MultiModBP Documentation

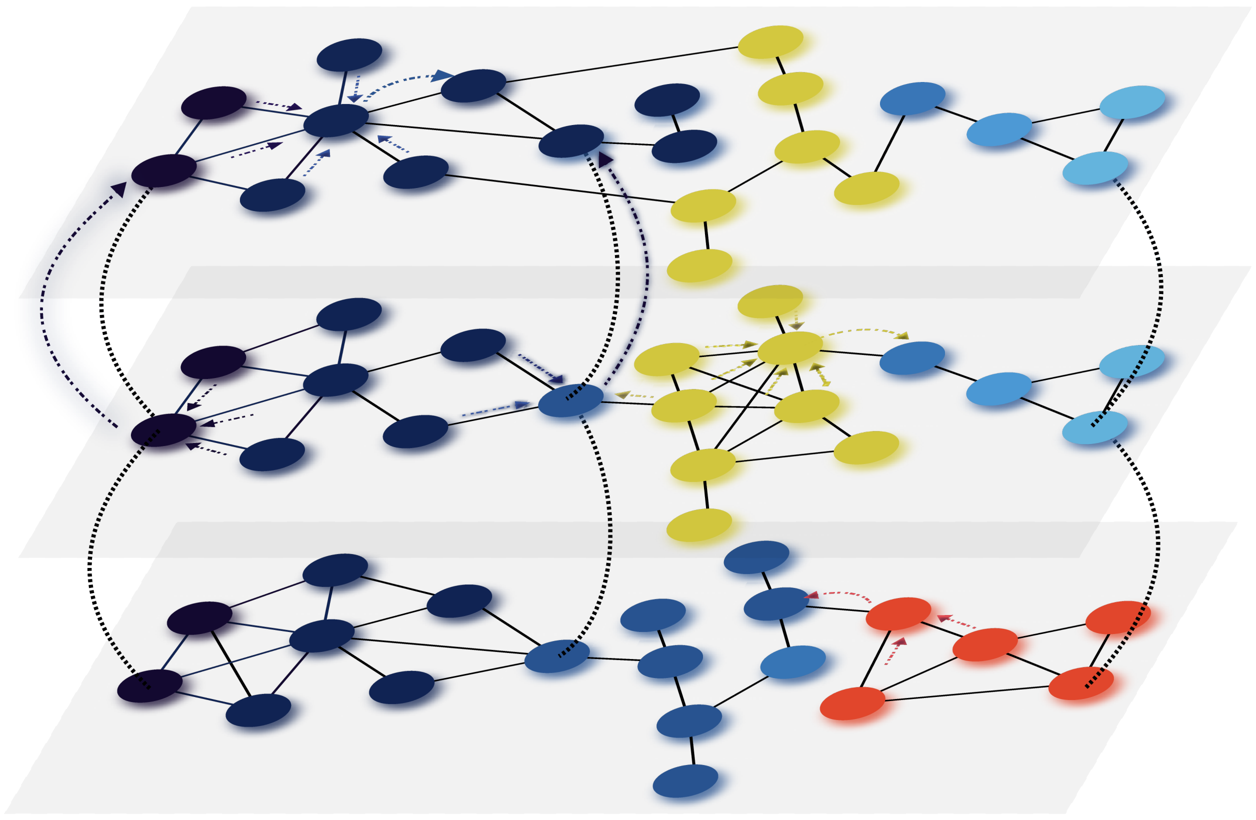
Release 1

William Weir

Aug 13, 2019

Contents

A belief propagation solution to multilay modularity community detection.



We have implemented a belief propagation solution for multilayer modularity in both C++ and Python. Our implementation allows for both weighted and unweighted single layer networksas well as a variety of multilayer topologies. The C++ backend provides significant performance increaseand allows for running the algorithm at larger scale networks. Our method extends the approach of Pan Zhang and Christopher Moore [1] and provides a convenient interface with the standard networks analysis library, igraph.

CHAPTER 1

Contents:

1.1 Background

We present *multimodbp*, a belief propagation implementation of multilayer modularity for communtiy detectionOur tool is a C++ based implentation of the belief propagation algorithm (also called the sum-product or message passing algorithm) wrapped in Python for convenient access and execution.

1.1.1 Introduction: Belief Propagation

1.1.2 Modularity

1.1.3 Multilayer Modularity

One of the strengths of modularity is that it has been extended in a principled way into a variety of network topologies in particular the multilayer context. The multilayer formulation [1] for modularity incorporates the interlayer connectivity of the network in the form of a second adjacency matrix C_{ij}

$$Q(\gamma) = \frac{1}{2m} \sum_{i,j} \left(A_{ij} - \gamma \frac{k_i k_j}{2m} + \omega C_{ij} \right) \delta(c_i, c_j) \tag{1.1}$$

Communities in this context group nodes within the layers and across the layers. The inclusion of the $C_i j$ boost the modularity for communites that include alot interlayer links. There is an additional parameter, ω that tunes how much weight these interlink ties contribute to the modularity.

