proxi Documentation

Release 1.0

Yasser El-Manzalawy

Jun 21, 2018

Contents

Proxi is a Python package for proximity graph construction. In proximity graphs, each node is connected by an edge (directed or undirected) to its nearest neighbors according to some distance metric *d*.

Proxi provides tools for inferring different types of proximity graphs from an OTU table including:

- k Nearest Neighbor kNN-graphs
- radius Nearest Neighbor rNN-graphs
- Perturbed k Nearest Neighbor pkNN-graphs

In addition, Proxi provides functionality for inferring pairwise graphs using virtually any user-defined proximity metric as well as support for aggregating graphs.

CHAPTER 1

Audience

The audience for Proxi includes bioinformaticians, mathematicians, physicists, biologists, computer scientists, and social scientists. Although Proxi was developed with metagenomics data in mind, the tool is applicable to other types of data including (but not limited to) gene expression, protein-protein interaction, wireless networks, images, etc.

CHAPTER 2

Citing

CHAPTER 3

Documentation

3.1 Installation

3.1.1 Dependencies

Proxi requires:

Python (>= 2.7 or >= 3.3) NumPy (>= 1.8.2) SciPy (>= 0.13.3) NetworkX (>= 2.1) Sklearn (>= 0.19.1)

3.1.2 User installation

If you already have a working installation of the required packages, the easiest way to install proxi is using pip:

```
$ pip install proxi
```

To upgrade to a newer release use the `--upgrade` flag:

```
$ pip install --upgrade proxi
```

3.2 ReadMe

Proxi is a Python package for proximity graph construction. In proximity graphs, each node is connected by an edge (directed or undirected) to its k nearest neighbors according to some distance metric d.

- **Website:** http://idsrlab.com/proxi/
- **Documentation:** https://proxi.readthedocs.io/en/latest/index.html
- **Tutorials:** https://proxi.readthedocs.io/en/latest/Tutorials.html
- **Source:** https://bitbucket.org/idsrlab/proxi/

3.2.1 Install

Install the latest version of Proxi:

```
$ pip install proxi
```

For additional details, please see 'INSTALL.rst'.

3.2.2 Bugs

Please report any bugs that you find here.

3.2.3 License

3.3 proxi package

3.3.1 Subpackages

3.3.1.1 proxi.algorithms package

Submodules

proxi.algorithms.knng module

Warrper for using sklearng kNN Graph (KNNG) construction method (see http://scikit-learn.org/stable/modules/generated/sklearn.neighbors.kneighbors_graph.html).

proxi.algorithms.knng.**get_knn_graph**(*data*, *k*, *metric='correlation'*, *p=2*, *metric_params=None*, *OTU_column=None*, *is_undirected=True*, *is_normalize_samples=True*, *is_standardize_otus=True*)

Compute the (directed/undirected) graph of k-Neighbors for points in the input data. The kNN-graph is constructed using sklearn method, sklearn.neighbors.kneighbors_graph.

Parameters

- **data** (*DataFrame*) – Input data as pandas DataFrame object. Each row is an OTU and each column is a sample
- **k** (*int*) – Number of neighbors for each node
- **metric** (*string or callable, default 'correlation'*) – metric to use for distance computation. Any metric from scikit-learn or scipy.spatial.distance can be used.

  If metric is a callable function, it is called on each pair of instances (rows) and the resulting value recorded. The callable should take two arrays as input and return one value indicating the distance between them.

  Valid values for metric are:

  – from scikit-learn: ['cityblock', 'cosine', 'euclidean', 'l1', 'l2', 'manhattan']

- from scipy.spatial.distance: ['braycurtis', 'canberra', 'chebyshev', 'correlation', 'dice', 'hamming', 'jaccard', 'kulsinski', 'mahalanobis', 'matching', 'minkowski', 'rogerstanimoto', 'russellrao', 'seuclidean', 'sokalmichener', 'sokalsneath', 'sqeuclidean', 'yule']

  See the documentation for scipy.spatial.distance for details on these metrics.

- any collable function (e.g., distance functions in proxi.utils.distance module)

p [int, optional, default = 2] Parameter for the Minkowski metric from sklearn.metrics.pairwise.pairwise_distances. When p = 1, this is equivalent to using manhattan_distance (l1), and euclidean_distance (l2) for p = 2. For arbitrary p, minkowski_distance (l_p) is used.

metric_params [dict, optional, default = None] Additional keyword arguments for the scipy metric function.

OTU_column [string, optional, default = None] Name of the DataFrame column that contains the OTUs IDs (i.e., nodes IDs). If OTU_column is None, the first column in the dataframe is treated as the OTU_column.

is_undirected [bool, optional, default = True] whether to compute undirected/directed graph. Default is undirected.

is_weighted [bool, optional, default = False] whether to compute weighted graph. Default is unweighted.

is_normalize_samples [bool, optional, default = True] whether to normalize each sample (i.e., column in the input data).

is_standardize_otus [bool, optional, default = True] whether to standardize each OTU (i.e., row in the input data)

Returns

- **nodes_id** (*array_like*) – list of nodes.
- **_A** (*scipy sparse matrix*) – Adjacency matrix of the constructed graph.

Examples

```
>>> df = pd.read_csv(in_file)
```

```
>>> # construct kNN-graph
>>> nodes, a = get_knn_graph(df, 5,  metric='braycurtis')
```

```
>>> # Note that a is a sparse matrix.
>>> # Use 'todense' to convert a into numpy matrix format required for NetworkX
>>> a = a.todense()
>>> print('Shape of adjacency matris is {}'.format(np.shape(a)))
```

```
>>> # save the constructed graph in graphml format
>>> save_graph(a, nodes, out_file)
```

proxi.algorithms.pairwise module

Construct a graph using a pairwise similarity metric (e.g. PCC).

proxi.algorithms.pairwise.**create_graph_using_pairwise_metric**(*data*, *similarity_metric*, *threshold*, *is_symmetric=True*, *OTU_column=None*, *is_normalize_samples=True*, *is_standardize_otus=True*, *is_weighted=False*)

Construct a graph using a pairwise similarity metric.

Parameters

- **data** (*DataFrame*) – Input data as pandas DataFrame object. Each row is an OTU and each column is a sample.
- **similarity_metric** (*collable similarity function*) – A collable function for computing the similarity between two vectors.
- **threshold** (*float*) – Minimum similarity score between two vectors required for having an edgle between their corresponding nodes.
- **is_symmetric** (*bool, optional, default=True*) – Set this parameter to False if the similarity function is not symmetric.
- **OTU_column** (*string, optional, default = None*) – Name of the DataFrame column that contains the OTUs IDs (i.e., nodes IDs). If OTU_column is None, the first column in the dataframe is treated as the OTU_column.

is_normalize_samples [bool, optional, default = True] whether to normalize each sample (i.e., column in the input data).

is_standardize_otus [bool, optional, default = True] whether to standardize each OTU (i.e., row in the input data)

is_weighted [bool, optional, default = False] whether to compute weighted graph. Default is unweighted.

