Co-occurrence Code Documentation

Jim Brunner

August 15, 2017

Contents

1 co_occ_funs.py

Module of python functions used in the rest of the project.

1.1 both_occ

Function that counts the number of times both `r1` and `r2` are within a range, and then returns the fraction of times this occurs. The range is half open: $(a, b]$.

- Input:
  - `r1`, `r2` - numpy arrays
- Optional input:
  - `lbd` - lower bound, default 0.
  - `ubd` - upper bound, default 1.
- Output: int count of indices i such that $r1(i) \in (lbd, ubd]$ and $r2(i) \in (lbd, ubd]$.

1.2 both_same_bin

Function that counts the number of times the two occur in the same abundance range.

- Input:
  - `r1`, `r2` - numpy arrays
  - `lthresh` - float, abundances lower are ignored.
  - `numthresh` - number of ranges (bins) to use.
- Optional input:
  - `rel` - bool, whether or not to normalize rows, default True.
- Output: count of indices i such that such that $r1(i) \in (lbd_j, ubd_j]$ and $r2(i) \in (lbd_j, ubd_j]$ for $j = 0, .., numthresh$.

Divides the interval $[0, 1]$ into $numthresh$ intervals $(a_j, b_j]$ and calls `both_occ(r1,r2,lbd = a_j, ubd = b_j)` for each.

1.3 color_picker

Function that classifies nodes by which type of sample they have the highest abundance in. If it's a dataframe it will return the column head of the winner'

- Input:
  - r - array or row of dataframe
- Optional input: none
- Output: index or column name of argmax of r, if the max is greater than $1.5 * \sigma + \mu$, where σ is the sample standard deviation, and μ is the sample mean

1.4 matchyn

- Input:
  - a, b - scalars or list to be compared
- Optional input: none
- Output: a if $a = b$, otherwise ['Intertype', grey], where grey is the hex value for grey

purpose is to compare two lists that look like ['Class', color].

1.5 occ_probs

Calculate probability of occurrence at an abundance level in a random graph, binomial distribution with parameters edge degree and sample degree/total edges, as in [4].

- Input:
  - abund_array - dataframe of GOTTCHA output
  - lthresh - float lower bound, abundances below are ignored
  - numthresh - int number of bins
- Optional input:
  - rel - bool, whether or not to normalize GOTTCHA data, default True
- Output: array occ_prob, the probability of an abundance level in a null model that assumes abundances are binomial with parameters coming from the number of taxa appearing in the sample and number of times a taxa appears.

1.6 random_coocc_prob

Calculate a poisson-binomial...$P(X > wij)$ where X is the number of times i and j co occur in random graph (X is a random variable) [4].

- Input:
  - occ - numpy array of probabilities (output of occ_probs)
  - wij - float the abundance to test
  - i, j - the indices of the taxa pair tested
- Optional input: none
- Output: float, probability that the null model produces taxa i and taxa j in the same bin more than wij times.

First gets the probability for each sample-level they are both present, then calculates a probability of the number of times this happens. Very very slow, do not use. This is because there are many ways 2 things can co-occur some number of times.

1.7 `approx_rand_prob`

Calculate an approximation of a poisson-binomial...$P(X > wij)$ where X is the number of timesi and j co occur in random graph (X is a random variable) The paper [4] calls it bi-binomial because its as if you had two probabilities in the above.

- Input:
  - `occ` - numpy array of probabilities (output of `occ_probs`)
  - `wij` - float the abundance to test
  - `i`, `j` - the indices of the taxa pair tested
- Optional input: none
- Output: float, approximate probability that the null model produces taxa i and taxa j in the same bin more than wij times.

First gets the probability for each sample-level they are both present, then calculates an approximate probability.

1.8 `mc_pearson`

MC approximation for pearson coefficient where X is a random vector of binomial(n1,p1) and Y is a random vector of binomial(n2,p2), where p1,p2 are vectors. Expected value is identity matrix.

