Bipartite Configuration Model for Python - Documentation

Release 1.2

Mika J. Straka

Aug 28, 2017

Contents

The Bipartite Configuration Model (BiCM) is a statistical null model for binary bipartite networks *[Squartini2011]*, *[Saracco2015]*. It offers an unbiased method for analyzing node similarities and obtaining statistically validated monopartite projections *[Saracco2017]*.

The BiCM belongs to a series of entropy-based null models for binary bipartite networks, see also

- BiPCM - Bipartite Partial Configuration Model
- BiRG - Bipartite Random Graph

Please consult the original articles for details about the underlying methods and applications to user-movie and international trade databases *[Saracco2017]*, *[Straka2017]*.

An example case is illustrated in the *Tutorial*.

CHAPTER 1

How to cite

If you use the `bicm` module, please cite its location on Github and the original articles *[Saracco2015]* and *[Saracco2017]*.

References

CHAPTER 2

Getting Started

Overview

The `bicm` module is an implementation of the Bipartite Configuration Model (BiCM) as described in the article [Saracco2016]. The BiCM can be used as a statistical null model to analyze the similarity of nodes in undirected bipartite networks. The similarity criterion is based on the number of common neighbors of nodes, which is expressed in terms of Λ-motifs in the original article [Saracco2016]. Subsequently, one can obtain unbiased statistically validated monopartite projections of the original bipartite network.

The construction of the BiCM, just like the related BiPCM and BiRG models, is based on the generation of a grand canonical ensemble of bipartite graphs subject to certain constraints. The constraints can be of different types. For instance, in the case of the BiCM the average degrees of the nodes of the input network are fixed. In the BiRG, on the other hand, the total number of edges is constrained. In general, these models are referred to as entropy-based null models.

The average graph of the ensemble can be calculated analytically using the entropy-maximization principle and provides a statistical null model, which can be used for establishing statistically significant node similarities. For more information and a detailed explanation of the underlying methods, please refer to [Saracco2016].

By using the `bicm` module, the user can obtain the BiCM null model which corresponds to the input matrix representing an undirected bipartite network. To address the question of node similarity, the p-values of the observed numbers of common neighbors (i.e. of the Λ-motifs) can be calculated and used for statistical verification. For an illustration and further details, please refer to [Saracco2016] and [Straka2016].

Dependencies

`bicm` is written in *Python 2.7* and uses the following modules:

- poibin Module for the Poisson Binomial probability distribution
- scipy
- numpy
- multiprocessing

- ctypes
- pytest for unit testing

BiCM Quickstart

The `bicm` module encompasses essentially two steps for the validation of node similarities in bipartite networks:

1. Given a binary input matrix, create the **biadjacency** matrix of the BiCM null model.
2. Calculate the **p-values** of the observed node similarities in the same bipartite layer.

Subsequently, a multiple hypothesis testing of the p-values can be performed. The statistically validated node similarities give rise to an unbiased monopartite projection of the original bipartite network, as illustrated in *[Saracco2017]*.

For more detailed explanations of the methods, please refer to *[Saracco2017]*, the *Tutorial* and the *API*.

Obtaining the biadjacency matrix of the BiCM null model

Be `mat` a two-dimensional binary NumPy array, which describes the biadjacency matrix of an undirected bipartite network. The nodes of the two bipartite layers are ordered along the columns and rows, respectively. In the algorithm, the two layers are identified by the boolean values `True` for the **row-nodes** and `False` for the **column-nodes**.

Import the module and initialize the Bipartite Configuration Model:

```
>>> from src.bicm import BiCM
>>> cm = BiCM(bin_mat=mat)
```

To create the biadjacency matrix of the BiCM, use:

```
>>> cm.make_bicm()
```

Note: Note that `make_bicm` outputs a *status message* in the console, which informs the user whether the underlying numerical solver has converged to a solution. The function is based on the `scipy.optimize.root` routine of the SciPy package to solve a log-likelihood maximization problem and uses thus the same arguments (except for *fun* and *args*, which are specified in our problem). This means that the user has full control over the selection of a solver, the initial conditions, tolerance, etc.