Returns

- **nodes_IDs** (*array_like*) – list of nodes.
- **A** (*array_like, Shape(N,N)*) – Adjacency matrix of the constructed graph.
- **W** (*array_like, Shape(N,N)*) – Weight matrics.

Examples

```
>>> df = pd.read_csv(in_file)
>>> nodes, a, weights = create_graph_using_pairwise_metric(df, similarity_
↪metric=abs_pcc,
>>>                             threshold=0.5, is_weighted=True)
>>> # save unweighted graph in graphml format
>>> save_graph(a, nodes, out_file)
>>> # save weighted graph in graphml format
>>> save_weighted_graph(a, nodes, weights, out_file2)
```

proxi.algorithms.pknng module

Implementation of Perturbed kNN Graph (PKNNG) [1].

1- Generate T bootstrapped kNN graphs where at each iteration a new dataset is generated by resampling with replacement from the original dataset.

2- Aggregate the T graphs into a single one by keeping edges that appear in more than cT of the bootstrapped graphs with the sample weights for those edges.

References

[1] Wagaman, A. (2013). Efficient kNN graph construction for graphs on variables. Statistical Analysis and Data Mining: The ASA Data Science Journal, 6(5), 443-455.

proxi.algorithms.pknng.**get_pknn_graph**(*data*, *k*, *c=0.5*, *T=100*, *metric='correlation'*, *p=2*, *metric_params=None*, *OTU_column=None*, *random_state=0*, *is_undirected=True*, *is_weighted=False*, *is_normalize_samples=True*, *is_standardize_otus=True*)

Compute the (directed/undirected) graph of k-Neighbors for points in the input data. Each kNN-graph is constructed using sklearn method, sklearn.neighbors.kneighbors_graph.

Parameters

- **data** (*DataFrame*) – Input data as pandas DataFrame object. Each row is an OTU and each column is a sample.
- **k** (*integer*) – Number of neighbors for each node.
- **c** (*float, optional, default=0.5*) – Graph aggregation tunning parameter.
- **T** (*integer, optional, default=100*) – Number of bootstrap iterations.
- **metric** (*string or callable, default='correlation'*) – metric to use for distance computation. Any metric from scikit-learn or scipy.spatial.distance can be used.

  If metric is a callable function, it is called on each pair of instances (rows) and the resulting value recorded. The callable should take two arrays as input and return one value indicating the distance between them.

  Valid values for metric are:

  – from scikit-learn: [‘cityblock’, ‘cosine’, ‘euclidean’, ‘l1’, ‘l2’, ‘manhattan’]

  – from scipy.spatial.distance: [‘braycurtis’, ‘canberra’, ‘chebyshev’, ‘correlation’, ‘dice’, ‘hamming’, ‘jaccard’, ‘kulsinski’, ‘mahalanobis’, ‘matching’, ‘minkowski’, ‘rogerstanimoto’, ‘russellrao’, ‘seuclidean’, ‘sokalmichener’, ‘sokalsneath’, ‘sqeuclidean’, ‘yule’]

  See the documentation for scipy.spatial.distance for details on these metrics.

  – any collable function (e.g., distance functions in utils.distance module)

p [int, optional, default = 2] Parameter for the Minkowski metric from sklearn.metrics.pairwise.pairwise_distances. When p = 1, this is equivalent to using manhattan_distance (l1), and euclidean_distance (l2) for p = 2. For arbitrary p, minkowski_distance (l_p) is used.

metric_params [dict, optional, default = None] Additional keyword arguments for the scipy metric function.

OTU_column [string, optional, default = None] Name of the DataFrame column that contains the OTUs IDs (i.e., nodes IDs). If OTU_column is None, the first column in the dataframe is treated as the OTU_column.

random_state [integer, optional, default=0] #TODO

is_undirected [bool, optional, default = True] whether to compute undirected/directed graph. Default is undirected.

is_weighted [bool, optional, default = False] whether to compute weighted graph. Default is unweighted.

is_normalize_samples [bool, optional, default = True] whether to normalize each sample (i.e., column in the input data).

is_standardize_otus [bool, optional, default = True] whether to standardize each OTU (i.e., row in the input data)

Returns

- **nodes_id** (*array_like*) – list of nodes.
- **_A** (*scipy sparse matrix*) – Adjacency matrix of the constructed graph.

Examples

```
>>> df = pd.read_csv(in_file)
```

```
>>> # construct kNN-graph
>>> nodes, a = get_pknn_graph(df, 5, metric='braycurtis', T=10, c=0.5, is_
↪weighted=True,
>>>                            OTU_column='SID')
```

```
>>> print('Shape of adjacency matris is {}'.format(np.shape(a)))
```

```
>>> # save the constructed graph in graphml format
>>> save_graph(a, nodes, out_file)
```

```
>>> # save the directed graph in graphml format
>>> save_graph(a, nodes, out_file2, create_using=nx.DiGraph())
```

References

proxi.algorithms.rng module

Computes a (weighted) graph of Neighbors for each data point. Neighborhoods are restricted to the points at a distance lower than radius. This is simply a warrper for using sklearng radius_neighbors_graph method.

proxi.algorithms.rng.**get_rn_graph**(*data*, *radius*, *metric='braycurtis'*, *p=2*, *metric_params=None*, *OTU_column=None*, *is_undirected=True*, *is_normalize_samples=True*, *is_standardize_otus=True*)

Computes the (weighted/directed) graph of k-Neighbors for points in data

Parameters

- **data** (*DataFrame*) – input data as pandas DataFrame object. Each row is an OTU and each column is a sample
- **radius** (*float*) – Radius of neighborhoods.
- **metric** – The distance metric used to calculate the neighbors within a given radius for each sample point. The DistanceMetric class gives a list of available metrics. The default distance is correlation.

p [int, optional, default = 2] Parameter for the Minkowski metric from sklearn.metrics.pairwise.pairwise_distances. When p = 1, this is equivalent to using manhattan_distance (l1), and euclidean_distance (l2) for p = 2. For arbitrary p, minkowski_distance (l_p) is used.

metric_params [dict, optional, default = None] Additional keyword arguments for the scipy metric function.

OTU_column [string, optional, default = None] Name of the DataFrame column that contains the OTUs IDs (i.e., nodes IDs). If OTU_column is None, the first column in the dataframe is treated as the OTU_column.

is_undirected [bool, optional, default = True] whether to compute undirected/directed graph. Default is undirected.

is_weighted [bool, optional, default = False] whether to compute weighted graph. Default is unweighted.

is_normalize_samples [bool, optional, default = True] whether to normalize each sample (i.e., column in the input data).

is_standardize_otus [bool, optional, default = True] whether to standardize each OTU (i.e., row in the input data)

Returns

- **nodes_id** (*array_like*) – list of nodes.
- **_A** (*scipy sparse matrix*) – Adjacency matrix of the constructed graph.