References

- genindex
- search

1.2 Multilayer Graph Class

We have created a Python class to store the graph information used by the multilayer belief propagation.the same class can be used for both the single layer and the multilayer case. In addition we have created a class for generating realization of the Dynamic Stochastic Block Model.

1.2.1 Multilayer Graph

class modbp.**MultilayerGraph**(*intralayer_edges*, *layer_vec*, *interlayer_edges=None*, *comm_vec=None*, *directed=False*)

Wrapper class for storing a 'multilayer graph' that can be used to call the modularity belief propagation A graph here is represented by a collection of igraphs (each one representing a "layer") as well as a set of edges between the layers. In this formulation, each node can only be present in a single layer

__init__(*intralayer_edges*, *layer_vec*, *interlayer_edges=None*, *comm_vec=None*, *directed=False*)

Parameters

- **intralayer_edges** – list of intralayer edges between the nodes. If intralayer_edges.shape[1] > 2 intralayer_edges[:,2] is assumed to represent the weights of the edges. Default weight is 1.
- **layer_vec** – vector denoting layer membership for each edge. Size of network is taken to be len(layer_vec)
- **interlayer_edges** – list of edges across layers. If interlayer_edges.shape[1] > 2 interlayer_edges[:,2] is assumed to represent the weights of the edges. Default weight is 1.
- **comm_vec** – Underlying known communitiies of the network. Default is None
- **directed** – Are intralayer and interlayer edges directed. #TODO: allow one or the other to be directed.

__weakref__

list of weak references to the object (if defined)

get_AMI_layer_avg_with_communities(*labels*)

Calculate AMI of each layer with corresponding community in labels. Return average AMI weighted by number of nodes in each layer. :param labels: commmunity to assess agreement with. Len(labels) must equal self.N :return:

get_AMI_with_communities(*labels*)

Calculate adjusted mutual information of labels with underlying community of network. :param labels: commmunity to assess agreement with. Len(labels) must equal self.N :return:

get_accuracy_with_communities(*labels*, *permute=True*)

Calculate accuracy between supplied labels and the known communities of the networks.

Parameters

- **labels** – commmunity to assess agreement with. Len(labels) must equal self.N
- **permute** – Should maximum accurracy across label permuations be identified?

Returns

get_layer_edgecounts()

m for each layer

plot_communities(*comvec=None*, *layers=None*, *ax=None*, *cmap=None*)
Plot communities as an nlayers by nodes/layer heatmap. Note this only works for the multiplex case where the number of nodes is fixed throughout each layer.

Parameters

- **comvec** – community label for each nodes. If none, used stored ground truth for the network.
- **layers** – Subset of the layers to plot. If None, plots all layers.
- **ax** – matplotlib.Axes to draw on
- **cmap** – color map to label communities with. Defaults to cube_helix.

Returns

reorder_nodes()
Resort all objects in the MultilayerGraph object by their community label :return:

to_scipy_csr()
Create sparse matrix representations of the multilayer network.

Returns (A_sparse,C_sparse) = interlayer adjacency , interlayer adjacency

1.2.2 Random Graph Classes

class modbp.**MultilayerSBM**(*n*, *comm_prob_mat*, *layers=2*, *transition_prob=0.1*, *block_sizes0=None*, *use_gcc=False*)
Subclass of MultilayerGraph to create the dynamic stochastic block model from Ghasemian et al. 2016.

__init__(*n*, *comm_prob_mat*, *layers=2*, *transition_prob=0.1*, *block_sizes0=None*, *use_gcc=False*)

Parameters

- **n** – number of nodes in each layer
- **comm_prob_mat** – probability of block connections in SBM(this is fixed across all layers)
- **layers** – number of layers
- **transition_prob** – probability of each node changing communities from one layer to next. if transistion_prob=0 then community structure is constant across all layers.
- **block_sizes0** – Initial size of the blocks. Default is to use even block sizes starting out. block sizes at subsequent layers are determined by number of nodes that randomly transition between communities
- **use_gcc** – use only giant connected component of starting SBM (option for single layer only)

get_all_layers_block()
returns a single vector with block id for each node across all of the layers :return:

get_interlayer_adj()
Singer interlayer adjencency matrix created by connecting each node to it's equivalent node in the next layer. This is a temporal multiplex topology. :return: np.array of interlayer adjacency representation

get_interlayer_edgelist()
Single list of edges giving the multilayer connections. For this model nodes are multiplex an connected to their neighboring slice identities.

