Aix-Marseille University

M1 Computational and Mathematical biology

Markov Clustering Algorithm

Author:
VASILEVA Svetlana

Supervisors:
Victor CEPOI

Graph theory

April 1, 2023

Introduction

The Markov Cluster Algorithm (MCL) is a graph clustering algorithm based on the theory of Markov chains. It was originally developed for clustering biological networks but can be applied to a variety of network structures. MCL iteratively performs two operations on the network: expansion and inflation. These operations cause the clusters to become more distinct, ultimately converging to a steady state.

MCL is based on the concept of random walks on a graph. The idea is that a random walker on a graph will tend to stay within densely connected regions, which correspond to clusters. MCL simulates these random walks by repeatedly multiplying the adjacency matrix of the graph by itself, effectively computing paths of length 2, 3, 4, and so on. This process is similar to computing the powers of a matrix, but with a twist: instead of using matrix multiplication, MCL uses the Hadamard product (element-wise product) and the normalization of rows to simulate the random walk.

Main steps of the algorithm: the **expansion** step generates new edges in the graph by raising the adjacency matrix to a power using the Hadamard product. $M_{i,j} \leftarrow \frac{M_{i,j}^p}{\sum_k (M_{k,j}^p)}$.The **inflation** step compresses the edges in the graph by raising the matrix to a power using element-wise multiplication, followed by row normalization $M \leftarrow (M \circ M)^q$.

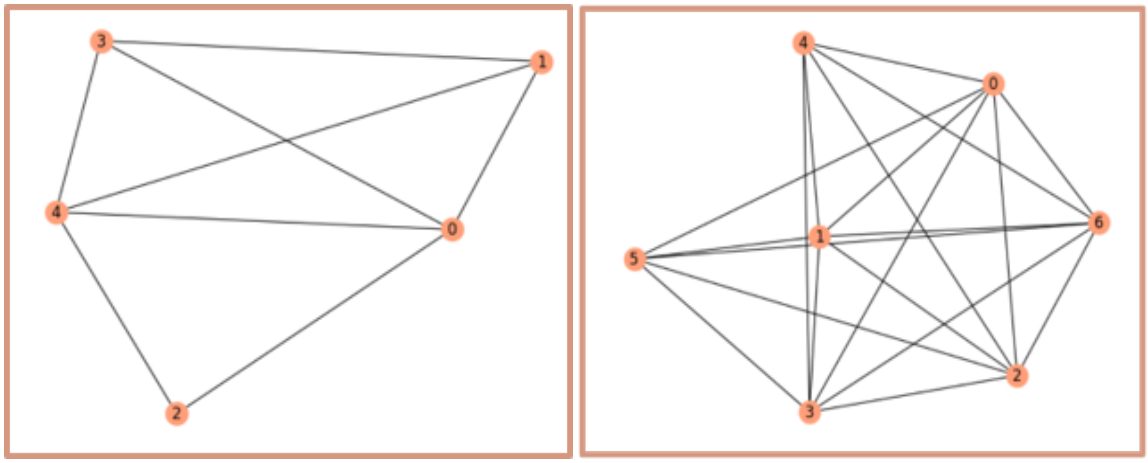
In this work the cluster map was built step by step and than all the clusters were distinguished with Markov Clustering method.

1 Map building

Let's simulate our own cluster work step by step. Firstly, we will pick some nodes (random amount from 3 to 10) and then simulate the connections between them with high probability. It was the most comfortable way to code firstly the extra function for adding the random edge and then the function for creating the county based on the first function.

```
import numpy as np
np.random.seed(0)

def add_random_edge(G, node1, node2, prob_road=0.9, mean_drive_time
    =20):
     if np.random.binomial(1, prob_road):
         drive_time = np.random.normal(mean_drive_time)
         G.add_edge(node1, node2, travel_time=round(drive_time,2))

nodes = list(G.nodes())
for node1 in nodes[:-1]:
     for node2 in nodes[node1 + 1:]:
         add_random_edge(G, node1, node2)

nx.draw(G, with_labels=True, node_color='lightsalmon')
plt.show()
```

