Tools for Community Detection

Complex Network Theory

Tutorial – 13/12/2021

In this lab, we will use the `NetworkX`[1] package from `Python` to observe some behaviours of the Modularity and implement and compare algorithms to perform community detection.

1 The Modularity and the Resolution Limit

In this section, we will observe a nondesirable behaviour of the Modularity, called the resolution limit, and see how we can mitigate against it.

Exercise 1. Implement a `Python` function that takes an undirected graph and a community structure[2] as inputs, and returns their Modularity.

The so-called resolution limit means that the Modularity does not consider small communities as "true" communities. It is often illustrated using ring of cliques graphs, which are a set of cliques (i.e. complete graphs) linked by a cycle, as illustrated in Figure 1. By increasing the number of cliques, one can observe that,

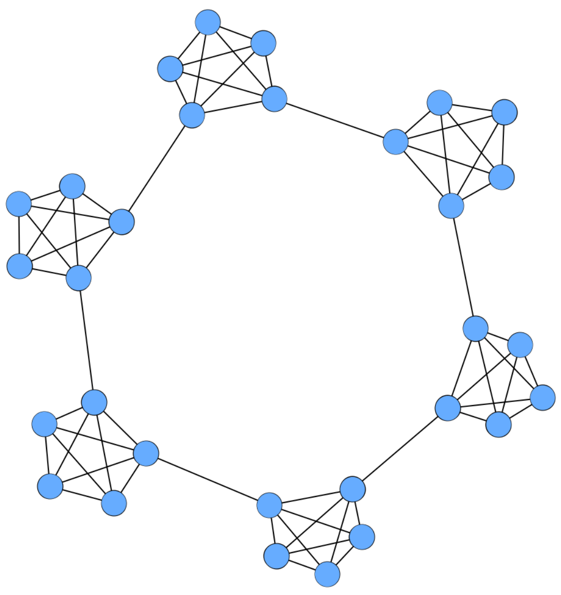

Figure 1: A ring-clique graph: 6 cliques of size 5.

after a certain number of cliques, the Modularity fails to correctly assess community structures. Namely, it considers that the community structure with two connected cliques per community is better than the community structure with one clique per community.

Exercise 2. Observe this behaviour using your implementation of the Modularity and the `ring_of_cliques` function from `NetworkX`.

To mitigate against this behaviour, a parametrised version of the Modularity has been proposed via the introduction of the "resolution parameter", which is a scalar $\gamma \geq 1$ to set up. This parametrised Modularity can be expressed as

$$\mathcal{Q}^{\gamma}(G,\mathcal{C}) = \frac{1}{2m} \sum_{C \in \mathcal{C}} \sum_{i,j \in C} \left(a_{i,j} - \gamma \times \frac{d^{\omega}(i) d^{\omega}(j)}{2m} \right),$$

using notations from slide 7.

Exercise 3. Adapt your implementation of the Modularity to introduce the resolution parameter, and try to observe the resolution limit again. What do you conclude?

One measure is known to have no resolution limit, namely the Potts Constant Model, which is a parametrised measure that can be expressed as:

$$\mathcal{P}^{\lambda}(G,\mathcal{C}) = \sum_{C \in \mathcal{C}} \sum_{i,j \in C} (a_{i,j} - \lambda), \qquad \text{with} \quad \lambda \in]0,1]$$

Exercise 4. Implement the Constant Potts Model, and see how it behaves with the rings of cliques.

[1] `https://networkx.org/documentation/stable/reference/index.html`

[2] In `NetworkX`, a community structure is a list of lists: on $V = \{1,2,3,4,5\}$, the community structure $\mathcal{C} = \{\{1,2\},\{3\},\{4,5\}\}$ is represented by $C = [[1,2],[3],[4,5]]$.

2 The Louvain Algorithm

We will now use `NetworkX` to implement the Louvain algorithm, that has been roughly introduced during the lesson. The pseudo-codes of this algorithm are given in Algorithm 1.

