FriendlyNets

Release 0.2

James D. Brunner and Nicholas Chia

Jun 29, 2023

CONTENTS:

FriendlyNets provides a method for assessing the promotion/inhibition effect on a microbe of a microbial community using a network of community interactions. At its core, FriendlyNets judges how much a network promotes or inhibits one of its nodes. It does this by assuming a set dynamical system represented by the network, and using the resulting dynamics. FriendlyNets is also packaged with functions for generating a network from a set of genome-scale metabolic by simulating pairwise growth using the methods from [KSC22].

Check out the *Usage* section for further information, including how to *install* the project.

Note: This project is under active development.

CHAPTER

ONE

USAGE

1.1 Installation

To use FriendlyNets, clone from github:

```
$ git clone https://github.com/jdbrunner/friendlyNets.git
```

(We plan to add pip installation in the future)

You will also need to add the directory to your python path, for example using

```
import sys
import os
sys.path.append(os.path.join(os.path.expanduser("~"),location,"friendlyNets"))
```

where `location` is the path to the folder that you cloned friendlyNets into.

Dependencies

We use joblib for parallel computing.

Our implementation of SteadyComX uses Gurobi and gurobi's python package for joint FBA, and expects cobrapy models as input.

Note: Unfortunately, we do not currently have access to a CPLEX license in order to implement CPLEX as a solver option.

1.2 Using the Method for a Set of Samples

Note: What follows is a plan of action. At this point, the data formatting and network building steps are incomplete.

FriendlyNets is designed to predict the invasion of a single species into a community. It is designed around microbiome studies, but written more generally so that nodes can be anything as long as interaction parameters between the nodes are known. For microbiome studies, we provide a method to generate the set of interaction parameters from a set of genome-scale metabolic models which must be provided by the user.

We assume that the user has a table in .csv format with rows indexed by species of interest and columns indexed by sample name that contains abundance data for each species in each sample (relative or absolute). Additionally, if an assessment of the predictive power of the method is desired (which may indicate the extent to which experimental

outcome depends network effects), a metadata file must be provided as .csv with rows indexed by sample name and a column indicating experimental outcome of an invasion experiment

1.2.1 Creating the full network

Note: Optional: A network can instead be loaded as pandas DataFrame with nodes and columns being names of the species in the data, formatted as `Network.loc[m_1,m_2]` contains the interaction with source `m_1` and target `m_2`.

For microbiome data paired with a set of genome-scale metabolic models, the method creates network of interactions for all co-occurring taxa that have an associated user-provided genome-scale model.

```
from make_gem_network import make_gem_network

full_interaction_network = make_gem_network(path_to_gsm_info)
```

This function requires the path to a file containing the paths to each genome-scale model.

By default, the interaction will be defined by the log-ratio of simulated growth in the pair to simulated growth alone.

1.2.2 Formatting the data

Note: Not Implemented

Warning: Removes any species from the data that do not have interaction parameters in the full interaction network.

We need to format the samples into a dict of dicts, or dict of tuples if predicitive power is to be assessed.

To format the data to compute friendliness scores with no known outcomes:

```
from format_data import format_data

experiment,scoretype = format_data(path_to_table,full_interaction_network,nodes_of_
↪interest)
```

The function requires the path to the table of abundances and the full interaction network so that species missing can be removed, and the name of the node(s) that we wish to assess for network friendliness.

Without known outcomes, `experiment` will be a dict (keyed by sample name - the column headers of the abundance table) of dicts (keyed by species), and `scoretype` will be None

For assessment of the predictive power of the method, the path to a metadata file with known outcomes for each sample is required, as is the name of the known outcome column in that file (default `Score`):

```
from format_data import format_data

experiment,scoretype = format_data(path_to_table,full_interaction_network,nodes_of_
↪interest,known_scores = path_to_metadata,score_column = column_of_score,scoretype =␣
↪score_type)
```

If the known outcome file is given, along with the name of the column of known outcome scores, `experiment` will be a dict (keyed by sample name - the column headers of the abundance table) of tuples with (known outcome score, dict of abundances). The dict of abundances is keyed by species. In this case, the function attempts to guess if the known scores are binary or continuous unless the scoretype is given. If the scoretype is given as binary and the data are continuous, the function binarizes the data.

