- (depending on the function you use).

2.4.2 Contents of this tutorial

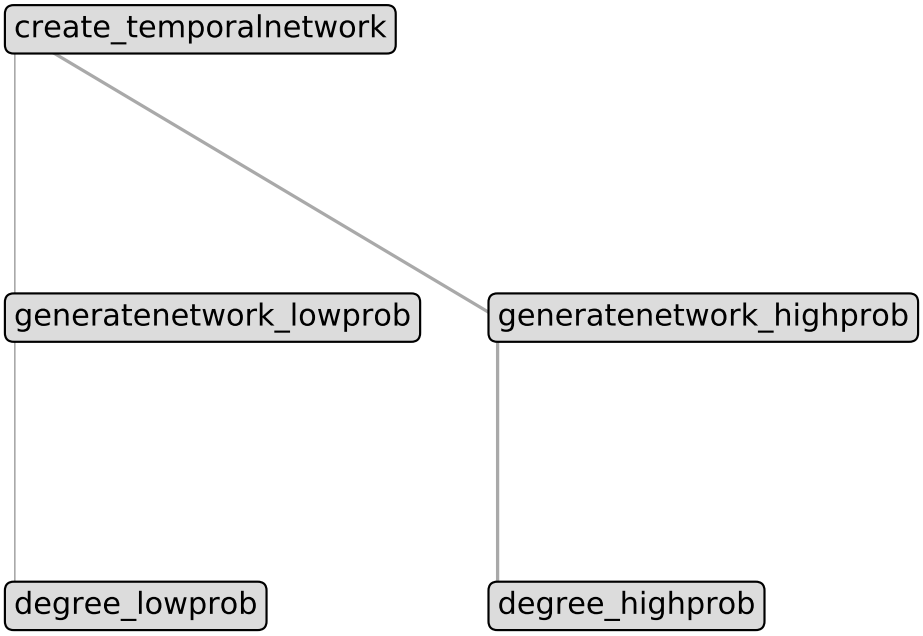
This tutorial will run a complete analysis on some test data.

For this tutorial, we will use some dummy data which is included with teneto. This section details what is in this data.

[1]:
```
import teneto
import os
dataset_path = teneto.__path__[0] + '/data/testdata/dummybids/'
print(os.listdir(dataset_path))
print(os.listdir(dataset_path + '/derivatives'))
```

```
['participants.tsv', 'dataset_description.json', 'sub-001', 'derivatives', 'sub-002']
['teneto-censor-timepoints', 'teneto-derive-temporalnetwork', 'teneto-volatility',
↪'teneto-exclude-runs', 'teneto-tests', 'teneto-make-parcellation', 'fmriprep',
↪'teneto-binarize', 'teneto-remove-confounds']
```

(continues on next page)


create_temporalnetwork
generatenetwork_lowprob
generatenetwork_highprob
degree_lowprob
degree_highprob

(continued from previous page)

From the above we can see that there are two subjects in our dataset, and there is a fMRIPrep folder in the derivatives section. Only subject 1 has any dummy data, so we will have to select subject 1.

2.4.3 A complete analysis

Below is a complete analysis of this test data. We will go through each step after it.

[2]:
```
#Imports.
from teneto import TenetoBIDS
from teneto import __path__ as tenetopath
import numpy as np
#Set the path of the dataset.
datdir = tenetopath[0] + '/data/testdata/dummybids/'

# Step 1:
bids_filter = {'subject': '001',
               'run': 1,
               'task': 'a'}
tnet = TenetoBIDS(datdir, selected_pipeline='fmriprep', bids_filter=bids_filter,␣
↪exist_ok=True)

# Step 2: create a parcellation
parcellation_params = {'atlas': 'Schaefer2018',
                       'atlas_desc': '100Parcels7Networks',
                       'parc_params': {'detrend': True}}
tnet.run('make_parcellation', parcellation_params)

# Step 3: Regress out confounds
remove_params = {'confound_selection': ['confound1']}
tnet.run('remove_confounds', remove_params)

# Step 4: Additonal preprocessing
exclude_params = {'confound_name': 'confound1',
                  'exclusion_criteria': '<-0.99'}
tnet.run('exclude_runs', exclude_params)
censor_params = {'confound_name': 'confound1',
                  'exclusion_criteria': '<-0.99',
                  'replace_with': 'cubicspline',
                  'tol': 0.25}
tnet.run('censor_timepoints', censor_params)

# Step 5: Calculats time-varying connectivity
derive_params = {'params': {'method': 'jackknife',
                            'postpro': 'standardize'}}
tnet.run('derive_temporalnetwork', derive_params)

# Step 6: Performs a binarization of the network
binaraize_params = {'threshold_type': 'percent',
                    'threshold_level': 0.1}
tnet.run('binarize', binaraize_params)

# Step 7: Calculate a network measure
measure_params = {'distance_func': 'hamming'}
```

(continues on next page)

(continued from previous page)

```
tnet.run('volatility', measure_params)

# Step 8: load data
vol = tnet.load_data()
print(vol)
```

```
{'sub-001_run-1_task-a_vol.tsv':           0
0  0.103733}
```

2.4.4 Big Picture

While the above code may seem overwhelming at first. It is quite little code for what it does. It starts with nifti images and ends with a single measure about a time-varying connectivity estimate of the network.

There is one recurring theme used in the code above:

`tnet.run(function_name, function_parameters)`

`function_name` is a string and function_parameters is a dictionary `function_name` can be most functions in teneto if the data is in the correct format. `function_parameters` are the inputs to that function. You never need to pass the input data (e.g. time series or network), or any functions that have a `sidecar` input.

TenetoBIDS will also automatically try and find a confounds file in the derivatives when needed, so, this does not need to be specified either.

Once you have grabbed the above, the rest is pretty straight forward. But we will go through each step in turn.

2.4.5 Step 1 - defining the TenetoBIDS object.

[3]:
```
#Set the path of the dataset.
datdir = tenetopath[0] + '/data/testdata/dummybids/'
# Step 1:
bids_filter = {'subject': '001',
               'run': 1,
               'task': 'a'}
tnet = TenetoBIDS(datdir, selected_pipeline='fmriprep', bids_filter=bids_filter,␣
↪exist_ok=True)
```

selected_pipeline

**This states where teneto will go looking for files. This example shows it should look in the fMRIPrep derivative directory. (i.e. in: datadir + '/derivatives/fmriprep/').

bids_filter

teneto uses pybids to select different files. The `bids_filter` argument is a dictionary of arguments that get passed into the `BIDSLayout.get`. In the example above, we are saying we want subject 001, run 1 and task a. If you do not provide any arguments for `bids_filter`, all data found within the derivatives folder gets selected for analyses.

exist_ok (default: False)

This checks that it is ok to overwrite any previous calculations. The output data is saved in a new directory. If the new output directory already exists, the teneto step has previously been run, and an error will be returned because otherwise data may be overwritten. To overrule this error, set `exists_ok` to True.

We can now look at what files are selected that will be run on the next step.

[4]:
```
tnet.get_selected_files()
```

[4]:
```
[<BIDSDataFile filename='/home/william/anaconda3/lib/python3.6/site-packages/teneto/
↪data/testdata/dummybids/derivatives/fmriprep/sub-001/func/sub-001_task-a_run-01_
↪desc-confounds_regressors.tsv'>,
 <BIDSImageFile filename='/home/william/anaconda3/lib/python3.6/site-packages/teneto/
↪data/testdata/dummybids/derivatives/fmriprep/sub-001/func/sub-001_task-a_run-01_
↪desc-preproc_bold.nii.gz'>]
```

If there are files here you do not want, you can add to the bids filter with `tnet.update_bids_filter` Or, you can set tnet.bids_filter to a new dictionary if you want.

Next, you might want to see what functions you can run on these selected files. The following will specify what functions can be run specifically on the selected data. If you want all options, you can add the `for_selected=False`.

[5]:
```
tnet.get_run_options()
```

[5]:
```
'make_parcellation, exclude_runs'
```

The output here (exclude_runs and make_parcellation) says which functions that, with the selected files, can be called in tnet.run. Once different functions have been called, the options change.

2.4.6 Step 2 Calling the run function to make a parcellation.

When selecting preprocessed files, these will often be nifti images. From these images, we want to make time-series of regions of interests. This can be done with :py:func:`.make_parcellation`. This function uses TemplateFlow atlases to make the parcellation.

[6]:
```
parcellation_params = {'atlas': 'Schaefer2018',
                       'atlas_desc': '100Parcels7Networks',
                       'parc_params': {'detrend': True}}
tnet.run('make_parcellation', parcellation_params)
```

The `atlas` and `atlas_desc` are used to identify TemplateFlow atlases.

Teneto uses nilearn's NiftiLabelsMasker to mark the parcellation. Any arguments to this function (e.g. preprocessing steps) can be passed in the argument using 'parc_params' (here detrend is used).

2.4.7 Step 3 Regress out confounds

[7]:
```
remove_params = {'confound_selection': ['confound1']}
tnet.run('remove_confounds', remove_params)
```

Confounds can be removed by calling :py:func:`.remove_confounds`.

The confounds tsv file is automatically located as long as it is in a derivatives folder and that there is only one

Here 'confound1' is a column namn in the confounds tsv file.

Similarly to make parcellation, it uses nilearn (nilean.signal.clean. `clean_params` is a possible argument, like `parc_params` these are inputs to the nilearn function.

2.4.8 Step 4: Additonal preprocessing

[8]:
```
exclude_params = {'confound_name': 'confound1',
                  'exclusion_criteria': '<-0.99'}
tnet.run('exclude_runs', exclude_params)
censor_params = {'confound_name': 'confound1',
                 'exclusion_criteria': '<-0.99',
                 'replace_with': 'cubicspline',
                 'tol': 0.25}
tnet.run('censor_timepoints', censor_params)
```

These two calls to tnet.run exclude both time-points and runs, which are problematic. The first, exclude_runs, rejects any run where the mean of confound1 is less than 0.99. Excluded runs will no longer be part of the loaded data in later calls of tnet.run().

Centoring time-points here says that whenever there is a time-point that is less than 0.99, it will be "censored" (set to not a number). We have also set argument replace_with to 'cubicspline'. This argument means that the values that have censored now get simulated using a cubic spline. The parameter *tol* says what percentage of time-points are allowed to be censored before the run gets ignored.

2.4.9 Step 5: Calculats time-varying connectivity

The code below now derives time-varying connectivity matrices. There are multiple different methods that can be called. See teneto.timeseries.derive_temporalnetwork for more options.

[9]:
```
derive_params = {'params': {'method': 'jackknife',
                            'postpro': 'standardize'}}
tnet.run('derive_temporalnetwork', derive_params)
```

2.4.10 Step 6: Performs a binarization of the network

Once you have a network representation, there are multiple ways this can be transformed. One example, is to binarize the network so all values are 0 or 1. The code below converts the top 10% of edges to 1s, the rest 0.

[10]:
```
binaraize_params = {'threshold_type': 'percent',
                    'threshold_level': 0.1}
tnet.run('binarize', binaraize_params)
```

2.4.11 Step 7: Calculate a network measure

We are now ready to calculate a property of the temproal network. Here we calculate volatility (i.e. how much the network changes per time-point). This generates one value per subject.

[11]:
```
measure_params = {'distance_func': 'hamming'}
tnet.run('volatility', measure_params)
```

2.4.12 Step 8: load data

[12]:
```
vol = tnet.load_data()
print(vol)
```

```
{'sub-001_run-1_task-a_vol.tsv':           0
0  0.103733}
```

Now that we have a measure of volatility for the network. We can now load it and view the measure.

2.5 TCTC

2.5.1 Backgorund

TCTC stands for *Temporal Communities by Trajectory Clustering*. It is an algorithm designed to find temporal communities on time series data.

The kind of data needed for TCTC are:

1. Multiple time series.
2. The time series are from nodes in a network.

Most community detection requires to first create an "edge inference" step where the edges of the different nodes are first calculated.

TCTC first finds clusters of trajectories in the time series without inferring edges. A trajectory is a time series moving through some space. Trajectory clustering tries to group together nodes that have similar paths through a space.

The hyperparameters of TCTC dictate what type of trajectory is found in the data. There are four hyperparameters:

1. A maximum distance parameter (ϵ). The distance between all nodes part of the same trajectory must be ϵ or lower.
2. A minimum size parameter (σ). All trajectories must include at least σ many nodes.
3. A minimum time parameter (τ). All trajectories must persist for τ time-points.
4. A tolerance parameter (κ). κ consecutive "exception" time-points can exist before the trajectory ends.

2.5.2 Outline

This example shows only how TCTC is run and how the different hyperparameters effect the community detection. These hyperparameters can be trained (saved for another example).

2.5.3 Read more

TCTC is outlined in more detail in this article

2.5.4 TCTC - example

We will start by generating some data and importing everything we need.

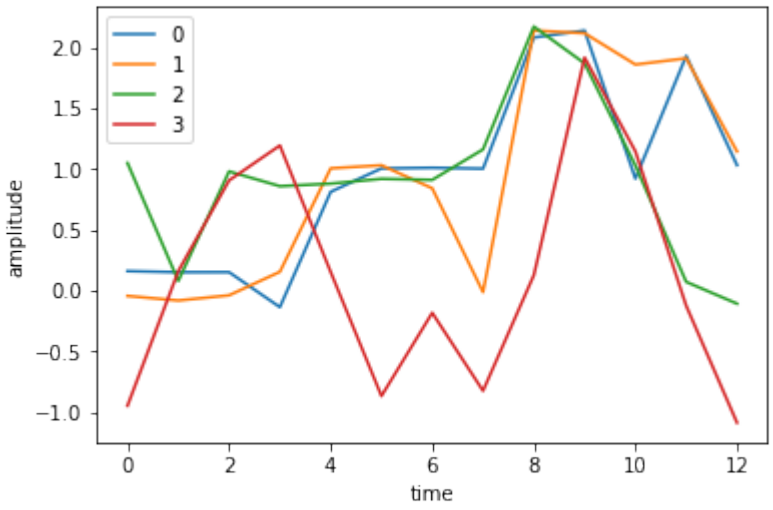
[1]:
```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
from teneto.communitydetection import tctc
import pandas as pd
```

```
Failed to import duecredit due to No module named 'duecredit'
/home/william/anaconda3/lib/python3.6/importlib/_bootstrap.py:219: RuntimeWarning:␣
↪numpy.ufunc size changed, may indicate binary incompatibility. Expected 216, got 192
  return f(*args, **kwds)
/home/william/anaconda3/lib/python3.6/importlib/_bootstrap.py:219: ImportWarning: can
↪'t resolve package from __spec__ or __package__, falling back on __name__ and __
↪path__
  return f(*args, **kwds)
/home/william/anaconda3/lib/python3.6/importlib/_bootstrap.py:219: RuntimeWarning:␣
↪numpy.ufunc size changed, may indicate binary incompatibility. Expected 192 from C␣
↪header, got 216 from PyObject
  return f(*args, **kwds)
```

[2]:
```
data = np.array([[0, 0, 0, 0, 1, 1, 1, 1, 2, 2, 1, 2, 1],
    [0, 0, 0, 0, 1, 1, 1, 0, 2, 2, 2, 2, 1],
    [1, 0, 1, 1, 1, 1, 1, 1, 2, 2, 1, 0, 0], [-1, 0, 1, 1, 0, -1, 0, -1, 0, 2, 1, 0, -
↪1]], dtype=float)
data = data.transpose()
np.random.seed(2019)
data += np.random.uniform(-0.2, 0.2, data.shape)
```

[3]:
```
# Lets have a look at the data
fig, ax = plt.subplots(1)
p = ax.plot(data)
ax.legend(p, [0,1,2,3])
ax.set_xlabel('time')
ax.set_ylabel('amplitude')
print(data.shape)
```

```
(13, 4)
```



There are two different outputs that TCTC can produce. TCTC allows for multilabel communities (i.e. the same node can belong to multiple communities). The output of TCTC can either be:

1. As a binary array (dimensions: node,node,time) where each 1 designates that two nodes are in the same community.
2. As a dataframe where each row is a community.