As a matter of fact, it may happen that the default function call `make_bicm()` results in an unsuccessful solver, which requires adjusting the function arguments. In this case, please refer to the more exhaustive note in the *Tutorial*, the description of the function `make_bicm` in the *API*, and the scipy.optimize.root documentation.

The biadjacency matrix of the BiCM null model can be saved in <*filename*>:

```
>>> cm.save_biadjacency(filename=<filename>, delim='\t')
```

By default, the file is saved in a human-readable `.csv` format. The matrix can also be saved as a binary NumPy file `.npy` by using:

```
>>> cm.save_biadjacency(filename=<filename>, binary=True)
```

If the file is not binary, it should end with, e.g., `.csv`. If it is binary instead, NumPy automatically appends the ending `.npy`.

Calculating the p-values of the node similarities

In order to analyze the similarities of the **row-nodes** and to save the p-values of the observed numbers of shared neighbors (i.e. of the Λ-motifs *[Saracco2017]*) in <*filename*>, use:

```
>>> cm.lambda_motifs(True, filename=<filename>)
```

By default, the file is saved as binary NumPy file to reduce disk space, and the format suffix `.npy` is appended. If the file should be saved in a human-readable `.csv` format, use:

```
>>> cm.lambda_motifs(True, filename=<filename>, delim='\t', binary=False)
```

Analogously for the **column-nodes**, use:

```
>>> cm.lambda_motifs(False, filename=<filename>)
```

or:

```
>>> cm.lambda_motifs(False, filename=<filename>, delim='\t', binary=False)
```

Note: The p-values are saved as a one-dimensional array with index $k \in \left[0, \ldots, \binom{N}{2} - 1\right]$ for a bipartite layer of N nodes. Please check the section *A Note on the Output Format* for details regarding the indexing.

Subsequently, the p-values can be used to perform a multiple hypotheses testing of the node similarities and to obtain statistically validated monopartite projections *[Saracco2017]*. The p-values are calculated in parallel by default, see *Parallel Computation and Memory Management* for details.

Tutorial

The tutorial will take you step by step from the biadjacency matrix of a real-data network to the calculation of the p-values. Our example bipartite network will be the following:

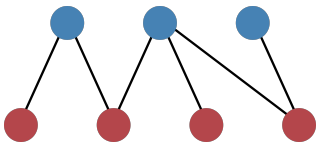

Fig. 2.1: Figure 1: Example network.

The structure of the network can be captured in a biadjacency matrix. In our case, the matrix is

$$\begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Note that the nodes of the top layer are ordered along the rows and the nodes of the bottom layer along columns. In the `bicm` module, the two layers are identified by the boolean values `True` for the **row-nodes** and `False` for the **column-nodes**. In our example image, the row-nodes are colored in blue (`True`, top layer) and the column-nodes in red (`False`, bottom layer).

The `bicm` module encompasses essentially two steps for the validation of node similarities in bipartite networks:

1. Given a binary input matrix capturing the network structure, create the **biadjacency matrix** of the BiCM null model.
2. Calculate the **p-values** of the node similarities for the same bipartite layer.

Subsequently, a multiple hypothesis testing of the p-values can be performed. The statistically validated node similarities give rise to a unbiased monopartite projection of the original bipartite network, as illustrated in *[Saracco2017]*.

For more detailed explanations of the methods, please refer to *[Saracco2017]*.