- Input:
  - `N` - numpy array of of number of trials parameter for binomial RV
  - `P` - numpy array of probabilities of success
  - `W` - the observed correlation matrix
- Optional input:
  - `num_samps`, default = 1000. Number of Monte Carlo draws
- Output: matrix giving the percentage of MC draws that had higher correlation than observed.

If n is the number of samples m the number of taxa in the network being built (number of taxa in data at given level, minus any that do not correlate with any other taxa), then N and P should be $m \times n$ matrices. The MC draw is an $m \times m$ matrix of "correlations" made from the null model.

1.9 `make_null`

Companion to `mc_pearson` for palatalization

- Input:
  - `N` - numpy array of of number of trials parameter for binomial RV
  - `P` - numpy array of probabilities of success
  - `W` - the observed correlation matrix
- Optional input: none
- Output: sample - A binary matrix of bools (simulated correlation ¿ observed correlation)

The null model assumes that the abundance of a taxa in a sample is binomial(n,p) where n is the number of times the taxa is seen in the real data, and p is the proportion of non-zero entries in the real data that occur in the sample. A simulated data array of abundances of taxa in sample is created. Then, the correlation matrix is computed for this. Finally, this is compared to the observed correlation matrix.

1.10 `mc_pearson_thr`

MC approximation for pearson coefficient where X is a random vector of binomial(n1,p1) and Y is a random vector of binomial(n2,p2), where p1,p2 are vectors. Expected value is identity.

- Input:
  - `N` - numpy array of of number of trials parameter for binomial RV
  - `P` - numpy array of probabilities of success
  - `W` - the observed correlation matrix
- Optional input:
  - `num_samps`, default = 1000. Number of Monte Carlo draws.
- Output: matrix giving the percentage of MC draws that had higher correlation than observed.

Same as the two above (in combination), with the only difference being that simulated data is thresholded before correlations are computed. Should compare to observed correlations of thresholded data.

1.11 `min_nz`

Find minimum non-zero value

- Input:
  - `arr` - numpy array
- Optional input:
  - `row` - bool, whether or not to compute minimum by row, default False.
- Output: the minimum nonzero value in a 1D array (if row is false), or an array containing the min nonzero in each of row of a 2D array

1.12 `build_network`

Build a cooccurrence network from a pandas dataframe. Data should all come from same taxonomic level, with columns `['LEVEL','TAXA','SAMPLE_1',...,'SAMPLE_N']`

- Input:
  - `abundance_array` - abundance array dataframe at one tax level
- Optional input:
  - `cotype` - str, way to compute correlations, default 'pearson', options 'pearson' and 'bins'.
  - `thr` - bool default False, whether to threshold data before computing correlations.
  - `list_too` - bool default True, whether or not to return an edge list as well as adjacency matrix
- Output: adjacency matrix as dataframe with index and columns taxa names. If `list_too`, a dataframe of edges, each row is Node, Node, Weight.

bins counts the number of times row have the same binned value in a columns, while pearson computes a correlation matrix. Networks are filtered by the null model described in the above functions `approx_rand_prob`, `mc_pearson`, or `mc_pearson_thr`. Edges are kept if $p < 0.05$ according to these null models. List output (result of `list_too` True) is formatted for easy loading into cytoscape, using the "import network from file" option.

1.13 make_meta

Make separate node attribute table, and add edge metadata to the existing network data frame (given as list of edges). Node table needs columns for node frequency, sample most commonly seen in, and sample type color. Edges need sample type and color.

- Input:
  - `edges` - dataframe list of Node, Node, Weight.
  - `ab_by_sample_type`, dataframe abundance array with columns of same class (location on body, for example) summed.
  - `orig_array` dataframe given to `build_network`.
- Optional input: none
- Output: dataframe list of Node, Node, Weight, Edge Classification, Edge Classification Color. dataframe of node attributes - sample most seen in, color with that sample.

The node data table is formatted for easy input into cytoscape using the "import table from file" option, after the network has been imported.