The default output is option one.

So let us run TCTC on the data we have above.

[4]:
```
parameters = {
    'epsilon': 0.5,
    'tau': 3,
    'sigma': 2,
    'kappa': 0
}
tctc_array = tctc(data, **parameters)
print(tctc_array.shape)
```

```
(4, 4, 13)
```

For now ignore the values in the "parameters" dictionary, we will go through that later.

In order to get the dataframe output, just add output='df'.

[5]:
```
parameters = {
    'epsilon': 0.5,
    'tau': 3,
    'sigma': 2,
    'kappa': 0
}
tctc_df = tctc(data, **parameters, output='df')
print(tctc_df.head())
```

```
   community  start  end  size  length
0     [0, 1]      0    7   2.0       7
```

(continues on next page)

(continued from previous page)

```
1      [2, 3]       1    4   2.0        3
2   [0, 1, 2]       4    7   3.0        3
3      [0, 2]       4   11   2.0        7
5      [2, 3]       9   12   2.0        3
```

Here we can see when the different communities start, end, the size, and the length.

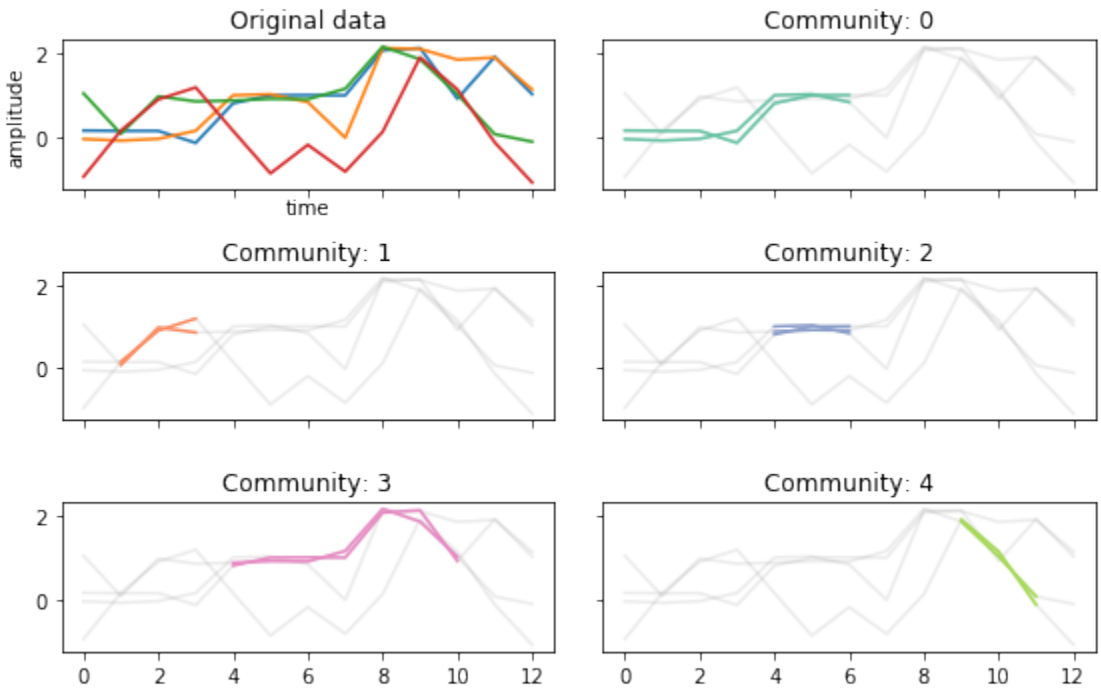
Below we define a function which plots each community on the original data.

[6]:
```
def community_plot(df, data):
    nrows = int(np.ceil((len(df)+1)/2))
    fig, ax = plt.subplots(nrows, 2, sharex=True, sharey=True, figsize=(8, 2+nrows))
    ax = ax.flatten()
    p = ax[0].plot(data)
    ax[0].set_xlabel('time')
    ax[0].set_ylabel('amplitude')
    ax[0].set_title('Original data')

    for i, row in enumerate(df.iterrows()):
        ax[i+1].plot(data, alpha=0.15, color='gray')
        ax[i+1].plot(np.arange(row[1]['start'],row[1]['end']),data[row[1]['start']:
↪row[1]['end'], row[1]['community']],color=plt.cm.Set2.colors[i])
        ax[i+1].set_title('Community: ' + str(i))

    plt.tight_layout()
    return fig, ax

fig, ax = community_plot(tctc_df, data)
```



The multiple community labels can be seed in 0 and 2 above. Where 2 contains three nodes and community 0 contains 2 nodes.

2.5.5 Changing the hyperparameters

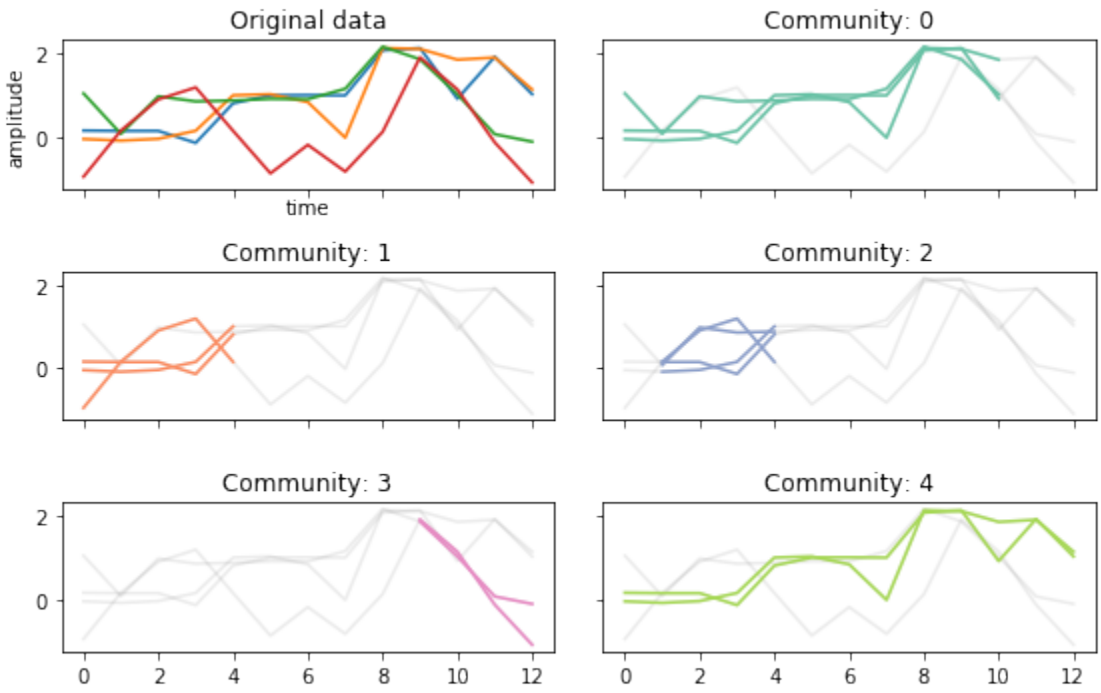
Now we will rerun TCTC but change each of the parameters in turn and then display them on a community plot.

2.5.6 Changing ϵ

If we make ϵ larger, we will include more time series in a trajectory.

This however can mean that the communities you detect are less "connected" than if ϵ was smaller

[7]:
```
parameters = {
    'epsilon': 1.5,
    'tau': 3,
    'sigma': 2,
    'kappa': 0
}
tctc_df_largeep = tctc(data, **parameters, output='df')
fig, ax = community_plot(tctc_df_largeep, data)
```



2.5.7 Changing τ

If we make τ larger, it requires that trajectories persist for more time points.

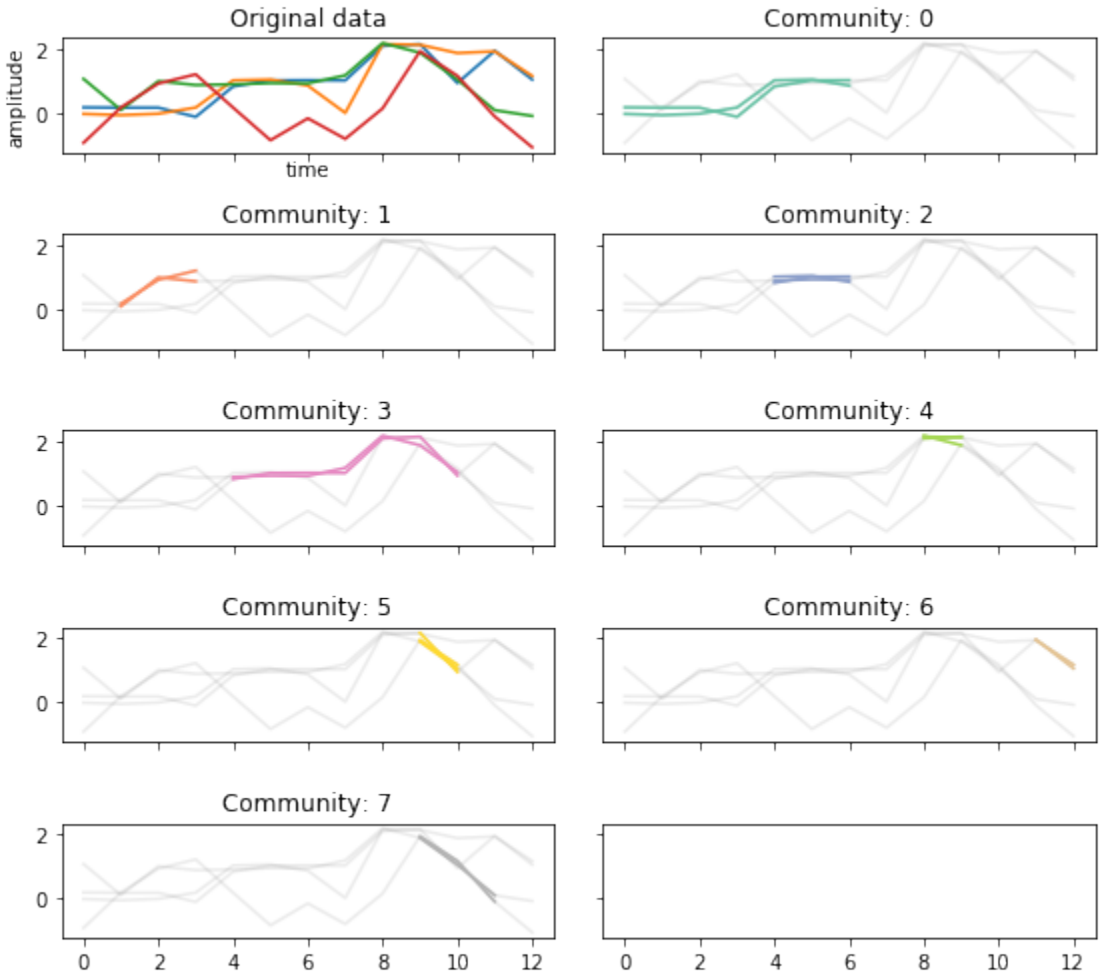
Shorter trajectories increase the change of more noisey connections.

[8]:
```
parameters = {
    'epsilon': 0.5,
    'tau': 2,
    'sigma': 2,
```

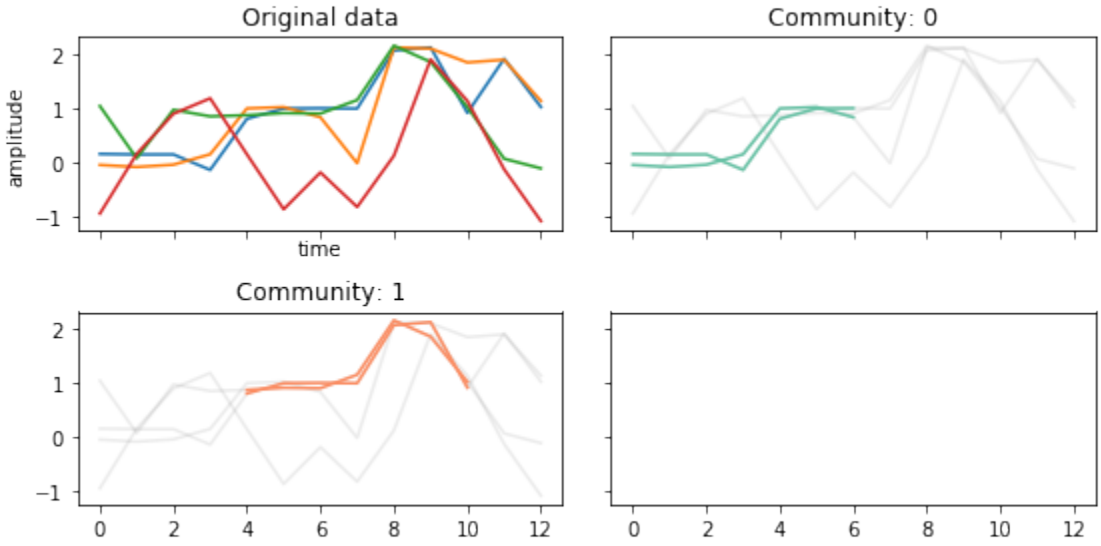
(continues on next page)

(continued from previous page)

```
    'kappa': 0
}
tctc_df_shorttau = tctc(data, **parameters, output='df')
fig, ax = community_plot(tctc_df_shorttau, data)
```



[9]:
```
parameters = {
    'epsilon': 0.5,
    'tau': 5,
    'sigma': 2,
    'kappa': 0
}
tctc_df_longtau = tctc(data, **parameters, output='df')
fig, ax = community_plot(tctc_df_longtau, data)
```

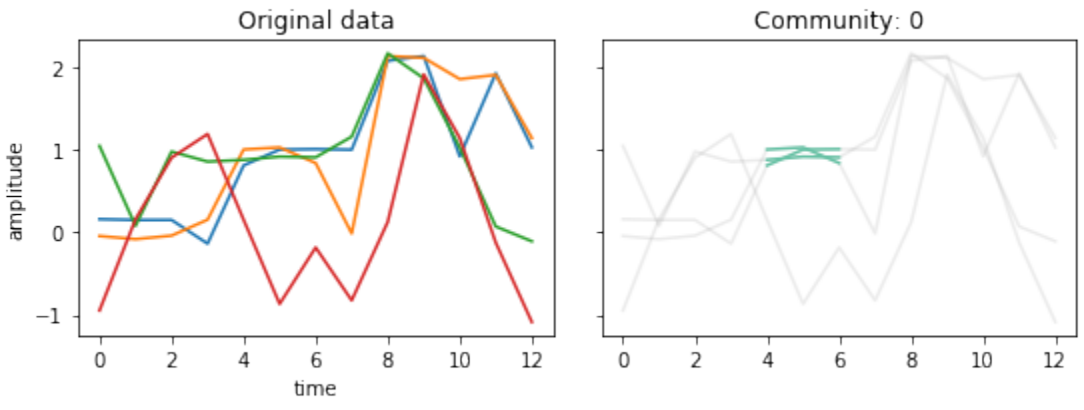


2.5.8 Changing σ

If we make σ larger, it requires that more nodes are part of the trajectory.

Smaller values of σ will result in possible noiser connections.