Getting started

We start by importing the necessary modules:

```
>>> import numpy as np
>>> from src.bicm import BiCM
```

The biadjacency matrix of our network can be captured by the two-dimensional NumPy array `mat`

```
>>> mat = np.array([[1, 1, 0, 0], [0, 1, 1, 1], [0, 0, 0, 1]])
```

and we initialize the Bipartite Configuration Model with:

```
>>> cm = BiCM(bin_mat=mat)
```

Biadjacency matrix

In order to obtain the individual link probabilities between the top and the bottom layer, which are captured by the biadjacency matrix of the BiCM null model, a log-likelihood maximization problem has to be solved, as described in *[Saracco2017]*. This can be done automatically by running

```
>>> cm.make_bicm()
```

In our example graph, the BiCM biadjacency matrix should be:

```
>>> print cm.adj_matrix
[[ 0.29433408  0.70566592  0.29433408  0.70566592]
 [ 0.60283296  0.89716704  0.60283296  0.89716704]
 [ 0.10283296  0.39716704  0.10283296  0.39716704]]
```

Note that `make_bicm` outputs a status message in the console of the form:

```
Solver successful: True
The solution converged.
```

The message informs the user whether the underlying numerical solver has successfully converged to a solution and prints its status message.

We can check the maximal degree difference between the input network and the BiCM model by running:

```
>>> cm.print_max_degree_differences()
```

Note: The function `make_bicm` uses the `scipy.optimize.root` routine of the SciPy package to solve the maximization problem. It accepts the same arguments as `scipy.optimize.root` except for *fun* and *args*, which

are specified in our problem. This means that the user has full control over the selection of a solver, the initial conditions, tolerance, etc.

As a matter of fact, in some situations it may happen that the function call `make_bicm()`, which uses default arguments, does not converge to a solution. In that case, the console will report *Solver successful: False* together with the status message returned by the numerical solver.

If this happens, the user should try different solvers, such as least-squares and/or different initial conditions or tolerance values.

Please consult the scipy.optimize.root documentation with a list of possible solvers and the description of the function `make_bicm` in the *API*.

After having successfully obtained the biadjacency matrix, we could save it in the file <*filename*> with

```
>>> cm.save_biadjacency(filename=<filename>, delim='\t')
```

The default delimiter is `\t` and does not have to be specified in the line above. Other delimiters such as `,` or `;` work fine as well. The matrix can either be saved as a human-readable `.csv` or as a binary NumPy `.npy` file, see `save_biadjacency()` in the *API*. In the latter case, we would run:

```
>>> cm.save_biadjacency(filename=<filename>, binary=True)
```

If `binary == True`, the file ending `.npy` is appended automatically.

P-values

Each entry in the biadjacency matrix of the null model corresponds to the probability of observing a link between the corresponding row- and column-nodes. If we take two nodes of the same layer, we can count the number of common neighbors that they share in the original input network and calculate the probability of observing the same or more common neighbors according to the BiCM *[Saracco2017]*. This corresponds to calculating the p-values for a right-sided hypothesis testing.

The number of common neighbors can be described in terms of Λ-motifs *[Saracco2017]*, as shown in Figure 2.

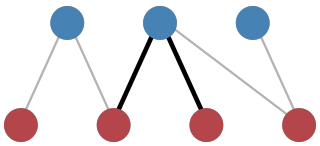

Fig. 2.2: Figure 2: Illustration of a Λ-motif between the two central red nodes.

The calculation of the p-values is computationally intensive and should be performed in parallel, see *Parallel Computation and Memory Management* for details. It can be executed by simply running

```
>>> cm.lambda_motifs(<bool>, filename=<filename>)
```

where `<bool>` is either `True` of `False` depending on whether we want to address the similarities of the **row-** or **column-nodes**, respectively. The results are written to <*filename*>. By default, the file is a binary NumPy file to reduce disk space, and the format suffix `.npy` is appended. If the file should be saved in a human-readable `.csv` format, we need to provide an appropriate name ending with `.csv` and use:

```
>>> cm.lambda_motifs(<bool>, filename=<filename>, delim='\t', binary=False)
```

Again the default delimiter is `\t`.

Note: The p-values are saved as a one-dimensional array with index $k \in \left[0, \ldots, \binom{N}{2} - 1\right]$ for a bipartite layer of N nodes. Please check the section *A Note on the Output Format* for details regarding the indexing.

Having calculated the p-values, it is possible to perform a multiple hypothesis testing of the node similarities and to obtain an unbiased monopartite projection of the original bipartite network. In the projection, only statistically significant edges are kept.