The second return value, `scoretype` indicates the type of known outcome scores, either binary or continuous.

1.2.3 Computing Friendliness Scores and Assessing Predictive Power

To compute friendliness scores for each sample on a node(s) of interest

```
from score_net import network_friendliness

friendliness = {}
for target_node in nodes_of_interest:
    friendliness[target_node] = network_friendliness(experiment,full_interaction_network,
↪target_node)
```

The return value is a pandas dataframe that can be saved as a .csv file.

To assess the predictive power of the method (for friendliness on a single `target_node` in `nodes_of_interest`)

```
from score_net import score_net

friendliness,predictive_power = score_net(experiment,full_interaction_network,target_
↪node,scoretype)
```

`predictive_power` is a dictionary of predictive power metrics, which depend on if the scoring is binary (in which case the ROC is used) or continuous (in which case correlation is used).

1.2.4 Plotting the Results

Note: I will probably add some functions to make plotting the results convenient.

1.2.5 Sensitivity to Parameters

We also provide functionality to assess the sensitivity of the predictions to two types of perturbations.

The first is sensitivity to community composition, which we test using simulated knock-outs (i.e. computing friendliness scores with nodes removed).

The second is sensitivity to the interaction parameter values. We test this using a dynamical system for $\frac{\partial x_i}{\partial a_{ij}}$. See *Functions for Sensitivity Testing*.

CHAPTER

TWO

FUNCTIONS FOR FORMATTING OTU TABLES

CHAPTER

THREE

FUNCTIONS FOR MAKING INTERATION NETWORKS FROM GEMS

make_gem_network.**check_co_occ**(*experiment*, *min_ra=1e-06*)

Check which nodes co-occurr in samples, so that we don't need to compute an interaction between those that don't

Parameters

- **experiment** (*dict[tuple[float,dict]]*) – set of sets of nodes, as a dictionary of tuples with (known score,data) keyed by sample identifier. The data should be a dictionary of abundances keyed by node names.
- **min_ra** (*float*) – cutoff to use for presence/abscence of a node in a sample. Default 10**-6

Returns

NxN array with bool indicating if the pair co-occures, and list of nodes giving ordering of this array.

Return type

array[bool],list[str]

CHAPTER

FOUR

FUNCTIONS TO GENERATE AND EVALUATE PREDICTIONS

score_net.**network_friendliness**(*experiment*, *full_net*, *target_node*, *models=None*, *min_ra=1e-06*, *odeTrials=None*)

This method computes the extent to which a set of sets of nodes are friendly to a particular node, using `friendlyNets`.

Parameters

- **experiment** (*dict[dict[float]]*) – Dictionary of samples. Each should be a dictionary of abundances keyed by node names. (Also supports dict of tuples as in *score_net*
- **full_net** (*pandas dataframe*) – Adjacency matrix of all interaction parameters between pairs of nodes in the entire experiment set.
- **target_node** (*str*) – name of node of interest. (must be in full_net, need not be in sample - will be added to induced subgraph)
- **models** (*list*) – list of models you wish to use for scoring. Leave as None for all 6.
- **min_ra** (*float*) – cutoff to use for presence/abscence of a node in a sample. Default 10**-6
- **odeTrials** (*int*) – number of ODE simulations in score estimation. If None, equal to number of non-zero taxa in a sample. Default None

Returns

friendliness scores for each sample and each model in models.

