UNIVERSITÀ DEGLI STUDI DI PADOVA

## Università degli Studi di Padova

DIPARTIMENTO DI MATEMATICA “TULLIO LEVI-CIVITA”

Corso di Laurea Magistrale in Matematica

# Louvain-like methods for community detection in multilayer hypergraphs

Relatore:
Prof. Francesco Rinaldi

Laureanda: Angelica Crepaldi
Matricola: 2028618

Correlatrice:
Dott.ssa Sara Venturini

Anno Accademico 2022/2023

21 Luglio 2023

# Contents

# Introduction

Community detection, i.e. grouping the nodes into sets of nodes in such a way that each set of vertices is densely connected internally and less connected with the other sets, is a relevant problem in graph theory and, in particular, in the study of complex networks. The reason why many researchers focus on this topic is that members of a community usually share common properties, therefore revealing the community structure in a network can provide a better understanding of the overall functioning of the network. In many fields like e.g. sociology, biology, computer science and economics, complex systems can be represented as graphs. In biology, for example, we can find communities in protein-protein interaction networks, where each cluster is a group of proteins that play the same role within the cell [4]. From a practical point of view, identification of clusters of customers with similar interests in the network of purchase relationships between customers and products of online retailers enables one to set up efficient recommendation systems [14], that better guide customers through the list of items of the retailer and enhance the business opportunities.

In literature, many algorithms for community detection are available, e.g. the most famous one is the Louvain method [2], which was developed in order to extract communities from large networks. Despite being a good algorithm, it is designed only for single-layer graphs, but assuming to represent every system as a single-layer graph is an oversimplification, therefore when we deal with real world applications, we have to consider also multilayer networks. We talk about multilayer networks when we describe multiple types of interactions among entities of the same type; going back to proteins' example, a description of the full protein-protein interactome (i.e. the totality of protein–protein interactions happening in a cell) involves, for some organisms, up to seven distinct modes of interaction among thousands of protein molecules [6]. Another example, drawn from everyday life, is given by social networks: each layer represents a different interaction between people (e.g. each layer is a different social network or, in the same social network setting, each layer is an interaction such as likes, comments, followers, etc).

Up to now, we have considered graphs with edges that connect pairs of nodes, but much of the structure in complex networks involves higher-order interactions and relationships between more than two entities. Neuronal dynamics display mesoscopic behaviors that require interactions among multiple neurons to be predicted [9]; three or more species routinely compete for food and territory in complex ecosystems [16]. As a consequence, it is necessary to find a mathematical framework to describe group interactions and hypergraphs are the natural candidates to provide such descriptions. What we pointed out above justifies our choice of focusing on multilayer hypergraphs in this work.

The aim of this thesis project is to develop an algorithm for community detection in multi-layer hypergraphs, taking as starting point a Louvain-like method for community detection in multi-layer networks [18] and hypergraph maximum likelihood Louvain [5]. They are both extensions of the well-known algorithm for single-layer graphs [2].
The work is organized as follows.

Chapter 1 contains basic notions about graphs, multi-layer graphs and hypergraphs; in the second part of this chapter we explain the concept of community, together with modularity measure.
Chapter 2 presents some of the most performing methods for community detection in multi-layer networks and hypergraphs and contains a summary of Louvain method and hypergraph maximum likelihood Louvain, mainly focusing on a particular affinity function.
Chapter 3 describes our Louvain-like method for multilayer hypergraphs, an extension of classic Louvain that uses another modularity function and the average of modularity values across the layers to evaluate partition.
In Chapter 4 we show some numerical experiments on synthetic multilayer hypergraphs, generated via DCHSBM.
We finish our work with some conclusions considering the experiments done.
In Appendix we report our algorithm's code together with the function it needs; then we also write codes that evaluate accuracy and NMI.

# Chapter 1

# Background

## 1.1 Preliminaries

Graphs are often used to model and analyse complex systems of interacting entities. First of all is useful to give some notions and definitions in order to better understand what will come later. For more details see [1], [13] and [17].