Examples

```
>>> df = pd.read_csv(in_file)
```

```
>>> # construct kNN-graph
>>> nodes, a = get_rn_graph(df, 0.3,  metric='braycurtis')
```

```
>>> # Note that a is a sparse matrix.
>>> # Use 'todense' to convert a into numpy matrix format required for NetworkX
>>> a = a.todense()
>>> print('Shape of adjacency matris is {}'.format(np.shape(a)))
```

```
>>> # save the constructed graph in graphml format
>>> save_graph(a, nodes, out_file)
```

Module contents

3.3.1.2 proxi.utils package

Submodules

proxi.utils.distance module

Distance functions for proxi project.

proxi.utils.distance.**abs_correlation**(*x*, *y*)
Compute absolute correlation distance between two vectors.

Parameters

- **x** (*array_like, Shape(N,)*) – First input vector.
- **y** (*array_like, Shape(N,)*) – Second input vector.

Returns

Return type 1-|pcc(x,y)|

proxi.utils.distance.**abs_kendall**(*x*, *y*)
Compute absolute Kendall correlation (tau) distance between two vectors.

Parameters

- **x** (*array_like, Shape(N,)*) – First input vector.
- **y** (*array_like, Shape(N,)*) – Second input vector.

Returns

Return type 1-|tau(x,y)|

proxi.utils.distance.**abs_spearmann**(*x*, *y*)
Compute absolute spearmann correlation (spc) distance between two vectors.

Parameters

- **x** (*array_like, Shape(N,)*) – First input vector.
- **y** (*array_like, Shape(N,)*) – Second input vector.

Returns

Return type 1-|spc(x,y)|

proxi.utils.distance.**neg_correlation**(*x*, *y*)
Compute negative correlation distance between two vectors.

Parameters

- **x** (*array_like, Shape(N,)*) – First input vector.
- **y** (*array_like, Shape(N,)*) – Second input vector.

Returns

Return type 1 if pcc is positive. Otherwise, the distance is 1+pcc(x,y)

proxi.utils.distance.**neg_kendall**(*x*, *y*)
Compute negative Kendall correlation (tau) distance between two vectors.

Parameters

- **x** (*array_like, Shape(N,)*) – First input vector.
- **y** (*array_like, Shape(N,)*) – Second input vector.

Returns

Return type 1 if tau is positive. Otherwise, the distance is 1+tau(x,y)

proxi.utils.distance.**neg_spearmann**(*x*, *y*)
Compute negative spearmann correlation (spc) distance between two vectors.

Parameters

- **x** (*array_like, Shape(N,)*) – First input vector.
- **y** (*array_like, Shape(N,)*) – Second input vector.

Returns

Return type 1 if spc is positive. Otherwise, the distance is 1+spc(x,y)

proxi.utils.distance.**pos_correlation**(*x*, *y*)
Compute positive correlation distance between two vectors.

Parameters

- **x** (*array_like, Shape(N,)*) – First input vector.
- **y** (*array_like, Shape(N,)*) – Second input vector.

Returns

Return type 1 if pcc is negative. Otherwise, the distance is 1-pcc(x,y)

proxi.utils.distance.**pos_kendall**(*x*, *y*)
Compute positive Kendall correlation (tau) distance between two vectors.

Parameters

- **x** (*array_like, Shape(N,)*) – First input vector.
- **y** (*array_like, Shape(N,)*) – Second input vector.

Returns

Return type 1 if tau is negative. Otherwise, the distance is 1-spc(x,y)

proxi.utils.distance.**pos_spearmann**(*x*, *y*)
Compute positive spearmann correlation (spc) distance between two vectors.

Parameters

- **x** (*array_like, Shape(N,)*) – First input vector.
- **y** (*array_like, Shape(N,)*) – Second input vector.

Returns

Return type 1 if spc is negative. Otherwise, the distance is 1-spc(x,y)

proxi.utils.misc module

Miscellaneous Python methods for proxi project.

proxi.utils.misc.**aggregate_graphs**(*G*, *min_num_edges*, *is_weighted=False*)
Aggregate the adjaceny matrices of graphs defined over the same set of nodes.

Parameters

- **G** (*list of array_like matrices of shape (N,N)*) – list of adjacency matrices.
- **min_num_edges** (*int*) – min number of edges between two nodes required to keep an edge between them in the aggregated graph.
- **is_weighted** (*bool, optional, default = False*) – whether to conmpute a weighted aggregated graph.

Returns

- **rVal** (*agregated graph*)
- **W** (*edge weights (None if is_weighted is False)*)

proxi.utils.misc.**filter_edges_by_votes**(*A*, *G*, *min_num_votes*)

Aggregate the adjaceny matrices of a list of graphs G and use the aggregated graph to decide which edges in the base graph A to keep. All graphs are assumed to be defined over the same set of nodes.

Parameters

- **A** (*array_like, shape(N,N)*) – adjaceny matrix of the base graph.
- **G** (*list of array_like matrices of shape (N,N)*) – list of adjacency matrices.
- **min_num_votes** (*int*) – minimum number of edges between two nodes in the aggregated graph required to keep their edge (if exist) in the base graph.

Returns

- **rVal** (*array_like, shape(N,N)*) – adjaceny matrix of the filtered base graph.
- **W** (*array_like, shape(N,N)*) – edge wesights associated with rVal graph

proxi.utils.misc.**save_graph**(*A*, *nodes_id*, *out_file*, *create_using=None*)

Save the graph in graphml format.

Parameters

- **A** (*array_like, shape(N,N)*) – adjaceny matrix of the base graph.
- **nodes_id** (*array-like, shape(N,)*) – list of modes id
- **out_file** (*file or string*) – File or filename to write. Filenames ending in .gz or .bz2 will be compressed.
- **create_using** (*Networkx Graph object, optional, default is Graph*) – User specified Networkx Graph type. Accepted types are: Undirected Simple Graph

  Directed Simple DiGraph With Self-loops Graph, DiGraph With Parallel edges MultiGraph, MultiDiGraph

Notes

This implementation, based on networkx write_graphml method, does not support mixed graphs (directed and unidirected edges together) hyperedges, nested graphs, or ports.

proxi.utils.misc.**summarize_graph**(*G*)

Report basic summary statistics of a networkx graph object.

Parameters **G** (*graph*) – A networkx graph object

Returns

Return type A dictionary of basic graph properties.

proxi.utils.misc.**jaccard_graph_similarity**(*G1*, *G2*)

Compute Jaccard similarity between two graphs over the same set of nodes.

Parameters

- **G1** (*graph*) – A networkx graph object.
- **G2** (*graph*) – A networkx graph pbject.
- **Returns** –
- **-------s** –

- **Jaccard similarity between two graphs over the same set of nodes.** (*Compute*) –

proxi.utils.misc.**get_graph_object**(*A*, *nodes_id=None*)
Construct a networkx graph object given an adjaceny matrix and nodes IDs.