1.14 make_meta_from_file

'Make a node attribute table using sample metadata file. edges is the network dataframe while metadata is a dataframe of metadata, and `orig_array` is the array of abundances (at the appropriate taxonomic level). The index set of the metadata should correspond columns [2:] of the `orig_array`

- Input:
  - `edges` - DF of Node, Node, Weight, (can have additional columns).
  - `metadata` - dataframe table of metadata indexed by sample ID.
  - `orig_array` - the dataframe used to build network.
- Optional input:
  - `existing_table` - dataframe of node data, default `[]`. If a dataframe is passed, node data columns will be added to this.
- Output: dataframe table of node data

The node data table is formatted for easy input into cytoscape using the "import table from file" option, after the network has been imported.

1.15 mc_network_stats

Construct a random network and compute statistics, compare these to the statistics of the statistics of the given adjacency matrix. abdata is the data values (numpy array) and network is the adjacency matrix. Random graph is made from null model detailed in the writeup

- Input:
  - `abdata` - numpy array, abundance array used to build network.
  - `network` - adjacency matrix as numpy array.
- Optional input:
  - `thr` - bool, whether or not data was thresholded in building the network, default False.
  - `bins` - bool, whether or not the binning method was used (as opposed to correlation), default False.
  - `sims` - number of MC draws used to estimate statistics, default 1000
- Output: the probability of seeing a greater mean degree, variance (in degree), minimum eigenvalue, maximum eigenvalue, and number of edges in a draw generated by the null model.

Null model is the same as in network construction, as described in `make_null`

1.16 sim_pears

Construct a MC draw simulating correlation

- Input:
  - `N` - numpy array of of number of trials parameter for binomial RV
  - `P` - numpy array of probabilities of success
- Optional input: none
- Output: mean degree, variance (in degree), minimum eigenvalue, maximum eigenvalue, and number of edges in the draw (given as list)

  Constructs an abundance array as in `make_null`

1.17 sim_pears_thr

Construct a MC draw simulating thresholded correlation

- Input:
  - `N` - numpy array of of number of trials parameter for binomial RV
  - `P` - numpy array of probabilities of success
- Optional input: none
- Output: mean degree, variance (in degree), minimum eigenvalue, maximum eigenvalue, and number of edges in the draw (given as list)

  Constructs an abundance array as in `make_null`

1.18 sim_bins

Construct a MC draw simulating binning

- Input:
  - `N` - numpy array of of number of trials parameter for binomial RV
  - `P` - numpy array of probabilities of success
- Optional input: none
- Output: mean degree, variance (in degree), minimum eigenvalue, maximum eigenvalue, and number of edges in the draw (given as list)

  Constructs an abundance array as in `make_null`

1.19 edge_prob

calculate probability of seeing an edge in a graph with that many edges if edge prob is p and we see m edges, $p \approx 2m/n^2$ (n nodes)

- Input:
  - `network` - dataframe list Node, Node, Weight (can have additional columns)
- Optional input: none
- Output: probability of an edge in a random graph with the same number of edges.

1.20 `random_sub_graph`

Calculate probability that the subnetwork has as many edges as it has

- Input:
  - `network` - dataframe list Node, Node, Weight, `edge_sample`
  - `types` - types of sample to look for, string
- Optional input:
  - `p` - float, probability of an edge, default 0.5
- Output: expected number of edges in a subgraph that size, actual number

1.21 `exp_cut_edges`

Calculate the probability of seeing that many intertype edges, or edges between some certain set of types, or out of some set of types.

- Input:
  - `network` - dataframe list Node, Node, Weight, `edge_sample`
  - `types` - types of sample to look for, string
- Optional input:
  - `p` - float, probability of an edge, default 0.5
- Output: expected number, actual number

1.22 `cut_cond`

Depreciated Calculate conductance of cuts that cut out a type or set of types

- Input:
  - `network` - dataframe of Node, Node, Weight, *with column for sample type - this column has been moved to the node data file and so this function must be rewritten*
  - `types` - string, type of nodes to look for
- Optional input: none
- Output: the conductance of a cut which removes the nodes of type specified.