For further information on the post-processing and the monopartite projections, please refer to *[Saracco2017]*.

Testing

The methods in the `bicm` module have been implemented using doctests. To run the tests, execute:

```
>>> python -m doctest bicm_tests.txt
```

from the folder `src` in the command line. If you want to run the tests in verbose mode, use:

```
>>> python -m doctest -v bicm_tests.txt
```

Note that `bicm.py` and `bicm_tests.txt` have to be in the same directory to run the test.

Parallel Computation and Memory Management

The calculation of the p-values of the Λ-motifs demands computation power as well as working memory which grow quickly with the number of nodes. To address this problem, the calculation is split into chunks and the Python multiprocessing package is used.

Parallel Computation

By default the computation is performed in parallel using the multiprocessing packge. The number of parallel processes depends on the number of CPUs of the work station and is defined by the variable `num_procs` in the method `BiCM.get_pvalues_q()` (see *API*).

If the calculation should **not** be performed in parallel, use:

```
>>> cm.lambda_motifs(<bool>, parallel=False)
```

instead of:

```
>>> cm.lambda_motifs(<bool>)
```

Memory Management

In order to calculate the p-values, information on the Λ-motifs and their probabilities has to be kept in the working memory. This can lead to memory allocation errors for large networks due to limited resources. To avoid this problem, the total number of p-value operations is split into chunks which are processed sequentially. The number of chunks can be defined when calling the motif function:

```
>>> cm.lambda_motifs(<bool>, num_chunks=<number_of_chunks>)
```

The default value is `num_chunks = 4`. As an example, for N nodes in a bipartite layer, $\binom{N}{2}$ p-values have to be computed. If `num_chunks = 4`, each chunk processes therefore about $\frac{1}{4}\binom{N}{2}$ p-values in parallel.

Note: Increasing `num_chunks` decreases the required working memory, but leads to longer processing times. For $N \leq 15$, `num_chunks = 1` by default.

A Note on the Output Format

A layer composed of N vertices requires the computation of $\binom{N}{2}$ p-values to assess the node similarities. They are saved in the output file as a one-dimensional array when running the method *:func:'BiCM.get_pvalues_q''cm.lambda_motifs'*. However, a more intuitive matrix representation can be easily recovered using some helper functions, since each element in the array can be mapped on an element in the matrix and vice versa.

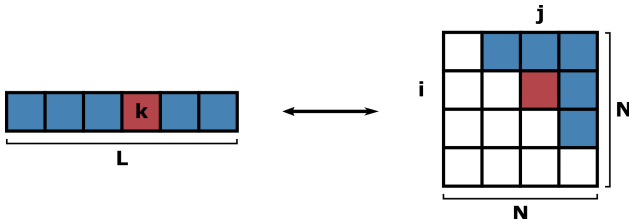

Fig. 2.3: Figure 1: Mapping of the one-dimensional array of length L onto a square matrix of dimension $N \times N$. Note that the matrix is symmetric.

Let's consider an array of length L as shown in Figure 1. The dimension N of the matrix, i.e. the number of nodes in the original bipartite network layer, can be obtained as:

```
>>> N = cm.flat2triumat_dim(L)
```

To convert an array index `k` to the corresponding matrix index couple `(i, j)` in the upper triangular part, use:

```
>>> (i, j) = cm.flat2triumat_idx(k, N)
```

Note: As illustrated in Figure 1, the output array contains the **upper triangular part** of the symmetric square matrix, excluding the diagonal. Thus

- $k \in [0, \ldots, N(N-1)/2 - 1]$
- $i \in [0, \ldots, N-1]$
- $j \in [i+1, \ldots, N-1]$

The inverse operations are given by:

```
>>> L = cm.triumat2flat_dim(N)
>>> k = cm.triumat2flat_idx(i, j, N)
```

API

API for the methods in the `bicm` module.

class `bicm.`**`BiCM`**(*bin_mat*)
Bases: `object`

Bipartite Configuration Model for undirected binary bipartite networks.