Parameters **A** (*array_like, shape(N,N)*) – adjaceny matrix of the base graph.

nodes_id [array-like, shape(N,)] list of modes id

Returns

Return type A networkx graph object.

proxi.utils.misc.**get_collable_name**(*func*)
Return the name of a collable function.

Parameters **func** (*collable function*) –

Returns

Return type The name of a collable function.

Notes

str(func) returns <function neg_correlation at 0x1085cdd08>.

proxi.utils.process module

Pre-processing methods for proxi project.

proxi.utils.process.**filter_OTUs_by_name**(*data*, *OTUs_to_keep*, *OTUs_column*)
Keeps only the OTUs in OTUs_to_keep list.

Parameters

- **data** (*DataFrame*) – Input data as a pandas DataFrame object. Each row is an OTU and each column is a sample
- **OTUs_to_keep** (*list*) – List of OTUs ID to select from the input dataframe.
- **OTU_column** (*string*) – Name of the DataFrame column that contains the OTUs IDs (i.e., nodes IDs).

Returns

Return type A dataframe derived from the input data by keeping only rows with specified OTUs IDs.

proxi.utils.process.**get_MAD**(*x*)
MAD is defined as the median of the absolute deviations from the data's median:

Parameters **x** (*array_like, Shape(N,)*) – Input 1-D array.

Returns

Return type The median of the absolute deviations (MAD) of x.

proxi.utils.process.**get_non_zero_percentage**(*x*)
The fraction of non-zero values in a 1-D array x.

Parameters **x** (*array_like, Shape(N,)*) – Input 1-D array.

Returns

Return type The percentage of non-zero elements in x.

proxi.utils.process.**get_variance**(*x*)

Compute the variance of an input vector x. Variance is the average of the squared deviations from the meanvar = mean(abs(x - x.mean())**2)

Parameters **x** (*array_like, Shape(N,)*) – Input 1-D array.

Returns

Return type The variance of x.

proxi.utils.process.**select_top_OTUs**(*data*, *score_function*, *threshold*, *OTUs_column*)

Filter OTUs using a scoring function and return top k OTUs or OTUs with scores greater than a threshold score.

Parameters

- **data** (*DataFrame*) – Input data as a pandas DataFrame object. Each row is an OTU and each column is a sample
- **score_function** (*collable function*) – Unsupervised scoring function (e.g., variance or percentage of non-zeros) of each OTU.
- **threshold** (*float*) – if threshold > 1, return top threshold OTUs. Otherwise, return OTUs with score > threshold.
- **OTU_column** (*string*) – Name of the DataFrame column that contains the OTUs IDs (i.e., nodes IDs).

Returns

Return type dataframe with selected OTUs

proxi.utils.similarity module

Similarity functions for proxi project.

proxi.utils.similarity.**abs_Kendall**(*x*, *y*)

Compute absolute Kendall correlation similarity between two vectors.

Parameters

- **x** (*array_like, Shape(N,)*) – First input vector.
- **y** (*array_like, Shape(N,)*) – Second input vector.

Returns

Return type **|kendalltau(x,y)|**

proxi.utils.similarity.**abs_pcc**(*x*, *y*)

Compute absolute Pearson correlation similarity between two vectors.

Parameters

- **x** (*array_like, Shape(N,)*) – First input vector.
- **y** (*array_like, Shape(N,)*) – Second input vector.

Returns

Return type **|pcc(x,y)|**

proxi.utils.similarity.**abs_spc**(*x*, *y*)

Compute absolute Spearman correlation similarity between two vectors.

Parameters

- **x** (*array_like, Shape(N,)*) – First input vector.
- **y** (*array_like, Shape(N,)*) – Second input vector.

Returns

Return type |spearmanr(x,y)|

proxi.utils.similarity.**distance_to_similarity**(*x*, *y*, *dist_func*)

Convert the distance functions in scipy.spatial.distance into similarity functions

Parameters

- **x** (*array_like, Shape(N,)*) – First input vector.
- **y** (*array_like, Shape(N,)*) – Second input vector.
- **dist_func** (*collable*) – collabel distance function (e.g., any distance function in scipy.spatial.distance)

Returns

Return type similarity between x and y.

Module contents

3.3.2 Module contents

3.4 Tutorials

Example simple uses and applications of Proxi are provided in the following tutorials

3.4.1 How to construct a proximity kNN graph?

by Yasser El-Manzalawy yasser@idsrlab.com

In this tutorial, we show how to construct undirected and directed kNN graphs from an Operational Taxonomic Unit (OUT) table.

An OTU Table is a form of the results that you will get from a metagenomics taxonomy classification pipeline. In that table, we are giving (for each sample) the number of sequences in each OTU and the taxonomy of that OTU. Samples correspond to columns and OTUs correspond to rows. OTUs taxonomy is the first column (by default) but it could be any column.

```
In [1]: import numpy as np
        import pandas as pd
        import networkx as nx

        from proxi.algorithms.knng import get_knn_graph
        from proxi.utils.misc import save_graph, save_weighted_graph
        from proxi.utils.process import *
        from proxi.utils.distance import abs_correlation
```

```
        import warnings
        warnings.filterwarnings("ignore")
```

3.4.1.1 Variables and Parameters settings

```
In [2]: # Input OTU Table
        healthy_file = './data/L6_healthy_train.txt'

        # Output file(s)
        healthy_graph_file = './graphs/L6_healthy_train.graphml'
        healthy_directed_graph_file = './graphs/L6_healthy_train_directed.graphml'


        # Parameters
        num_neighbors = 5       # number of nearest neighbors in the kNN graph
        dist = abs_correlation  # distance function
```

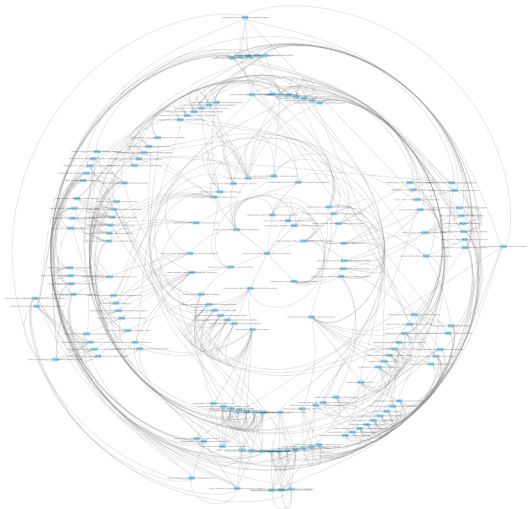
3.4.1.2 Load OTU Table and remove useless OTUs

```
In [3]: # Load OTU Table
        df = pd.read_csv(healthy_file, sep='\t')

        # Delete OTUs with less than 5% non-zero values
        df = select_top_OTUs(df, get_non_zero_percentage, 0.05, 'OTU_ID')
```

3.4.1.3 Construct an undirected kNN graph

```
In [4]: # Construct kNN-graph
        nodes, a = get_knn_graph(df, k=num_neighbors,  metric=dist)

        # Save the constructed graph in an edge list format
        save_graph(a.todense(), nodes, healthy_graph_file)
```

Like other graph inference tools, proxi doesn't support any network visualization functionality. Here, we used Cytoscape to open our graphml file and change the network layout to 'Radial layout' (see Figure 1). Moreover, Cytoscape has many tools and plugins that could be used for downstream analyses of our constructed networks.