  Cut conductance [2] is

$$\phi(S,\overline{S}) = \frac{\sum_{i\in S, j\in \overline{S}} a_{ij}}{\min(a(S), a(\overline{S}))}$$

where

$$a(S) = \sum_{i \in S, j \in V} a_{ij}$$

1.23 `com_clust`

Clustering using Girvan and Newman algortithm. This means we try to maximize the quantity *modularity* over all possible community groupings. Modularity is the quantity: (Fraction of edges that are inside a community) - (expected fraction in a random graph) I'd like to weight this, so I'll maximize ((δ is Kronecher, d is sum of weights of edges on that vertex (generalized degree)) Also, the undirectedness means I only have to take the sum over the subdiagonal.

- Input:
  - `network` - dataframe of adjacency matrix for graph

- Optional input: none
- Output: list of such that entry i tells us what cluster node i is in, and the modularity Q

Community clustering minimizes a function of the graph called modularity [1][5]. That is

$$Q = \frac{1}{2m} \sum_{i,j} \left(w_{ij} + \frac{d_i d_j}{2m} \right) \delta(c_i, c_j)$$

where $m = \sum_{i \sim j} w_{ij}$, d_i is the (weighted) degree of vertex i, c_i is the community containing vertex i, and δ is the Kronecker δ. This done by a gradient search/greedy algorithm.

1.24 `spectral_cluster`

Clustering using spectral clustering.

- Input:
  - `adj_mat` - numpy array of adjacency matrix for graph
- Optional input: none
- Output: list of such that entry i tells us what cluster node i is in.

Spectral clustering performs a k-means clustering on the rows of the matrix whose columns are the k eigenvectors of the graph Laplacian corresponding to the smallest k eigenvalues [6] . Intuitively, this means it clusters points together that are close in the first k (slowest decaying) modes of the diffusion equation on the graph.

1.25 `clust_judge`

Create a dataframe containing the percentage of each class in the metadata category appears in each cluster. `clust_type` should be either commun or spect. `node_data` should be a dataframe indexed by nodes, with a column for clusters and columns classifying nodes by meta data categories. `meta_col` should be the name of a column of `node_data`.

- Input:
  - `node_data` - dataframe of node data (including cluster number)
  - `meta_col` - string, name of a column of `node_data` to asses
  - `clust_type` - string, options ‘commun’ or ‘spect’
- Optional input: none
- Output: the percentage of each class that appears in each cluster (dataframe)

1.26 `color_picker2`

Takes in vector r and maps to hex colors

- Input:
  - `r` - a 1D array or list of values to be mapped to colors
- Optional input:
  - `the_map` - matplotlib colormap to pull colors from, default `cm.rainbow`.
  - `weighted` - bool, whether color values should be taken according values, as opposed to evenly distributed.
- Output: A list of colors (hex values)

1.27 `est_prob`

Unclear if this function is working correctly

Use distribution factorization from random markov field to estimate the probability of seeing the subset of taxa in the sample. Takes in the ROW NUMBERS of the represented edges, along with the whole network.

- Input:
  - `source` - list of row numbers of edges represented in the sample
  - `induced` - list of edges such that both nodes are in sample
  - `whole` - dataframe adjacency matrix of the network
  - `N` - number of nodes
- Optional input: none
- Output: log probability of a sample.

We can consider the network a Markov random field [3]. This can give us a way to calculate the probability a group of taxa occurs together. The main idea of a MRF is that nodes are conditionally independent of nodes they aren't neighbors of (conditioned on ones they are neighbors of). If c are the (maximal) cliques of the graph (complete sub-graphs), then the probability of configuration $\boldsymbol{x}$ is

$$P(\boldsymbol{x}) = \frac{1}{Z}\prod_c \psi_c(x_c)$$

where Z is a normalizing constant and ψ_c are some functions, often called "potential functions"[3].

