Bipartite Configuration Model Documentation

Release 1.0

Mika J. Straka

January 23, 2017

CONTENTS

The Bipartite Configuration Model (BiCM) is a statistical null model for binary bipartite networks [Squartini2011] [Saracco2015]. It offers an unbiased method of analyzing node similarities and obtaining statistically validated monopartite projections [Saracco2016].

The BiCM belongs to a series of entropy-based null model 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 [Saracco2016], [Straka2016].

An example case is illustrated in the *Tutorial*.

CHAPTER

ONE

HOW TO CITE

If you use the `bicm` module, please cite its location on Github and the original articles [Saracco2015] and [Saracco2016].

1.1 References

CHAPTER

TWO

GETTING STARTED

2.1 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.

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. In general, they are referred to as entropy-based null models. 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 can be calculated and used for statistical verification. For an illustration and further details, please refer to [Saracco2016] and [Straka2016].

2.1.1 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
- doctest For unit testing

2.2 BiCM Quickstart

If you want to get started right away, go ahead and follow the summary below. The `bicm` module encompasses essentially two steps for the analysis of node similarities in bipartite networks:

1. given an input matrix, create the biadjacency matrix of the BiCM null model
2. perform a statistical validation of the similarities of nodes in the same layer

The validated node similarities can be used to obtain an unbiased monopartite projection of the bipartite network, as illustrated in [Saracco2016].

For more detailed explanations of the methods, please refer to [Saracco2016], the *Tutorial* and the *API*.

2.2.1 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()
```

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 information can also be saved as a binary NumPy file `.npy` by using:

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

The names of the files should reflect the format, i.e. end with `.csv` or `.npy`.

2.2.2 Calculating the p-values of the node similarities

In order to analyze the similarity of the row-layer nodes and to save the p-values of the corresponding Λ-motifs, i.e. of the number of shared neighbors [Saracco2016], use:

```
>>> cm.lambda_motifs(True, filename='p_values_True.csv', delim='\t')
```

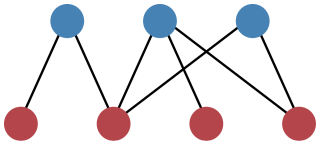
For the column-layer nodes, use:

```
>>> cm.lambda_motifs(False, filename='p_values_False.csv', delim='\t')
```

Subsequently, the p-values can be used to perform a multiple hypotheses testing and to obtain statistically validated monopartite projections [Saracco2016]. The p-values are calculated in parallel by default, see *Parallel Computation* for details.

2.3 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:



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

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

Note that the nodes of the layers of the bipartite network are ordered along the rows and the columns, respectively. In the algorithms, 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 (top layer) and the column-nodes in red (bottom layer).

Let's get started by importing the necessary modules:

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

The biadjacency matrix of our toy network will be saved in the two-dimensional NumPy array `mat`:

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

and we initialize the Bipartite Configuration Model with:

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

In order to obtain the biadjacency matrix of the BiCM null model corresponding to the input network, a number of equations have to be solved. However, this is done automatically by running:

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

You can now save the bidajacency matrix in the file *<filename>* as:

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

Note that the default delimiter is `\t`. 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 our example graph, the BiCM matrix should be:

```
>>> cm.adj_matrix
array([[ 0.21602144,  0.99855239,  0.21602144,  0.56873952],
       [ 0.56845256,  0.99969684,  0.56845256,  0.86309703],
       [ 0.21602144,  0.99855239,  0.21602144,  0.56873952]])
```

Each entry in the matrix corresponds to the probability of observing a link between the corresponding row- and column-nodes. If we take two nodes in 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 of more common neighbors according to the BiCM [Saracco2016]. This corresponds to calculating the p-values for a right-sided hypothesis testing.

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

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

where <bool> is either True of False depending on whether one wants to address the similarities of the **row-** or **column-nodes**, respectively, and <filename> is the name of the output file.

Having calculated the p-values, it is possible to perform a multiple hypothesis testing with FDR control 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 [Saracco2016].

2.4 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.

2.5 Parallel Computation

Since the calculation of the p-values is computationally demanding, the bicm module uses the Python multiprocessing package by default for this purpose. The number of parallel processes depends on the number of CPUs of the work station (see variable numprocs in the method BiCM.get_pvalues_q() in the *API*).

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

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

instead of:

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

2.6 API

API for the methods in the bicm module.

class bicm.**BiCM**(*bin_mat*)

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 to calculate the biadjacency matrix of the null model and to quantify node similarities in terms of p-values.

add2inqueue(*nprocs*, *plam_mat*, *nlam_mat*)

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

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 reference mentioned in the header.

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

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

Return type numpy.array

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 $< G >^*_{rc} = p_{rc}$, $p_{rc} = \frac{x_r \cdot x_c}{1.0 + x_r \cdot x_c}$. 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

static get_lambda_motif_matrix(*mm*, *bip_set*)
Return the number of Λ-motifs as found in `mm`.

Given the binary input matrix `mm`, count the number of Λ-motifs between node couples of the bipartite layer specified by `bip_set`.

Parameters

- **mm** (*numpy.array*) – binary matrix
- **bip_set** (*bool*) – selects row-nodes (`True`) or column-nodes (`False`)

Returns square matrix of observed Λ-motifs

Return type numpy.array

Raises NameError raise an error if the parameter `bip_set` is neither `True` nor `False`

static get_plambda_matrix(*biad_mat*, *bip_set*)
Return the Λ-motif probability tensor for `bip_set`.

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

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

where (i, j) are two nodes of the bipartite layer `bip_set` and α_k runs over the nodes in the opposite layer.

Parameters

- **biad_mat** (*numpy.array*) – biadjacency matrix
- **bip_set** (*bool*) – selects row-nodes (`True`) or column-nodes (`False`)

Returns Λ-motif probability tensor for `bip_set`

Return type numpy.array

get_pvalues_q(*plam_mat*, *nlam_mat*, *parallel=True*)

Calculate the p-values of the observed Λ-motifs.

For each number of Λ-motifs in `nlam_mat`, construct the Poisson Binomial distribution using the corresponding probabilities in `plam_mat` and calculate the p-value.

Note:

•the p-values are saved in the matrix `self.pval_mat`.

•the lower-triangular part of the output matrix is null since the matrix is symmetric by definition.

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
- **parallel** (*bool*) – if `True`, the calculation is executed in parallel; if `False`, only one process is started

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'*)

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

For each node couple in the bipartite layer specified by `bip_set`, Λ-motifs and calculate the corresponding p-value.

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 file which will contain the p-values
- **delim** (*str*) – delimiter between entries in file, default is tab

make_bicm()

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.

Raises AssertionError raise an error if the adjacency matrix of the null model has different dimensions than the input matrix

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

pval_process_worker()
Calculate one p-value and add the result to the out-queue.

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/biadjacency_matrix.csv*

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 CSV a file

static save_matrix(*mat*, *filename*, *delim='t'*, *binary=False*)
Save the matrix `mat` in the file `filename`.

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*

Parameters

- **mat** (*numpy.array*) – two-dimensional matrix
- **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(*eq*, *jac*)
Solve the system of equations of the maximum log-likelihood problem.

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

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

Parameters

- **eq** (*numpy.array*) – system of equations ($f(x) = 0$)

- **jac** (*numpy.ndarray*) – Jacobian of the system

Returns solution of the equation system

test_average_degrees()
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.

static triumat2flat_idx(i, j, n)
Convert an matrix index couple to a flattened array index.

Given a square matrix of dimension n and an index couple (i, j) *of the upper triangular part* of the matrix, the function returns 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

2.7 License

2.8 Contact

For questions or input, please write to mika.straka@imtlucca.it.

CHAPTER

THREE

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

[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)

[Saracco2016] F. Saracco, M. J. Straka, R. Di Clemente, A. Gabrielli, G. Caldarelli, T. Squartini, Inferring monopartite projections of bipartite networks: an entropy-based approach, arXiv preprint arXiv:1607.02481

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

[Straka2016] M. J. Straka, F. Saracco, G. Caldarelli, Product Similarities in International Trade from Entropy-based Null Models, Complex Networks 2016, 130-132 (11 2016), ISBN 978-2-9557050-1-8

A

B

C

E

G

J

L

M

O

P

S

T