**Fit a weighted configuration model to a bipartite network using the code Fit\_WBiCM.py.**

The key reference for the Python code **Fit\_WBiCM.py** is the published Python package NEMtropy which is fully explained and tested at

<https://pypi.org/project/NEMtropy/>

For all the details check:

https://meh.imtlucca.it/codes/nemtropy

Before running the script **Fit\_WBiCM.py** follows all the instructions given at

<https://pypi.org/project/NEMtropy/>

to run the basic routines of NEMtropy. All the information on the Python version and dependencies necessary to run bicm2.0 are provided at that link, including basic testing of the routines.

The script **Fit\_WBiCM.py** implements a specific application of NEMtropy, that is an application of the CReM model as in Parisi et al. 2020

https://iopscience.iop.org/article/10.1088/1367-2630/ab74a7

The script first calls the Python packages “pandas”, “os”, “network”, and ‘numpy’ and ‘NEMtropy’ with the starting lines (below, script lines in Italics):

*import pandas as pd*

*import numpy as np*

*import os*

*import networkx as nx*

*from NEMtropy import DirectedGraph*

*from NEMtropy.network\_functions import build\_adjacency\_from\_edgelist*

The script then defines a Python class for a matrix initialization to separate information on the degree and strength sequence using the lines

*class weighted\_bipartite():*

*def \_\_init\_\_(self, matrix,Degree\_preserving\_randomization=False):*

*self.matrix = matrix*

*self.n\_rows = np.shape(self.matrix)[0]*

*self.n\_cols = np.shape(self.matrix)[1]*

*self.full\_matrix = self.\_\_inizialize\_full\_matrix()*

*self.randomize\_degree = Degree\_preserving\_randomization*

*self.directed\_graph = self.\_\_inizialize\_directed\_graph()*

*def \_\_inizialize\_full\_matrix(self):*

*zeros\_1 = np.zeros((self.n\_rows, self.n\_rows))*

*zeros\_2 = np.zeros((self.n\_cols, self.n\_cols))*

*zeros\_3 = np.zeros((self.n\_cols, self.n\_rows))*

*full\_matrix\_1 = np.concatenate((zeros\_1, zeros\_3))*

*full\_matrix\_2 = np.concatenate((self.matrix, zeros\_2))*

*full\_matrix = np.concatenate((full\_matrix\_1, full\_matrix\_2), axis=1)*

*return full\_matrix*

*def \_\_inizialize\_directed\_graph(self):*

*self.out\_degree = self.full\_matrix.astype(bool).astype(float).sum(axis=1)*

*self.in\_degree = self.full\_matrix.astype(bool).astype(float).sum(axis=0)*

*self.out\_strength = self.full\_matrix.sum(axis=1)*

*self.in\_strength = self.full\_matrix.sum(axis=0)*

*self.topology = self.full\_matrix.astype(bool).astype(float)*

*if self.randomize\_degree:*

*return DirectedGraph(degree\_sequence=np.concatenate([self.out\_degree, self.in\_degree]),strength\_sequence=np.concatenate([self.out\_strength, self.in\_strength]))*

*else:*

*return DirectedGraph(strength\_sequence=np.concatenate([self.out\_strength, self.in\_strength]))*

This approach allows multiple implementations. In particular, one can either randomise or not the degree sequence before randomising the strength sequence.

*def solve(self):*

*if self.randomize\_degree:*

*self.directed\_graph.solve\_tool(model="crema",*

*method="newton",*

*initial\_guess="random",*

*adjacency="dcm\_exp",*

*method\_adjacency="newton")*

*else:*

*self.directed\_graph.solve\_tool(model="crema",*

*method="newton",*

*initial\_guess="random",*

*adjacency=self.topology)*

and the script implements the case in which both degree sequences and strength sequence are randomised (but this can be changed: see below).

Finally, the script generates a routine to sample the ensemble with the lines:

*def sampler(self,n\_samples,folder\_name = 'samples'):*

*if self.randomize\_degree:*

*try:*

*os.mkdir(folder\_name)*

*except:*

*print('\033[91m'+'The folder',folder\_name, 'exists. Before running the sampler, change the folder name or delete the existing folder.'+'\033[0m')*

*return*

*for ii in range(n\_samples):*

*self.directed\_graph.ensemble\_sampler(1, cpu\_n=2, output\_dir="sample/")*

*edgelist\_ens = np.loadtxt("sample/0.txt")*

*# then we can build the adjacency matrix*

*ens\_adj = build\_adjacency\_from\_edgelist(edgelist=edgelist\_ens,*

*is\_directed=True,*

*is\_sparse=False,*

*is\_weighted=True)*

*adj\_matrix = np.zeros((self.n\_rows, self.n\_cols))*

*for i,j,w in edgelist\_ens:*

*adj\_matrix[int(i),int(j)-self.n\_rows] = w*

*np.savetxt('samples/sample\_' + repr(ii) + '.csv', adj\_matrix, fmt='%s', delimiter=',')*

*else:*

*try:*

*os.mkdir(folder\_name)*

*except:*

*print('\033[91m'+'The folder', folder\_name, 'exists. Before running the sampler, change the folder name or delete the existing folder.'+'\033[0m')*

*return*

*for i in range(n\_samples):*

*# print(i)*

*self.directed\_graph.ensemble\_sampler(1, cpu\_n=2, output\_dir="sample/")*

*edgelist\_ens = np.loadtxt("sample/0.txt")*

*# then we can build the adjacency matrix*

*ens\_adj = build\_adjacency\_from\_edgelist(edgelist=edgelist\_ens,*

*is\_directed=True,*

*is\_sparse=False,*

*is\_weighted=True)*

*G = nx.from\_numpy\_array(ens\_adj)*

*adj\_matrix = nx.adjacency\_matrix(G).A*

*np.savetxt('samples/sample\_' + repr(i) + '.csv', adj\_matrix[:self.n\_rows, self.n\_rows:], fmt='%s', delimiter=',')*

The last five lines of the script allows:

loading the particular dataset to which to fit the model (note that the current lines will make the script upload the matrix *'bicm\_matW.csv'* in the data repository)

*weighted\_matrix = np.loadtxt('bicm\_matW.csv', delimiter=',',dtype=str)*

*weighted\_matrix = weighted\_matrix[1:, 1:].astype(float)*

setting up whether degree sequencing must be preserved or not

*fitWeightedBi = weighted\_bipartite(weighted\_matrix,Degree\_preserving\_randomization= True)*

executing the model fitting routine

*fitWeightedBi.solve()*

and sampling the desired number of matrices (here 10)

*fitWeightedBi.sampler(10)*

The user just needs to set these five last lines of the script, and then run the script on the data matrix. The folder “samples\_W” available in the repository was generated by applying the script to *'bicm\_matW.csv'* , setting the last line to 999 samples.