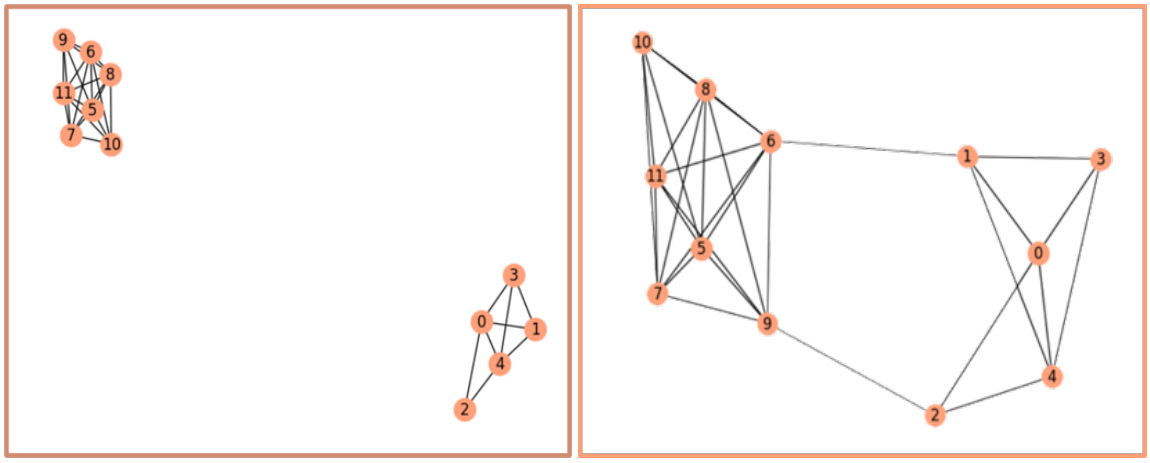
```
16
17 np.random.seed(0)
18 def random_county(county_id):
19     num_towns = np.random.randint(3, 10)
20     G = nx.Graph()
21     nodes = [(node_id, {'county_id': county_id})
22             for node_id in range(num_towns)]
23     G.add_nodes_from(nodes)
24     for node1, _ in nodes[:-1]:
25         for node2, _ in nodes[node1 + 1:]:
26             add_random_edge(G, node1, node2)
27     return G
28
29 G2 = random_county(1)
30 nx.draw(G2, with_labels=True, node_color='lightsalmon')
31 plt.show()
```



(a) 1st county (b) 2nd county

Now it's time to define inter-counties links with low probabilities. There is natural introduction of clusters property about strong connection of the elements inside the cluster and weak connection with all the members of other dense regions.

```
1 np.random.seed(0)
2 def add_intracounty_edges(G):
3     nodes = list(G.nodes(data=True))
4     for node1, attributes1 in nodes[:-1]:
5         county1 = attributes1['county_id']
6         for node2, attributes2 in nodes[node1:]:
7             if county1 != attributes2['county_id']:
8                 add_random_edge(G, node1, node2, prob_road=0.05,
    mean_drive_time=45)
9     return G
10
11 G = add_intracounty_edges(G)
12 np.random.seed(0)
13 nx.draw(G, with_labels=True, node_color='lightsalmon')
```



(a) 2 first counties on the global map (b) Weak connection between clusters

As you can see from the code, the probability of inter-county connection is 0.05. There is the simulation for our region with 6 counties.

```
np.random.seed(1)
G = random_county(0)
for county_id in range(1, 6):
    G2 = random_county(county_id)
    G = nx.disjoint_union(G,G2)

G = add_intracounty_edges(G)
np.random.seed(1)
G.remove_edge(3,9) # for more showable results
nx.draw(G, with_labels=True, node_color='lightsalmon')
```

