Using the Python package `motifcluster`

William G. Underwood

April 29, 2020

Contents

1 Introduction

This tutorial demonstrates how to use the Python package `motifcluster`. These methods are detailed in the paper *Motif-Based Spectral Clustering of Weighted Directed Networks*, which is available at `arXiv:2004.01293` [1]. The functionality of the `motifcluster` package falls into a few main categories:

- Building motif adjacency matrices
- Sampling random weighted directed networks
- Spectral embedding with motif adjacency matrices
- Motif-based spectral clustering

This tutorial comprehensibly demonstrates all of these functionalities, showcasing the full capability of the `motifcluster` package. The package can be installed from PyPI with:

```
> pip install motifcluster
```

We load some other helpful packages for this tutorial:

```
>>> import numpy as np
>>> import scipy
>>> import sklearn
```

The `motifcluster` package can then be loaded from within Python with:

```
>>> from motifcluster import clustering as mccl
>>> from motifcluster import motifadjacency as mcmo
>>> from motifcluster import sampling as mcsa
>>> from motifcluster import spectral as mcsp
>>> from motifcluster import utils as mcut
```

2 Building motif adjacency matrices

The main novelty in the `motifcluster` package is its ability to build a wide variety of motif adjacency matrices (MAMs), and to do so quickly. There are several options to consider when building an MAM, which are covered in this section.

2.1 An example network

In order to demonstrate the construction of MAMs, we first need a small weighted directed network $\mathcal{G}_1$ to use as an example. Note that throughout this package we represent networks by their weighted directed adjacency matrices (possibly in sparse form). This means that for use alongside Python packages such as `networkx`, one must manually convert between adjacency matrices and `networkx` objects.

```
>>> G1 = np.array([
...   0, 2, 0, 0,
...   0, 0, 2, 3,
...   0, 4, 0, 0,
...   4, 0, 5, 0
... ]).reshape((4, 4))
```

2.2 Basic motif adjacency matrix construction

The `build_motif_adjacency_matrix` function is the main workhorse for building MAMs with `motifcluster`. Let's use it to build an MAM for the network $\mathcal{G}_1$. First we must choose a motif to look for. A full list can be obtained with:

```
mcut.get_motif_names()
['Ms', 'Md', 'M1', 'M2', 'M3', 'M4', 'M5', 'M6', 'M7']
['M8', 'M9', 'M10', 'M11', 'M12', 'M13', 'Mcoll', 'Mexpa']
```

Let's use the 3-cycle motif $\mathcal{M}_1$.

```
>>> mcmo.build_motif_adjacency_matrix(G1, motif_name="M1").todense()
matrix([[0., 1., 0., 1.],
        [1., 0., 0., 1.],
        [0., 0., 0., 0.],
        [1., 1., 0., 0.]])
```

Note that all the entries are zero except for entries $(1,2)$, $(1,4)$, $(4,1)$, and $(4,2)$. This is because vertices 1, 2 and 4 form an exact copy of the motif $\mathcal{M}_1$ in the network $\mathcal{G}_1$, and the (i,j)th MAM entry simply counts the number of instances containing both vertices i and j.

2.3 Functional and structural motif adjacency matrices

Looking at our example network $\mathcal{G}_1$ again, you might notice that there is seemingly another instance of the motif $\mathcal{M}_1$ in our network $\mathcal{G}_1$, on the vertices 2, 3 and 4, albeit with an "extra" edge from 2 to 3. The reason for this is that we instructed `build_motif_adjacency_matrix` to look for *structural* motif instances (this is the default). Structural instances require an exact match, with no extra edges. If we want to also include instances which may have extra edges present, we must instead use functional motif instances:

```
>>> mcmo.build_motif_adjacency_matrix(G1, motif_name="M1",
...   motif_type="func").todense()
matrix([[0., 1., 0., 1.],
        [1., 0., 1., 2.],
        [0., 1., 0., 1.],
        [1., 2., 1., 0.]])
```

This time we also pick up the 3-cycle on vertices 2, 3 and 4. Vertices 2 and 4 therefore occur in two distinct instances of the motif, and so their motif adjacency matrix entries are equal to two.

2.4 Weighted motif adjacency matrices

Our example network $\mathcal{G}_1$ has weighted edges, which we have not yet used: so far our MAMs have been simply counting instances of motifs. This is because the default weighting scheme is "unweighted", assigning every instance a weight of one.