This class implements the Bipartite Configuration Model (BiCM), which can be used as a null model for the analysis of undirected and binary bipartite networks. The class provides methods for calculating the biadjacency matrix of the null model and for quantifying node similarities in terms of p-values.

`add2inqueue`(*nprocs*, *plam_mat*, *nlam_mat*, *k1*, *k2*)
Add elements to the in-queue to calculate the p-values.

Parameters

- **`nprocs`** (*int*) – number of processes running in parallel
- **`plam_mat`** (*numpy.array (square matrix)*) – array containing the list of probabilities for the single observations of Λ-motifs
- **`nlam_mat`** (*numpy.array (square matrix)*) – array containing the observations of Λ-motifs
- **`k1`** (*int*) – lower interval limit
- **`k2`** (*int*) – upper interval limit

`check_input_matrix_is_binary`()
Check that the input matrix is binary, i.e. entries are 0 or 1.

Raises **`AssertionError`** – raise an error if the input matrix is not binary

`equations`(*xx*)
Return the equations of the log-likelihood maximization problem.

Note that the equations for the row-nodes depend only on the column-nodes and vice versa, see *[Saracco2015]*.

Parameters **`xx`** (*numpy.array*) – Lagrange multipliers which have to be solved

Returns equations to be solved ($f(x) = 0$)

Return type numpy.array

static **`flat2triumat_dim`**(*k*)
Return the dimension of the matrix hosting `k` triangular elements.

Parameters **`k`** (*int*) – the number of elements in the upper triangular part of the corresponding square matrix, excluding the diagonal

Returns dimension of the corresponding square matrix

Return type int

static **`flat2triumat_idx`**(*k*, *n*)
Convert an array index into the index couple of a triangular matrix.

`k` is the index of an array of length $\binom{n}{2}2$, which contains the elements of an upper triangular matrix of dimension `n` excluding the diagonal. The function returns the index couple (i, j) that corresponds to the entry `k` of the flat array.

Note:

•$k \in \left[0, \ldots, \binom{n}{2} - 1\right]$

•**returned indices:**

– $i \in [0, \ldots, n-1]$

– $j \in [i+1, \ldots, n-1]$

Parameters

- **k** (*int*) – flattened array index
- **n** (*int*) – dimension of the square matrix

Returns matrix index tuple (row, column)

Return type tuple

get_biadjacency_matrix(*xx*)

Calculate the biadjacency matrix of the null model.

The biadjacency matrix describes the BiCM null model, i.e. the optimal average graph $< G >^*$ with the average link probabilities $< G >^*_{rc} = p_{rc}$, $p_{rc} = \frac{x_r \cdot x_c}{1 + x_r \cdot x_c}$. x are the solutions of the equation system which has to be solved for the null model. Note that r and c are taken from opposite bipartite node sets, thus $r \neq c$.

Parameters **xx** (*numpy.array*) – solutions of the equation system (Lagrange multipliers)

Returns biadjacency matrix of the null model

Return type numpy.array

Raises **ValueError** – raise an error if $p_{rc} < 0$ or $p_{rc} > 1$ for any r, c

get_lambda_motif_block(*mm*, *k1*, *k2*)

Return a subset of Λ-motifs as observed in mm.

Given the binary input matrix mm, count the number of Λ-motifs for all the node couples specified by the interval $[k_1, k_2[$.

Note:

•The Λ-motifs are counted between the **row-nodes** of the input matrix mm.

•If $k_2 \equiv \binom{mm.shape[0]}{2}$, the interval becomes $[k_1, k_2]$.

Parameters

- **mm** (*numpy.array*) – binary matrix
- **k1** (*int*) – lower interval limit
- **k2** (*int*) – upper interval limit

Returns array of observed Λ-motifs

Return type numpy.array

get_plambda_block(*biad_mat*, *k1*, *k2*)
Return a subset of the Λ probability matrix.