[10]:
```
parameters = {
    'epsilon': 0.5,
    'tau': 3,
    'sigma': 3,
    'kappa': 0
}
tctc_df_longsigma = tctc(data, **parameters, output='df')
fig, ax = community_plot(tctc_df_longsigma, data)
```

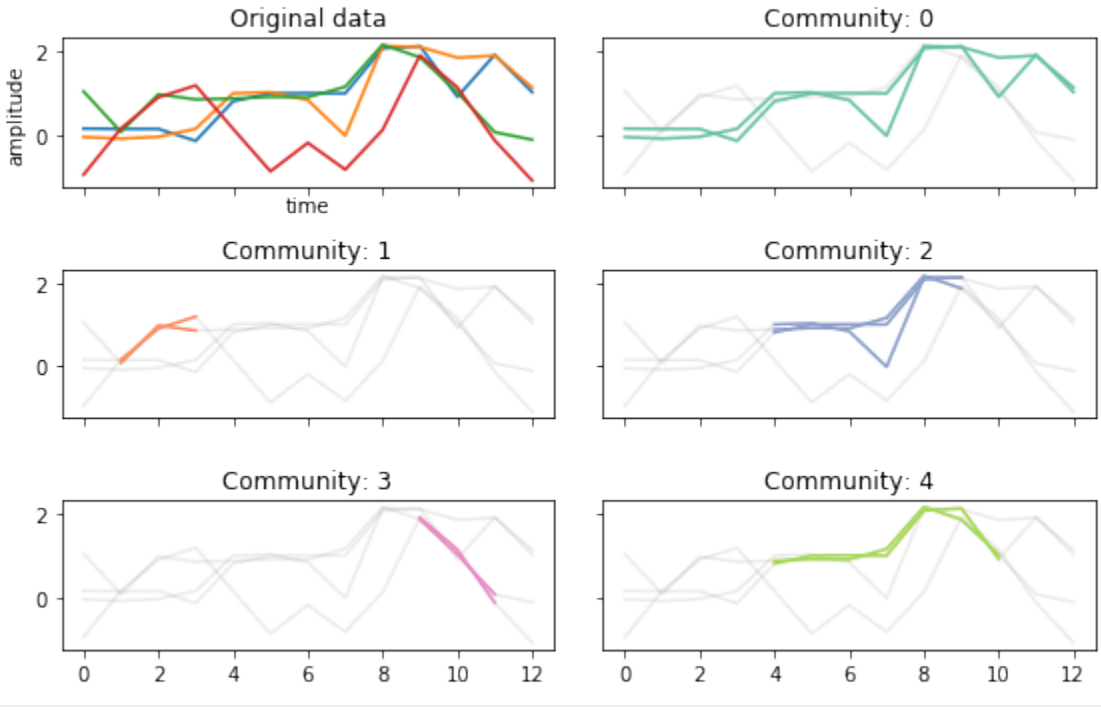


2.5.9 Changing κ

If we make κ larger, it allows for that many number of "noisey" time-points to exist to see if the trajectory continues.

In the data we have been looking at, node 0 and 1 are close to each other except for time-point 7 and 10. If we let κ be 1, if will ignore these time-points and allow the trajectory to continue.

[11]:
```
parameters = {
    'epsilon': 0.5,
    'tau': 3,
    'sigma': 2,
    'kappa': 1
}
tctc_df_withkappa = tctc(data, **parameters, output='df')
fig, ax = community_plot(tctc_df_withkappa, data)
```



CHAPTER 3

API

3.1 TemporalNetwork, TenetoBIDS and TenetoWorkflow

3.1.1 teneto.classes Package

Classes in Teneto

Classes

TenetoBIDS(bids_dir, selected_pipeline[, ...])	Class for analysing data in BIDS.
TemporalNetwork([N, T, nettype, from_df, ...])	A class for temporal networks.
TenetoWorkflow([remove_nonterminal_output])	

TenetoBIDS

class TenetoBIDS(*bids_dir*, *selected_pipeline*, *bids_filter=None*, *bidsvalidator=False*, *update_pipeline=True*, *history=None*, *exist_ok=False*, *layout=None*, *nettsv='nn-t'*)

Bases: `object`

Class for analysing data in BIDS.

TenetoBIDS allows for an analysis to be performed across a dataset. All different functions from Teneto can be applied to all files in a dataset organized in BIDS. Data should be first preprocessed (e.g. fMRIPrep).

Parameters

- **bids_dir** (*str*) – string to BIDS directory
- **selected_pipeline** (*str or dict*) – the directory that is in the bids_dir/derivatives/<selected_pipeline>/. This fine will be used as the input to any teneto function (first argument). If multiple inputs are required for a function, then you can specify:

{'netin': 'tvc', 'communities': 'coms'}

With this, the input for netin with be from bids_dir/derivatives/[teneto-]tvc/, and the input for communities will be from bids_dir/derivatives/[teneto-]coms/. The keys in this dictionary must match the names of the teneto funciton inputs.

- **bids_filter** (*dict*) –
- **history** (*bool*) –
- **update_pipeline** (*bool*) – If true, the output_pipeline becomes the new selected_pipeline
- **exist_ok** (*bool*) – If False, will raise an error if the output directory already exist_ok. If True, will not raise an error. This can lead to files being overwritten, if desc is not set.
- **nettsv** (*str can be nn-t or ijt.*) – nn-t means networks are node-node x time. ijt means daframs are ijt columns.

Methods Summary

create_output_pipeline(runc_func, ...[, ...])	Creates the directories of the saved file.
get_aux_file(bidsfile[, filetype])	Tries to automatically get auxiliary data for input files, and loads it
get_run_options([for_selected])	Returns the different function names that can be called using TenetoBIDS.run()
get_selected_files([output])	Uses information in selected_pipeline and the bids layout and shows the files that will be processed when calling TenetoBIDS.run().
load_data([bids_filter])	Returns data, default is the input data.
load_events()	Loads event data for selected files
load_file(bidsfile)	Aux function to load the data and sidecar from a BIDSFile
run(run_func, input_params[, output_desc, ...])	Runs a runction on the selected files.
troubleshoot(stepname, status)	Prints ongoing info to assist with troubleshooting
update_bids_filter(filter_addons)	Updates TenetoBIDS.bids_filter
update_bids_layout()	Function that upddates to new bids l

Methods Documentation

create_output_pipeline(*runc_func*, *output_pipeline_name*, *exist_ok=None*)

Creates the directories of the saved file.

Parameters

- **output_pipeline** (*str*) – name of output pipeline
- **exist_ok** (*bool*) – If False, will raise error if pipeline already exist_ok. If True, will not raise an error. This can lead to files being overwritten, if desc is not set. If None, will use the exist_ok set during init.

Returns bids_dir/teneto-[output_pipeline]/

Return type Creates the output pipeline directory in

get_aux_file(*bidsfile*, *filetype='confounds'*)

Tries to automatically get auxiliary data for input files, and loads it

bidsfile [BIDSDataFile or BIDSImageFile] The BIDS file that the confound file is gong to be matched.

filetype [string] Can be confounds, events. Specified if you want to get the confound or events data.

get_run_options(*for_selected=True*)
Returns the different function names that can be called using TenetoBIDS.run()

Parameters **for_selected** (*bool*) – If True, only return run options for the selected files. If False, returns all options.

Returns **options** – a list of options that can be run.

Return type str

get_selected_files(*output=None*)
Uses information in selected_pipeline and the bids layout and shows the files that will be processed when calling TenetoBIDS.run().

If you specify a particular output, it will tell you which files will get selected for that output

load_data(*bids_filter=None*)
Returns data, default is the input data.

bids_filter [dict] default is None. If set, load data will load all files found by the bids_filter. Any preset BIDS filter is used as well, but will get overwritten by this input.

load_events()
Loads event data for selected files

load_file(*bidsfile*)
Aux function to load the data and sidecar from a BIDSFile

bidsfile [BIDSDataFile or BIDSImageFile] The BIDS file that the confound file is gong to be matched.

run(*run_func*, *input_params*, *output_desc=None*, *output_pipeline_name=None*, *bids_filter=None*, *update_pipeline=True*, *exist_ok=None*, *troubleshoot=False*)
Runs a runction on the selected files.

Parameters

- **run_func** (*str*) – str should correspond to a teneto function. So to run the funciton teneto.timeseries.derive_temporalnetwork the input should be: 'timeseries.derive_temporalnetwork'
- **input_params** (*dict*) – keyword and value pairing of arguments for the function being run. The input data to each function will be located automatically. This input_params does not need to include the input network. For any other input that needs to be loaded loaded within the teneto_bidsstructure (communities, events, confounds), you can pass the value "bids" if they can be found within the current selected_pipeline. If they are found within a different selected_pipeline, type "bids_[selected_pipeline]".
- **output_desc** (*str*) – If none, no desc is used (removed any previous file) If 'keep', then desc is preserved. If any other str, desc is set to that string
- **output_pipeline_name** (*str*) – If set, then the data is saved in teneto_[functionname]_[output_pipeline_name]. If run_func is teneto.timeseries.derive_temporalnetwork and output_pipeline_name is jackknife then then the pipeline the data is saved in is teneto-generatetemporalnetwork_jackknife
- **update_pipeline** (*bool*) – If set to True (default), then the selected_pipeline updates to output of function
- **exist_ok** (*bool*) – If set to True, then overwrites direcotry is possible.

- **troubleshoot** (*bool*) – If True, prints out certain information during running. Useful to run if reporting a bug.

troubleshoot(*stepname*, *status*)
Prints ongoing info to assist with troubleshooting

update_bids_filter(*filter_addons*)
Updates TenetoBIDS.bids_filter

Parameters **filter_addons** (*dict*) – dictionary that updates TenetoBIDS.bids_filter

update_bids_layout()
Function that upddates to new bids l

TemporalNetwork

class TemporalNetwork(*N=None*, *T=None*, *nettype=None*, *from_df=None*, *from_array=None*, *from_dict=None*, *from_edgelist=None*, *timetype=None*, *diagonal=False*, *timeunit=None*, *desc=None*, *starttime=None*, *nodelabels=None*, *timelabels=None*, *hdf5=False*, *hdf5path=None*, *forcesparse=False*, *dense_threshold=0.25*)
Bases: object

A class for temporal networks.

This class allows to call different teneto functions within the class and store the network representation.

Parameters

- **N** (*int*) – number of nodes in network
- **T** (*int*) – number of time-points in network
- **nettype** (*str*) – description of network. Can be: bu, bd, wu, wd where the letters stand for binary, weighted, undirected and directed. Default is weighted and undirected.
- **from_df** (*pandas df*) – input data frame with i,j,t,[weight] columns
- **from_array** (*array*) – input data from an array with dimesnions node,node,time
- **from_dict** (*dict*) – input data is a contact sequence dictionary.
- **from_edgelist** (*list*) – input data is a list of lists where each item in main list consists of [i,j,t,[weight]].
- **timetype** (*str*) – discrete or continuous
- **diagonal** (*bool*) – if the diagonal should be included in the edge list.
- **timeunit** (*str*) – string (used in plots)
- **desc** (*str*) – string to describe network.
- **startime** (*int*) – integer represents time of first index.
- **nodelabels** (*list*) – list of labels for naming the nodes
- **timelabels** (*list*) – list of labels for time-points
- **hdf5** (*bool*) – if true, pandas dataframe is stored and queried as a h5 file.
- **hdf5path** (*str*) – Where the h5 files is saved if hdf5 is True. If left unset, the default is ./teneto_temporalnetwork.h5

- **forcesparse** (*bool*) – When forsesparse if False (default), if importing array and if dense_threshold% (default%) edges are present, tnet.network is an array. If forsesparse is True, then this inhibts arrays being created.
- **dense_threshold** (*float*) – If forsesparse == False, what percentage (as a decimal) of edges need to be present in order for representation to be dense.

Methods Summary

add_edge(edgelist)	Adds an edge from network.
binarize(threshold_type, threshold_level, ...)	Binarizes the network.
calc_networkmeasure(networkmeasure, ...)	Calculate network measure.
df_to_array([start_at])	Turns datafram to array.
drop_edge(edgelist)	Removes an edge from network.
generatenetwork(networktype, **network-params)	Generate a network
get_network_when(**kwargs)	
hdf5_setup(hdf5path)	
network_from_array(array[, forcesparse, ...])	Defines a network from an array.
network_from_df(df)	Defines a network from an array.
network_from_dict(contact)	
network_from_edgelist(edgelist)	Defines a network from an array.
plot(plottype[, ij, t, ax])	

Methods Documentation

add_edge(*edgelist*)
Adds an edge from network.

Parameters **edgelist** (*list*) – a list (or list of lists) containing the i,j and t indicies to be added. For weighted networks list should also contain a 'weight' key.

Returns

Return type Updates TenetoBIDS.network dataframe with new edge

binarize(*threshold_type*, *threshold_level*, ***kwargs*)
Binarizes the network.

Parameters

- **threshold_type** (*str*) – What type of thresholds to make binarization. Options: 'rdp', 'percent', 'magnitude'.
- **threshold_level** (*str*) – Paramter dependent on threshold type. If 'rdp', it is the delta (i.e. error allowed in compression). If 'percent', it is the percentage to keep (e.g. 0.1, means keep 10% of signal). If 'magnitude', it is the amplitude of signal to keep.
- **teneto.utils.binarize for kwarg arguments.** (*See*) –

Returns

Return type Updates tnet.network to be binarized

calc_networkmeasure(*networkmeasure*, ***measureparams*)
Calculate network measure.

Parameters

- **networkmeasure** (*str*) – Function to call. Functions available are in teneto.networkmeasures
- **measureparams** (*kwargs*) – kwargs for teneto.networkmeasure.[networkmeasure]

df_to_array(*start_at='auto'*)
Turns datafram to array. See teneto.utils.df_to_array for more information.

Parameters **start_at** (*str*) – 'min' or 'zero'. If auto, the 0th time-point is tnet.starttime. If min, the 0th time-point in the array is the minimum time-point found. If zero, the 0th time-point in the array is 0.

drop_edge(*edgelist*)
Removes an edge from network.

Parameters **edgelist** (*list*) – a list (or list of lists) containing the i,j and t indicies to be removes.

Returns

Return type Updates TenetoBIDS.network dataframe

generatenetwork(*networktype*, ***networkparams*)
Generate a network

Parameters

- **networktype** (*str*) – Function to call. Functions available are in teneto.generatenetwork
- **measureparams** (*kwargs*) – kwargs for teneto.generatenetwork.[networktype]

Returns

Return type TenetoBIDS.network is made with the generated network.

get_network_when(***kwargs*)

hdf5_setup(*hdf5path*)

network_from_array(*array*, *forcesparse=False*, *dense_threshold=0.25*)
Defines a network from an array.

Parameters

- **array** (*array*) – 3D numpy array.
- **forcespace** (*bool*) – If true, will always make the array sparse (can be slow). If false, dense form will be kept if more than dense_threshold% of edges are present.
- **dense_threshold** (*float*) – Threshold for when array representation is kept as an array instead of sparse. Only done if forcesparse is False.

network_from_df(*df*)
Defines a network from an array.

Parameters **array** (*array*) – Pandas dataframe. Should have columns: 'i', 'j', 't' where i and j are node indicies and t is the temporal index. If weighted, should also include 'weight'. Each row is an edge.

network_from_dict(*contact*)

network_from_edgelist(*edgelist*)
Defines a network from an array.