Returns np.array

get_intralayer_adj()

Single adjacency matrix representing all layers. :return: np.array of intralayer adjacency representation

get_intralayer_edgelist()

Single list of edges treating the group of single layer SBM's as a surpra-adjacency format

Returns np.array

get_node_layer_vec()

Returns np.array denoting which layer each node is in

- genindex
- search

1.3 ModularityBP Object

We have created a single Python object, `modbp.ModularityBP`

1. *Store graphs on ModularityBP Object*
2. *Run modularity belief propagation*
3. *Access discovered partitions and marginals*
4. *Compile results across many different runs*

1.3.1 Storing Graphs on ModularityBP Object

A representation of a network must be supplied at the instantiation of the `modbp.ModularityBP` object. Each instance can only be associated with one network. A network can be supplied in several different formats:

- A `igraph.Graph` object is supplied. In this case network will be treated as single layer. This is pass in through the mlgraph parameter:

```
rand_g=igraph.Graph.ErdosRenyi(n=100,p=.05)
modbp_obj=modbp.ModularityBP(mlgraph=rand_g)
```

- A *`modbp.MultilayerGraph`* object can be supplied, also through the mlgraph parameter. See *Multilayer Graph* for more details.
- Finally, one can pass in an array for intralayer_edges, interlayer_edges, and layer_vec, from which `modbp.ModularityBP` will internally construct a *`modbp.MultilayerGraph`*:

```
intra_edges=np.array([[0,1],[0,2],[3,4]])
inter_edges=np.array([[2,3]])
layer_vec=[0,0,0,1,1]
modbp_obj=modbp.ModularityBP(intra_edges=intra_edges,
    inter_edges=inter_edges,
    layer_vec=layer_vec)
```

The graph is stored internally each `modbp.ModularityBP` as a *`modbp.MultilayerGraph`* and is accessible through `modbp.ModularityBP.graph` variable.

1.3.2 Running Multilayer Modularity Belief Propagation

`ModularityBP.`**`run_modbp`**(*beta*, *q*, *niter=100*, *resgamma=1.0*, *omega=1.0*, *reset=False*)

Parameters

- **`beta`** – The inverse tempature parameter at which to run the modularity belief propagation algorithm. Must be specified each time BP is run.
- **`q`** – The number of mariginals used for the run of modbp. Note that if self.use_effective is true, The final number of reported communities could be lower.
- **`niter`** – Maximum number of iterations allowed. If self._align_communities_across_layers is true, the actual number of runs could be higher than this upper bound though at most 2*niter
- **`resgamma`** – The resolution parameter at which to run modbp. Default is resgamma=1.0
- **`omega`** – The coupling strength used in running multimodbp. This represent how strongly the algorithm tries to assign nodes connected by an interlayer connection to the same community.
- **`reset`** – If true, the marginals will be rerandomized when this method is called. Otherwise the state will be maintained from previous runs if existing (assuming q hasn't changed).

Returns None

1.3.3 Accessing partitions and marginals

1.3.4 Assessing Results Across Many Different Runs

- genindex
- search

1.4 References

bibtex

- genindex
- search

CHAPTER 2

Download and Installation:

The *modbp* module is hosted on PyPi. The easiest way to install is via the pip command:

```
pip install modbp
```

For installation from source, the latest version of modbp can be downloaded from GitHub:

> Multimodbp Github

For basic installation:

```
python setup.py install
```

2.1 Dependencies

To make our code run as quickly as possible, the underlying belief propagation algorithm has been written in C++. Wrapping and interfacing this code with the Python tools requires swig, a tool for creating Python classes from C++ objects.

The python dependencies for *modbp* are fairly standard tools for data analysis in Python:

- NumPy : Python numerical analysis library.
- sklearn :Machine learning tools for python.
- python-igraph :igraph python version for manipulation of networks.
- matplotlib :Python data visualization library.
- pandas :data structures and data analysis tools for python.

These should all be handled automatically if using pip to install.

We are also working on creating a conda recipe for easy installation through conda forge.

CHAPTER 3

Citation

bibtex

For more details and results see our manuscript

- genindex
- search

CHAPTER 4

Acknowledgements

Bibliography

[1] Peter J Mucha, Thomas Richardson, Kevin Macon, Mason A Porter, and Jukka-Pekka Onnela. Community structure in time-dependent, multiscale, and multiplex networks. *Science*, 328(5980):876–878, May 2010.

[1] Pan Zhang and Cristopher Moore. Scalable detection of statistically significant communities and hierarchies, using message passing for modularity. *Proceedings of the National Academy of Sciences*, 111(51):18144–18149, 2014.

Index