Return type

pandas dataframe

score_net.**score_net**(*experiment*, *full_net*, *target_node*, *scoretype*, *models=None*, *min_ra=1e-06*, *odeTrials=None*)

This method computes the extent to which a set of sets of nodes are friendly to a particular node, using `friendlyNets`, and **compares** this with a known effect of the community, which can be binary (good/bad) or continuous (e.g. relative abundance at a later time point) to compute predictive performance

Parameters

- **experiment** (*dict[tuple[float,dict]]*) – Dictionary of samples, each a tuple with (known score,data). The data should be a dictionary of abundances keyed by node names.
- **full_net** (*pandas dataframe*) – Adjacency matrix of all interaction parameters between pairs of nodes in the entire experiment set.
- **target_node** (*str*) – name of node of interest. (must be in full_net, need not be in sample - will be added to induced subgraph)
- **scoretype** (*str*) – Type of known score (*b* for binary, *c* for continuous)

- **models** (*list*) – list of models you wish to use for scoring. Leave as None for all 6.
- **min_ra** (*float*) – cutoff to use for presence/abscence of a node in a sample. Default 10**-6
- **odeTrials** (*int*) – number of ODE simulations in score estimation. If None, equal to number of non-zero taxa in a sample. Default None

Returns

friendliness scores, predictive performance dictionary. If binary scoring, predictive performance is AUROC, ROC curves, and the mean AUROC.If continuous scoring, this is pearson correlation between friendliness score and known score, pearson p valule, kendall correlation, and kendall p value,spearman correlation, spearman p value. Correlation values are rescaled to [0,1] (from [-1,1]) to better match AUCROC scores.

Return type

tuple of pandas dataframe, {dict, dict, float} OR tuple of pandas dataframe, {dict,dict,dict,dict,dict,dict}

score_net.**score_light**(*experiment*, *full_net*, *target_node*, *scoretype*, *score_model*, *self_inhibit=0*, *min_ra=1e-06*, *odeTrials=None*, *lvshift=0*, *cntbu=False*, *keepscores=False*, *KO=None*)

This method computes the extent to which a set of sets of nodes are friendly to a particular node, using `friendlyNets`, and compares this with a known effect of the community, which can be binary (good/bad) or continuous (e.g. relative abundance at a later time point). This version only computes a score for a single type of model, but offers more parameter flexibility, including Shift and Self Inhibition in the *Lotka-Volterra system*

Parameters

- **experiment** (*dict[tuple[float,dict]]*) – set of sets of nodes, each a tuple with (known score,data). The data should be a dictionary of abundances keyed by node names.
- **full_net** (*pandas dataframe*) – Adjacency matrix of all interaction parameters between pairs of nodes in the entire experiment set.
- **target_node** (*str*) – name of node of interest. (must be in full_net, need not be in sample - will be added to induced subgraph)
- **scoretype** (*str*) – Type of known score (*b* for binary, *c* for continuous)
- **score_model** (*str*) – Dynamical model to use for scoring. Choices LV, AntLV, InhibitLV, Replicator, NodeBalance, Stochastic, Composite, as detailed in *score_node*
- **self_inhibit** (*float*) – extent to which the model should include self inhibition - self interaction terms (A_{ii}) will be set to negative `self_inhibit`. Default 0
- **min_ra** (*float*) – cutoff to use for presence/abscence of a node in a sample. Default 10**-6
- **odeTrials** (*int*) – number of ODE simulations in score estimation. If None, equal to number of non-zero taxa in a sample. Default None
- **lvshift** (*float*) – Uniform (subtracted) modifier to interactions. Lotka-Volterra parameters will be *Adjacency* - `shift`. Default 0
- **cntbu** (*bool*) – Whether or not to count the number of blow-ups in the simulations. Default False
- **keepscores** (*bool*) – Wether or not to return the friendliness scores. If False, only returns the predictive performance.
- **KO** (*str*) – Knockout nodes to remove from data

Returns

evaluation of prediction, as AUCROC or (kendall, spearman), optionally count of ODE blowups, and optionally sample ordering, friendliness scores (in that order)

Return type

tuple float (or float,float), optional float, optional list[str], optional array[float]

CHAPTER

FIVE

FUNCTIONS FOR SENSITIVITY TESTING

5.1 Functions for Lotka-Volterra Interaction Parameter Sensitivity Testing

sensitivity.**get_all_sensitivity**(*target_node*, *fnet*, *entries='all'*, *shift=0*, *self_inhibit=0*, *numtrials=100*, *nj=1*, *mxTime=1000*, *wpts=40*, *base_we=1.5*)

Compute sensitivity of a node to interaction parameters, and average over simulations. Computes weighted average, using *final* `wpts` timepoints with increasing weight. Weight s is computed as b^s where b is `base_we` and then weights are rescaled to sum to 1.