Parameters **edgelist** (*list of lists.*) – A list of lists which are 3 or 4 in length. For binary networks each sublist should be [i, j ,t] where i and j are node indicies and t is the temporal index. For weighted networks each sublist should be [i, j, t, weight].

plot (*plottype*, *ij=None*, *t=None*, *ax=None*, ***plotparams*)

TenetoWorkflow

class TenetoWorkflow (*remove_nonterminal_output=True*)

Bases: object

Methods Summary

add_node(nodename, func[, depends_on, params])	Adds a node to the workflow graph.
calc_runorder()	Calculate the run order of the different nodes on the graph.
delete_output_from_level(level)	Delete the output found after calling TenetoWorkflow.run().
make_workflow_figure([fig, ax])	Creates a figure depicting the workflow figure.
remove_node(nodename)	Remove a node from the graph.
run()	Runs the entire graph.

Methods Documentation

add_node (*nodename*, *func*, *depends_on=None*, *params=None*)

Adds a node to the workflow graph.

Parameters

- **nodename** (*str*) – Name of the node
- **func** (*str*) – The function that is to be called. The alternatives here are 'TemporalNetwork' or 'TenetoBIDS', or any of the functions that can be called within these classes.
- **depends_on** (*str*) – which step the node depends on. If empty, is considered to preceed the previous step. If 'isroot' is specified, it is considered an input variable.
- **params** (*dict*) – Parameters that are passed into func.

Note: These functions are not run until TenetoWorkflow.run() is called.

calc_runorder ()

Calculate the run order of the different nodes on the graph.

delete_output_from_level (*level*)

Delete the output found after calling TenetoWorkflow.run().

make_workflow_figure (*fig=None*, *ax=None*)

Creates a figure depicting the workflow figure.

Parameters

- **fig** (*matplotlib*) –

- **ax** (*matplotlib*) –
- **fig is used as input, ax should be too.** (*if*) –

Returns **fig, ax** – matplotlib figure and axis

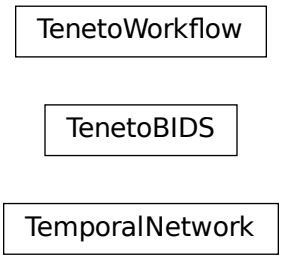
Return type matplotlib

remove_node(*nodename*)
Remove a node from the graph.

Parameters **nodename** (*str*) – Name of node that is to be removed.

run()
Runs the entire graph.

Class Inheritance Diagram



3.2 teneto.communitydetection

3.2.1 Louvain

make_consensus_matrix(*com_membership*, *th=0.5*)

Makes the consensus matrix.

From multiple iterations, finds a consensus partition.

.

com_membership [array] Shape should be node, time, iteration.

th [float] threshold to cancel noisey edges

D [array] consensus matrix

make_temporal_consensus(*com_membership*)
Matches community labels accross time-points.

Jaccard matching is in a greedy fashiong. Matching the largest community at t with the community at t-1.

Parameters **com_membership** (*array*) – Shape should be node, time.

Returns **D** – temporal consensus matrix using Jaccard distance

Return type array

temporal_louvain(*tnet*, *resolution=1*, *intersliceweight=1*, *n_iter=100*, *negativeedge='ignore'*, *randomseed=None*, *consensus_threshold=0.5*, *temporal_consensus=True*, *njobs=1*)

Louvain clustering for a temporal network.

Parameters

- **tnet** (*array, dict, TemporalNetwork*) – Input network
- **resolution** (*int*) – resolution of Louvain clustering ($gamma$)
- **intersliceweight** (*int*) – interslice weight of multilayer clustering ($omega$). Must be positive.
- **n_iter** (*int*) – Number of iterations to run louvain for
- **randomseed** (*int*) – Set for reproduceability
- **negativeedge** (*str*) – If there are negative edges, what should be done with them. Options: 'ignore' (i.e. set to 0). More options to be added.
- **consensus** (*float (0.5 default)*) – When creating consensus matrix to average over number of iterations, keep values when the consensus is this amount.

Returns **communities** – node,time array of community assignment

Return type array (node,time)

Notes

References

3.3 teneto.communitymeasures

3.3.1 teneto.communitymeasures Package

Functions to quantify temporal communities

Functions

flexibility(communities)	Amount a node changes community
allegiance(community)	Computes allience of communities.
recruitment(temporalcommunities, ...)	Calculates recruitment in relation to static communities.
integration(temporalcommunities, ...)	Calculates the integration coefficient for each node.
promiscuity(communities)	Calculates promiscuity of communities.
persistence(communities[, calc])	Persistence is the proportion of consecutive time-points that a temporal community is in the same community at the next time-point

flexibility

flexibility (*communities*)
Amount a node changes community

Parameters **communities** (*array*) – Community array of shape (node,time)

Returns **flex** – Size with the flexibility of each node.

Return type array

Notes

Flexbility calculates the number of times a node switches its community label during a time series [flex-1]. It is normalized by the number of possible changes which could occur. It is important to make sure that the different community labels accross time points are not artbirary.

References

allegiance

allegiance (*community*)
Computes allience of communities.

The allegiance matrix with values representing the probability that nodes i and j were assigned to the same community by time-varying clustering methods.[alleg-1]_

Parameters **community** (*array*) – array of community assignment of size node,time

Returns

- **P** (*array*) – module allegiance matrix, with P_ij probability that area i and j are in the same community
- *Reference*
- ———-
- .. *[alleg-1]* – Bassett, et al. (2013) “Robust detection of dynamic community structure in networks”, Chaos, 23, 1

recruitment

recruitment (*temporalcommunities*, *staticcommunities*)
Calculates recruitment in relation to static communities.

Calculates recruitment coefficient for each node. Recruitment coefficient is the average probability of nodes from the same static communities being in the same temporal communities at other time-points or during different tasks.

temporalcommunities [array] temporal communities vector (node,time)

staticcommunities [array] Static communities vector for each node

recruit [array] recruitment coefficient for each node

integration

integration (*temporalcommunities*, *staticcommunities*)

Calculates the integration coefficient for each node. Measures the average probability that a node is in the same community as nodes from other systems.

temporalcommunities [array] temporal communities vector (node,time)

staticcommunities [array] Static communities vector for each node

integration_coeff [array] integration coefficient for each node

Danielle S. Bassett, Muzhi Yang, Nicholas F. Wymbs, Scott T. Grafton. Learning-Induced Autonomy of Sensorimotor Systems. Nat Neurosci. 2015 May;18(5):744-51.

Marcelo Mattar, Michael W. Cole, Sharon Thompson-Schill, Danielle S. Bassett. A Functional Cartography of Cognitive Systems. PLoS Comput Biol. 2015 Dec 2;11(12):e1004533.

promiscuity

promiscuity (*communities*)

Calculates promiscuity of communities.

Promiscuity calculates the number of communities each node is a member of. 0 entails only 1 community. 1 entails all communities [prom-1].

Parameters **communities** (*array*) – temporal communities labels of type (node,time). Temporal communities labels should be non-trivial through snapshots (i.e. temporal consensus clustering should be run)

Returns **promiscuity_coeff** – promiscuity of each node

Return type array

References

persistence

persistence (*communities*, *calc='global'*)

Persistence is the proportion of consecutive time-points that a temporal community is in the same community at the next time-point

Parameters

- **communities** (*array*) – temporal communities of type: node,time (singlelabel) or node,node,time (for multilabel) communities
- **calc** (*str*) – can be 'global', 'time', or 'node'

Returns **persit_coeff** – the percentage of nodes that calculate the overall persistence (calc=global), or each node (calc=node), or for each time-point (calc=time)

Return type array

References

Bazzi, Marya, et al. "Community detection in temporal multilayer networks, with an application to correlation networks." Multiscale Modeling & Simulation 14.1 (2016): 1-41.

Note: Bazzi et al present a non-normalized version with the global output.

3.4 teneto.networkmeasures

3.4.1 teneto.networkmeasures Package

Imports from networkmeasures

Functions

temporal_degree_centrality(tnet[, axis, ...])	Temporal degree of network.
shortest_temporal_path(tnet[, steps_per_t, ...])	Shortest temporal path
temporal_closeness_centrality([tnet, paths])	Returns temporal closeness centrality per node.
intercontacttimes(tnet)	Calculates the intercontacttimes of each edge in a network.
volatility(tnet[, distance_func, calc, ...])	Volatility of temporal networks.
bursty_coeff(data[, calc, nodes, ...])	Calculates the bursty coefficient.[1][2]
fluctuability(netin[, calc])	Fluctuability of temporal networks.
temporal_efficiency([tnet, paths, calc])	Returns temporal efficiency estimate.
temporal_efficiency([tnet, paths, calc])	Returns temporal efficiency estimate.
reachability_latency([tnet, paths, rratio, calc])	Reachability latency.
sid(tnet, communities[, axis, calc, decay])	Segregation integration difference (SID).
temporal_participation_coeff(tnet[, ...])	Calculates the temporal participation coefficient
topological_overlap(tnet[, calc])	Topological overlap quantifies the persistency of edges through time.
local_variation(data)	Calculates the local variaiont of inter-contact times.
temporal_betweenness_centrality([tnet, ...])	Returns temporal betweenness centrality per node.

temporal_degree_centrality

temporal_degree_centrality(*tnet*, *axis=0*, *calc='overtime'*, *communities=None*, *decay=0*, *ignorediagonal=True*)

Temporal degree of network.

The sum of all connections each node has through time (either per timepoint or over the entire temporal sequence).

Parameters

- **net** (*array, dict*) – Temporal network input (graphlet or contact). Can have nettype: 'bu', 'bd', 'wu', 'wd'

- **axis** (*int*) – Dimension that is returned 0 or 1 (default 0). Note, only relevant for directed networks. i.e. if 0, node i has Aijt summed over j and t. and if 1, node j has Aijt summed over i and t.
- **calc** (*str*) – Can be following alternatives:

  'overtime' : returns a 1 x node vector. Returns the degree/stregnth over all time points.

  'pertime' : returns a node x time array. Returns the degree/strength per time point.

  'module_degree_zscore' : returns the Z-scored within community degree centrality (communities argument required). This is done for each time-point i.e. 'pertime' returns static degree centrality per time-point.
- **ignorediagonal** (*bool*) – if True, diagonal is made to 0.
- **communities** (*array (Nx1)*) – Vector of community assignment. If this is given and calc='pertime', then the strength within and between each communities is returned. (Note, this is not technically degree centrality).
- **decay** (*int*) – if calc = 'pertime', then decay is possible where the centrality of the previous time point is carried over to the next time point but decays at a value of e^decay such that $D_d(t+1) = e^{-decay}D_d(t) + D(t+1)$. If decay is 0 then the final D will equal D when calc='overtime', if decay = inf then this will equal calc='pertime'.

Returns **D** – temporal degree centrality (nodal measure). Array is 1D ('overtime'), 2D ('pertime', 'module_degree_zscore'), or 3D ('pertime' + communities (non-nodal/community measures)).

Return type array

Notes

When the network is weighted, this could also be called "temporal strength" or "temporal strength centrality". This is a simple extension of the static definition. At times this has been defined slightly differently. Here we followed the definitions in [**?**] or [**?**]. There are however many authors prior to this that have used temporal degree centrality.

There are two basic versions of temporal degree centrality implemented: the average temporal degree centrality (`calc='overtime'`) and temporal degree centrality (`calc='pertime'`).

When `calc='pertime'`:

$$D_{it} = \sum_{j} A_{ijt}$$

where A is the multi-layer connectivity matrix of the temporal network.

This entails that D_{it} is the sum of a node i's degree/strength at t. This has also been called the instantaneous degree centrality [**?**].

When `calc='overtime'`:

$$D_{i} = \sum_{t} \sum_{j} A_{ijt}$$

i.e. D_i is the sum of a node i's degree/strength over all time points.

There are some additional options which can modify the estimate. One way is to add a decay term. This entails that ..math::*D_{it}*, uses some of the previous time-points estimate. An exponential decay is used here.

$$D_{it} = e^{-b} D_{i(t-1)} + \sum_{j} A_{ijt}$$

where b is the deay parameter specified in the function. This, to my knowledge, was first introdueced by [**?**].

References

shortest_temporal_path

shortest_temporal_path(*tnet*, *steps_per_t='all'*, *i=None*, *j=None*, *it=None*, *minimise='temporal_distance'*)

Shortest temporal path

Parameters

- **tnet** (*tnet obj, array or dict*) – input network. nettype: bu, bd.
- **steps_per_t** (*int or str*) – If str, should be 'all'. How many edges can be travelled during a single time-point.
- **i** (*list*) – List of node indicies to restrict analysis. These are nodes the paths start from. Default is all nodes.
- **j** (*list*) – List of node indicies to restrict analysis. There are nodes the paths end on. Default is all nodes.
- **it** (*None, int, list*) – Time points for parts. Either None (default) which takes all time points, an integer to indicate which time point to start at, or a list of time-points that is included in analysis (including end time-point).
- **minimise** (*str*) – Can be "temporal_distance", returns the path that has the smallest temporal distance. It is possible there can be a path that is a smaller topological distance (this option currently not available).

Returns **paths** – Dataframe consisting of information about all the paths found.

Return type pandas df

Notes

The shortest temporal path calculates the temporal and topological distance there to be a path between nodes.

The argument steps_per_t allows for multiple nodes to be travelled per time-point.

Topological distance is the number of edges that are travelled. Temporal distance is the number of time-points.

This function returns the path that is the shortest temporal distance away.

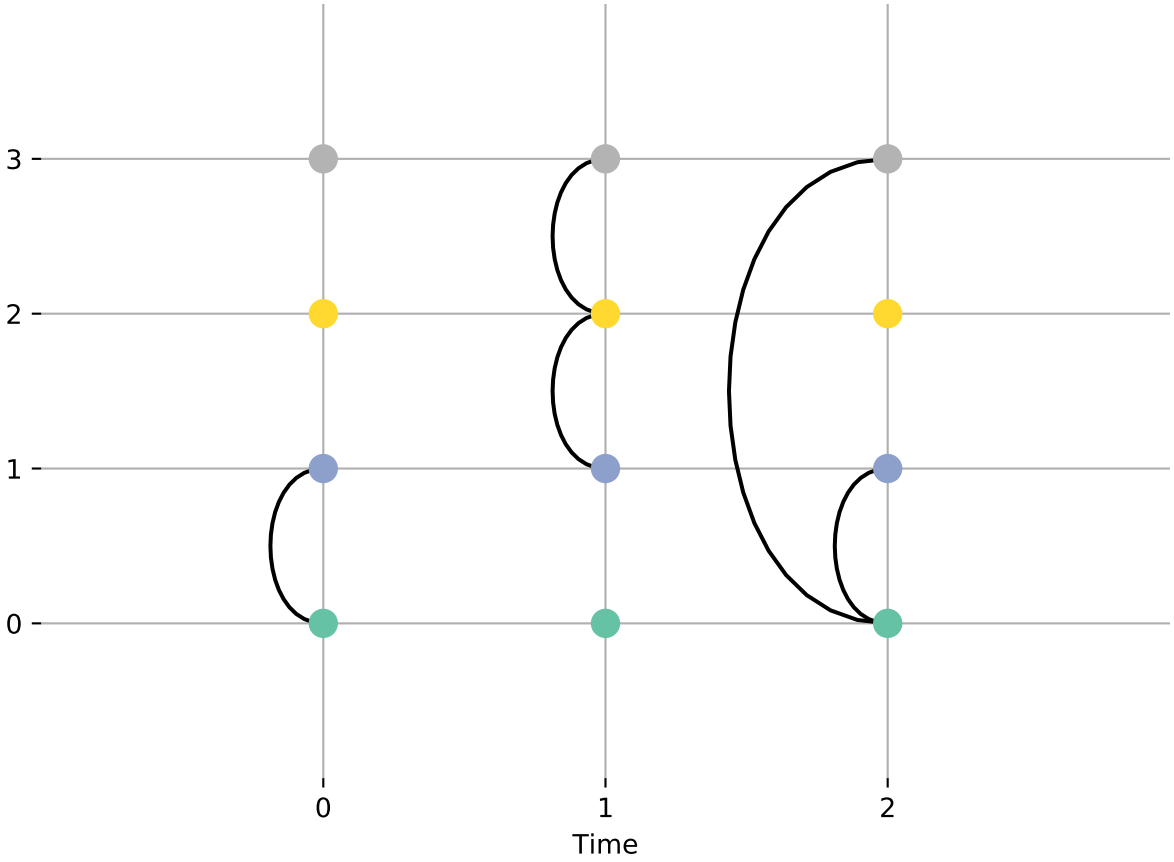
Examples

Let us start by creating a small network.