A *graph* is a tuple $G = (V, E)$, where $V$ is the set of nodes and $E \subseteq V \times V$ is the set of edges that connect pairs of nodes. Definition of $E$ would be slightly different in case of directed graphs, but we don't consider them and we focus on undirected graphs.
If there is an edge between a pair of nodes, those nodes are *adjacent* and we say that this edge is *incident* to each of the two nodes. If we consider a system with multiple types of interactions among entities of the same type, then we also consider *layers* (each layer represents a type of interaction) and when there is more than one layer, we call the system *multi-layer network*. Unfortunately networks, even multi-layer networks, with only pairwise interactions are not enough to describe in a good way real-world phenomena and dynamics, since many times interactions take place within groups of nodes (and not only between two nodes); therefore we need to define hypergraphs, that describe higher-order interactions. In particular, a *hypergraph* is a tuple $G = (V, H)$, where $V$ is the set of nodes and $H$ is the set of hyperedges that specify which nodes participate in which way within an interaction.

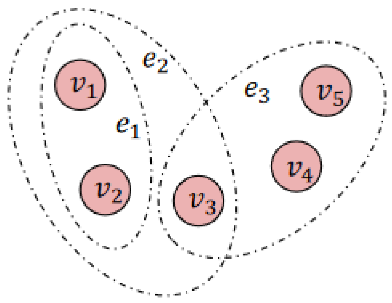


Figure 1.1: Hypergraph with three hyperedges $e_1, e_2, e_3$ that group respectively two, three and three nodes [23].

The setting of our work is a combination of the last two notions, i.e. a multi-layer hypergraph with $k$ layers $G_1, \dots, G_k$, where $G_s = (V, H_s)$ is the hypergraph forming the $s$-th layer. We assume that the set of nodes is the same for each layer, that's why we write $V$ in the tuple instead of $V_s$.

In the algorithms, instead of directly working with the graph, we work with its representation in matrix form. The *adjacency matrix* is a square matrix having as many rows and columns as nodes in the graph; assume that the matrix is called $A$, then the element $A_{ij}$ is defined as follows:

$$A_{ij} = \begin{cases} 1, & \text{if } i \text{ and } j \text{ are adjacent} \\ 0, & \text{otherwise} \end{cases} \tag{1.1}$$

When dealing with hypergraphs, rather than considering tha adjacency matrix, we will rather use the notion of *incidence matrix*, i.e. a matrix with as many rows as nodes and as many columns as hyperedges; assume now that the matrix is called $B$, then the element $B_{ij}$ is defined as follows:

$$B_{ij} = \begin{cases} 1, & \text{if node } v_i \text{ is incident with hyperedge } e_j \\ 0, & \text{otherwise} \end{cases} \tag{1.2}$$

|  | $e_1$ | $e_2$ | $e_3$ |
|---|---|---|---|
| $v_1$ | 1 | 1 | 0 |
| $v_2$ | 1 | 1 | 0 |
| $v_3$ | 0 | 1 | 1 |
| $v_4$ | 0 | 0 | 1 |
| $v_5$ | 0 | 0 | 1 |

Table 1.1: Incidence matrix of the hypergraph represented in 1.1.

## 1.2 Community detection problem

After the basic notions, we are ready to describe the problem we want to focus on: we want to find *communities* (also called *clusters*) in graphs. There is no precise definition of what a community is, but in general we consider communities as densely connected groups of vertices, with only sparser connections between groups. The output of an algorithm developed in order to find clusters is a set of communities $\boldsymbol{C} = \{C_1, \dots, C_k\}$ such that each community contains a non-empty subset of $V$. The set $\boldsymbol{C}$ can be different according to the algorithm used: it is *total* if every node in $V$ belongs to at least one community, otherwise it is *partial*; $\boldsymbol{C}$ is *node-overlapping* if there is at least a node that belongs to more than one cluster, otherwise it is *node-disjoint*. If our system is a multilayer graph, there is also another notion: $\boldsymbol{C}$ is *pillar* if each node together with its counterparts in the other layers belong all to the same community. In this work, our attention is focused on total and pillar clusterings.