1.28 `find_cliques`

find all the maximal cliques containing a given node, where network is an adjacency graph given as a numpy array

- Input:
  - `node` - indices of the node of interest
  - `network` - numpy array of adjacency matrix
- Optional input: none
- Output: list of maximal cliques involving the given node

1.29 `psi_over_psi`

calculate ratio between $\psi(k+1)$ & $\psi(k)$

- Input:
  - `k1`, `k2` - scalar values, should be number of nodes present in the clique in sample 1 and 2
  - `cliq` - list of nodes in a clique
- Optional input:
  - `rm` - scalar constant effecting formula, default 0.8
- Output: value of $\frac{\psi(k+1)}{\psi(k)}$ estimate

1.30 `diff_cliques`

Take two different samples and identify which cliques are different. Then, compute the ratio of probabilities between the two configurations based on the rule that $\frac{\psi(k+1)}{\psi(k)}$ depends on whether or not k is above or below half the size of the clique.

- Input:
  - `s1`, `s2` - abundance samples, given as numpy 1D arrays
  - `network` - numpy array of adjacency matrix
- Optional input: none
- Output: comparison of two samples.

1.31 `diffusion_ivp`

Using diffusion on the graph, rank the nodes we don't know about. Solve diffusion with initial condition being 1 for known on nodes and -1 for known off nodes. Network should be the adjacency graph (probably unweighted) given as a numpy array. Output is list of node indices ordered by the ranking procedure, with ties grouped in sublists

- Input:
  - `known_on` - list of nodes considered present, can be empty
  - `known_off` - list of nodes considered absent, can be empty
  - `network` - numpy array of adjacency matrix
- Optional input:
  - `suspected` - value to give nodes not in `known_on` or `known_off`, default 0.5.
  - `non_suspected` - value to give nodes in `known_off`, default 0.
  - `probably` - value to give nodes in `known_on`, default 1.
  - `sample` - a sample of data covering all nodes, to use as the initial condition. Default `[]`
  - `all` - bool, whether or not to rank all the nodes as opposed to just those not in `known_on` or `known_off`, default False
- Output: list of nodes in rank order, with nodes on the same connected component grouped and ties grouped. Rank is first over equilibrium solution, and secondly over transient.

1.32 `diffusion_bvp`

Using diffusion on the graph, rank the nodes we don't know about. Find equilibrium of solution with ”boundary values” given by known node values (0,1). Network should be the adjacency graph (probably unweighted) given as a numpy array.

- Input:
  - `known_on` - list of nodes considered present, can be empty
  - `known_off` - list of nodes considered absent, can be empty
  - `network` - numpy array of adjacency matrix
- Optional input: none
- Output: list of nodes in rank order, with ties grouped.

1.33 diffusion_forced

Using diffusion on the graph, rank the nodes we don't know about. Find equilibrium of solution with forcing (pm 1) on the known nodes. Network should be the adjacency graph (probably unweighted) given as a numpy array

- Input:
  - known_on - list of nodes considered present, can be empty
  - known_off - list of nodes considered absent, can be empty
  - network - numpy array of adjacency matrix
- Optional input: none
- Output: list of nodes in rank order, with ties grouped.

1.34 ivp_score

Calculate a fit score based on IVP ranking

- Input:
  - network_adj - dataframe of network adjacency matrix
  - the_samp - dataframe of sample to be judged
- Optional input:
  - con - scalar constant used in score calculation.
- Output: fit score:

  $$F_j(\boldsymbol{s}) = \frac{1}{\|\boldsymbol{s}\|}\sum_{i=0}^{n-1} c^i s_i$$

  where $\boldsymbol{s} = (u_{j_1}(0), u_{j_2}(0), ..., u_{j_n}(0))$ such that if

  $$\frac{d}{dt}\boldsymbol{u} = -L\boldsymbol{u}$$

  and $U_l = \int_0^\infty u_l(t)dt$ then

  $$U_{j_1} \geq U_{j_2} \geq \cdots \geq U_{j_n}$$

1.35 get_sample

create a random sample with the organsims in the real data. Made with the null model