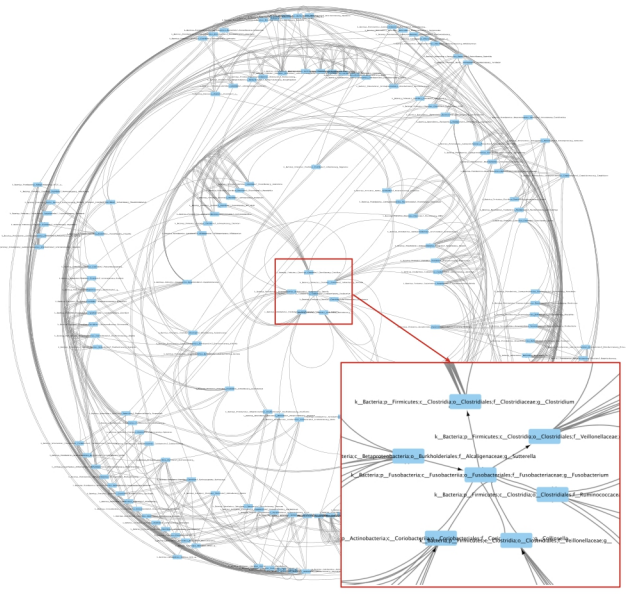


!

Figure 1: kNN undirected proximity graph constructed from healthy OTU table using k = 5.

3.4.1.4 Construct a directed kNN graph

```
In [5]: # construct directed kNN-graph
        nodes, a = get_knn_graph(df, k=num_neighbors,  metric=dist, is_undirected=False)

        # save the constructed graph in an edge list format
        save_graph(a.todense(), nodes, healthy_directed_graph_file, create_using=nx.DiGraph())
```



Now, let's visualize the constructed directed network using Cytoscape.
Figure 2: kNN directed proximity graph constructed from healthy OTU table using k = 5.

3.4.1.5 Limitation of kNN graphs

A major limitation of the constructed kNN graphs in Figures 1 and 2 is that the constructed graphs might not be sparse. This limitation could be addressed using different approaches including:

```
<li> Using smaller k. </li>
<li> Using Perturbed kNN Graphs (see Tutorial 2). </li>
<li> Using aggregated graphs constructed using different distance functions (see␣
↪Tutorial 3).</li>
```

3.4.2 How to construct a perturbed kNN graph?

by Yasser El-Manzalawy yasser@idsrlab.com

In this tutorial, we show how to construct directed/undirected perturbed kNN graphs [1]. This algorithm simply uses bootstrapping to perturb the graph (i.e., obtain several boostrapped graphs and aggregate them). An important property of the resulting perturbed kNN graphs is that it may not have the same k for every vertex.

References: [1] Wagaman, A. (2013). Efficient kNN graph construction for graphs on variables. Statistical Analysis and Data Mining: The ASA Data Science Journal, 6(5), 443-455.

```
In [1]: import numpy as np
        import pandas as pd
        import networkx as nx

        from proxi.algorithms.pknng import get_pknn_graph
        from proxi.utils.misc import save_graph, save_weighted_graph
        from proxi.utils.process import *
        from proxi.utils.distance import abs_correlation

        import warnings
        warnings.filterwarnings("ignore")
```

3.4.2.1 Variables and Parameters settings

```
In [2]: # Input OTU Table
        healthy_file = './data/L6_healthy_train.txt'

        # Output file(s)
        healthy_graph_file = './graphs/L6_healthy_train_pknng.graphml'   # Output file for pkNN graph
        # Output file for weighted and directed pkNN graph
        healthy_weighted_directed_graph_file = './graphs/L6_healthy_train_weighted_directed_pknng.gra

        # Parameters
        num_neighbors = 5        # Number of neighbors, k, for kNN graphs
        dist = abs_correlation   # distance function
        T=100                    # No of iterations
        c=0.6                    # control parameter for pknng algorithm
```

3.4.2.2 Load OTU Table and remove useless OTUs

```
In [3]: # load OTU Table
        df = pd.read_csv(healthy_file, sep='\t')

        # proprocess OTU Table by deleting OTUs with less than 5% non-zero values
        df = select_top_OTUs(df, get_non_zero_percentage, 0.05, 'OTU_ID')
        IDs = df['OTU_ID'].values
```

3.4.2.3 Construct an undirected pkNN graph

```
In [4]: # construct kNN-graph
        nodes, a,_ = get_pknn_graph(df, k=num_neighbors, metric=dist, T=T, c=c)

        # save the constructed graph in an edge list format
        save_graph(a, nodes, healthy_graph_file)
```

```
Shape of original data is (161, 200)
```

Now, you can use Cytocscape to visualize (and analyze) the constructed graph (See Fig. 1).

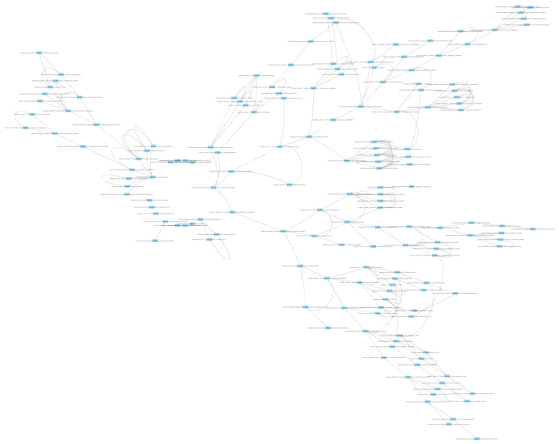
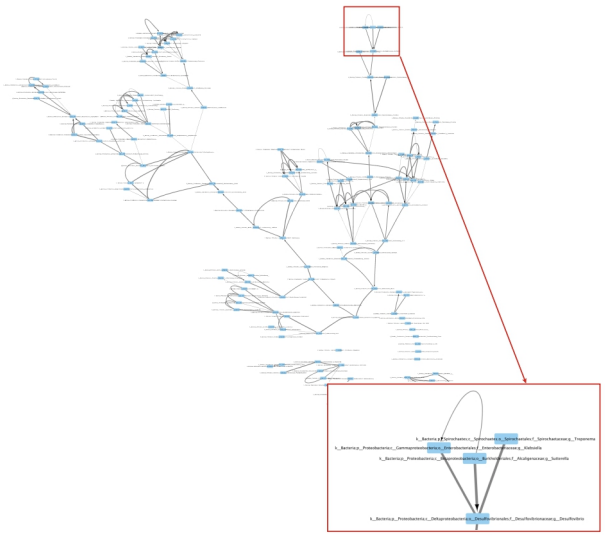


Figure 1: Perturbed kNN undirected proximity graph constructed from healthy OTU table using k=5, T=100, and c=0.6.