If we talk about community, we have to talk about *partition* too. A partition is a division of a graph in clusters, such that each vertex belongs to just one cluster. It is necessary to find a criterion to establish whether a partition is better than another one; the idea is to use a *quality function*, whose job is to assign a number to each partition of a graph.

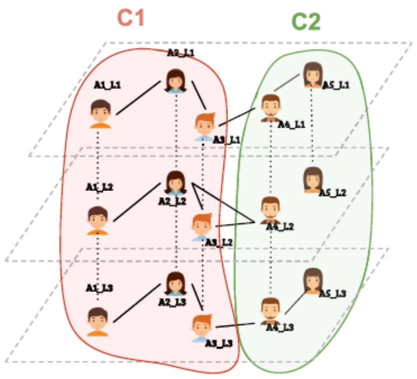


Figure 1.2: Multilayer graph with three layers, five nodes on each layer and two pillar communities. In this case $\boldsymbol{C}$ is also node-disjoint [17].

We can choose among many different quality functions, but the most popular one is the *modularity* of Newman and Girvan [20], the higher is the modularity, the better is the partition found. We have to keep in mind the fact that if a graph can be divided into different communities, it is obviously different from a random graph. Due to the absence of a rigorous definition of communities, we need to compare the graph we are working on with a *null model*, a random graph that shares some structural properties with the original graph; in order to do so, we will sum over all pairs of vertices $v_i$ and $v_j$ that fall in the same group, the difference between the real number of edges between nodes $v_i$ and $v_j$ and the expected number of edges between them. Modularity is defined in the following way:

$$Q = \frac{1}{2m} \sum_{i,j} \left( A_{ij} - \frac{k_i k_j}{2m} \right) \delta \left( C_i, C_j \right), \tag{1.3}$$

where $A$ is the adjacency matrix, $m$ is the total number of edges in the graph, $k_i$ is the sum of all the edges incident to $i$ and $C_i$ is the community node $i$ belongs to. $\delta$ has the obvious meaning of Kronecker delta, i.e. its value is 1 if the two communities coincide, 0 otherwise. The result will be positive if the number of edges within a community is bigger than the number expected according to the null model, negative otherwise. Its value lies in the range [-1/2,1].
Expression 1.3 can be used only for a single-layer graph, this means that for multi-layer graphs we need to modify it. Some researchers have thought about taking modularity average on the layers as quality function to evaluate the partition, while other reasearchers have shown good performance in choosing as quality function a linear combination of average and sampled variance of the modularity of the layers ([22]).

# Chapter 2

# Related work

In this chapter we present an overview of the most important community detection algorithms studied in literature so far. We will focus on methods designed for multi-layer networks and hypergraphs, that are the main characters of this work. To date, we are not aware of any existing methods for multilayer hypergraphs. In principal, we can extend single-layer algorithms to multi-layer networks: we can merge all the layers, reducing the network to a single-layer graph; otherwise we can apply single-layer algorithms to each layer and then combine communities in a proper way. Despite being intuitive ideas, the result is not accurate, because inevitably these steps lead to information loss, so it is better to think about methods that take into account the multilayer structure of these networks. Nevertheless, our discussion will start from the well-known Louvain algorithm for single-layer graphs and its extension to multilayer graphs, since it is one of the most famous methods. Single-layer methods can be extended to hypergraphs too and this can be done if we consider hypergraph's corresponding clique expansion graph, obtained by replacing hyperedges with cliques.