2.4.1 Mean-weighted instances

We could instead use the "mean" weighting scheme, where every instance is assigned a weight equal to its mean edge weight. The (i,j)th MAM entry is then defined as the sum of these instance weights across all instances containing both vertices i and j:

```
>>> mcmo.build_motif_adjacency_matrix(G1, motif_name="M1",
...   motif_type="func", mam_weight_type="mean").todense()
matrix([[0., 3., 0., 3.],
        [3., 0., 4., 7.],
        [0., 4., 0., 4.],
        [3., 7., 4., 0.]])
```

The 3-cycle on vertices 1, 2 and 4 has edge weights of 2, 3 and 4, so its mean edge weight is 3. Similarly the 3-cycle on vertices 2, 3 and 4 has mean edge weight of 4. Vertices 2 and 4 appear in both, so their mutual MAM entries are the sum of these two mean weights, which is 7.

2.4.2 Product-weighted instances

We can also use the "product" weighting scheme, where every instance is assigned a weight equal to the product of its edge weights. The (i, j)th MAM entry is then defined as the sum of these instance weights across all instances containing both vertices i and j:

```
>>> mcmo.build_motif_adjacency_matrix(G1, motif_name="M1",
...   motif_type="func", mam_weight_type="product").todense()
matrix([[ 0, 24,  0, 24],
        [24,  0, 60, 84],
        [ 0, 60,  0, 60],
        [24, 84, 60,  0]], dtype=int64)
```

The 3-cycle on vertices 1, 2 and 4 has edge weights of 2, 3 and 4, so the product of its edge weights is 24. Similarly the 3-cycle on vertices 2, 3 and 4 has mean edge weight of 60. Vertices 2 and 4 appear in both, so their shared MAM entries are the sum of these two product weights, which is 84.

2.4.3 Computation method

The final argument to `build_motif_adjacency_matrix` is the "mam_method" argument. This does not affect the value returned but may impact the amount of time taken to return the MAM. In general the "sparse" method (the default) is faster on large sparse networks. The "dense" method, which uses fewer operations but on denser matrices, tends to be faster for small dense networks.

```
>>> mam_sparse = mcmo.build_motif_adjacency_matrix(G1, motif_name="M1",
...   mam_method="sparse").toarray()
>>> mam_dense  = mcmo.build_motif_adjacency_matrix(G1, motif_name="M1",
...   mam_method="dense").toarray()
>>> (mam_sparse == mam_dense).all()
True
```

3 Sampling random weighted directed networks

Building adjacency matrices by hand is tedious, so it is useful to have methods for generating the adjacency matrices of networks drawn from some probabilistic model. We use (weighted) directed stochastic block models (DSBMs) and (weighted) bipartite stochastic block models (BSBMs).

3.1 Directed stochastic block models

First let's sample the adjacency matrix of a DSBM which has two blocks of vertices; the first containing 5 vertices and the second containing 3. We use strong within-block connections, with the diagonal entries of the connection matrix set to 0.9. The between-block connections are weaker, with the off-diagonal connection matrix entries set to 0.2. Note how the resulting adjacency matrix is denser on its diagonal blocks $\{1, \ldots, 5\} \times \{1, \ldots, 5\}$ and $\{6, \ldots, 8\} \times \{6, \ldots, 8\}$, and is sparser on its off-diagonal blocks $\{1, \ldots, 5\} \times \{6, \ldots, 8\}$ and $\{6, \ldots, 8\} \times \{1, \ldots, 5\}$. The entries which lie exactly on the diagonal will always be zero, since we only consider networks without self-loops.

```
>>> block_sizes = [5, 3]
>>> connection_matrix = np.array([
...   0.9, 0.2,
...   0.2, 0.9
... ]).reshape((2, 2))
```

```
>>> mcsa.sample_dsbm(block_sizes, connection_matrix).todense()
matrix([[0., 1., 1., 1., 0., 1., 0., 0.],
        [1., 0., 1., 1., 1., 0., 0., 0.],
        [1., 0., 0., 1., 1., 0., 1., 0.],
        [1., 1., 1., 0., 1., 1., 0., 0.],
        [1., 1., 0., 1., 0., 0., 1., 0.],
        [0., 0., 0., 0., 0., 0., 1., 1.],
        [1., 1., 0., 0., 0., 1., 0., 1.],
        [1., 0., 0., 0., 0., 1., 1., 0.]])
```

3.1.1 Constant-weighted directed stochastic block models

The matrix above has binary entries, indicating that it is the adjacency matrix of an unweighted directed network. The `motifcluster` package also allows sampling of weighted directed networks. The simplest example of this is "constant" weighting, where we simply multiply each block of the adjacency matrix by a constant.

```
>>> weight_matrix = np.array([
...   5, 2,
...   2, 5
... ]).reshape((2, 2))
>>> mcsa.sample_dsbm(block_sizes, connection_matrix, weight_matrix,
...   sample_weight_type="constant").todense()
matrix([[0., 5., 5., 5., 5., 0., 0., 0.],
        [5., 0., 5., 0., 5., 0., 2., 0.],
        [5., 5., 0., 5., 5., 0., 0., 2.],
        [5., 5., 0., 0., 5., 2., 0., 0.],
        [5., 5., 5., 5., 0., 2., 0., 0.],
        [0., 0., 0., 0., 0., 0., 5., 5.],
        [0., 0., 0., 0., 2., 5., 0., 5.],
        [2., 0., 0., 2., 0., 5., 5., 0.]])
```