Parameters

- **target_node** (*str or int*) – name node of interest (nodes are usually named with str, but can be named with other objects, most commonly int equal to node index.)
- **fnet** (*friendlyNet*) – network of interactions for the model
- **entries** (*list[tuple[str,str]]*) – Which interaction parameters to test sensitivity to (names). If 'all', tests for every parameter. Otherwise, should be a list of tuples of names (source,target). Default 'all'
- **shift** (*float*) – Uniform (subtracted) modifier to interactions. Lotka-Volterra parameters will be *Adjacency* - `shift`
- **self_inhibit** (*float*) – extent to which the model should include self inhibition - self interaction terms (a_{ii}) will be set to -`self_inhibit`
- **numtrials** (*int*) – Number of simulations to average over
- **nj** (*int*) – number of trials to run in parallel (using joblib)
- **mxTime** (*float*) – time length of simulations
- **wpts** (*int*) – number of time-points in each simulation to average over (will be final time-points). Default 40
- **base_we** (*float*) – base for weights of time-averaging - should be > 1 for increasing weight, equal to 1 for uniform weight on last `wpt` time-points. Default 1.5

Returns

Average value of $\partial x_i/\partial a_{kl}$ averaged first over time points in each simulation and next over simulations. If `pars` == 'all', returns NxN array indexed by [source,target]. Otherwise, returns 1d array corresponding to 'pars'

Return type

float

sensitivity.**get_all_sensitivity_single_trajectory**(*fnet*, *i*, *shift=0*, *self_inhibit=0*, *weights=None*, *mxTime=1000*, *pars='all'*)

Compute sensitivity of node i to every interaction parameter. To do this, we need to solve an ODE that arises from the chain rule. Because we can reuse the solution to the lotka-volerra system, we don't want to repeatedly call get_sensitivity on a each parameter.

Parameters

- **fnet** (*friendlyNet*) – network of interactions for the model
- **i** (*int*) – index of node of interest
- **shift** (*float*) – Uniform (subtracted) modifier to interactions. Lotka-Volterra parameters will be *Adjacency* - shift
- **self_inhibit** (*float*) – extent to which the model should include self inhibition - self interaction terms (a_{ii}) will be set to -self_inhibit
- **weights** (*array[float]*) – weights for time-points of the ODE solution. This allows us to weight the later timepoints (closer to equilibrium) higher, or not, with some granularity. Default unweighted.
- **mxTime** (*float*) – time length of simulations
- **pars** (*list[tuple[int,int]]*) – Which interaction parameters to test sensitivity to. If 'all', tests for every parameter. Otherwise, should be a list of tuples of indices (source,target). Default 'all'

Returns

(possibly weighted) average value of $\partial x_i/\partial a_{kl}$ over time points in simulation for each. If pars == 'all', returns NxN array indexed by [source,target]. Otherwise, returns 1d array corresponding to 'pars'

Return type

array[float]

sensitivity.**get_sensitivity_single_trajectory**(*fnet*, *i*, *k*, *l*, *soln=None*, *shift=0*, *self_inhibit=0*, *weights=None*, *mxTime=1000*)

Compute sensitivity of node i to parameter a_{kl}. To do this, we need to solve an ODE that arises from the chain rule.

Parameters

- **fnet** (*friendlyNet*) – network of interactions for the model
- **i** (*int*) – index of node of interest
- **k** (*int*) – source node of interaction of interest
- **l** (*int*) – target node of interaction of interest
- **soln** (*scipy.integrate.solve_ivp object*) – solution to lotka-volterra system *must use same shift and self_inhibit parameters*
- **shift** (*float*) – Uniform (subtracted) modifier to interactions. Lotka-Volterra parameters will be *Adjacency* - shift
- **self_inhibit** (*float*) – extent to which the model should include self inhibition - self interaction terms (a_{ii}) will be set to -self_inhibit
- **weights** (*array[float]*) – weights for time-points of the ODE solution. This allows us to weight the later timepoints (closer to equilibrium) higher, or not, with some granularity. Default unweighted.