## 2.1 Louvain method

First of all we briefly introduce the Louvain method and its main steps (see [2] for the whole description). The objective function that has to be optimized is the modularity function 1.3 and the method consists of two phases that are repeated iteratively. At the beginning, we assign a different community to each node of the network; after that, we evaluate how much the modularity value would change if we removed a node $v_i$ from its community and we placed it in the community of one of its neighbours $v_j$. Once we are done with all the possible moves, we choose to place node $v_i$ in the community of the neighbour that gives the maximum gain in terms of modularity value. We don't move $v_i$ to another community if, in doing this, there is no positive gain. In the second phase, each community found in the previous stage becomes a supernode and there is an edge between two supernodes if at least one node of one of the two communities is adjacent, in the original graph, to at least one node of the other community. This edge is weighted and its weight is the sum of the weights of the edges between the nodes of the two communities (we don't consider edges within the same community). We then start again with the first phase applied to this modified network, treating it as the

new weighted graph whose partition to find and keeping in mind that each node of the new network contains nodes of the starting graph. This observation is crucial to get an output from the algorithm that regards not only the reduced graph, but the starting one, which is the one of our interest.
Look at 1 for a pseudocode of Louvain method.

**Algorithm 1** Louvain algorithm

```
Input: $G$ graph
Output: $C$ label vector that assigns nodes to communities

repeat
    PHASE 1
    $C \leftarrow$ initial partition, each node of $G$ is a community
    $Q \leftarrow$ modularity value of the initial partition
    $NB \leftarrow$ neighbour nodes vector
    repeat
        for each node $i$ do
            remove node $i$ from its community
            $\Delta Q_1 \leftarrow$ modularity gain for removing node $i$ from its community
            for each neighbour $j$ of node $i$ do
                insert node $i$ into community of node $j$
                $\Delta Q_2 \leftarrow$ modularity gain for inserting $i$ into community of $j$
                $\Delta Q = \Delta Q_1 + \Delta Q_2 \leftarrow$ total modularity gain
            end for
            $\Delta Q^* \leftarrow$ maximum modularity gain, corresponding to node $j^*$
            if $\Delta Q^* > 0$ then
                move node $i$ into community of node $j^*$
                $Q + \Delta Q^* \leftarrow$ modularity value of the new partition
            end if
        end for
    until no more improvements are possible moving one node
    PHASE 2
    Apply Phase 1 to the reduced graph $G$, where each community becomes
    a supernode
until no improved clustering found
return $C \leftarrow$ final partition vector
```

As mentioned before, we can extend this method to multilayer networks. One of the main downsides of modularity approach is that it doesn't work for multilayer networks, but only for single-layer graphs. Since there are many layers, the aim is to maximize the modularity of all layers at the same time, treating the problem as a multiobjective problem. Therefore there is a modularity value $Q_s$ associated to each layer $s$ and expression is the same given in 1.3; the only difference is that in this case adjacency matrix, number $m$ and degree of nodes $k_i$ must refer, in turn, to the specific layer, so expression is:

$$Q = \frac{1}{2m_s} \sum_{i,j} \left( A_{ij}^{(s)} - \frac{k_i^{(s)} k_j^{(s)}}{2m_s} \right) \delta\left(C_i, C_j\right). \tag{2.1}$$

The goal is to maximize all the entries of the modularity vector $Q = \{Q_1, \dots, Q_k\}$; it is necessary to have a quality function $F$ that evaluates the quality of the partition. The idea behind this method is choosing $F$ as the average of the modularity functions across the layers. In [22] a different quality function is used: it is a linear combination of the average and the sampled variance of the modularity of the layers. In the same paper they also focus on the multiobjective structure of the problem, proposing a filter type method that uses the concept of Pareto dominance to delete the least promising result from the filter and keep only the best solutions.

## 2.2 Community detection in multilayer graphs

The most intuitive ideas to detect communities in multilayer graphs are extensions of methods for single-layer graphs: either we merge all the layers, obtaining a collapsed single-layer graph, from which we know how to extract communities, or we apply single-layer network algorithms to each layer and then combine all the outputs using consensus clustering [15]. In both cases we don't preserve the original structure of the system, resulting in low accuracy of results. A possible solution is developing algorithms tailored for mutlilayer graphs that simultaneously take into account all the layers. This section follows survey [12], where we find distinction between algorithms that can only support two-layers and algorithms that perform well for systems with an arbitrary number of layers. Knowing that we don't want to restrict to two-layers case, we directly consider methods developed for a generic number of layers.