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> import teneto
>>> G = np.zeros([4, 4, 3])
>>> G[0, 1, [0, 2]] = 1
>>> G[0, 3, [2]] = 1
>>> G[1, 2, [1]] = 1
>>> G[2, 3, [1]] = 1
```

Let us look at this network to see what is there.

```
>>> fig, ax = plt.subplots(1)
>>> ax = teneto.plot.slice_plot(G, ax, nodelabels=[0,1,2,3], timelabels=[0,1,2],␣
↪cmap='Set2')
>>> plt.tight_layout()
>>> fig.show()
```



Here we can visualize what the shortest paths are. Let us start by starting at node 0 we want to find the path to node 3, starting at time 0. To do this we write:

```
>>> sp = teneto.networkmeasures.shortest_temporal_path(G, i=0, j=3, it=0)
>>> sp['temporal-distance']
0    2
Name: temporal-distance, dtype: int64
>>> sp['topological-distance']
0    3
Name: topological-distance, dtype: int64
>>> sp['path includes']
0    [[0, 1], [1, 2], [2, 3]]
Name: path includes, dtype: object
```

Here we see that the shortest path takes 3 steps (topological distance of 3) at 2 time points.

It starts by going from node 0 to 1 at t=0, then 1 to 2 and 2 to 3 at t=1. We can see all the nodes that were travelled in the "path includes" list.

In the above example, it was possible to traverse multiple edges at a single time-point. It is possible to restrain

that by setting the steps_per_t argument

```
>>> sp = teneto.networkmeasures.shortest_temporal_path(G, i=0, j=3, it=0, steps_
↪per_t=1)
>>> sp['temporal-distance']
0    3
Name: temporal-distance, dtype: int64
>>> sp['topological-distance']
0    1
Name: topological-distance, dtype: int64
>>> sp['path includes']
0    [[0, 3]]
Name: path includes, dtype: object
```

Here we see that the path is now only one edge, 0 to 3 at t=2. The quicker path is no longer possible.

temporal_closeness_centrality

temporal_closeness_centrality (*tnet=None*, *paths=None*)

Returns temporal closeness centrality per node.

Temporal closeness centrlaity is the sum of a node's average temporal paths with all other nodes.

Parameters

- **tnet** (*array, dict, object*) – Temporal network input with nettype: 'bu', 'bd'.
- **paths** (*pandas dataframe*) – Output of Teneto-BIDS.networkmeasure.shortest_temporal_paths

Note: Only one input (tnet or paths) can be supplied to the function.

Returns temporal closness centrality (nodal measure)

Return type close: array

Notes

Temporal closeness centrality is defined in [**?**]:

$$C_i^T = \frac{1}{N-1} \sum_j \frac{1}{tau_{ij}}$$

Where
tau_{ij} is the average temporal paths between node i and j.

Note, there are multiple different types of temporal distance measures that can be used in temporal networks. If a temporal network is used as input (i.e. not the paths), then teneto uses `shortest_temporal_path()` to calculates the shortest paths. See `shortest_temporal_path()` for more details.

intercontacttimes

intercontacttimes(*tnet*)

Calculates the intercontacttimes of each edge in a network.

Parameters **tnet** (*array, dict*) – Temporal network (craphlet or contact). Nettype: 'bu',

Returns **contacts** – Intercontact times as numpy array in dictionary. contacts['intercontacttimes']

Return type dict

Notes

The inter-contact times is calculated by the time between consequecutive "active" edges (where active means that the value is 1 in a binary network).

Examples

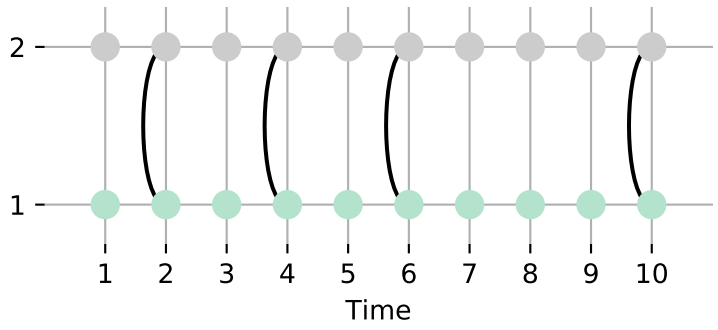
This example goes through how inter-contact times are calculated.

```
>>> import teneto
>>> import numpy as np
```

Make a network with 2 nodes and 4 time-points with 4 edges spaced out.

```
>>> G = np.zeros([2,2,10])
>>> edge_on = [1,3,5,9]
>>> G[0,1,edge_on] = 1
```

The network visualised below make it clear what the inter-contact times are between the two nodes:



Calculating the inter-contact times of these edges becomes: 2,2,4 between nodes 0 and 1.

```
>>> ict = teneto.networkmeasures.intercontacttimes(G)
```

The function returns a dictionary with the icts in the key: intercontacttimes. This is of the size NxN. So the icts between nodes 0 and 1 are found by:

```
>>> ict['intercontacttimes'][0,1]
array([2, 2, 4])
```

volatility

volatility(*tnet*, *distance_func='default'*, *calc='overtime'*, *communities=None*, *event_displacement=None*)

Volatility of temporal networks.

Volatility is the average distance between consecutive time points (difference is caclualted either globally or per edge).

Parameters

- **tnet** (*array or dict*) – temporal network input (graphlet or contact). Nettype: 'bu','bd','wu','wd'
- **D** (*str*) – Distance function. Following options available: 'default', 'hamming', 'euclidean'. (Default implies hamming for binary networks, euclidean for weighted).
- **calc** (*str*) – Version of volaitility to caclulate. Possibilities include: 'overtime' - (default): the average distance of all nodes for each consecutive time point). 'edge' - average distance between consecutive time points for each edge). Takes considerably longer 'node' - (i.e. returns the average per node output when calculating volatility per 'edge'). 'pertime' - returns volatility per time point 'communities' - returns volatility per communitieswork id (see communities). Also is returned per time-point and this may be changed in the future (additional options are then required) 'event_displacement' - calculates the volatility from a specified point. Returns time-series.
- **communities** (*array*) – Array of indicies for community (eiter (node) or (node,time) dimensions).
- **event_displacement** (*int*) – if calc = event_displacement specify the temporal index. All other time-points are calculated in relation to this time point.

Notes

Volatility calculates the difference between network snapshots.

$$V_t = D(G_t, G_{t+1})$$

Where D is some distance function (e.g. Hamming distance for binary matrices).

V can be calculated for the entire network (global), but can also be calculated for individual edges, nodes or given a community vector.

Index of communities are returned "as is" with a shape of: (max(communities)+1, max(communities)+1). So if the indexes used are [1,2,3,5], V.shape==(6,6). The returning V[1,2] will correspond indexes 1 and 2. And missing index (e.g. here 0 and 4 will be NANs in rows and columns). If this behaviour is unwanted, call clean_communitiesdexes first.

Examples

Import everything needed.

```
>>> import teneto
>>> import numpy
>>> np.random.seed(1)
>>> tnet = teneto.TemporalNetwork(nettype='bu')
```

Here we generate a binary network where edges have a 0.5 change of going "on", and once on a 0.2 change to go "off"

```
>>> tnet.generatenetwork('rand_binomial', size=(3,10), prob=(0.5,0.2))
```

Calculate the volatility

```
>>> tnet.calc_networkmeasure('volatility', distance_func='hamming')
0.5555555555555556
```

If we change the probabilities to instead be certain edges disapeared the time-point after the appeared:

```
>>> tnet.generatenetwork('rand_binomial', size=(3,10), prob=(0.5,1))
```

This will make a more volatile network

```
>>> tnet.calc_networkmeasure('volatility', distance_func='hamming')
0.1111111111111111
```

We can calculate the volatility per time instead

```
>>> vol_time = tnet.calc_networkmeasure('volatility', calc='pertime', distance_
↪func='hamming')
>>> len(vol_time)
9
>>> vol_time[0]
0.3333333333333333
```

Or per node:

```
>>> vol_node = tnet.calc_networkmeasure('volatility', calc='node', distance_func=
↪'hamming')
>>> vol_node
array([0.07407407, 0.07407407, 0.07407407])
```

Here we see the volatility for each node was the same.

It is also possible to pass a community vector. The function will return volatility both within and between each community. So the following has two communities:

```
>>> vol_com = tnet.calc_networkmeasure('volatility', calc='communities',␣
↪communities=[0,1,1], distance_func='hamming')
>>> vol_com.shape
(2, 2, 9)
>>> vol_com[:,:,0]
array([[nan, 0.5],
       [0.5, 0. ]])
```

And we see that, at time-point 0, there is some volatility between community 0 and 1. Further, there is no volatility within community 1. The reason for nan appearing is due to there only being 1 node in community 0.

vol : array

bursty_coeff

bursty_coeff(*data*, *calc='edge'*, *nodes='all'*, *communities=None*, *threshold_type=None*, *threshold_level=None*, *threshold_params=None*)

Calculates the bursty coefficient.[1][2]

Parameters

- **data** (*array, dict*) – This is either (1) temporal network input with nettype: 'bu', 'bd'. (2) dictionary of ICTs (output of *intercontacttimes*). (3) temporal network input with nettype: 'wu', 'wd'. If weighted, you must also specify threshold_type and threshold_value which will make it binary.
- **calc** (*str*) – Caclulate the bursty coeff over what. Options include 'edge': calculate B on all ICTs between node i and j. (Default); 'node': caclulate B on all ICTs connected to node i.; 'communities': calculate B for each communities (argument communities then required); 'meanEdgePerNode': first calculate ICTs between i and j, then take the mean over all j.
- **nodes** (*list or str*) – Options: 'all': do for all nodes (default) or list of node indexes to calculate.
- **communities** (*array, optional*) – None (default) or Nx1 vector of communities assignment. This returns a "centrality" per communities instead of per node.
- **threshold_type** (*str, optional*) – If input is weighted. Specify binarizing threshold type. See teneto.utils.binarize
- **threshold_level** (*str, optional*) – If input is weighted. Specify binarizing threshold level. See teneto.utils.binarize
- **threhsold_params** (*dict*) – If input is weighted. Dictionawy with kwargs for teneto.utils.binarize

Returns **B** – Bursty coefficienct per (edge or node measure).

Return type array

Notes

The burstiness coefficent, B, is defined in refs [1,2] as:

$$B = \frac{\sigma_{ICT} - \mu_{ICT}}{\sigma_{ICT} + \mu_{ICT}}$$

Where σ_{ICT} and μ_{ICT} are the standard deviation and mean of the inter-contact times respectively (see teneto.networkmeasures.intercontacttimes)

When B > 0, indicates bursty intercontact times. When B < 0, indicates periodic/tonic intercontact times. When B = 0, indicates random.

Examples

First import all necessary packages

```
>>> import teneto
>>> import numpy as np
```

Now create 2 temporal network of 2 nodes and 60 time points. The first has periodict edges, repeating every other time-point:

```
>>> G_periodic = np.zeros([2, 2, 60])
>>> ts_periodic = np.arange(0, 60, 2)
>>> G_periodic[:,:,ts_periodic] = 1
```

The second has a more bursty pattern of edges:

```
>>> ts_bursty = [1, 8, 9, 32, 33, 34, 39, 40, 50, 51, 52, 55]
>>> G_bursty = np.zeros([2, 2, 60])
>>> G_bursty[:,:,ts_bursty] = 1
```

The two networks look like this:

Now we call bursty_coeff.

```
>>> B_periodic = teneto.networkmeasures.bursty_coeff(G_periodic)
>>> B_periodic
array([[nan, -1.],
       [-1., nan]])
```

Above we can see that between node 0 and 1, B=-1 (the diagonal is nan). Doing the same for the second example:

```
>>> B_bursty = teneto.networkmeasures.bursty_coeff(G_bursty)
>>> B_bursty
array([[       nan, 0.13311003],
       [0.13311003,        nan]])
```

gives a positive value, indicating the inter-contact times between node 0 and 1 is bursty.

References

fluctuability

fluctuability (*netin*, *calc='overtime'*)

Fluctuability of temporal networks.

This is the variation of the network's edges over time. [**?**] This is the unique number of edges through time divided by the overall number of edges.

Parameters

- **netin** (*array or dict*) – Temporal network input (graphlet or contact) (nettype: 'bd', 'bu', 'wu', 'wd')
- **calc** (*str*) – Version of fluctuabiility to calcualte. 'overtime'

Returns **fluct** – Fluctuability

Return type array

Notes

Fluctuability quantifies the variability of edges. Given x number of edges, F is higher when those are repeated edges among a smaller set of edges and lower when there are distributed across more edges.

$$F = \frac{\sum_{i,j} H_{i,j}}{\sum_{i,j,t} G_{i,j,t}}$$

where $H_{i,j}$ is a binary matrix where it is 1 if there is at least one t such that G_{i,j,t} = 1 (i.e. at least one temporal edge exists).

F is not normalized which makes comparisions of F across very different networks difficult (could be added).

Examples

This example compares the fluctability of two different networks with the same number of edges. Below two temporal networks, both with 3 nodes and 3 time-points. Both get 3 connections.

```
>>> import teneto
>>> import numpy as np
>>> # Manually specify node (i,j) and temporal (t) indicies.
>>> ind_highF_i = [0,0,1]
>>> ind_highF_j = [1,2,2]
>>> ind_highF_t = [1,2,2]
>>> ind_lowF_i = [0,0,0]
>>> ind_lowF_j = [1,1,1]
>>> ind_lowF_t = [0,1,2]
>>> # Define 2 networks below and set above edges to 1
>>> G_highF = np.zeros([3,3,3])
>>> G_lowF = np.zeros([3,3,3])
>>> G_highF[ind_highF_i,ind_highF_j,ind_highF_t] = 1
>>> G_lowF[ind_lowF_i,ind_lowF_j,ind_lowF_t] = 1
```

The two different networks look like this:

Now calculate the fluctability of the two networks above.