- **mxTime** (*float*) – time length of simulations

Returns
(possibly weighted) average value of $\partial x_i / \partial a_{kl}$ over time points in simulation.

Return type
float

sensitivity.**sense_kl**(*t*, *ps*, *x*, *k*, *l*, *net*)

Dynamical system for senstivity $\partial x_i / \partial a_{kl}$ for a Lotka-Volterra interaction parameter.

Parameters

- **t** (*float*) – time in simulation
- **ps** (*array[float]*) – state in simulatoin
- **x** (*function*) – Lotka-Volterra solution (dense output of scipy.integrate.solve_ivp)
- **k** (*int*) – index of source of interaction
- **l** (*int*) – index of target of interaction
- **net** (*array[float]*) – adjacency matrix for Lotka-Volterra system

Returns
right-hand side of dynamical system

Return type
array[float]

sensitivity.**compute_j**(*x*, *net*)

Helper function to compute a term in the dynamical system defined by *sense_kl*.

Parameters

- **x** (*array[float]*) – Lotka-Volterra state at time t
- **net** (*array[float]*) – adjacency matrix for Lotka-Volterra system

Returns
term in RHS

Return type
array[float]

CHAPTER

SIX

THE FRIENDLYNETS CLASS

This is the core class of the method, a network class built from an adjacency matrix that contains methods for measuring friendliness to the nodes.

class friendlyNet.**friendlyNet**(*adj*)

Network class designed for testing how positive a directed network is for a node. Provides several ways to score the network's friendliness for a given node. By friendliness, we mean some measure of how much the network promotes/inhibits a node based on a choice of interaction model represented by the graph (e.g. Lotka-Volterra or Diffusion). Rescales the adjacency matrix so that all weights are in $[0, 1]$

Parameters

adj (*(N,N) array[float]*) – Adjacency matrix of the graph.

Adjacency

Network adjacency matrix, rescaled so that all weights are in $[0, 1]$

NodeNames

List of names of the nodes, corresponding to ordering of adjacency matrix

InDegree

Total weight of edges into each node

OutDegree

Total weight of edges out of each node

NodeScores

Pandas DataFrame of friendliness scores for each node for a set of chosen dynamical models.

EdgeList

Pandas DataFrame of edges (with weights) for the network that can be easily saved and loaded into cytoscape. See *make_edge_list*

make_edge_list()

Create a list of edges with columns:

- Source
- Target
- Weight
- ABS_Weight (absolute value)
- Sign_Weight

Modifies

- *EdgeList*

Returns
None

lotka_volterra_system(*t*, *s*, *shift*, *self_inhibit*)

Right-Hand side of generalized Lotka-Volterra dynamical system, using *Adjacency* for interactions:

$$\dot{x}_i = x_i(1 + \sum_{j=1}^{N} a_{ij}x_j)$$

Parameters

- **t** (*float*) – time in simulation
- **s** (*array[float]*) – State of simulation (species abundance)
- **shift** (*float*) – Uniform (subtracted) modifier to interactions. Lotka-Volterra parameters will be *Adjacency* - shift
- **self_inhibit** (*float*) – extent to which the model should include self inhibition - self interaction terms (a_{ii}) will be set to -self_inhibit

Returns
value of vector field at t,s

Return type
array[float]

solve_lotka_volterra(*s0*, *T*, *shift=0*, *bup=1000*, *self_inhibit=0*)

Solves the generalized Lotka-Volterra dynamical system, using *Adjacency* for interactions