Given the biadjacency matrix `biad_mat` with $\mathbf{M}_{rc} = p_{rc}$, which describes the probabilities of row-node `r` and column-node `c` being linked, the method returns the matrix

$P(\Lambda)_{ij} = (M_{i\alpha_1} \cdot M_{j\alpha_1}, M_{i\alpha_2} \cdot M_{j\alpha_2}, \ldots)$,

for all the node couples in the interval $[k_1, k_2[$. (i, j) are two **row-nodes** of `biad_mat` and α_k runs over the nodes in the opposite layer.

Note:

•The probabilities are calculated between the **row-nodes** of the input matrix `biad_mat`.

•If $k_2 \equiv \binom{biad\_mat.shape[0]}{2}$, the interval becomes $[k1, k2]$.

Parameters

- **biad_mat** (*numpy.array*) – biadjacency matrix
- **k1** (*int*) – lower interval limit
- **k2** (*int*) – upper interval limit

Returns Λ-motif probability matrix

Return type numpy.array

get_pvalues_q(*plam_mat*, *nlam_mat*, *k1*, *k2*, *parallel=True*)
Calculate the p-values of the observed Λ-motifs.

For each number of Λ-motifs in `nlam_mat` for the node interval $[k1, k2[$, construct the Poisson Binomial distribution using the corresponding probabilities in `plam_mat` and calculate the p-value.

Parameters

- **plam_mat** (*numpy.array (square matrix)*) – array containing the list of probabilities for the single observations of Λ-motifs
- **nlam_mat** (*numpy.array (square matrix)*) – array containing the observations of Λ-motifs
- **k1** (*int*) – lower interval limit
- **k2** (*int*) – upper interval limit
- **parallel** (*bool*) – if `True`, the calculation is executed in parallel; if `False`, only one process is started

get_triup_dim(*bip_set*)
Return the number of possible node couples in `bip_set`.

Parameters **bip_set** (*bool*) – selects row-nodes (`True`) or column-nodes (`False`)

Returns return the number of node couple combinations corresponding to the layer `bip_set`

Return type int

Raises **ValueError** – raise an error if the parameter `bip_set` is neither `True` nor `False`

jacobian(*xx*)
Return a NumPy array with the Jacobian of the equation system.

Parameters **xx** (*numpy.array*) – Lagrange multipliers which have to be solved

Returns Jacobian

Return type numpy.array

lambda_motifs(*bip_set*, *parallel=True*, *filename=None*, *delim='\t'*, *binary=True*, *num_chunks=4*)

Calculate and save the p-values of the Λ-motifs.

For each node couple in the bipartite layer specified by `bip_set`, calculate the p-values of the corresponding Λ-motifs according to the link probabilities in the biadjacency matrix of the BiCM null model.

The results can be saved either as a binary `.npy` or a human-readable `.csv` file, depending on `binary`.

Note:

- The total number of p-values that are calculated is split into `num_chunks` chunks, which are processed sequentially in order to avoid memory allocation errors. Note that a larger value of `num_chunks` will lead to less memory occupation, but comes at the cost of slower processing speed.
- The output consists of a one-dimensional array of p-values. If the bipartite layer `bip_set` contains `n` nodes, this means that the array will contain $\binom{n}{2}$ entries. The indices `(i, j)` of the nodes corresponding to entry `k` in the array can be reconstructed using the method `BiCM.flat2_triumat_idx()`. The number of nodes `n` can be recovered from the length of the array with `BiCM.flat2_triumat_dim()`
- If `binary == False`, the `filename` should end with `.csv`. If `binary == True`, it will be saved in binary NumPy `.npy` format and the suffix `.npy` will be appended automatically. By default, the file is saved in binary format.