3.1.2 Poisson-weighted directed stochastic block models

We can also use weights drawn randomly from a Poisson distribution, where each block in the adjacency matrix has its own mean parameter. This returns an adjacency matrix with weights which could be any natural number, but is equal in expectation to the constant version. Note that in this scheme it is possible for the weight to be zero, removing an edge which might have otherwise been present.

```
>>> mcsa.sample_dsbm(block_sizes, connection_matrix, weight_matrix,
...   sample_weight_type="poisson").todense()
matrix([[ 0.,  4.,  7., 10.,  5.,  0.,  2.,  0.],
        [ 5.,  0.,  6.,  4.,  0.,  0.,  0.,  0.],
        [ 5.,  1.,  0.,  0.,  4.,  1.,  0.,  0.],
        [ 1.,  3.,  8.,  0.,  4.,  0.,  0.,  0.],
        [ 6.,  6.,  3.,  3.,  0.,  0.,  0.,  0.],
        [ 0.,  0.,  0.,  0.,  0.,  0.,  3.,  6.],
        [ 3.,  0.,  0.,  3.,  0.,  4.,  0.,  7.],
        [ 0.,  0.,  0.,  0.,  2.,  3.,  3.,  0.]])
```

3.2 Bipartite stochastic block models

The `motifcluster` package can also be used to sample bipartite networks. The vertices of a bipartite network are partitioned into "source" and "destination" vertices, and edges are only permitted to go from source vertices to destination vertices. Let's sample a DSBM with a single block of two source vertices and two blocks of destination vertices, with sizes of three and two respectively. We can use a strong connection probability of 0.9 to the first block of destination vertices, and a weaker connection probability of 0.2 to the second.

```
>>> source_block_sizes = [2]
>>> destination_block_sizes = [3, 2]
>>> bipartite_connection_matrix = np.array([0.9, 0.2]).reshape((1, 2))
>>> mcsa.sample_bsbm(source_block_sizes, destination_block_sizes,
...   bipartite_connection_matrix).todense()
matrix([[0., 0., 1., 1., 1., 0., 1.],
        [0., 0., 1., 1., 1., 0., 1.],
        [0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0.]])
```

3.2.1 Weighted bipartite stochastic block models

Similarly to the more general directed stochastic block models, we can also use constant-weighted or Poisson-weighted edges for bipartite stochastic block models.

```
>>> bipartite_weight_matrix = np.array([7, 2]).reshape((1, 2))
>>> mcsa.sample_bsbm(source_block_sizes, destination_block_sizes,
...   bipartite_connection_matrix, bipartite_weight_matrix,
...   sample_weight_type="poisson").todense()
matrix([[ 0.,  0.,  0.,  0., 13.,  2.,  0.],
        [ 0.,  0.,  7.,  3., 11.,  3.,  0.],
        [ 0.,  0.,  0.,  0.,  0.,  0.,  0.],
        [ 0.,  0.,  0.,  0.,  0.,  0.,  0.],
        [ 0.,  0.,  0.,  0.,  0.,  0.,  0.],
        [ 0.,  0.,  0.,  0.,  0.,  0.,  0.],
        [ 0.,  0.,  0.,  0.,  0.,  0.,  0.]])
```

4 Spectral embedding with motif adjacency matrices

Spectral methods involve performing eigenvalue and eigenvector operations on matrices related to networks. We work here with weighted *undirected* networks (which have symmetric adjacency matrices), because motif adjacency matrices are always symmetric.

4.1 Laplacian matrices

We can construct two types of Laplacian matrix for a network using the `motifcluster` package. First we create an example of a weighted undirected network $\mathcal{G}_2$.

```
>>> G2 = np.array([
...   0, 2, 0, 0,
...   2, 0, 4, 3,
...   0, 4, 0, 5,
...   0, 3, 5, 0
... ]).reshape((4, 4))
```

4.1.1 Combinatorial Laplacian

The combinatorial Laplacian of an adjacency matrix G is $L_c = D - G$, where D is the diagonal matrix of weighted vertex degrees:

```
>>> mcsp.build_laplacian(G2, type_lap="comb").todense()
matrix([[ 2., -2.,  0.,  0.],
        [-2.,  9., -4., -3.],
        [ 0., -4.,  9., -5.],
        [ 0., -3., -5.,  8.]])
```