3.4.2.4 Construct a weighted and directed pkNN graph

```
In [5]: # construct directed kNN-graph
        nodes, a, weights = get_pknn_graph(df, k=num_neighbors,  metric=dist, T=T, c=c, is_undirecte

        # save the constructed graph in an edge list format
        save_weighted_graph(a, nodes, weights, healthy_weighted_directed_graph_file)
```

```
Shape of original data is (161, 200)
```



Now, use Cytoscape to visualize the graph (See Fig. 2).
Figure 2: Perturbed kNN weighted and directed proximity graph constructed from healthy OTU table using k=5, T=100, and c=0.6.

3.4.3 How to construct an aggregated kNN graph?

by Yasser El-Manzalawy yasser@idsrlab.com

In tutorial 1, we showed how to construct a kNN graph. To construct such graphs, you need to decide on k (number of neighbors) and d (the dissimilarity metric). Selecting a dissimilarity metric is not trivial and should be taken into account when interpreting the resulting kNN graph. Proxi allows the following distance functions to be used:

```
- from scikit-learn: ['cityblock', 'cosine', 'euclidean', 'l1', 'l2', 'manhattan']

- from scipy.spatial.distance: ['braycurtis', 'canberra', 'chebyshev',
  'correlation', 'dice', 'hamming', 'jaccard', 'kulsinski',
  'mahalanobis', 'matching', 'minkowski', 'rogerstanimoto',
  'russellrao', 'seuclidean', 'sokalmichener', 'sokalsneath',
  'sqeuclidean', 'yule']

- any callable function (e.g., distance functions in proxi.utils.distance module)
```

Moreover, Proxi supports any user-defined callable function. For example, a user might define a new function that is the average or weighted combination of some of the functions listed above. Finally, Proxi aggregate_graphs and filter_edges_by_votes methods allow user to construct different kNN graphs using different distance functions and/or ks. In what follows, we show how to aggregate three graphs constructed using k=5 and three different distance functions.

```
In [1]: import numpy as np
        import pandas as pd
        import networkx as nx

        from proxi.algorithms.knng import get_knn_graph
        from proxi.utils.misc import save_graph, save_weighted_graph, aggregate_graphs, filter_edges_
        from proxi.utils.process import *
        from proxi.utils.distance import abs_correlation, abs_spearmann, abs_kendall

        import warnings
        warnings.filterwarnings("ignore")
```

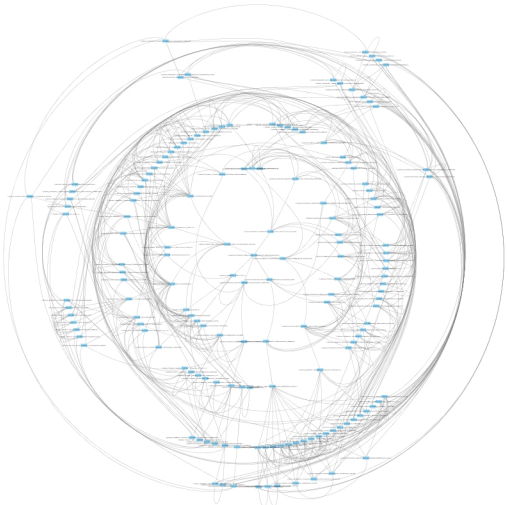
```
In [2]: # Input file(s)
        healthy_file = './data/L6_healthy_train.txt'    # OTU table

        # Output file(s)
        healthy_graph_file = './graphs/L6_healthy_train_aknng.graphml'    # Output file for aggregate

        # Graph aggregation parameters
        num_neighbors = 5  # Number of neighbors, k, for kNN graphs
        dists = [abs_correlation, abs_spearmann, abs_kendall]    # distance functions
        min = 2       # minimum number of edges needed to have an edge in the aggregated graph
```

3.4.3.1 Load OTU Table and remove useless OTUs

```
In [3]: # Load OTU Table
        df = pd.read_csv(healthy_file, sep='\t')

        # Proprocess OTU Table by deleting OTUs with less than 5% non-zero values
        df = select_top_OTUs(df, get_non_zero_percentage, 0.05, 'OTU_ID')
```

3.4.3.2 Method 1 for constructing an undirected aggregated kNN graph

```
In [4]: graphs = []
        # Construct kNN-graphs using different distance fucntions
        for dist in dists:
            nodes, a = get_knn_graph(df, k=num_neighbors, metric=dist)
            graphs.append(a.todense())

        aggregated_graph,_ = aggregate_graphs(graphs, min)

        # Save the constructed graph in an edge list format
        save_graph(aggregated_graph, nodes, healthy_graph_file)
```



Now, we can visualize the graph using Cytoscape (See Fig. 1) Figure 1: Aggregated kNN graph obtained by aggregating three kNN graphs consutucted using three distance functions, abs_correlation, abs_spearmann, and abs_kendall.

An interesting property of the aggregated graph in Fig. 1 is that each edge is supported by at least 2 distance functions. Alternatively, one can aggregate the three graphs such that each edge is supported by one fixed base distance function

(e.g., abs_correlation) plus at least one of the remaining two functions. Therefore, each edge in the resulting aggregated graph (Fig. 2) is supported by at least two functions such that one of them is abs_correlation.

3.4.3.3 Method 2 for constructing an undirected aggregated kNN graph

```
In [5]: # Specify input/output files and parameters

        # Output file
        healthy_graph_file2 = './graphs/L6_healthy_aknng2.graphml'    # Output file for aggregated pkl

        # Graph aggregation parameters
        base_distance = abs_correlation
        dists = [abs_spearmann, abs_kendall]    # distance functions
        min_votes = 1
```

```
In [6]: graphs = []
        # Construct kNN-graphs using different distance fucntions
        for dist in dists:
            nodes, a = get_knn_graph(df, k=num_neighbors, metric=dist)
            graphs.append(a.todense())

        nodes, a = get_knn_graph(df, k=num_neighbors, metric=base_distance)
        aggregated_graph,_ = filter_edges_by_votes(a.todense(), graphs, min)

        # Save the constructed graph in an edge list format
        save_graph(aggregated_graph, nodes, healthy_graph_file2)
```

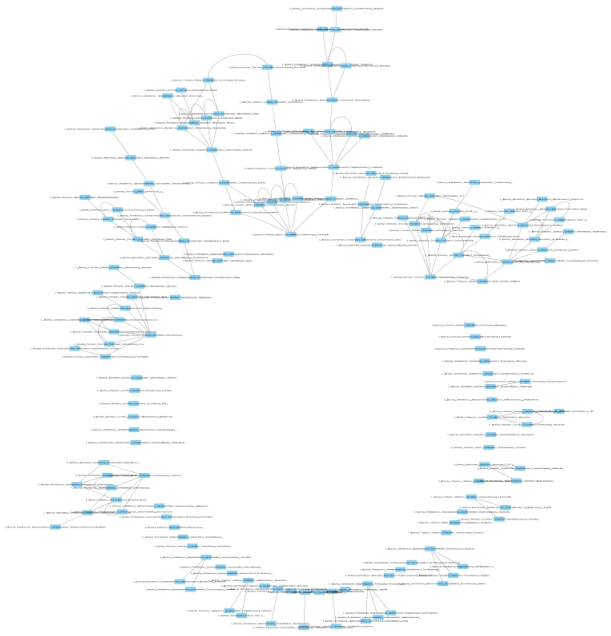


Figure 2: Sparse base kNN graph (using abs_correlation) and remaining two graphs are used for filtering out unsupported edges.