Parameters

- **s0** (*array[float]*) – Inital state of simulation (species abundance)
- **T** (*float*) – Simulation length
- **shift** (*float*) – Uniform (subtracted) modifier to interactions. Lotka-Volterra parameters will be *Adjacency* - shift
- **bup** (*float*) – maximum abundance to allow in simulation. If any state variable reaches this value, it is assumed that the simulation has exhibited finite-time blowup and the simulation is stopped.
- **self_inhibit** (*float*) – extent to which the model should include self inhibition - self interaction terms (A_{ii}) will be set to negative self_inhibit

Returns
Solution to the Lotka-Volterra dynamics

Return type
scipy.integrate.solve_ivp solution

lotka_volterra_score_single(*node*, *mxTime=100*, *shift=0*, *self_inhibit=0*)

Uses the Lotka-Volterra dynamical system to determine the network's friendliness to a particular node. Solves the system using *solve_lotka_volterra*. Solves the Lotka-Volterra system with random initial conditions and computes a score. The score is based on final relative abundance, but for finer scoring we also account for time to extinction and time to domination (e.g. relative abundance near 1). The score is computed as

where T_e is the proportion of the time internal that the species is *not* extinct, T_d is the proportion of the time interval that the species *is* dominant, and r is the final relative abundance of the species.

Parameters

- **node** (*str or int*) – name or index of node
- **mxTime** (*float*) – time length of simulations
- **shift** (*float*) – Uniform (subtracted) modifier to interactions. Lotka-Volterra parameters will be *Adjacency* - `shift`
- **self_inhibit** (*float*) – extent to which the model should include self inhibition - self interaction terms (A_{ii}) will be set to negative `self_inhibit`

Returns
Friendliness of the network to the node, according to the single Lotka-Volterra simulation, and the status of the ODE solution

Return type
tuple[float,str]

lotka_volterra_score(*node*, *mxTime=100*, *numtrials=1000*, *nj=-1*, *shift=0*, *self_inhibit=0*, *cntbu=False*)

Provides a score using repeated trials of the Lotka-Volterra system. Scores using *lotka_volterra_score_single*

Parameters

- **node** (*str or int*) – name or index of node
- **mxTime** (*float*) – time length of simulations
- **numtrials** (*int*) – Number of ODE solutions and corresponding scores to compute
- **nj** (*int*) – Number of parrallel simulations to run concurrently (uses joblib)
- **shift** (*float*) – Uniform (subtracted) modifier to interactions. Lotka-Volterra parameters will be *Adjacency* - `shift`
- **self_inhibit** (*float*) – extent to which the model should include self inhibition - self interaction terms (A_{ii}) will be set to negative `self_inhibit`
- **cntbu** (*bool*) – Whether or not to count the number of blow-ups in the simulations

Returns
score from *lotka_volterra_score_single* averaged over all trials, optinoally number of blowups in simulation

Return type
float,int

replicator_system(*t*, *s*)

Right-Hand side of replicator dynamical system, using *Adjacency* for interactions:

$$\dot{x}_i = x_i(\sum_{j=1}^{N} a_{ij}x_j - x^T Ax)$$

Parameters

- **t** (*float*) – time in simulation
- **s** (*array[float]*) – State of simulation (species abundance)

Returns
value of vector field at t,s

Return type
array[float]

solve_replicator(*s0*, *T*)

Solves the replicator dynamical system, using *Adjacency* for interactions

Parameters

- **s0** (*array[float]*) – Inital state of simulation (species abundance)
- **T** (*float*) – Simulation length

Returns
Solution to the replicator dynamics

Return type
scipy.integrate.solve_ivp solution

replicator_score_single(*node*, *mxTime=100*)

Uses the replicator dynamical system to determine the network's friendliness to a particular node. Solves the system using *solve_replicator*. Solves the replicator system with random initial conditions and computes a score. The score is based on final relative abundance, but for finer scoring we also account for time to extinction and time to domination (e.g. relative abundance near 1). The score is computed as

where T_e is the proportion of the time internal that the species is *not* extinct, T_d is the proportion of the time interval that the species *is* dominant, and r is the final relative abundance of the species.