```
>>> F_high = teneto.networkmeasures.fluctuability(G_highF)
>>> F_high
1.0
>>> F_low = teneto.networkmeasures.fluctuability(G_lowF)
>>> F_low
0.3333333333333333
```

Here we see that the network with more unique connections has the higher fluctuability.

temporal_efficiency

temporal_efficiency (*tnet=None*, *paths=None*, *calc='overtime'*)

Returns temporal efficiency estimate. BU networks only.

Parameters

- **should be either tnet or paths.** (*Input*) –
- **data** (*array or dict*) – Temporal network input (graphlet or contact). nettype: 'bu', 'bd'.
- **paths** (*pandas dataframe*) – Output of Teneto-BIDS.networkmeasure.shortest_temporal_paths
- **calc** (*str*) – Options: 'overtime' (default) - measure averages over time and nodes; 'node' or 'node_from' average over nodes (i) and time. Giving average efficiency for i to j; 'node_to' measure average over nodes j and time;

  Giving average efficiency using paths to j from i;

Returns **E** – Global temporal efficiency

Return type array

reachability_latency

reachability_latency (*tnet=None*, *paths=None*, *rratio=1*, *calc='global'*)

Reachability latency. This is the r-th longest temporal path.

Parameters

- **data** (*array or dict*) – Can either be a network (graphlet or contact), binary unidrected only. Alternative can be a paths dictionary (output of teneto.networkmeasure.shortest_temporal_path)
- **rratio** (*float (default: 1)*) – reachability ratio that the latency is calculated in relation to. Value must be over 0 and up to 1. 1 (default) - all nodes must be reached. Other values (e.g. .5 imply that 50% of nodes are reached) This is rounded to the nearest node inter. E.g. if there are 6 nodes [1,2,3,4,5,6], it will be node 4 (due to round upwards)
- **calc** (*str*) – what to calculate. Alternatives: 'global' entire network; 'nodes': for each node.

Returns **reach_lat** – Reachability latency

Return type array

Notes

Reachability latency calculates the time it takes for the paths.

sid

sid (*tnet*, *communities*, *axis=0*, *calc='overtime'*, *decay=0*)

Segregation integration difference (SID). An estimation of each community or global difference of within versus between community strength.[sid-1]_

Parameters

- **tnet** (*array, dict*) – Temporal network input (graphlet or contact). Allowerd nettype: 'bu', 'bd', 'wu', 'wd'
- **communities** (*array*) – a Nx1 vector or NxT array of community assignment.
- **axis** (*int*) – Dimension that is returned 0 or 1 (default 0). Note, only relevant for directed networks. i.e. if 0, node i has Aijt summed over j and t. and if 1, node j has Aijt summed over i and t.
- **calc** (*str*) – 'overtime' returns SID over time (a 1 x community vector) (default);

  'community_pairs' returns a community x community x time matrix, which is the SID for each community pairing;

  'community_avg' (returns a community x time matrix). Which is the normalized average of each community to all other communities.

  'community_pairs_norm' (returns a community x time matrix). Which is the normalized average of each community pair. Each pair is normalized to the average of both communities in the pair.
- **decay** (*int*) – if calc = 'community_pairs' or 'community_avg', then decay is possible where the centrality of the previous time point is carried over to the next time point but

decays at a value of e^decay such that the temporal centrality measure becomes: $D(t+1) = e^{-decay}D(t) + D(t+1)$.

Returns **sid** – segregation-integration difference. Format: 2d or 3d numpy array (depending on calc) representing (community,community,time) or (community,time)

Return type array

Notes

SID tries to quantify if there is more segergation or intgration compared to other time-points. If SID > 0, then there is more segregation than usual. If SID < 0, then there is more integration than usual.

There are three different variants of SID, one is a global measure (calc='overtime'), the second is a value per community (calc='community_avg'), the third is a value for each community-community pairing (calc='community_pairs').

First we calculate the temporal strength for each edge. This is calculate by

$$S_{i,t} = \sum_j G_{i,j,t}$$

The pairwise SID, when the network is undirected, is calculated by

$$SID_{A,B,t} = (\frac{2}{N_A(N_A-1)})S_{A,t} - (\frac{1}{N_A * N_B})S_{A,B,t})$$

Where $S_{A,t}$ is the average temporal strength at time-point t for community A. N_A is the number of nodes in community A.

When calculating the SID for a community, it is calculated byL

$$SID_{A,t} = \sum_b^C (\frac{2}{N_A(N_A-1)})S_{A,t} - (\frac{1}{N_A * N_b})S_{A,b,t})$$

Where C is the number of communities.

When calculating the SID globally, it is calculated byL

$$SID_t = \sum_a^C \sum_b^C (\frac{2}{N_a(N_a-1)})S_{A,t} - (\frac{1}{N_a * N_b})S_{a,b,t})$$

References

temporal_participation_coeff

temporal_participation_coeff(*tnet*, *communities=None*, *decay=None*, *removeneg=False*)

Calculates the temporal participation coefficient

Temporal participation coefficient is a measure of diversity of connections across communities for individual nodes.

Parameters

- **tnet** (*array, dict*) – graphlet or contact sequence input. Only positive matrices considered.

- **communities** (*array*) – community vector. Either 1D (node) community index or 2D (node,time).
- **removeneg** (*bool (default false)*) – If true, all values < 0 are made to be 0.

Returns **P** – participation coefficient

Return type array

Notes

Static participatoin coefficient is:

$$P_i = 1 - \sum_s^{N_M} (\frac{k_{is}}{k_i})^2$$

Where s is the index of each community (N_M). k_i is total degree of node. And k_{is} is degree of connections within community.[part-1]_

This "temporal" version only loops through temporal snapshots and calculates P_i for each t.

If directed, function sums axis=1, so tnet may need to be transposed before hand depending on what type of directed part_coef you are interested in.

References

topological_overlap

topological_overlap(*tnet*, *calc='pertime'*)

Topological overlap quantifies the persistency of edges through time.

If two consequtive time-points have similar edges, this becomes high (max 1). If there is high change, this becomes 0.

References: [**?**], [**?**]

Parameters

- **tnet** (*array, dict*) – graphlet or contact sequence input. Nettype: 'bu'.
- **calc** (*str*) – which version of topological overlap to calculate: 'node' - calculates for each node, averaging over time. 'pertime' - (default) calculates for each node per time points. 'overtime' - calculates for each node per time points.

Returns **topo_overlap** – if calc = 'pertime', array is (node,time) in size. if calc = 'node', array is (node) in size. if calc = 'overtime', array is (1) in size. The final time point returns as nan.

Return type array

Notes

When edges persist over time, the topological overlap increases. It can be calculated as a global valu, per node, per node-time.

When calc='pertime', then the topological overlap is:

$$TopoOverlap_{i,t} = \frac{\sum_j G_{i,j,t} G_{i,j,t+1}}{\sqrt{\sum_j G_{i,j,t} \sum_j G_{i,j,t+1}}}$$

When calc='node', then the topological overlap is the mean of math:*TopoOverlap_{i,t}*:

$$AvgTopoOverlap_i = \frac{1}{T-1}\sum_t TopoOverlap_{i,t}$$

where T is the number of time-points. This is called the *average topological overlap*.

When calc='overtime', the *temporal-correlation coefficient* is calculated

$$TempCorrCoeff = \frac{1}{N}\sum_i AvgTopoOverlap_i$$

where N is the number of nodes.

For all the three measures above, the value is between 0 and 1 where 0 entails "all edges changes" and 1 entails "no edges change".

Examples

First import all necessary packages

```
>>> import teneto
>>> import numpy as np
```

Then make an temporal network with 3 nodes and 4 time-points.

```
>>> G = np.zeros([3, 3, 3])
>>> i_ind = np.array([0, 0, 0, 0,])
>>> j_ind = np.array([1, 1, 1, 2,])
>>> t_ind = np.array([0, 1, 2, 2,])
>>> G[i_ind, j_ind, t_ind] = 1
>>> G = G + G.transpose([1,0,2]) # Make symmetric
```

Now the topological overlap can be calculated:

```
>>> topo_overlap = teneto.networkmeasures.topological_overlap(G)
```

This returns *topo_overlap* which is a (node,time) array. Looking above at how we defined G, when t = 0, there is only the edge (0,1). When t = 1, this edge still remains. This means topo_overlap should equal 1 for node 0 at t=0 and 0 for node 2:

```
>>> topo_overlap[0,0]
1.0
>>> topo_overlap[2,0]
0.0
```

At t=2, there is now also an edge between (0,2), this means node 0's topological overlap at t=1 decreases as its edges have decreased in their persistency at the next time point (i.e. some change has occured). It equals ca. 0.71

```
>>> topo_overlap[0,1]
0.7071067811865475
```

If we want the average topological overlap, we simply add the calc argument to be 'node'.

```
>>> avg_topo_overlap = teneto.networkmeasures.topological_overlap(G, calc='node')
```

Now this is an array with a length of 3 (one per node).

```
>>> avg_topo_overlap
array([0.85355339, 1.        , 0.        ])
```

Here we see that node 1 had all its connections persist, node 2 had no connections persisting, and node 0 was in between.

To calculate the temporal correlation coefficient,

```
>>> temp_corr_coeff = teneto.networkmeasures.topological_overlap(G, calc='overtime
↪')
```

This produces one value reflecting all of G

```
>>> temp_corr_coeff
0.617851130197758
```

References

local_variation

local_variation(*data*)

Calculates the local variaiont of inter-contact times. [**?**], [**?**]

Parameters **data** (*array, dict*) – This is either (1) temporal network input (graphlet or contact) with nettype: 'bu', 'bd'. (2) dictionary of ICTs (output of *intercontacttimes*).

Returns **LV** – Local variation per edge.

Return type array

Notes

The local variation is like the bursty coefficient and quantifies if a series of inter-contact times are periodic, random or Poisson distributed or bursty.

It is defined as:

$$LV = \frac{3}{n-1}\sum_{i=1}^{n-1}\frac{\iota_i - \iota_{i+1}}{\iota_i + \iota_{i+1}}^2$$

Where ι are inter-contact times and i is the index of the inter-contact time (not a node index). n is the number of events, making n-1 the number of inter-contact times.

The possible range is: $0 \geq LV3$.

When periodic, LV=0, Poisson, LV=1 Larger LVs indicate bursty process.

Examples

First import all necessary packages

```
>>> import teneto
>>> import numpy as np
```

Now create 2 temporal network of 2 nodes and 60 time points. The first has periodict edges, repeating every other time-point:

```
>>> G_periodic = np.zeros([2, 2, 60])
>>> ts_periodic = np.arange(0, 60, 2)
>>> G_periodic[:,:,ts_periodic] = 1
```

The second has a more bursty pattern of edges:

```
>>> ts_bursty = [1, 8, 9, 32, 33, 34, 39, 40, 50, 51, 52, 55]
>>> G_bursty = np.zeros([2, 2, 60])
>>> G_bursty[:,:,ts_bursty] = 1
```

Now we call local variation for each edge.

```
>>> LV_periodic = teneto.networkmeasures.local_variation(G_periodic)
>>> LV_periodic
array([[nan,  0.],
       [ 0., nan]])
```

Above we can see that between node 0 and 1, LV=0 (the diagonal is nan). This is indicative of a periodic contacts (which is what we defined). Doing the same for the second example:

```
>>> LV_bursty = teneto.networkmeasures.local_variation(G_bursty)
>>> LV_bursty
array([[       nan, 1.28748748],
       [1.28748748,        nan]])
```

When the value is greater than 1, it indicates a bursty process.

nans are returned if there are no intercontacttimes

References

temporal_betweenness_centrality

temporal_betweenness_centrality (*tnet=None*, *paths=None*, *calc='pertime'*)

Returns temporal betweenness centrality per node.

Parameters

- **data** (*array or dict*) – Temporal network input (graphlet or contact). nettype: 'bu', 'bd'.
- **calc** (*str*) – either 'overtime' or 'pertime'
- **paths** (*pandas dataframe*) – Output of Teneto-BIDS.networkmeasure.shortest_temporal_paths

Note: Input should be *either* tnet or paths.

Returns

normalized temporal betweenness centrality.

If calc = 'pertime', returns (node,time)

If calc = 'overtime', returns (node)

Return type close: array

Notes

Temporal betweenness centrality uses the shortest temporal paths and calculates betweennesss from it.

Teneto returns a normalized betweenness centrality value, defined as [Bet-1]:

$$B_{it} = \frac{1}{(N-1)(N-2)} \sum_{j=1;j\neq i} \sum_{k=1;k\neq i,j} \frac{\sigma^i_{jkt}}{\sigma_{jk}}$$

If there is a shortest temporal path from j to k, starting at t that goes through node i, then σ^i_{jkt} is 1, otherwise 0. σ_{jk} is the total number of paths that exist from j to k. The remaining part of the equation normalizes by the number of nodes.

If a temporal network is used as input (i.e. not the paths), then teneto uses `shortest_temporal_path()` to calculates the shortest paths. See `shortest_temporal_path()` for more details.

If `calc=overtime` then the average B over time is returned.

References

3.5 teneto.plot

3.5.1 teneto.plot Package

Imports when importing plot

Functions

slice_plot(netin, ax[, nodelabels, ...])	Fuction draws "slice graph".
circle_plot(netIn, ax[, nodelabels, ...])	Function draws "circle plot" and exports axis handles
graphlet_stack_plot(netin, ax[, q, cmap, ...])	Returns matplotlib axis handle for graphlet_stack_plot.

slice_plot

slice_plot(*netin*, *ax*, *nodelabels=None*, *timelabels=None*, *communities=None*, *plotedgeweights=False*, *edgeweightscalar=1*, *timeunit=''*, *linestyle='k-'*, *cmap=None*, *nodesize=100*, *nodekwargs=None*, *edgekwargs=None*)

Fuction draws "slice graph".

A slice plot plots all the nodes per time point as a column with Bezier curvers connecting connected nodes.

Parameters

- **netin** (*array, dict*) – temporal network input (graphlet or contact)
- **ax** (*matplotlib figure handles.*) –
- **nodelabels** (*list*) – nodes labels. List of strings.
- **timelabels** (*list*) – labels of dimension Graph is expressed across. List of strings.
- **communities** (*array*) – array of size: (time) or (node,time). Nodes will be coloured accordingly.
- **plotedgeweights** (*bool*) – if True, edges will vary in size (default False)

- **edgeweightscalar** (*int*) – scalar to multiply all edges if tweaking is needed.
- **timeunit** (*string*) – unit time axis is in.
- **linestyle** (*string*) – line style of Bezier curves.
- **nodesize** (*int*) – size of nodes
- **nodekwargs** (*dict*) – any additional kwargs for matplotlib.plt.scatter for the nodes
- **edgekwargs** (*dict*) – any additional kwargs for matplotlib.plt.plots for the edges

Returns ax

Return type axis handle of slice graph

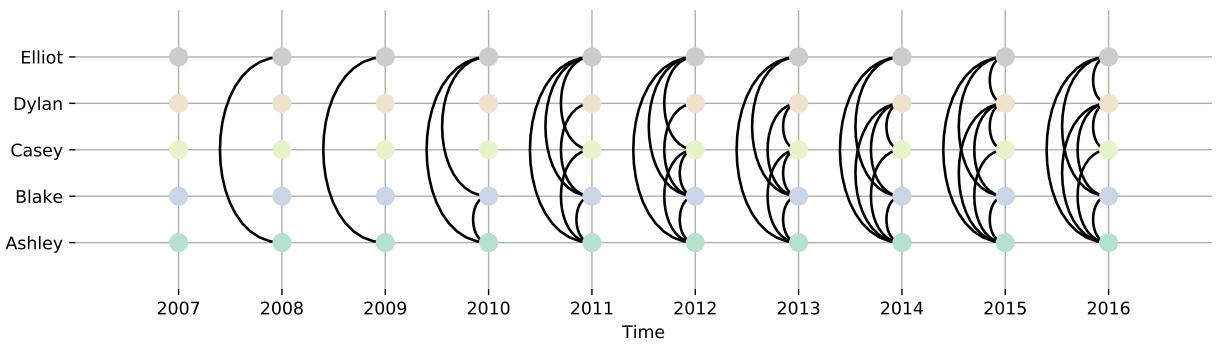
Examples

Create a network with some metadata