It worths to mention that these two methods of aggregating graphs could also be applied to aggregate the following graphs:

```
<li> kNN graphs constructed with different <i>k</i> values</li>
<li> radius graphs rNN graphs with different <i>radius</i> values</i>
```

(continues on next page)

(continued from previous page)

```
<li> different perturbed kNN graphs obtained using different T, c, k, or distance␣
↪parameters</li>
```

3.4.4 Comparative network analysis of perturbed kNN graphs

by Yasser El-Manzalawy yasser@idsrlab.com

In this tutorial, we construct two perturbed kNN graph for IBD and healthy controls (respectively) and then present examples of possible comparative network analysis that could be apply to the two graphs using Cytoscape. In particular, we compare the two graphs using: - Their global topological properties obtained using Cytoscape NetworkAnalyzer tool - Their top modules obtained using MCODE plugins - Their most varying nodes using DyNet Analyzer plugins and we report the subnetwork of top most varying 20 nodes (potential IBD biomarkers)

```
In [1]: import numpy as np
        import pandas as pd
        import networkx as nx

        from proxi.algorithms.pknng import get_pknn_graph
        from proxi.utils.misc import save_graph, save_weighted_graph
        from proxi.utils.process import *
        from proxi.utils.distance import abs_correlation

        import warnings
        warnings.filterwarnings("ignore")
```

3.4.4.1 Construct an undirected pkNN graph using IBD OTU table

```
In [2]: # Input file(s)
        ibd_file = './data/L6_IBD_train.txt'    # OTU table

        # Ouput file(s)
        ibd_graph_file = './graphs/L6_IBD_train_pknng.graphml'    # Output file for pkNN graph

        # Parameters
        num_neighbors = 5         # Number of neighbors, k, for kNN graphs
        dist = abs_correlation    # distance function
        T=100                     # No of iterations
        c=0.6                     # control parameter for pknng algorithm
```

```
In [3]: # Load OTU Table
        df = pd.read_csv(ibd_file, sep='\t')

        # Proprocess OTU Table by deleting OTUs with less than 5% non-zero values
        df = select_top_OTUs(df, get_non_zero_percentage, 0.05, 'OTU_ID')

        # Construct kNN-graph
        nodes, a,_ = get_pknn_graph(df, k=num_neighbors, metric=dist, T=T, c=c)

        # Save the constructed graph in an edge list format
        save_graph(a, nodes, ibd_graph_file)
```

```
Shape of original data is (178, 200)
```

Fig. 1 shows the constructed perturbed kNN graph from IBD samples.

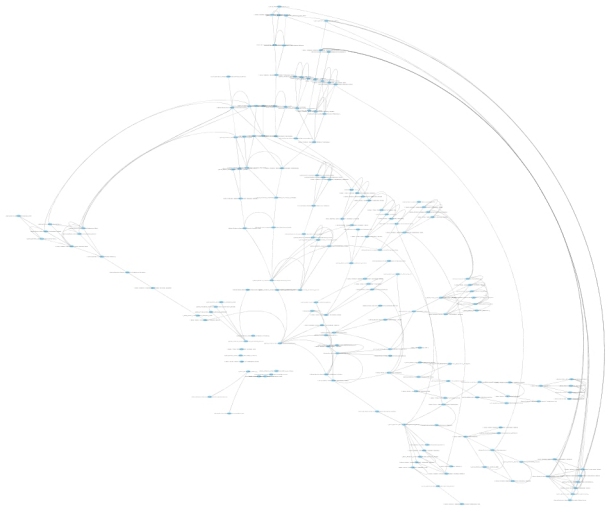


Figure 1: Perturbed kNN undirected proximity graph constructed from IBD OTU table using k=5, T=100, and c=0.6.

Fig. 2 shows the constructed perturbed kNN graph from healthy control samples. Note that we don't need to construct this network since it has been generated in tutorial 2.

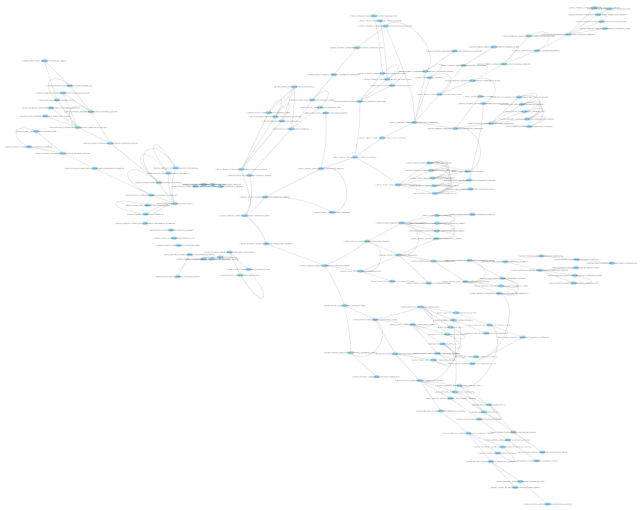


Figure 2: Perturbed kNN undirected proximity graph constructed from healthy OTU table using k=5, T=100, and c=0.6 (See Example_2).

Now, we can use cytoscape and some of its plugins to compare the two graphs in Figures 1 and 2.

3.4.4.2 Analysis of global topological properties

First, we used Cytoscape NetworkAnalyzer tool (1) to get several global properties of each network. Fig. 3 shows that IBD network has higher average node degree, clustering coefficient, network centralization, and number of nodes.

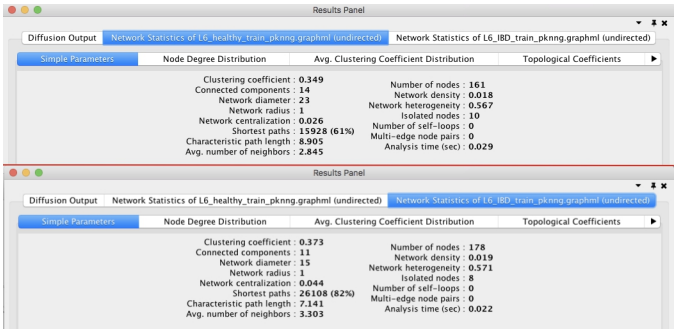


Figure 3: Global network properties for healthy (top) and IBD (bottom) networks.