### 2.2.1 Matrix factorization

The main contribution for this method can be found in [7] and in [21]. They share the same idea of combining different information by extracting common factors from multiple layers; then they apply general clustering methods. The first one approximates Laplacian matrices, while the second one approximates adjacency matrices. Approximation is made for each layer through a low-rank matrix factorization. Recall that Laplacian matrix $L$ has as many rows and columns as number of graph vertices and it is defined as $L = D - A$, where $D$ is the diagonal degree matrix and $A$ is the adjacency matrix.

### 2.2.2 Pattern mining

In [24] we can find a subgraph mining algorithm for finding quasi-cliques (i.e. a generalization of clique notion) that appear on multiple layers with a frequency above a given threshold. The goal is finding cross-graph quasi-cliques (i.e. a set of vertices belonging to a quasi-clique that appears on all layers and must be the maximal set) in a multi-layer graph that are frequent, coherent, and closed.

The above mentioned algorithms share some properties, but they are also very different. For the sake of brevity, we call the first class of algorithms MF and the second one PM. In a multilayer graph, it is important to find the importance of each layer based on its characteristics; MF considers this aspect, while PM doesn't. However, layer's importance can vary across communities, so it would be useful to distinguish the layer participation in each community; in this case, roles are reversed and PM performs better.
Using MF, the user is free to choose any graph clustering algorithm, resulting in an improvement of community detection quality; PM shows limits in this case.
We talk about locality assumption if an algorithm starts by finding communities from a layer and then discover final communities by expanding the initial communities on the other layers; neither MF nor PM does that. They are also both independent from layers' order, which is good because user doesn't have to worry about how to order them.
Last property concerns overlapping layers; using PM, it can happen that a node belongs to a certain community on a layer and to another community on another layer.

## 2.3 Community detection in hypergraphs

There is short literature about community detection in hypergraphs. The easiest approach to extract communities from hypergraphs is to project the hypergraph to a weighted graph; the weight of an edge between two nodes is given by the number of hyperedges involving that nodes in the original network. Another approach is replacing hyperedges with cliques, still leading back to the graphs. After this step, common community detection methods for graph networks can be applied. It is straightforward to see that this approach leads to oversimplification and information loss.

In [25], the proposed method consists of introducing a null vertex in order to augment a non-uniform hypergraph (i.e., a hypergraph where hyperedges have different sizes) into a uniform multi-hypergraph, and then embeds the multi-hypergraph in a low-dimensional vector space such that vertices within the same community are close to each other.

Another common approach (see [10]) is to conduct higher-order singular value decomposition (HOSVD) on the adjacency tensor and then perform clustering on the output factor matrix. In addition to its slow rate of convergence, the problem of this method is that it applies to uniform hypergraphs and requires degree homogeneity, that is a restrictive feature. Unlike the method cited above (where, in some way, we force uniformity and homogeneity, changing the original structure), in this case there is no information loss during the algorithm, since we decompose adjacency tensor without modifying it.

We can also talk about random walk process on a hypergraph. Assume that the agents are located on the nodes and hop between nodes at discrete times. Hyperedges are considered more or less important depending on their size. The link between random walks and community detection is that a walker should stay for long times inside good communities before escaping them. In this framework, Markov stability is used to extract communities (see [3] for more details).

We now focus on a modularity method (described in detail in [5]) that uses the concept of maximum likelihood together with Louvain algorithm. This latter method has already been presented in the previous section; now we mainly focus on the first one, which applies to hypergraphs. In this framework, a great importance is given to the vector $\mathbf{z}$ of group assignment of nodes, the affinity function $\Omega$ and the degree parameter $\boldsymbol{\theta}$. Affinity function's role is to control the probability of placing a hyperedge at a given node tuple. The goal of maximum likelihood is to solve the following optimization problem:

$$\hat{\mathbf{z}}, \hat{\Omega}, \hat{\boldsymbol{\theta}} = \text{argmax}_{z,\Omega,\theta} P(\mathbf{A}|\mathbf{z}, \Omega, \boldsymbol{\theta}) \tag{2.2}$$