- Input:
  - templ - dataframe template that the fake sample should look like
- Optional input:
  - numb - number of samples to make, default 1
- Output: dataframe of sample generated by the null model, as in make_null

1.36 make_sample

grab a real sample (column) from the template data. Option to choose the sample from a set of holdout columns that are not training data

- Input:
  - daata - template data to get a column of
- Optional input:
  - holdouts - list of columns held out of the network building, one of these will be chosen if there are any, default [].
- Output: chosen sample and its type.

1.37 flat_two_deep

Flattens doubly nested lists to lists. Allows originals like `[x,[x,x],[x,[x,x,x]]]`

- Input:
  - `li` - list with entries that may be lists or lists of lists.
- Optional input: none
- Output: flattened list

1.38 flat_one_deep

Flattens nested lists to lists. Allows originals like `[x,[x,x],x,x,x,x]`

- Input:
  - `li` - list with entries that may be lists.
- Optional input: none
- Output: flattened list

2 co_occurrence.py

This script builds three networks at each specified taxonomic level. Those are, a network build from the binning procedure, a network built from correlation, and a network built from correlations of thresholded data.

Command line input (all required):

- `csv_name` - The name of the GOTTCHA output data file. This should be a .txt or .csv with *space as separation character between columns.* To change the separation character, change line 61. Uses pandas `pandas.read_csv` to import the data.
- `level` - list of taxonomic levels to make networks for, or 'all'.
- `net_name` - name of a *folder* that the networks will be saved in.
- `sample_types` - bool that specifies whether networks should be made with subsets of the data corresponding to different body locations the samples were taken from.
- `numhld` - the number of columns that should be removed from the data for future testing.

The GOTTCHA output data should have a column specifying taxonomic level headed "LEVEL", and another column with taxa names headed "TAXA". The remaining columns should be of the form "`Body_Part_gender_LLL#######`" where the gender is optional, and can be "male", "female', or "NA".

The script combines abundances of samples from the same body part (referred to as the sample type) in order to classify nodes in the network according to where they are found in the highest abundance. This is saved in the pandas dataframe `samp_type_abund`.

Samples can be separated by metadata to make separate networks. If `sample_types` is true, then new data matrices are made, containing only samples of a single type(body location), and only types with more than 50 samples. The full array is also kept. This script can instead separate by gender, if this is in the column headings of data, by inputting `sample_types` as false, and switching `gender` to true (line 117). There is also the option to use some other type of metadata to create subsets of data and make networks, by setting `gender` to false and `other_meta` to true. Then, one must also input `meta_file` file name of a file that *can be parsed by pandas* `read_csv`, as well as `meta_column`, the name of the column of interest in the metadata. The file should be separated by spaces and the first column should be sample IDs. *This last possibility has not been tested and may contain bugs.*

Networks are made using `build_network` and `make_meta` from `co_occ_funs.py`.

Networks and node data .tsv files are created. The folder specified must have subfolders "bins" and "pears". The bin networks will be saved in "bins" and correlation in "pears". with the network file (list of edges: Node, Node, Weight, Edge Data, Edge Data) saved as "...`_list.tsv`", the adjacency matrix "...`_adj.tsv`", and the node data table as "...`_node_data.tsv`". In the "pears" folder, correlation of abundances and correlation of thresholded abundances can be distinguished by "cor" and "thr" respectively. If there were held out columns, a list of their indices is saved in "...`_held.tsv`". All of these files are saved as tab-separated text files using pandas `to_csv`.

3 `cluster_net.py`

This script takes networks and adds to the node data table the result of two clustering algorithms.