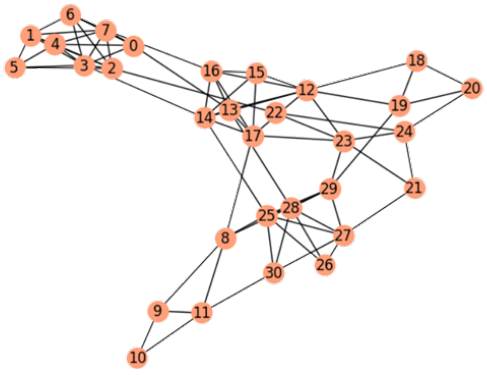


Figure 3: Whole region

Our next step is to simulate random walks of cars between towns in the whole region. We will identify the function for the random drive and use it for simulation of 20000 cars moving.

```
def random_drive(num_stops=10):
    town = np.random.choice(G.nodes)
```

```
    print(town)
    for _ in range (num_stops):
        town = np.random.choice(G[town])

    return town

import time
np.random.seed(0)
car_counts = np.zeros(len(G.nodes))
num_cars = 20000

for _ in range(num_cars):
    destination = random_drive()
    car_counts[destination] += 1

central_town = car_counts.argmax()
traffic = car_counts[central_town]
```

As a result of this part of code we can gain the amount of cars in each town. With this step we model transport load in every city, so in real projects it's sometimes useful to get absolute values, but for this work it's important to estimate relative values. That's why we will realise the probabilities of ending up in the certain town (just with normalizing by the amount of cars $probabilities = carcounts/numcars$).

We also can do the same simulation just with application of normalized adjacency matrix as a transition matrix. Due to the same probability of choice of the starting point we can realise this mechanism quite simply in the next strings of code:

```
adjacency_matrix = nx.to_numpy_array(G)
transition_matrix = adjacency_matrix / adjacency_matrix.sum(axis=0)
assert np.array_equal(transition_vector, transition_matrix[:,0])

v = np.array([1 / 31] * 31)
for _ in range(10):
    v = transition_matrix @ v
```

2 MCL run

After simulation of potentially clusterable graph, let's try to use Markov Clustering Algorithm to identify these clusters. Firstly, the term *flow* should be introduced. Flow is the probability of ending up in the town after two moves (probability of ending of the random walk in the certain node after two iterations). Counting of the flow also corresponds to the expansion operator of MCL. So we will assume that flow is big for intra-counties movements and low for inter-counties drives. There is the visualisation of flows in histogram:

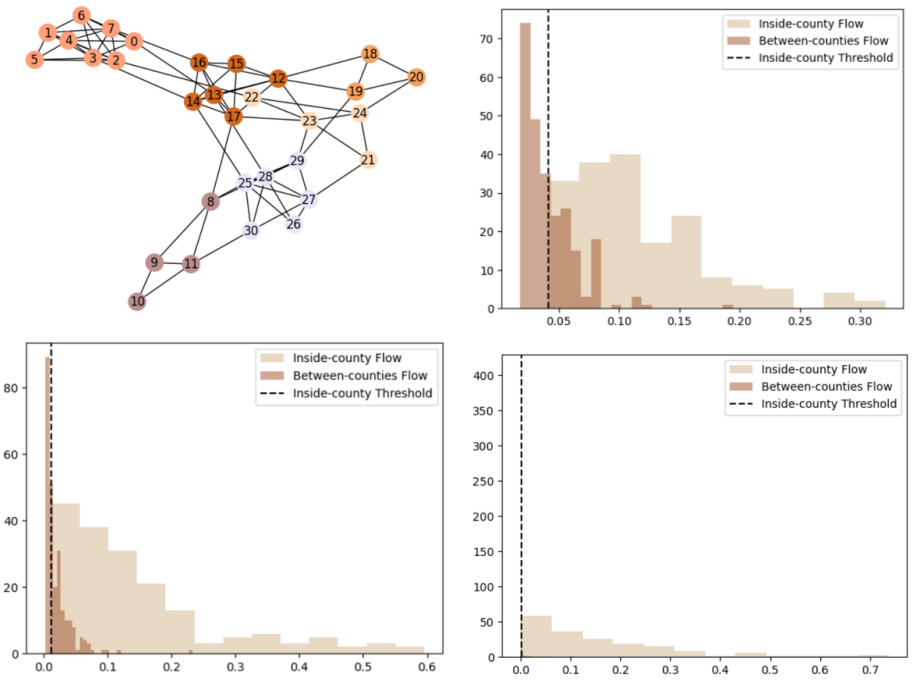


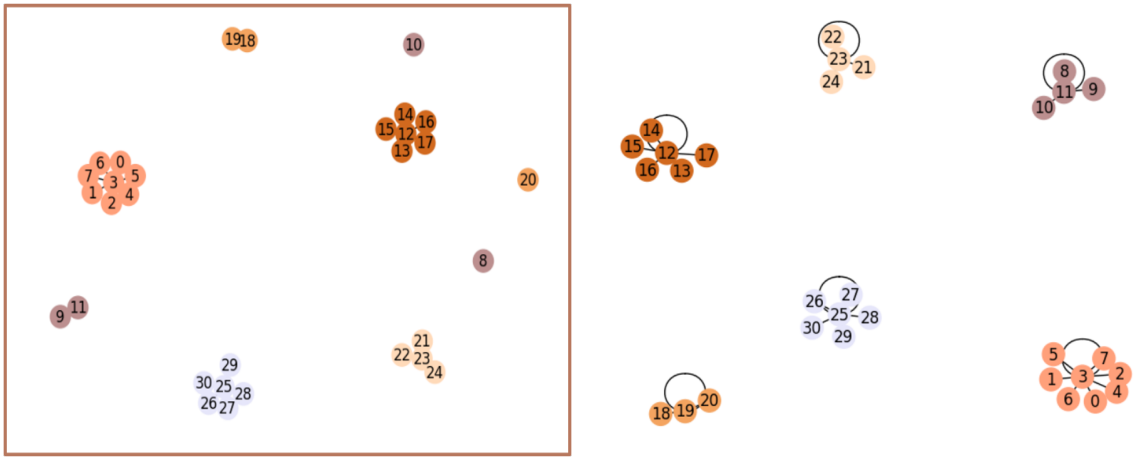
Figure 4: Histograms of the flow for initial data and after two steps of inflation

Due to the histogram data we assumed the correlation between two types of trips pretty clear. But still we cannot algorithmically clime the sectors, because of inaccuracy in the zone next to the threshold zone. In the first histogram threshold is the smallest flow of drives inside the counties. We know the threshold precisely only because of our own simulation, but in the general case it is more appropriate to use 0.01 threshold. To clarify the low-probability movements we will use the second operator of MCL $-inflation$. With this operation we will make big values bigger and small values smaller without changing the mean of our transition matrix.

One more nuance of the work with flows. When we talk about 2-steps movement the probability often miss the first step behaviour. That's why we add self-loops to each node of the graph to unlock the possibility to stay in the same town due to the random walk. There is the main function that is corresponding to the core of the Markov Clustering Algorithm with each nuance mentioning.

```
def inflate(matrix):
    matrix = matrix ** 2
    return matrix / matrix.sum(axis=0)

def run_mcl(G):
    for i in G.nodes:
        G.add_edge(i, i)

    adjacency_matrix = nx.to_numpy_array(G)
    transition_matrix = adjacency_matrix / adjacency_matrix.sum(
   axis=0)
    flow_matrix = inflate(transition_matrix @ transition_matrix)
```

```
    for _ in range(20):
        flow_matrix = inflate(flow_matrix @ flow_matrix)

    G.remove_edges_from([(i, j) for i, j in G.edges()
                         if not (flow_matrix[i][j] or flow_matrix[j
    ][i])])

G_copy = G.copy()
run_mcl(G_copy)
nx.draw(G_copy, with_labels=True, node_color=node_colors)
plt.show()
```



(a) Clustering without self-looping (b) Clustering with self-looping

Hence, we can see the perfect clustering with our result code.

3 References

[1] Van Dongen, S. (2008). Graph clustering via a discrete uncoupling process. SIAM Journal on Matrix Analysis and Applications, 30(1), 121-141.

[2] Van Dongen, S. M. (2000). Graph clustering by flow simulation (Doctoral dissertation).

[3] Van Dongen, S., Abreu-Goodger, C. (2012). Using MCL to extract clusters from networks. Bacterial molecular networks: Methods and protocols, 281-295.