Parameters

- **bip_set** (*bool*) – select row-nodes (`True`) or column-nodes (`False`)
- **parallel** (*bool*) – select whether the calculation of the p-values should be run in parallel (`True`) or not (`False`)
- **filename** (*str*) – name of the output file
- **delim** (*str*) – delimiter between entries in the `.csv``file, default is ``\t`
- **binary** (*bool*) – if `True`, the file will be saved in the binary NumPy format `.npy`, otherwise as `.csv`
- **num_chunks** (*int*) – number of chunks of p-value calculations that are performed sequentially

Raises **ValueError** – raise an error if the parameter `bip_set` is neither `True` nor `False`

make_bicm(*x0=None*, *method='hybr'*, *jac=None*, *tol=None*, *callback=None*, *options=None*)

Create the biadjacency matrix of the BiCM null model.

Solve the log-likelihood maximization problem to obtain the BiCM null model which respects constraints on the degree sequence of the input matrix.

The problem is solved using `scipy`'s root function with the solver defined by `method`. The status of the solver after running ``scipy.root``and the difference between the network and BiCM degrees are printed in the console.

The default solver is the modified Powell method `hybr`. Least-squares can be chosen with `method='lm'` for the Levenberg-Marquardt approach.

Depending on the solver, keyword arguments `kwargs` can be passed to the solver. Please refer to the scipy.optimize.root documentation for detailed descriptions.

Note: It can happen that the solver `method` used by `scipy.root` does not converge to a solution. In this case, please try another `method` or different initial conditions and refer to the scipy.optimize.root documentation.

Parameters

- **x0** (*1d numpy.array, optional*) – initial guesses for the solutions. The first entries are the initial guesses for the row-nodes, followed by the initial guesses for the column-nodes.
- **method** (*str, optional*) – type of solver, default is 'hybr'. For other solvers, see the scipy.optimize.root documentation.
- **jac** (*bool or callable, optional*) – Jacobian of the system
- **tol** (*float, optional*) – tolerance for termination. For detailed control, use solver-specific options.
- **callback** (*function, optional*) – optional callback function to be called at every iteration as `callback(self.equations, x)`, see `scipy.root` documentation
- **options** (*dict, optional*) – a dictionary of solver options, e.g. `xtol` or `maxiter`, see scipy.root documentation
- **kwargs** – solver-specific options, please refer to the SciPy documentation

Raises **ValueError** – raise an error if not enough initial conditions are provided

outqueue2pval_mat(*nprocs*, *pvalmat*)
Put the results from the out-queue into the p-value array.

print_max_degree_differences()
Print the maximal differences between input network and BiCM degrees.

Check that the degree sequence of the solved BiCM null model graph corresponds to the degree sequence of the input graph.

pval_process_worker()
Calculate p-values and add them to the out-queue.

static save_array(*mat*, *filename*, *delim='\t'*, *binary=False*)
Save the array `mat` in the file `filename`.

The array can either be saved as a binary NumPy `.npy` file or as a human-readable `.npy` file.

Note:

•The relative path has to be provided in the filename, e.g. *../data/pvalue_matrix.csv*.

•If `binary==True`, NumPy automatically appends the format ending `.npy` to the file.

Parameters

- **mat** (*numpy.array*) – array
- **filename** (*str*) – name of the output file

- **delim** (*str*) – delimiter between values in file
- **binary** (*bool*) – if `True`, save as binary `.npy`, otherwise as a `.csv` file

save_biadjacency (*filename*, *delim='\t'*, *binary=False*)
Save the biadjacendy matrix of the BiCM null model.

The matrix can either be saved as a binary NumPy `.npy` file or as a human-readable `.csv` file.

Note:

•The relative path has to be provided in the filename, e.g. *../data/pvalue_matrix.csv*.

•If `binary==True`, NumPy automatically appends the format ending `.npy` to the file.

Parameters

- **filename** (*str*) – name of the output file
- **delim** (*str*) – delimiter between values in file
- **binary** (*bool*) – if `True`, save as binary `.npy`, otherwise as a `.csv` file

set_degree_seq()
Return the node degree sequence of the input matrix.

Returns node degree sequence [degrees row-nodes, degrees column-nodes]

Return type numpy.array

Raises **AssertionError** – raise an error if the length of the returned degree sequence does not correspond to the total number of nodes

solve_equations (*x0=None*, *method='hybr'*, *jac=None*, *tol=None*, *callback=None*, *options=None*)
Solve the system of equations of the maximum log-likelihood problem.