Command line input (required):

- `folder` - The folder where the network or networks in question are saved. Networks are assumed to have been saved as by `co_occurrence.py`, so adjacency matrices must end with "adj.tsv" and node data must end with "data.tsv".

This script will cluster *all the networks in the folder.* For each network found in the file, community clustering is run using `com_clust`, and spectral clustering is run using `spectral_cluster`, both from `co_occ_funs.py`. Both of these return a list such that entry i is the cluster number of node i.

Rather than producing a new output file, this script adds columns to the existing node data table. It adds four columns - the spectral cluster, the community cluster, a color (hex value) assigned to the spectral cluster, and a color (hex value) assigned to the community cluster. These colors can be shown in cytoscape using a "passthrough" mapping for node color.

4 `network_stats.py`

The purpose of this script is to determine the significance of the network build compared to the null model described in `make_null`.

Command line input (all required)

- `csv_name` - the data file used to build the network
- `level` - the taxonomic level at which the network was built
- `sample_types` - whether or not a network was made for each sample type.
- `adj_name` - name of the adjacency matrix file
- `type` - '`pears`' - correlation, '`pears_thr`' - thresholded correlation, or '`binned`' - binned.

Computes the following statistics for the network given:

- Number of edges
- Mean node degree
- Probability the null model produces a higher mean degree
- Probability the null model produces a higher degree variance
- Probability the null model produces a larger maximum eigenvalue
- Probability the null model produces a larger minimum eigenvalue
- Probability the null model produces more edges

The function `mc_network_stats` from `co_occ_funs.py` is used.

It outputs these to a file called `stats.txt` in the same folder as the network. If the file already exists, this will append to the end of it.

These statistics are meant to give some indication of how expected the network would be given the null model:

Let

$$N_j = |\{i : r_{ij} \neq 0\}|$$

and

$$P_i = \frac{|\{j : r_{ij} \neq 0\}|}{|\{(i,j) : r_{ij} \neq 0\}|}$$

then I take

$$\hat{r}_{ij} = binom(N_j, P_i)$$

as randomly generated abundance data. The values N_i are the counts of appearances of taxi i, while the values P_j are the proportions of all appearances which happen within each sample.

Then, I construct our random graph from this random data. If $\hat{\boldsymbol{r}}_i$ is the vector of random "abundances" of taxa i, I have edges

$$w_{ik}^{null} = \frac{1}{N}\frac{(\hat{\boldsymbol{r}}_{\boldsymbol{i}} - \hat{\mu}_i\mathbf{1}) \cdot (\hat{\boldsymbol{r}}_k - \hat{\mu}_k\mathbf{1})}{\hat{\sigma}_i\hat{\sigma}_k}$$

5 falsepm.py

This script is used for verification of the methods. It attempts to measure how well the network fitting procedure works, and how whether or not the ranking procedure can identify false negatives in a sample.

Command line input (all required):

- `abund_name` - the name to the original array used to construct the networks
- `folder` - the folder all the networks are stored in.

The folder should have networks made from all of the data as well as subsets by sample type (body location) - adjacency matrices saved as files ending with "adj.tsv". It should also have at least one file saved ending in "held.tsv" listing the nodes that were held out of network making.

The held out nodes are all tested for fit to the full network as well as fit to each of the networks made from subsets of the data, using `ivp_scores` from `co_occ_funs.py`. Then, each of the 30 highest abundance entries are set to 0 from the sample (one at a time) and the rank of the node is found using `diffusion_ivp`. We record this rank (minus the number of non-zero nodes).

Next, random samples are generate (equal in number to the number of hold outs) and tested for fit to the whole network.

Finally, this script produces the following plots, saved in a subfolder "`validation_plots`":

- histogram of fit scores to the full (correlation) network of the holdouts and the randomly generated samples
- histogram of fit scores to the full (thresholded) network of the holdouts and the randomly generated samples
- A bar chart of the proportion of samples of each type that were correctly classified (by the best fit score)
- Bar charts of the proportion of classification of the samples of each type
- scatter plot of abundance rank vs diffusion rank (after abundance is set to 0)

6 net_making.sh

This script creates a folder and networks (`co_occurrence.py`) at the genus and species level using the data in `merged2.txt`. It also runs clustering on these networks (`cluster_net.py`), computes network significance statistics (`network_stats.py`). Finally, it runs method validation (`falsepm.py`).