4.1.2 Random-walk Laplacian

The random-walk Laplacian of an adjacency matrix G is $L_{rw} = I - D^{-1}G$, where D is the diagonal matrix of weighted vertex degrees and I is the identity matrix:

```
>>> mcsp.build_laplacian(G2, type_lap="rw").todense()
matrix([[ 1.  , -1.  ,  0.  ,  0.  ],
        [-0.22,  1.  , -0.44, -0.33],
        [ 0.  , -0.44,  1.  , -0.56],
        [ 0.  , -0.38, -0.62,  1.  ]])
```

4.2 Laplace embedding

Once we have constructed the desired Laplacian matrix, we use it to embed each vertex into $\mathbb{R}^l$ by finding the eigenvectors associated with its first (smallest magnitude) few eigenvalues. Below we use the random-walk Laplacian, and embedding dimension $l = 2$:

```
>>> spectrum = mcsp.run_laplace_embedding(G2, num_eigs=2, type_lap="rw")
>>> spectrum["vals"]
array([0.  , 0.79])
>>> spectrum["vects"]
array([[ 0.5 , -0.93],
       [ 0.5 , -0.2 ],
       [ 0.5 ,  0.2 ],
       [ 0.5 ,  0.23]])
```

For a random-walk Laplacian, the first eigenvalue is always zero (up to machine precision) and its corresponding eigenvector is constant.

4.3 Motif embedding

Motif embedding is simply the process of building an MAM and performing Laplace embedding with it. As an example we use the `run_motif_embedding` function on the network $\mathcal{G}_3$ below.

```
>>> G3 = np.array([
...   0, 0, 0, 0,
...   0, 0, 2, 3,
...   0, 4, 0, 0,
...   4, 0, 5, 0
... ]).reshape((4, 4))
```

An artefact of building MAMs is that although the original network may be connected, there is no guarantee that the MAM is also connected. Hence the MAM is restricted to its largest connected component before the Laplacian is formed. We observe this with the network $\mathcal{G}_3$, in which only three of the four vertices are embedded.

```
>>> spectrum = mcsp.run_motif_embedding(G3, motif_name="M1", motif_type="func",
...   mam_weight_type="unweighted", mam_method="sparse", num_eigs=2,
...   restrict = True, type_lap="rw")
>>> spectrum["vals"]
array([-0. ,  1.5])
>>> spectrum["vects"]
array([[-0.58,  0.29],
       [-0.58, -0.81],
       [-0.58,  0.51]])
```

5 Motif-based spectral clustering

The overall aim of `motifcluster` is to use motifs for spectral clustering, so now we see how to extract clusters from the motif-based eigenvector embeddings. The `run_motif_clustering` function handles the entire process of building an MAM, restricting it to its largest connected component, performing eigenvector embedding, and extracting clusters. We therefore take the opportunity to showcase the ability of `motifcluster` to recover the blocks of a DSBM, demonstrating all of the methods outlined in this vignette.

Let's use a DSBM with three blocks of 10 nodes each.

```
>>> block_sizes = 3 * [10]
```

We use strong connections of 0.8 within the blocks, and weaker connections of 0.2 between the blocks.

```
>>> connection_matrix = np.array([
...   0.8, 0.2, 0.2,
...   0.2, 0.8, 0.2,
...   0.2, 0.2, 0.8
... ]).reshape((3, 3))
```

We also set the within-block edges to be Poisson-weighted with mean 20, and the between-block edges to be Poisson-weighted with smaller mean 10.

```
>>> weight_matrix = np.array([
...   20, 10, 10,
...   10, 20, 10,
...   10, 10, 20
... ]).reshape((3, 3))

>>> G4 = mcsa.sample_dsbm(block_sizes, connection_matrix, weight_matrix,
...   sample_weight_type="poisson")
```

Now we can run the motif-based spectral clustering algorithm on this network with the 3-cycle motif $\mathcal{M}_1$. We build a functional MAM, weighting the instances by their mean edge weights, using the sparse formulation. We restrict this MAM to its largest connected component. Then we construct a random-walk Laplacian and embed it using the first four eigenvalues and eigenvectors. Finally we extract three clusters.

```
>>> motif_cluster = mccl.run_motif_clustering(G4, motif_name="M1", motif_type="func",
...   mam_weight_type="mean", mam_method="sparse", type_lap="rw", num_eigs=4,
...   restrict=True, num_clusts=3
... )
```

We can evaluate the performance by comparing it to the ground-truth labels using the adjusted Rand index from the `sklearn` package:

```
>>> truth = 10 * [0] + 10 * [1] + 10 * [2]
>>> sklearn.metrics.adjusted_rand_score(motif_cluster["clusts"], truth)
1.0
```

A larger value indicates better recovery of the blocks, with a value of one indicating perfect agreement.

References

[1] W. G. Underwood, A. Elliott, and M. Cucuringu. Motif-based spectral clustering of weighted directed networks. April 2020. arXiv:2004.01293.