in order to learn estimates of $\hat{\mathbf{z}}$, $\hat{\boldsymbol{\theta}}$ and $\hat{\Omega}$, where $\mathbf{A}$ is a given dataset represented by a collection of hyperedges. As frequently happens in such cases, it is preferable to work with the log-likelihood, since the two functions share the same local optima. The coordinate ascent approach to maximum likelihood alternates between two stages: in the first one, we assume to know an estimate $\hat{\mathbf{z}}$ and we want to obtain new estimates of $\Omega$ and $\boldsymbol{\theta}$; in the second stage, we do the opposite, we assume to know $\hat{\Omega}$ and $\hat{\boldsymbol{\theta}}$ and we want to find an estimate of $\mathbf{z}$. This procedure has to be repeated until convergence. If an estimate for $\mathbf{z}$ is given, then it is easy to solve the first stage and we obtain the following estimates:

$$\hat{\boldsymbol{\theta}} = \mathbf{d} \qquad \hat{\omega} = \frac{\sum_{\mathbf{y}\in Y} \sum_{R\in\mathcal{R}} a_R \delta(\mathbf{z}_R, \mathbf{y})}{\sum_{\mathbf{y}\in Y} \prod_{y\in\mathbf{y}} \mathbf{vol}(y)}, \tag{2.3}$$

where volume term has the following expression: $\mathbf{vol}(l) = \sum_{i=n}^{n} d_i \delta(z_i, l), \quad l = 1, \dots, \bar{l}$; $d_i$ is the (weighted) number of hyperedges in which node $v_i$ appears and $\bar{l}$ is the number of groups.
In order to find $\hat{\mathbf{z}}$, we have, first of all, to make an important assumption: $\Omega$ symmetric with respect to permutations of node labels. The function to optimize to get the estimate we are looking for can be rewritten, after some calculations, in the following form:

$$Q(\mathbf{z}, \Omega, \mathbf{d}) = \sum_{\mathbf{p}\in\mathcal{P}} [\mathbf{cut}_{\mathbf{p}}(\mathbf{z}) \log \Omega(\mathbf{p}) - \mathbf{vol}_{\mathbf{p}}(\mathbf{z})\Omega(\mathbf{p})], \tag{2.4}$$

where $\mathbf{cut}_{\mathbf{p}}(\mathbf{z})$ and $\mathbf{vol}_{\mathbf{p}}(\mathbf{z})$ are defined as below:

$$\mathbf{cut}_{\mathbf{p}}(\mathbf{z}) = \sum_{R\in\mathcal{R}^k} a_R \delta(\mathbf{p}, \phi(\mathbf{z}_R)) \tag{2.5}$$

$$\mathbf{vol}_{\mathbf{p}}(\mathbf{z}) = \sum_{\mathbf{y}\in[\bar{l}]^k} \delta(\mathbf{p}, \phi(\mathbf{y})) \prod_{y\in\mathbf{y}} \mathbf{vol}(y) \tag{2.6}$$

Note that in 2.4, which now becomes our new modularity function, we write $\mathbf{d}$ instead of $\boldsymbol{\theta}$, since the maximum likelihood estimate for $\boldsymbol{\theta}$ when $\mathbf{z}$ is known is $\hat{\boldsymbol{\theta}} = \mathbf{d}$. In particular, $\mathbf{cut}_{\mathbf{p}}(\mathbf{z})$ counts the number of hyperedges that are split by $\mathbf{z}$ into the partition $\mathbf{p}$, while $\mathbf{vol}_{\mathbf{p}}(\mathbf{z})$ is a sum-product of volumes over all grouping vectors $\mathbf{y}$ that induce partition $\mathbf{p}$.