7 rand_samp.py

This script creates modified samples for method validation with `sample_analysis.py`.

Command line input (all required):

- `template_name` - file name of data used to create network, or similar
- `level` - taxonomic level the sample should "look like"
- `flder` - folder the sample should be saved in
- `cols_not_used` - numbers of the columns held out of network building.

Takes a column of the data (possibly a hold-out column) used to the create the full network. Creates a dataframe with the column's abundances, and a bool column of whether or not the abundance is nonzero. Optionally can modify the sample, and creates two columns of bools whether or not the abundance has been artificially set to 0 or artificially set to non-zero.

Additionally, if `rand_samp` is true, creates a sample from the null model and does the same to it.

Saves the sample or samples as .tsv files.

8 sample_analysis.py

Runs the diffusion procedures on a given sample.

Command line input (all required):

- sample_name - filename of sample - should be able to be interpreted by pandas read_csv.
- node_atts_name - the filename of the node attribute table of the network to be used
- cocc_mat_name - name of adjacency matrix file.
- level - taxonomic level of the network

This script ranks the nodes but all three possible diffusion rankings, diffusion_ivp, diffustion_bdvp, and diffusion_forced. It also assigns colors to these ranks. It saves the sample with abundances, non-zero nodes identified, rankings, and colors in a .tsv that can be imported into cytoscape as a node table.

9 sample_test.sh

Script that makes folders for samples, creates samples using rand_samp.py, and runs sample_analysis.py on these samples.

10 examples.py

Script to create some examples. If

Command line input (only if tiny is false):

- sample_name - file output from sample_analysis.py

If tiny is true, runs diffusion_ivp on three small toy networks. Then, computes their fit score. Finally, it plots abundance vs rank and displays the fit score.

If tiny is false, it takes a sample that has rankings associated with it already, computes the fit score, and plots abundance vs fit score and displays fit score. It also tries to classify the sample by the clusters of the network, and which clusters the highest rank nodes appear in.

11 adding_data.py

Script to build networks using growing subsets of the data.

Command line input (all required):

- csv_name - the GOTTCHA output data array
- level - the taxonomic level to be tested
- sample_types - whether or not the data should be separated by sample types (body locations).

Builds a network and, if monte is true, runs mc_network_stats using random subset of the data, increasing in size. Creates plots of how number of edges, mean degree, and if monte is true, the stats computed by mc_network_stats change as data is added.

12 mc_speed.py

A script to test how the length of time it takes to build a network and run mc_network_stats takes as data is added.

Command line input (required):

- csv_name - the GOTTCHA output data

Script is similar to adding_data.py, except that it repeats the process. Rather than saving the stats, it saves the length of time it takes to build and compute the stats of each network. It plots how time increases as data is added.

References

[1] Aaron Clauset, M. E. J. Newman, and Cristopher Moore. Finding community structure in very large networks. *Phys. Rev. E*, 70:066111, Dec 2004.

[2] Reinhard Diestel. *Graph Theory*, volume 173 of *Graduate Texts in Mathematics*. Springer, 2 edition, 2000.

[3] Kevin P. Murphy. *Machine Learning: A Probabilistic Perspective*. The MIT Press, 2012.

[4] Heiko Mller and Francesco Mancuso. Identification and analysis of co-occurrence networks with netcutter. *PLOS ONE*, 3(9):1–16, 09 2008.

[5] M. E. J. Newman. Analysis of weighted networks. *Phys. Rev. E*, 70:056131, Nov 2004.

[6] Ulrike von Luxburg. A tutorial on spectral clustering. *Statistics and Computing*, 17(4):395–416, 2007.