Using the `motifclustr` package

William George Underwood

2020-04-16

Contents

1 Introduction **1**

2 Building motif adjacency matrices **2**
2.1 An example network 2
2.2 Basic motif adjacency matrix construction 2
2.3 Functional and structural motif adjacency matrices 3
2.4 Weighted motif adjacency matrices 3
2.4.1 Mean-weighted instances 3
2.4.2 Product-weighted instances 3
2.4.3 Computation method 4

3 Sampling random weighted directed networks **4**
3.1 Directed stochastic block models 4
3.1.1 Constant-weighted directed stochastic block models 5
3.1.2 Poisson-weighted directed stochastic block models 5

4 Spectral embedding with motif adjacency matrices **6**
4.1 Laplacian matrices 6
4.1.1 Combinatorial Laplacian 6
4.1.2 Random-walk Laplacian 6
4.2 Laplace embedding 6
4.3 Motif embedding 7

5 Motif-based spectral clustering **8**

1 Introduction

TODO consistent variable names TODO name args

This vignette demonstrates how to use the `motifclustr` package for motif-based spectral clustering of weighted directed networks. These methods are detailed in the paper *Motif-Based Spectral Clustering of Weighted Directed Networks*, available on the arXiv. TODO format arxiv link The functionality of the `motifclustr` package falls into two main categories:

- Sampling random weighted directed networks
- Building motif adjacency matrices (MAMs)
- Spectral clustering with MAMs

This vignette comprehensibly demonstrates all of these functionalities, showcasing the full capability of the `motifclustr` package. The package is loaded with

```
library(motifclustr)
```

2 Building motif adjacency matrices

The main novelty in the `motifclustr` package is its ability to build a wide variety of MAMs, and to do so quickly. There are several options to consider when building an MAM, and these options 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 $\mathcal{G}$ 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 packages such as `igraph`, one must manually convert between adjacency matrices and `igraph` objects.

```
toy_adj_mat <- matrix(c(
  0, 2, 0, 0,
  0, 0, 2, 3,
  0, 4, 0, 0,
  4, 0, 5, 0
), byrow = TRUE, nrow = 4)
```

TODO plot network

2.2 Basic motif adjacency matrix construction

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

```
get_motif_names()
#  [1] "Ms"    "Md"    "M1"    "M2"    "M3"    "M4"    "M5"    "M6"    "M7"
# [10] "M8"    "M9"    "M10"   "M11"   "M12"   "M13"   "Mcoll" "Mexpa"
```

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

```
mam <- build_motif_adjacency_matrix(toy_adj_mat, "M1")
# Warning in if (!(motif_type %in% c("func", "struc"))) {: the condition has
# length > 1 and only the first element will be used
# Warning in if (weight_type == "unweighted") {: the condition has length > 1 and
# only the first element will be used
# Warning in if (motif_type == "func") {: the condition has length > 1 and only
# the first element will be used
# Warning in if (motif_type == "struc") {: the condition has length > 1 and only
# the first element will be used
prmatrix(mam)
#      [,1] [,2] [,3] [,4]
# [1,]    0    1    0    1
# [2,]    1    0    0    1
# [3,]    0    0    0    0
# [4,]    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 $\mathcal{G}$, 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 $\mathcal{G}$ again, you might notice that there is seemingly another instance of the motif $\mathcal{M}_1$ in our network $\mathcal{G}$, 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 can instead use functional motif instances:

```
mam <- build_motif_adjacency_matrix(toy_adj_mat, "M1", motif_type = "func")
# Warning in if (weight_type == "unweighted") {: the condition has length > 1 and
# only the first element will be used
prmatrix(mam)
#      [,1] [,2] [,3] [,4]
# [1,]    0    1    0    1
# [2,]    1    0    1    2
# [3,]    0    1    0    1
# [4,]    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 has weighted edges, which we have not yet taken into account: 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 can 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:

```
mam <- build_motif_adjacency_matrix(toy_adj_mat, "M1", motif_type = "func",
  weight_type = "mean")
prmatrix(mam)
#      [,1] [,2] [,3] [,4]
# [1,]    0    3    0    3
# [2,]    3    0    4    7
# [3,]    0    4    0    4
# [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 shared 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:

```
mam <- build_motif_adjacency_matrix(toy_adj_mat, "M1", motif_type = "func",
  weight_type = "product")
prmatrix(mam)
```

```
#      [,1] [,2] [,3] [,4]
# [1,]    0   24    0   24
# [2,]   24    0   60   84
# [3,]    0   60    0   60
# [4,]   24   84   60    0
```

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 "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_1 <- build_motif_adjacency_matrix(toy_adj_mat, "M1", method = "sparse")
# Warning in if (!(motif_type %in% c("func", "struc"))) {: the condition has
# length > 1 and only the first element will be used
# Warning in if (weight_type == "unweighted") {: the condition has length > 1 and
# only the first element will be used
# Warning in if (motif_type == "func") {: the condition has length > 1 and only
# the first element will be used
# Warning in if (motif_type == "struc") {: the condition has length > 1 and only
# the first element will be used
mam_2 <- build_motif_adjacency_matrix(toy_adj_mat, "M1", method = "dense")
# Warning in if (!(motif_type %in% c("func", "struc"))) {: the condition has
# length > 1 and only the first element will be used
# Warning in if (weight_type == "unweighted") {: the condition has length > 1 and
# only the first element will be used
# Warning in if (motif_type == "func") {: the condition has length > 1 and only
# the first element will be used
# Warning in if (motif_type == "struc") {: the condition has length > 1 and only
# the first element will be used
print(all(mam_1 == mam_2))
# [1] 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 a 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 much weaker, with the off-diagonal connection matrix entries set to 0.1. Note how the resulting adjacency matrix is dense on its diagonal blocks $\{1,\ldots,5\}\times\{1,\ldots,5\}$ and $\{6,\ldots,8\}\times\{6,\ldots,8\}$, and is sparse on its off-diagonal blocks $\{1,\ldots,5\}\times\{6,\ldots,8\}$ and $\{6,\ldots,8\}\times\{1,\ldots,5\}$. All the entries which lie exactly on the diagonal will always be zero, since we consider networks without self-loops.

```
block_sizes = c(5, 3)
connection_matrix = matrix(c(
  0.9, 0.2,
  0.2, 0.9
), nrow = 2, byrow = TRUE)
adj_mat = sample_dsbm(block_sizes, connection_matrix)
prmatrix(adj_mat)
#      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8]
# [1,]    0    1    1    1    1    0    0    0
# [2,]    1    0    0    1    1    1    0    0
# [3,]    1    1    0    1    1    0    0    0
# [4,]    1    1    1    0    0    0    0    0
# [5,]    1    1    1    1    0    0    0    0
# [6,]    0    0    0    0    1    0    1    1
# [7,]    0    0    0    1    1    1    0    1
# [8,]    0    1    0    0    1    1    0    0
```

3.1.1 Constant-weighted directed stochastic block models

The adjacency matrix above has binary entries, indicating that it is the adjacency matrix of an unweighted directed network. The `motifclustr` 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 = matrix(c(
  5, 2,
  2, 5
), nrow = 2, byrow = TRUE)
adj_mat = sample_dsbm(block_sizes, connection_matrix, weight_matrix,
                      weight_type = "constant")
prmatrix(adj_mat)
#      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8]
# [1,]    0    5    5    5    5    0    0    0
# [2,]    5    0    5    5    5    0    0    0
# [3,]    5    5    0    5    5    0    0    0
# [4,]    5    5    5    0    5    0    0    0
# [5,]    5    5    0    5    0    0    2    0
# [6,]    0    0    0    0    0    0    5    5
# [7,]    2    0    0    0    0    5    0    5
# [8,]    0    0    0    0    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.