Algorithm 1: Louvain Algorithm

```
Data: $G = (V, E, \omega)$ an undirected weighted graph.
Result: $\mathcal{C} = \{C_1, ..., C_k\}$ a partitioning of $V$.
begin
    $\mathcal{C} \leftarrow \{\{i\}, i = 1..|V|\}$ ;                    /* Start with one community/node. */
    while $True$ do
        $\mathcal{C}_{crt} \leftarrow$__ONELEVEL($G$);
        if $|\mathcal{C}_{crt}| == |V|$ then
            break ;          /* Nothing has been done in ONELEVEL: end of Louvain. */
        end
        $\mathcal{C} \leftarrow$ __UPDATE($\mathcal{C}$,$\mathcal{C}_{crt}$) ;                          /* Not described here. */
        $G \leftarrow$__METAGRAPH($G$,$\mathcal{C}_{crt}$) ;     /* Metagraph on which ONELEVEL is reapplied. */
    end
end

__METAGRAPH($G = (V, E, \omega)$,$\mathcal{C}$):
begin
    $V_{meta} \leftarrow \{1, ..., |\mathcal{C}|\}$ ;                       /* Metanodes are the communities. */
    $E_{meta} \leftarrow \{\}$;
    for $v = 1..|\mathcal{C}|$ do
        $E_{meta} \leftarrow E_{meta} \cup \{\{v, v\}\}$ ;              /* We add self loops and their weights. */
        $\omega_{meta}(\{v, v\}) \leftarrow \sum_{i \in C_v} \sum_{j \in C_v, j \sim i} \omega(\{i, j\})$;
        for $u = v + 1..|\mathcal{C}|$ do
            if $Cut(C_v, C_u) > 0$ then
                $E_{meta} \leftarrow E_{meta} \cup \{\{v, u\}\}$;   /* An edge when 2 cties have non empty cut. */
                $\omega_{meta}(\{v, u\}) \leftarrow \sum_{i \in C_v} \sum_{j \in C_u, j \sim i} \omega(\{i, j\})$ ;       /* Weight is size of the cut. */
            end
        end
    end
    return $G_{meta} = (V_{meta}, E_{meta}, \omega_{meta})$;
end

__ONELEVEL($G = (V, E)$):
begin
    $\mathcal{C} \leftarrow \{\{i\}, i = 1..|V|\}$ ;                    /* Start with one community/node. */
    $2m \leftarrow \sum_{i=1}^{|V|} d^{\omega}(i)$;
    $increase \leftarrow True$;
    while $increase$ do
        $increase \leftarrow False$ ;           /* To update when a node changes its community. */
        for $i = 1..|V|$ do
            $C_{old} \leftarrow C \in \mathcal{C} : i \in C$;
            $C_{old} \leftarrow C_{old} \setminus i$ ;           /* Removes the current node from its community. */
            $\mathcal{N}_{comm} \leftarrow \{C \in \mathcal{C} : \exists u \in C, u \sim i\}$;        /* Finds its adjacent communities. */
            $GAIN_{max} \leftarrow 0, \quad C_{max} \leftarrow C_{old}$;
            for $C_n \in \mathcal{N}_{comm}$ do
                $GAIN \leftarrow Cut(C_n, \{i\}) - \frac{1}{2m} \times d^{\omega}(i) \times \sum_{j \in C_n} d^{\omega}(j)$;     /* Increase of modularity
                 resulting of adding the current node in this community. */
                if $GAIN > GAIN_{max}$ then
                    $GAIN_{max} \leftarrow GAIN, \quad C_{max} \leftarrow C_n$;
                end
            end
            $C_{max} \leftarrow C_{max} \cup \{i\}$ ;      /* Node added to the cty with the best increase. */
            $increase \leftarrow C_{max} \neq C_{old}$;
        end
    end
end
```

Exercise 5. Prove that the value `GAIN` is what we want. The fact that the increase of modularity can be written like this is the reason that makes Louvain as numerically efficient.

Exercise 6. In the `Python` file *Louvain.py* on Moodle, fill the `### TODO`s (lines 20, 77, 84 and 88) to get a functional version of Louvain[3].

3 Comparing Algorithms

We would like to compare the Louvain algorithm to other community detection algorithms implemented in `NetworkX`, by comparing the correctness of the communities they discover on the LFR benchmark. This requires a metric to assess the closeness between the discovered community structure and the groundtruth one.

Exercise 7. Implement a function that takes two community structures and outputs their Adjusted Rand Index.

The LFR benchmark is implemented in `NetworkX` and we will use it to assess the algorithms. By varying the so-called mixing parameter μ, one can control the blurriness of the community structure: $\mu = 0$ returns a graph in which communities are pairwise disconnected, while one generally considers that there is no community structure for $\mu > 0.75$.

Exercise 8. Use the function `LFR_benchmark_graph` with the parameters as stated below

```
LFR_benchmark_graph(500,2,1.5,μ, average_degree=5, min_community=10,seed=10)
```

to build a sequence of 7 random graphs with $\mu \in \{0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7\}$. By computing the Adjusted Rand Index of the community structures returned by the Louvain algorithm[4], observe that it is harder to uncover communities with large values of μ.

Exercise 9. Compare the community structures returned by Louvain algorithm on this benchmark, with those of some algorithms described here `https://networkx.org/documentation/stable/reference/algorithms/community.html`, e.g. `greedy_modularity_communities`, and conclude.

[3]Note the imports of functions `volume` and `cut` that may be useful.

[4]This will require you to carefully read the doc of the LFR function to understand how to extract the groundtruth community structure of the generated graph.