3.4.4.3 Analysis of top first modules

Second, we used MCODE (2) to extract top modules from each network. Fig. 4 compare the top first module from healthy (top) and IBD (bottom) networks. For healthy network, the top module includes interactions between 4 different genera of Firmicutes and 2 different genera of Actionbacteria. For IBD network, the top module includes interactions among different genara belonging to Actionbacteria, Proteobacteria, Firmicutes, and Bacteriodetes phylum.

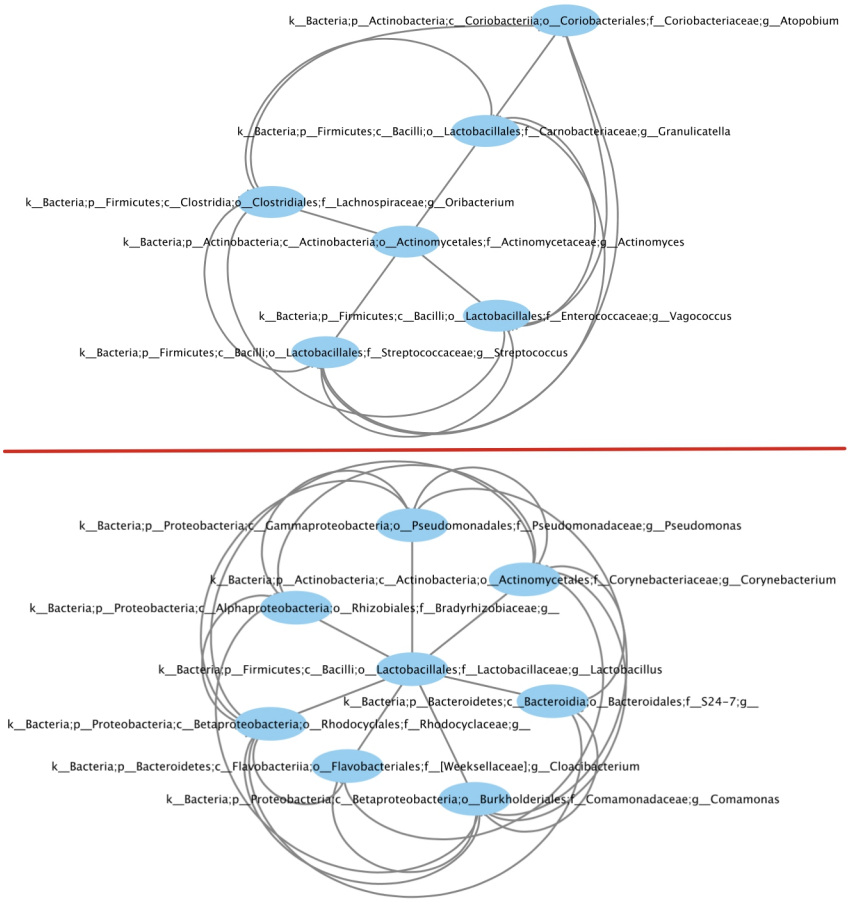


Figure 4: Top module extracted from healthy (top) and IBD (bottom) networks.

3.4.4.4 Analysis of most varying nodes

Third, we used DyNet Analyzer (3) to compare the the networks in healthy and IBD states. The results are visualized in Fig. 5 where: green edges represent edges present only in healthy network; red edges represent edges present only in IBD network; and gray edges represent edges present in both networks. DyNet also associates a rewiring score with each node that quantifies the amount of change in the identity of the node interacting neighbors. We then ranked nodes by their DyNet score and generated a subnetwork of the top 20 nodes (See Fig. 6). Interestingly, 13 out of 20 nodes form a single connected module. In this module, two nodes corresponding to corynebacterium genera and Rhodocyclaceae family have the highest node degrees of 5 and 4 (respectively).

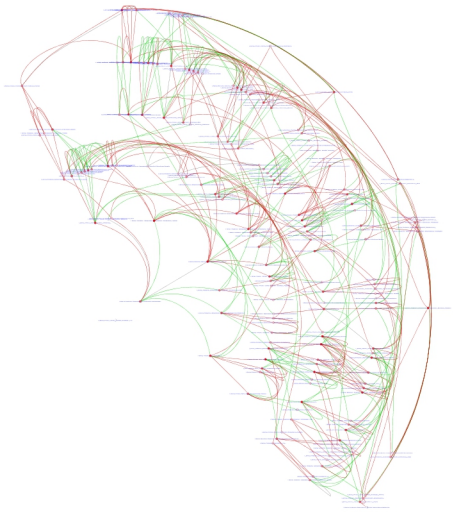


Figure 5: DynNet Analyzer. Healthy (green) and IBD (red).

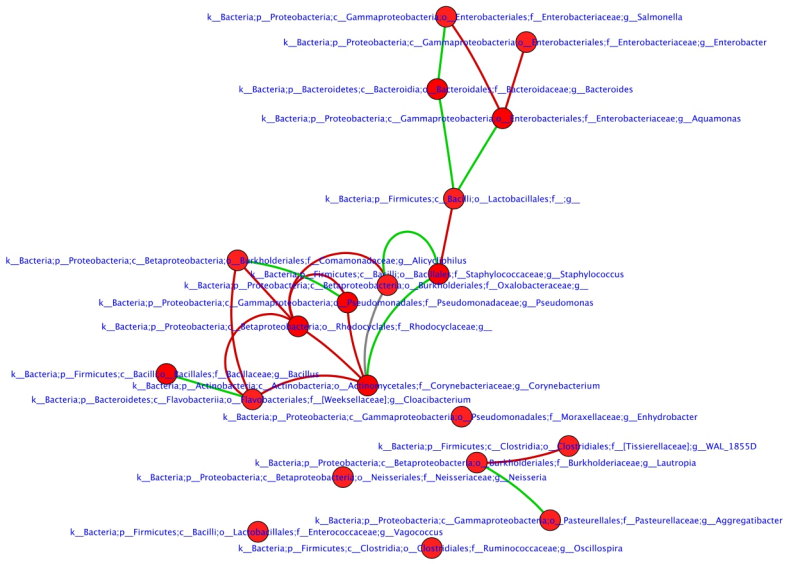


Figure 6: Subnetwork of top 20 varying nodes determined using DyNet score.

References:

[1] Assenov, Yassen, et al. "Computing topological parameters of biological networks." Bioinformatics 24.2 (2007): 282-284.

[2] Bader, Gary D., and Christopher WV Hogue. "An automated method for finding molecular complexes in large protein interaction networks." BMC bioinformatics 4.1 (2003): 2.

[3] Goenawan, Ivan H., Kenneth Bryan, and David J. Lynn. "DyNet: visualization and analysis of dynamic molecular

interaction networks." Bioinformatics 32.17 (2016): 2713-2715.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index