```
>>> import numpy as np
>>> import teneto
>>> import matplotlib.pyplot as plt
>>> np.random.seed(2017) # For reproduceability
>>> N = 5 # Number of nodes
>>> T = 10 # Number of timepoints
>>> # Probability of edge activation
>>> birth_rate = 0.2
>>> death_rate = .9
>>> # Add node names into the network and say time units are years, go 1 year per␣
↪graphlet and startyear is 2007
>>> cfg={}
>>> cfg['Fs'] = 1
>>> cfg['timeunit'] = 'Years'
>>> cfg['t0'] = 2007 #First year in network
>>> cfg['nodelabels'] = ['Ashley','Blake','Casey','Dylan','Elliot'] # Node names
>>> #Generate network
>>> C = teneto.generatenetwork.rand_binomial([N,T],[birth_rate, death_rate],
↪'contact','bu',netinfo=cfg)
```

Now this network can be plotted

```
>>> fig,ax = plt.subplots(figsize=(10,3))
>>> ax = teneto.plot.slice_plot(C, ax, cmap='Pastel2')
>>> plt.tight_layout()
>>> fig.show()
```



circle_plot

circle_plot (*netIn*, *ax*, *nodelabels=None*, *linestyle='k-'*, *nodesize=1000*, *cmap='Set2'*)

Function draws "circle plot" and exports axis handles

Parameters

- **netIn** (*temporal network input (graphlet or contact)*) –
- **ax** (*matplotlib ax handles.*) –
- **nodelabels** (*list*) – nodes labels. List of strings
- **linestyle** (*str*) – line style
- **nodesize** (*int*) – size of nodes
- **cmap** (*str*) – matplotlib colormap

Returns ax

Return type axis handle

Example

```
>>> import teneto
>>> import numpy
>>> import matplotlib.pyplot as plt
>>> G = np.zeros([6, 6])
>>> i = [0, 0, 0, 1, 2, 3, 4]
>>> j = [3, 4, 5, 5, 4, 5, 5]
>>> G[i, j] = 1
>>> fig, ax = plt.subplots(1)
>>> ax = teneto.plot.circle_plot(G, ax)
>>> fig.show()
```

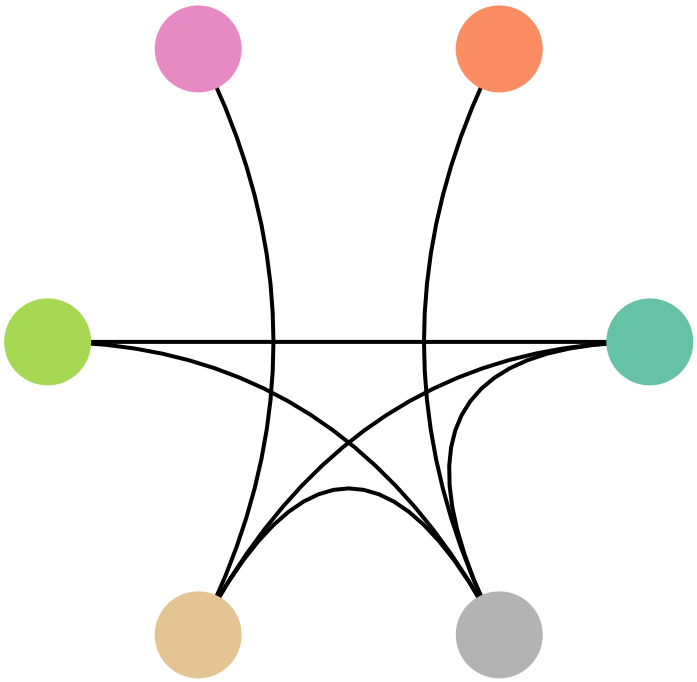
graphlet_stack_plot

graphlet_stack_plot (*netin*, *ax*, *q=10*, *cmap='Reds'*, *gridcolor='k'*, *borderwidth=2*, *bordercolor=None*, *Fs=1*, *timeunit=''*, *t0=1*, *sharpen='yes'*, *vminmax='minmax'*)

Returns matplotlib axis handle for graphlet_stack_plot. This is a row of transformed connectivity matrices to look like a 3D stack.

Parameters

- **netin** (*array, dict*) – network input (graphlet or contact)
- **ax** (*matplotlib ax handles.*) –
- **q** (*int*) – Quality. Increaseing this will lead to smoother axis but take up more memory.
- **cmap** (*str*) – Colormap (matplotlib) of graphlets
- **Fs** (*int*) – Sampling rate. Same as contact-representation (if netin is contact, and input is unset, contact dictionary is used)
- **timeunit** (*str*) – Unit of time for xlabel. Same as contact-representation (if netin is contact, and input is unset, contact dictionary is used)
- **t0** (*int*) – What should the first time point be called. Should be integer. Default 1.



- **gridcolor** (*str*) – The color of the grid section of the graphlets. Set to 'none' if not wanted.
- **borderwidth** (*int*) – Scales the size of border.
- **bordorcolor** – color of the border (at the moment it must be in RGB values between 0 and 1 -> this will be changed sometime in the future). Default: black.
- **vminmax** (*str*) – 'maxabs', 'minmax' (default), or list/array with length of 2. Specifies the min and max colormap value of graphlets. Maxabs entails [-max(abs(G)),max(abs(G))], minmax entails [min(G), max(G)].

Returns ax

Return type matplotlib ax handle

Note: This function can require a lot of RAM with larger networks.

Note: At the momenet bordercolor cannot be set to zero. To remove border, set bordorwidth=1 and bordercolor=[1,1,1] for temporay workaround.

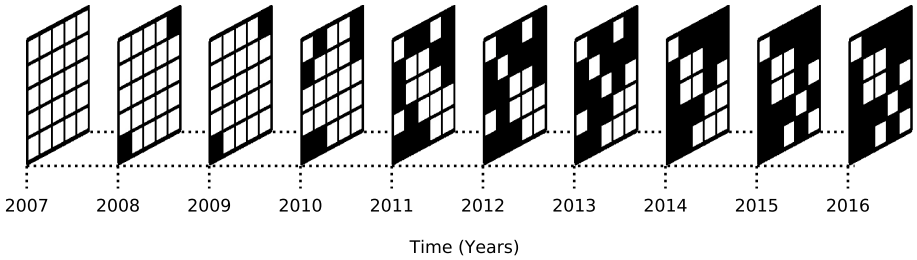
Examples

Create a network with some metadata

```
>>> import numpy as np
>>> import teneto
>>> import matplotlib.pyplot as plt
>>> np.random.seed(2017) # For reproduceability
>>> N = 5 # Number of nodes
>>> T = 10 # Number of timepoints
>>> # Probability of edge activation
>>> birth_rate = 0.2
>>> death_rate = .9
>>> # Add node names into the network and say time units are years, go 1 year per
graphlet and startyear is 2007
>>> cfg={}
>>> cfg['Fs'] = 1
>>> cfg['timeunit'] = 'Years'
>>> cfg['t0'] = 2007 #First year in network
>>> #Generate network
>>> C = teneto.generatenetwork.rand_binomial([N,T],[birth_rate, death_rate],
↪'contact','bu',netinfo=cfg)
```

Now this network can be plotted