```
adj_mat = sample_dsbm(block_sizes, connection_matrix, weight_matrix,
                      weight_type = "poisson")
prmatrix(adj_mat)
#      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8]
# [1,]    0    4    5    5    4    0    0    1
# [2,]    4    0    3    5    6    0    0    0
# [3,]    0    8    0    4    7    0    0    0
# [4,]    5    3    3    0    5    0    2    0
# [5,]    8    9    5    4    0    1    0    0
# [6,]    0    0    1    0    0    0    0    2
# [7,]    4    0    0    0    0    6    0    4
# [8,]    0    0    1    0    1    6    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 `motifclustr` package. First we create a toy example of a weighted undirected adjacency matrix.

```
toy_adj_mat_sym <- matrix(c(
  0, 2, 0, 0,
  2, 0, 4, 3,
  0, 4, 0, 5,
  0, 3, 5, 0
), byrow = TRUE, nrow = 4)
```

4.1.1 Combinatorial Laplacian

The combinatorial Laplacian of an adjacency matrix G is $L_\mathrm{c} = D - G$, where D is the diagonal matrix of weighted vertex degrees:

```
Lc = build_laplacian(toy_adj_mat_sym, "comb")
prmatrix(Lc)
#      [,1] [,2] [,3] [,4]
# [1,]    2   -2    0    0
# [2,]   -2    9   -4   -3
# [3,]    0   -4    9   -5
# [4,]    0   -3   -5    8
```

4.1.2 Random-walk Laplacian

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

```
Lrw = build_laplacian(toy_adj_mat_sym, "rw")
prmatrix(round(Lrw, 2))
#       [,1]  [,2]  [,3]  [,4]
# [1,]  1.00 -1.00  0.00  0.00
# [2,] -0.22  1.00 -0.44 -0.33
# [3,]  0.00 -0.44  1.00 -0.56
# [4,]  0.00 -0.38 -0.62  1.00
```

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) eigenvalues. Below we use the random-walk Laplacian, and embedding dimension $l = 2$:

```
spectrum = run_laplace_embedding(toy_adj_mat_sym, 2, "rw")
# [1] "matrix"
# [1] "numeric"
round(spectrum$vects, 2)
#      [,1]  [,2]
# [1,]  0.5  0.93
# [2,]  0.5  0.20
# [3,]  0.5 -0.20
# [4,]  0.5 -0.23
round(spectrum$vals, 4)
# [1] 0.0000 0.7883
```

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 a MAM and then performing Laplace embedding with it. This function also makes the MAM available. As an example we go back to the first toy network, which is weighted and directed.

```
spectrum = run_motif_embedding(toy_adj_mat, "M1", "func", "unweighted", "sparse", 2, "rw")
# [1] "matrix"
# [1] "numeric"
round(spectrum$vals, 4)
# [1] 0 1
round(spectrum$vects, 2)
#      [,1]  [,2]
# [1,]  0.5  0.71
# [2,]  0.5  0.00
# [3,]  0.5 -0.71
# [4,]  0.5  0.00
prmatrix(spectrum$motif_adj_mat)
#      [,1] [,2] [,3] [,4]
# [1,]    0    1    0    1
# [2,]    1    0    1    2
# [3,]    0    1    0    1
# [4,]    1    2    1    0
```

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. This is demonstrated below, where only three of the four vertices are embedded.

```
toy_adj_mat <- matrix(c(
  0, 0, 0, 0,
  0, 0, 2, 3,
  0, 4, 0, 0,
  4, 0, 5, 0
), byrow = TRUE, nrow = 4)
spectrum = run_motif_embedding(toy_adj_mat, "M1", "func", "unweighted", "sparse", 2, "rw")
# [1] "matrix"
# [1] "numeric"
round(spectrum$vals, 4)
# [1] 0.0 1.5
round(spectrum$vects, 2)
```

```
#       [,1]  [,2]
# [1,] 0.58 -0.52
# [2,] 0.58 -0.29
# [3,] 0.58  0.81
```

5 Motif-based spectral clustering

The overall aim is to use motifs for spectral clustering, so now we see how to extract clusters from the motif-based eigenvector embeddings.

TODO write this after writing the code for it