Parameters

- **node** (*str or int*) – name or index of node
- **mxTime** (*float*) – time length of simulations

Returns
Friendliness of the network to the node, according to the single replicator simulation, and the status of the ODE solution

Return type
tuple[float,str]

replicator_score(*node*, *mxTime=100*, *numtrials=1000*, *nj=-1*)

Provides a score using repeated trials of the replicator system. Scores using *replicator_score_single*

Parameters

- **node** (*str or int*) – name or index of node
- **mxTime** (*float*) – time length of simulations
- **numtrials** (*int*) – Number of ODE solutions and corresponding scores to compute
- **nj** (*int*) – Number of parrallel simulations to run concurrently (uses joblib)

Returns
score from *lotka_volterra_score_single* averaged over all trials

Return type
float

node_balanced_score(*node*)

Provides a score based on the linear system

where $L = A^T - D$ is the graph laplance matrix for the graph after weights have been rescaled to the interval $[0, 1]$. This dynamical system arises from the notion of node-balancing the graph. Because this is linear, we can use the dominant eigenvector of the laplacian to compute equilibrium.

Parameters
node (*str or int*) – name or index of node

Returns
Value of node in dominant eigenvector (i.e. equilibrium solution)

Return type
float

node_balanced_system(*T*)

Function to provide a simulation for the node-balancing linear system

where $L = A^T - D$ is the graph laplance matrix for the graph after weights have been rescaled to the interval $[0, 1]$.

Parameters
T (*float*) – End time of simulation

Returns
Solution to the node balance dynamics

Return type
scipy.integrate.solve_ivp solution

stochastic_score(*node*)

Provides a score based on the linear system that simulates concurrent random walks (or, equivalently, diffusion) on the graph. We rescale the adjacency matrix to build a stochastic matrix that represents the transition probabilities in a random walk on the graph. The eigenvectors of this matrix provide a stationary distribution for the cuncurrent random walks.

Parameters
node (*str or int*) – name or index of node

Returns
Value of node in stationary distribution

Return type
float

score_node(*node*, *scores=None*, *odeTrials=None*)

Function to score a node using a set of scores. Choose any list of the following:

- **LV** The *Lotka-Volterra system*
- **InhibitLV** *Lotka-Volterra system* with self inhibition = 1
- **AntLV** The *Lotka-Volterra system* with all interactions shifted by -1 to make them antagonistic.
- **Replicator** The *replicator equation dynamics*
- **NodeBalance** The linear *node balancing dynamical system*
- **Stochastic** The linear *random walk dynamical system*

A composite score will be included, which is simply the mean of each score included.

Parameters

- **node** (*int or str*) – name or index of node
- **scores** (*list*) – list of score types you wish to use. Leave as None for all 6.
- **odeTrials** (*int*) – Number of simulations of the Lotka-Volterra and replicator dynamics to use to estimate the score. Leave as None for number of trials equal to number of nodes in the network.

Returns
Dictionary of scores keyed by score type.

Trype
dict[str,float]

`score_all_nodes`(*scores=None*, *odeTrials=None*)

Function to score all nodes using a set of scores. Choose any list of the following:

- **LV** The *Lotka-Volterra system*
- **InhibitLV** *Lotka-Volterra system* with self inhibition = 1
- **AntLV** The *Lotka-Volterra system* with all interactions shifted by -1 to make them antagonistic.
- **Replicator** The *replicator equation dynamics*
- **NodeBalance** The linear *node balancing dynamical system*
- **Stochastic** The linear *random walk dynamical system*

A composite score will be included, which is simply the mean of each score included.

Parameters

- **scores** (*list*) – list of score types you wish to use. Leave as None for all 6.
- **odeTrials** (*int*) – Number of simulations of the Lotka-Volterra and replicator dynamics to use to estimate the score. Leave as None for number of trials equal to number of nodes in the network.

Returns
Table of scores for each node and score type chosen

Trype
pandas dataframe

Modifies

- *NodeScores*

BIBLIOGRAPHY

[KSC22] Minsuk Kim, Jaeyun Sung, and Nicholas Chia. Resource-allocation constraint governs structure and function of microbial communities in metabolic modeling. *Metabolic Engineering*, 70:12–22, 2022.

INDEX