```
>>> fig,ax = plt.subplots(figsize=(10,3))
>>> ax = teneto.plot.graphlet_stack_plot(C,ax,q=10,cmap='Greys')
>>> fig.show()
```



3.6 teneto.timeseries

3.6.1 teneto.timeseries Package

Import functions from time series module

Functions

derive_temporalnetwork(data, params)	Derives connectivity from the data.
postpro_pipeline(data, pipeline[, report])	Function to call multiple postprocessing steps.
postpro_fisher(data[, report])	Performs fisher transform on everything in data.
postpro_standardize(data[, report])	Standardizes everything in data (along axis -1).
postpro_boxcox(data[, report])	Performs box cox transform on everything in data.
remove_confounds(timeseries, confounds[, ...])	Removes specified confounds using nilearn.signal.clean
gen_report(report[, sdir, report_name])	Generates report of derivation and postprocess steps in teneto.timeseries

derive_temporalnetwork

derive_temporalnetwork (*data*, *params*)

Derives connectivity from the data.

A lot of data is inherently built with edges (e.g. communication between two individuals).

However other networks are derived from the covariance of time series (e.g. brain networks between two regions).

Covariance based metrics deriving time-resolved networks can be done in multiple ways. There are other methods apart from covariance based.

Derive a weight vector for each time point and then the corrrelation coefficient for each time point.

data [array] Time series data to perform connectivity derivation on. (Default dimensions are: (time as rows, nodes as columns). Change params{'dimord'} if you want it the other way (see below).

params [dict] Parameters for each method (see below).

method [str]

method: "distance","slidingwindow", "taperedslidingwindow",

"jackknife", "multiplytemporalderivative". Alternatively, method can be a weight matrix of size time x time.

Different methods have method specific paramaters (see below)

postpro : "no" (default). Other alternatives are: "fisher", "boxcox", "standardize"

and any combination seperated by a + (e.g. "fisher+boxcox"). See postpro_pipeline for more in

report [bool] False by default. If true, A report is saved in ./report/[analysis_id]/derivation_report.html if "yes"

report_path [str] String where the report is saved. Default is ./report/[analysis_id]/derivation_report.html

Distance metric calculates 1/Distance metric weights, and scales between 0 and 1. W[t,t] is excluded from the scaling and then set to 1.

params['distance']: str Distance metric (e.g. 'euclidean'). See teneto.utils.get_distance_function for more info

params['windowsize'] [int] Size of window.

params['windowsize'] [int] Size of window.

params['distribution'] [str] Scipy distribution (e.g. 'norm','expon'). Any distribution here: https://docs.scipy.org/doc/scipy/reference/stats.html

params['distribution_params'] [dict] Dictionary of distribution parameter, excluding the data "x" to generate pdf.

The data x should be considered to be centered at 0 and have a length of window size. (i.e. a window size of 5 entails x is [-2, -1, 0, 1, 2] a window size of 6 entails [-2.5, -1.5, 0.5, 0.5, 1.5, 2.5])

Given x params['distribution_params'] contains the remaining parameters.

e.g. normal distribution requires pdf(x, loc, scale) where loc=mean and scale=std.

Say we have a gaussian distribution, a window size of 21 and params['distribution_params'] = {'loc': 0, 'scale': 5}. This will lead to a gaussian with its peak at in the middle of each window with a standard deviation of 5.

params['windowsize'] [int] Size of window.

No parameters are necessary.

Optional parameters:

params['weight-var'] [array, (optional)] NxN array to weight the JC estimates (standerdized-JC*W). If weightby is selected, do not standerdize in postpro.

params['weight-mean'] [array, (optional)] NxN array to weight the JC estimates (standerdized-JC+W). If weightby is selected, do not standerdize in postpro.

No parameters are necessary.

Returns **G** – Connectivity estimates (nodes x nodes x time)

Return type array

About the general weighted pearson approach used for most methods, see: Thompson & Fransson (2019) A common framework for the problem of deriving estimates of dynamic functional brain connectivity. Neuroimage. (https://doi.org/10.1016/j.neuroimage.2017.12.057)

See also:

postpro_pipeline, *gen_report*

postpro_pipeline

postpro_pipeline(*data*, *pipeline*, *report=None*)

Function to call multiple postprocessing steps.

Parameters

- **data** (*array*) – pearson correlation values in temporal matrix form (node,node,time)
- **pipeline** (*list or str*) – (if string, each steps seperated by + sign).

  options 'fisher','boxcox','standardize'

  Each of the above 3 can be specified. If fisher is used, it must be before boxcox. If standardize is used it must be after boxcox and fisher.
- **report** (*bool*) – If true, appended to report.

Returns

- **postpro_data** (*array*) – postprocessed data
- **postprocessing_info** (*dict*) – Information about postprocessing

postpro_fisher

postpro_fisher(*data*, *report=None*)

Performs fisher transform on everything in data.

If report variable is passed, this is added to the report.

postpro_standardize

postpro_standardize(*data*, *report=None*)

Standardizes everything in data (along axis -1).

If report variable is passed, this is added to the report.

postpro_boxcox

postpro_boxcox(*data*, *report=None*)

Performs box cox transform on everything in data.

If report variable is passed, this is added to the report.

remove_confounds

remove_confounds(*timeseries*, *confounds*, *confound_selection=None*, *clean_params=None*)

Removes specified confounds using nilearn.signal.clean

Parameters

- **timeseries** (*array or dataframe*) – input timeseries with dimensions: (node,time)
- **confounds** (*array or dataframe*) – List of confounds. Expected format is (confound, time). If using TenetoBIDS, this does not need to be specified.
- **confound_selection** (*list*) – List of confounds. If None, all confoudns are removed
- **clean_params** (*dict*) – Dictionary of kawgs to pass to nilearn.signal.clean

Returns

Return type Says all TenetBIDS.get_selected_files with confounds removed with _rmconfounds at the end.

Note: There may be some issues regarding loading non-cleaned data through the TenetoBIDS functions instead of the cleaned data. This depeneds on when you clean the data.

gen_report

gen_report (*report*, *sdir='./'*, *report_name='report.html'*)
Generates report of derivation and postprocess steps in teneto.timeseries

3.7 teneto.trajectory

3.7.1 compression

Calculate compression of trajectory.

create_traj_ranges (*start*, *stop*, *N*)
Fills in the trajectory range.

Adapted from https://stackoverflow.com/a/40624614

rdp (*datin*, *delta=1*, *report=10*, *quiet=True*)

3.7.2 Module contents

Trajectory module

3.8 teneto.utils

3.8.1 teneto.utils Package

Many helper functions for Teneto

Functions

graphlet2contact(tnet[, params])	Converts array representation to contact representation.
contact2graphlet(C)	Converts contact representation to array representation.
binarize_percent(netin, level[, sign, axis])	Binarizes a network proprtionally.
binarize_rdp(netin, level[, sign, axis])	Binarizes a network based on RDP compression.
binarize_magnitude(netin, level[, sign])	Make binary network based on magnitude thresholding.
binarize(netin, threshold_type, threshold_level)	Binarizes a network, returning the network.
set_diagonal(tnet[, val])	Generally diagonal is set to 0.
gen_nettype(tnet[, weightonly])	Attempts to identify what nettype input graphlet tnet is.

Continued on next page

Table 9 – continued from previous page

check_input(netin[, rasie_if_undirected, conmat])	This function checks that netin input is either graphlet (tnet) or contact (C).
get_distance_function(requested_metric)	This function returns a specified distance function.
process_input(netin, allowedformats[, ...])	Takes input network and checks what the input is.
clean_community_indexes(communityID)	Takes input of community assignments.
multiple_contacts_get_values(C)	Given an contact representation with repeated contacts, this function removes duplicates and creates a value
df_to_array(df, netshape, nettype[, start_at])	Returns a numpy array (snapshot representation) from thedataframe contact list
check_distance_funciton_input(...)	Function checks distance_func_name, if it is specified as 'default'.
get_dimord(measure[, calc, community])	Get the dimension order of a network measure.
get_network_when(tnet[, i, j, t, ij, logic, ...])	Returns subset of dataframe that matches index
create_supraadjacency_matrix(tnet[, ...])	Returns a supraadjacency matrix from a temporal network structure
df_drop_ij_duplicates(df)	
tnet_to_nx(df[, t])	Creates undirected networkx object
is_jsonable(x)	Check if a dict is jsonable.

graphlet2contact

graphlet2contact (*tnet*, *params=None*)

Converts array representation to contact representation.

Contact representation are more efficient for memory storing. Also includes metadata which can made it easier for plotting. A contact representation contains all non-zero edges.

Parameters

- **tnet** (*array_like*) – Temporal network.
- **params** (*dict, optional*) – Dictionary of parameters for contact representation.

  Fs [int, default=1] sampling rate.

  timeunit [str, default=''] Sampling rate in for units (e.g. seconds, minutes, years).

  nettype [str, default='auto'] Define what type of network. Can be: 'auto': detects automatically; 'wd': weighted, directed; 'bd': binary, directed; 'wu': weighted, undirected; 'bu': binary, undirected.

  diagonal [int, default = 0.] What should the diagonal be.

  timetype [str, default='discrete'] Time units can be The params file becomes the foundation of 'C'. Any other information in params, will added to C.

  nodelabels [list] Set nod labels.

  t0**: int** Time label at first index.

Returns **C** – Contact representation of temporal network. Includes 'contacts', 'values' (if nettype[0]='w'),'nettype','netshape', 'Fs', 'dimord' and 'timeunit', 'timetype'.

Return type dict

contact2graphlet

contact2graphlet(*C*)

Converts contact representation to array representation.

Graphlet representation discards all meta information in contacts.

Parameters **C** (*dict*) – A contact representation. Must include keys: 'dimord', 'netshape', 'nettype', 'contacts' and, if weighted, 'values'.

Returns **tnet** – Graphlet representation of temporal network.

Return type array

Note: Returning elements of tnet will be float, even if binary graph.

---

binarize_percent

binarize_percent(*netin*, *level*, *sign='pos'*, *axis='time'*)

Binarizes a network proprtionally. When axis='time' (only one available at the moment) then the top values for each edge time series are considered.

Parameters

- **netin** (*array or dict*) – network (graphlet or contact representation),
- **level** (*float*) – Percent to keep (expressed as decimal, e.g. 0.1 = top 10%)
- **sign** (*str, default='pos'*) – States the sign of the thresholding. Can be 'pos', 'neg' or 'both'. If "neg", only negative values are thresholded and vice versa.
- **axis** (*str, default='time'*) – Specify which dimension thresholding is applied against. Can be 'time' (takes top % for each edge time-series) or 'graphlet' (takes top % for each graphlet)

Returns **netout** – Binarized network

Return type array or dict (depending on input)

binarize_rdp

binarize_rdp(*netin*, *level*, *sign='pos'*, *axis='time'*)

Binarizes a network based on RDP compression.

Parameters

- **netin** (*array or dict*) – Network (graphlet or contact representation),
- **level** (*float*) – Delta parameter which is the tolorated error in RDP compression.
- **sign** (*str, default='pos'*) – States the sign of the thresholding. Can be 'pos', 'neg' or 'both'. If "neg", only negative values are thresholded and vice versa.

Returns **netout** – Binarized network

Return type array or dict (dependning on input)

binarize_magnitude

binarize_magnitude (*netin*, *level*, *sign='pos'*)
Make binary network based on magnitude thresholding.

Parameters

- **netin** (*array or dict*) – Network (graphlet or contact representation),
- **level** (*float*) – Magnitude level threshold at.
- **sign** (*str, default='pos'*) – States the sign of the thresholding. Can be 'pos', 'neg' or 'both'. If "neg", only negative values are thresholded and vice versa.
- **axis** (*str, default='time'*) – Specify which dimension thresholding is applied against. Only 'time' option exists at present.

Returns **netout** – Binarized network

Return type array or dict (depending on input)

binarize

binarize (*netin*, *threshold_type*, *threshold_level*, *outputformat='auto'*, *sign='pos'*, *axis='time'*)
Binarizes a network, returning the network. General wrapper function for different binarization functions.

Parameters

- **netin** (*array or dict*) – Network (graphlet or contact representation),
- **threshold_type** (*str*) – What type of thresholds to make binarization. Options: 'rdp', 'percent', 'magnitude'.
- **threshold_level** (*str*) – Paramter dependent on threshold type. If 'rdp', it is the delta (i.e. error allowed in compression). If 'percent', it is the percentage to keep (e.g. 0.1, means keep 10% of signal). If 'magnitude', it is the amplitude of signal to keep.
- **outputformat** (*str*) – specify what format you want the output in: G, C, TN, or DF. If 'auto', input form is returned.
- **sign** (*str, default='pos'*) – States the sign of the thresholding. Can be 'pos', 'neg' or 'both'. If "neg", only negative values are thresholded and vice versa.
- **axis** (*str*) – Threshold over specfied axis. Valid for percent and rdp. Can be time or graphlet.

Returns **netout** – Binarized network

Return type array or dict (depending on input)

set_diagonal

set_diagonal (*tnet*, *val=0*)
Generally diagonal is set to 0. This function helps set the diagonal across time.

Parameters

- **tnet** (*array*) – temporal network (graphlet)
- **val** (*value to set diagonal to (default 0)*) –

Returns **tnet** – Graphlet representation with new diagonal

Return type array

gen_nettype

gen_nettype (*tnet*, *weightonly=False*)
Attempts to identify what nettype input graphlet tnet is. Diagonal is ignored.

tnet [array] temporal network (graphlet)

Returns **nettype** – 'wu', 'bu', 'wd', or 'bd'

Return type str

check_input

check_input (*netin*, *rasie_if_undirected=1*, *conmat=0*)
This function checks that netin input is either graphlet (tnet) or contact (C).

Parameters

- **netin** (*array or dict*) – temporal network, (graphlet or contact).
- **rasie_if_undirected** (*int, default=1.*) – Options 1 or 0. Error is raised if not found to be tnet or C
- **conmat** (*int, default=0.*) – Options 1 or 0. If 1, input is allowed to be a 2 dimensional connectivity matrix. Allows output to be 'M'

Returns **inputtype** – String indicating input type. 'G','C', 'M' or 'U' (unknown). M is special case only allowed when conmat=1 for a 2D connectivity matrix.

Return type str

get_distance_function

get_distance_function (*requested_metric*)
This function returns a specified distance function.

requested_metric: str Distance function. Can be any function in: https://docs.scipy.org/doc/scipy/reference/spatial.distance.html.

Returns **requested_metric**

Return type distance function

process_input

process_input (*netin*, *allowedformats*, *outputformat='G'*, *forcesparse=False*)
Takes input network and checks what the input is.

Parameters

- **netin** (*array, dict, or teneto.TemporalNetwork*) – Network (graphlet, contact or object)
- **allowedformats** (*list or str*) – Which format of network objects that are allowed. Options: 'C', 'TN', 'G'.

- **outputformat** (*str, default=G*) – Target output format. Options: 'C' or 'G'.

Returns

- **C** (*dict*)
- *OR*
- **tnet** (*array*) – Graphlet representation.
- **netinfo** (*dict*) – Metainformation about network.
- *OR*
- **tnet** (*object*) – object of teneto.TemporalNetwork class

clean_community_indexes

clean_community_indexes(*communityID*)

Takes input of community assignments. Returns reindexed community assignment by using smallest numbers possible.

Parameters **communityID** (*array-like*) – list or array of integers. Output from community detection algorithems.

Returns **new_communityID** – cleaned list going from 0 to len(np.unique(communityID))-1

Return type array

Note: Behaviour of funciton entails that the lowest community integer in communityID will recieve the lowest integer in new_communityID.

---

multiple_contacts_get_values

multiple_contacts_get_values(*C*)

Given an contact representation with repeated contacts, this function removes duplicates and creates a value

Parameters **C** (*dict*) – contact representation with multiple repeated contacts.

Returns Contact representation with duplicate contacts removed and the number of duplicates is now in the 'values' field.

Return type C_out: dict

df_to_array

df_to_array(*df*, *netshape*, *nettype*, *start_at='min'*)

Returns a numpy array (snapshot representation) from thedataframe contact list

Parameters

- **df** – pandas df pandas df with columns, i,j,t.
- **netshape** – tuple network shape, format: (node, time)
- **nettype** – str 'wu', 'wd', 'bu', 'bd'

- **start_at** – str ‘min’ or ‘zero’ or int. If min, the 0th time-point in the array is min t value. If zero, the 0th time-point in the array is 0. If int, the 0th time-point in array starts at int in df.

tnet [array] (node,node,time) array for the network

check_distance_funciton_input

check_distance_funciton_input(*distance_func_name*, *netinfo*)

Function checks distance_func_name, if it is specified as ‘default’. Then given the type of the network selects a default distance function.

Parameters

- **distance_func_name** (*str*) – distance function name.
- **netinfo** (*dict*) – the output of utils.process_input

Returns **distance_func_name** – distance function name.

Return type str

get_dimord

get_dimord(*measure*, *calc=None*, *community=None*)

Get the dimension order of a network measure.

Parameters

- **measure** (*str*) – Name of funciton in teneto.networkmeasures.
- **calc** (*str, default=None*) – Calc parameter for the function
- **community** (*bool, default=None*) – If not null, then community property is assumed to be believed.

Returns **dimord** – Dimension order. So “node,node,time” would define the dimensions of the network measure.

Return type str

get_network_when

get_network_when(*tnet*, *i=None*, *j=None*, *t=None*, *ij=None*, *logic='and'*, *copy=False*, *asarray=False*, *netshape=None*, *nettype=None*)

Returns subset of dataframe that matches index

Parameters

- **tnet** (*df, array or teneto.TemporalNetwork*) – teneto.TemporalNetwork object or pandas dataframe edgelist
- **i** (*list or int*) – get nodes in column i (source nodes in directed networks)
- **j** (*list or int*) – get nodes in column j (target nodes in directed networks)
- **t** (*list or int*) – get edges at this time-points.
- **ij** (*list or int*) – get nodes for column i or j (logic and can still persist for t). Cannot be specified along with i or j

- **logic** (*str*) – options: 'and' or 'or'. If 'and', functions returns rows that corrspond that match all i,j,t arguments. If 'or', only has to match one of them
- **copy** (*bool*) – default False. If True, returns a copy of the dataframe. Note relevant if hd5 data.
- **asarray** (*bool*) – default False. If True, returns the list of edges as a numpy array.

Returns **df** – Unless asarray are set to true.

Return type pandas dataframe

create_supraadjacency_matrix

create_supraadjacency_matrix(*tnet*, *intersliceweight=1*)

Returns a supraadjacency matrix from a temporal network structure

Parameters

- **tnet** (*teneto.TemporalNetwork*) – Temporal network (any network type)
- **intersliceweight** (*int*) – Weight that links the same node from adjacent time-points

Returns **supranet** – Supraadjacency matrix

Return type dataframe

df_drop_ij_duplicates

df_drop_ij_duplicates(*df*)

tnet_to_nx

tnet_to_nx(*df*, *t=None*)

Creates undirected networkx object

is_jsonable

is_jsonable(*x*)

Check if a dict is jsonable.

Credit: https://stackoverflow.com/a/53112659

CHAPTER 4

Contributers

- William Hedley Thompson
- Peter Fransson
- Vatika Harlalka

4.1 Contribute to teneto?

Found a bug or want to add a feature? Feel free to contribute! Open up an issue on github with a suggestion/fix and then leave a pull request to submit your code.

At the github page you can find suggested enhancements that you can contribute to: https://github.com/wiheto/teneto/issues

Suggestions of other things that need to be added:

- Control theory.
- Weighted shortest paths.
- More network measures.
- More derive_temporalnetwork alternatives.
- Null models.
- Better documentation/tutorials
- More plot alternatives
- Complete HDF5 compatibility
- Freesurfer output in TenetoBIDS
- Implement continous time for all networkmeasures (where possible)

CHAPTER 5

FAQ

5.1 What is the dimension order for dense arrays in *Teneto*?

Inputs/outputs in Teneto can be in both Numpy arrays (time series or temporal works) or Pandas Dataframes (time series). The default dimension order runs from node to time. This means that if you have a temporal network array in Teneto, than the array should have the dimension order *(node,node,time)*. If using time series than the dimension order *(node,time)*. This entails that the nodes are the *rows* in a pandas array and the time-points are the *columns*. Different software can organize their dimension orders differently (e.g. Nilearn uses a time,node dimension order).

CHAPTER 6

Teneto

Temporal **ne**twork **to**ols.

6.1 What is the package

Package includes various tools for analyzing temporal network data. Temporal network measures, temporal network generation, derivation of time-varying/dynamic connectivities, plotting functions.

Some extra focus is placed on neuroimaging data (e.g. compatible with BIDS - *NB: currently not compliant with latest release candidate of BIDS Derivatives*).

6.2 Installation

With pip installed:

```
pip install teneto
```

to upgrade teneto:

```
pip install teneto -U
```

Requires: Python 3.6+

Installing teneto via pip installs all python package requirements as well.

6.3 Documentation

More detailed documentation can be found at teneto.readthedocs.io and includes tutorials.

6.4 Outlook

This package is under active development. And a lot of changes will still be made.

6.5 Contributers

For a list of contributors to teneto, see: teneto.readthedocs.io

6.6 Cite

If using this, please cite us. At present we do not have a dedicated article about teneto, but you can cite the software using the Zenodo DOI and/or the article where teneto is introduced, along with a considerable discussion about many of the measures in teneto:

Thompson et al (2017) "From static to temporal network theory applications to functional brain connectivity." *Network Neuroscience*, 2: 1. p.69-99 Link

Bibliography

[flex-1] Bassett, DS, Wymbs N, Porter MA, Mucha P, Carlson JM, Grafton ST. Dynamic reconfiguration of human brain networks during learning. PNAS, 2011, 108(18):7641-6.

[recruit-1] Danielle S. Bassett, Muzhi Yang, Nicholas F. Wymbs, Scott T. Grafton. Learning-Induced Autonomy of Sensorimotor Systems. Nat Neurosci. 2015 May;18(5):744-51.

[recruit-2] Marcelo Mattar, Michael W. Cole, Sharon Thompson-Schill, Danielle S. Bassett. A Functional Cartography of Cognitive Systems. PLoS Comput Biol. 2015 Dec 2;11(12):e1004533.

[prom-1] Papadopoulos, Lia, et al. "Evolution of network architecture in a granular material under compression." Physical Review E 94.3 (2016): 032908.

[Degree-1] Thompson, et al (2017). From static to temporal network theory: Applications to functional brain connectivity. Network Neuroscience, 1(2), 69-99. [Link]

[Degree-2] Masuda, N., & Lambiotte, R. (2016). A Guidance to Temporal Networks. [Link to book's publisher]

[Close-1] Pan, R. K., & Saramäki, J. (2011). Path lengths, correlations, and centrality in temporal networks. Physical Review E - Statistical, Nonlinear, and Soft Matter Physics, 84(1). [**'Link https://doi.org/10.1103/PhysRevE.84.016105'_**]

[fluct-1] Thompson et al (2017) "From static to temporal network theory applications to functional brain connectivity." Network Neuroscience, 2: 1. p.69-99 [Link]

[sid-1] Fransson et al (2018) Brain network segregation and integration during an epoch-related working memory fMRI experiment. Neuroimage. 178. [Link]

[part-1] Guimera et al (2005) Functional cartography of complex metabolic networks. Nature. 433: 7028, p895-900. [Link]

[topo-1] Tang et al (2010) Small-world behavior in time-varying graphs. Phys. Rev. E 81, 055101(R) [arxiv link]

[topo-2] Nicosia et al (2013) "Graph Metrics for Temporal Networks" In: Holme P., Saramäki J. (eds) Temporal Networks. Understanding Complex Systems. Springer. [arxiv link]

[LV-1] Shinomoto et al (2003) Differences in spiking patterns among cortical neurons. Neural Computation 15.12 [Link]

[LV-2] Followed eq., 4.34 in Masuda N & Lambiotte (2016) A guide to temporal networks. World Scientific. Series on Complex Networks. Vol 4 [Link]

[Bet-1] Tang, J., Musolesi, M., Mascolo, C., Latora, V., & Nicosia, V. (2010). Analysing Information Flows and Key Mediators through Temporal Centrality Metrics Categories and Subject Descriptors. Proceedings of the 3rd Workshop on Social Network Systems. [**'Link https://doi.org/10.1145/1852658.1852661'_**]

Python Module Index

t

Index

L

M

N

P

R

S

T

U

V