The system of equations is solved using `scipy`'s root function with the solver defined by `method`. The solutions correspond to the Lagrange multipliers

$$x_i = \exp(-\theta_i).$$

Depending on the solver, keyword arguments `kwargs` can be passed to the solver. Please refer to the scipy.optimize.root documentation for detailed descriptions.

The default solver is the modified Powell method `hybr`. Least-squares can be chosen with `method='lm'` for the Levenberg-Marquardt approach.

Note: It can happen that the solver `method` used by `scipy.root` does not converge to a solution. In this case, please try another `method` or different initial conditions and refer to the scipy.optimize.root documentation.

Parameters

- **x0** (*1d numpy.array, optional*) – initial guesses for the solutions. The first entries are the initial guesses for the row-nodes, followed by the initial guesses for the column-nodes.
- **method** (*str, optional*) – type of solver, default is 'hybr'. For other solvers, see the scipy.optimize.root documentation.

- **jac** (*bool or callable, optional*) – Jacobian of the system
- **tol** (*float, optional*) – tolerance for termination. For detailed control, use solver-specific options.
- **callback** (*function, optional*) – optional callback function to be called at every iteration as `callback(self.equations, x)`, see `scipy.root` documentation
- **options** (*dict, optional*) – a dictionary of solver options, e.g. `xtol` or `maxiter`, see scipy.root documentation
- **kwargs** – solver-specific options, please refer to the SciPy documentation

Returns solution of the equation system

Return type scipy.optimize.OptimizeResult

Raises **ValueError** – raise an error if not enough initial conditions are provided

split_range(*n*, *m=4*)

Split the interval $[0, \ldots, n]$ in `m` parts.

Parameters

- **n** (*int*) – upper limit of the range
- **m** – number of part in which range should be split

Returns delimiter indices for the `m` parts

Return type list

test_average_degrees(*eps=0.01*)

Test the constraints on the node degrees.

Check that the degree sequence of the solved BiCM null model graph corresponds to the degree sequence of the input graph.

Parameters **eps** (*float*) – maximum difference between degrees of the real network and the BiCM

static **triumat2flat_dim**(*n*)

Return the size of the triangular part of a `n x n` matrix.

Parameters **n** (*int*) – the dimension of the square matrix

Returns number of elements in the upper triangular part of the matrix (excluding the diagonal)

Return type int

static **triumat2flat_idx**(*i*, *j*, *n*)

Convert an matrix index couple to a flattened array index.

Given a square matrix of dimension `n` and the index couple `(i, j)` *of the upper triangular part* of the matrix, return the index which the matrix element would have in a flattened array.

Note:

•$i \in [0, ..., n-1]$

•$j \in [i+1, ..., n-1]$

•returned index $\in [0,\ n(n-1)/2-1]$

Parameters

- **i** (*int*) – row index
- **j** (*int*) – column index
- **n** (*int*) – dimension of the square matrix

Returns flattened array index

Return type int

License

Contact

For questions or input, please write to mika.straka [at] imtlucca.it.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Bibliography

[Saracco2015] F. Saracco, R. Di Clemente, A. Gabrielli, T. Squartini, Randomizing bipartite networks: the case of the World Trade Web, Scientific Reports 5, 10595 (2015)

[Saracco2017] F. Saracco, M. J. Straka, R. Di Clemente, A. Gabrielli, G. Caldarelli, and T. Squartini, Inferring monopartite projections of bipartite networks: an entropy-based approach, New J. Phys. 19, 053022 (2017)

[Squartini2011] T. Squartini, D. Garlaschelli, Analytical maximum-likelihood method to detect patterns in real networks, New Journal of Physics 13, (2011)

[Straka2017] M. J. Straka, G. Caldarelli, F. Saracco, Grand canonical validation of the bipartite international trade network, Phys. Rev. E 96, 022306 (2017)

Index