There are many affinity functions $\Omega$ that can be used, but one is preferable for its easier implementation and update, i.e. the All-Or-Nothing (AON) affinity function, that distinguishes only whether a given edge is contained entirely within a single cluster and is defined as:

$$\Omega(\mathbf{p}) = \begin{cases} \omega_{k1}, & \text{if } \|\mathbf{p}\|_0 = 1 \\ \omega_{k0}, & \text{otherwise} \end{cases} \tag{2.7}$$

where $k$ is the number of nodes in partition $\mathbf{p}$; therefore $\Omega$ takes value $\omega_{k1}$ if partition vector is characterized by just one community (i.e. the hyperedge is contained in a single cluster), and takes value $\omega_{k0}$ otherwise. There are also many other affinity functions, each one with a different meaning: the Group Number affinity depends on the number of distinct groups represented in a hyperedge, the Relative Plurality affinity considers the relative difference between the size of the largest group represented in a hyperedge and the next largest group, the Pairwise affinity counts the number of pairs of nodes within the hyperedge whose clusters differ.
Replacing in 2.4 the affinity function $\Omega$ with the AON affinity function, we get the following AON modularity:

$$Q(\mathbf{z}, \Omega, \mathbf{d}) = -\sum_{k=1}^{\bar{k}} \beta_k \left[ \mathbf{cut}_k(\mathbf{z}) + \gamma_k \sum_{l=1}^{\bar{l}} \mathbf{vol}(l)^k \right] + J(\boldsymbol{\omega}), \tag{2.8}$$

where

$$\mathbf{cut}_k(\mathbf{z}) = m_k - \sum_{R \in R^k} a_R \delta(\mathbf{z}_R) \qquad \beta_k = \log \omega_{k1} - \log \omega_{k0} \qquad \gamma_k = \beta_k^{-1}(\omega_{k1} - \omega_{k0}) \tag{2.9}$$

and $m_k$ is the (weighted) number of hyperedges of size $k$. The term $J(\boldsymbol{\omega})$ has the following expression and does not depend on the cluster label vector $\mathbf{z}$; once we know hyperedges, their weights and node degrees, this term remains fixed:

$$J(\boldsymbol{\omega}) = \left[ \sum_{k=1}^{\bar{k}} \beta_k m_k \right] + \left[ \sum_{k=1}^{\bar{k}} \sum_{R \in \mathcal{R}_k} (a_R \log \omega_{k0} - b_R \pi(\mathbf{d}_R) \omega_{ko}) \right].$$

Nevertheless, we don't use $\beta_k$ and $\gamma_k$ values as in 2.9, since we follow the idea of "strict modularity" proposed in [11]; according to this article, values of the AON affinity function have to be chosen so that $\beta_k = 1$ and $\gamma_k = \frac{m_k}{\mathrm{vol}(H)^k}$, where $\mathrm{vol}(H)$ is the sum of all node degrees in hypergraph $H$. This choice implies that AON affinity function values are:

$$\omega_{k0} = \frac{m_k}{\mathrm{vol}(H)^k (e-1)} \qquad \omega_{k1} = e\omega_{k0}$$

Anyway, the user is free to choose whether to specify those parameters' values a priori or not; generally, if the user specify $\boldsymbol{\beta}$ value, his aim is to focus on the hyperedges sizes that are more interesting for the clustering problem he is facing. On the other hand, specifying $\boldsymbol{\gamma}$ leads to modify the sizes of clusters favored by the objective.

*Derivation of 2.8.*
Expression of AON affinity function for a hyperedge $R$ of size $k$ can be written as

$$\Omega(\mathbf{z}_R) = \omega_{k0} + \delta(\mathbf{z}_R)(\omega_{k1} - \omega_{k0}) = \omega_{k0} + \delta(\mathbf{z}_R)\gamma_k \beta_k,$$

and it holds

$$\log \Omega(\mathbf{z}_R) = \log \omega_{k0} + \delta(\mathbf{z}_R)(\log \omega_{k1} - \log \omega_{k0}) = \log \omega_{k0} + \delta(\mathbf{z}_R)\beta_k,$$

Now we call $\mathcal{R}_k$ the set of unordered $k$-tuples of nodes and $\mathcal{T}_k$ the set of ordered $k$-tuples. Knowing that $b_R$ is the number of distinct ways to order the nodes of the hyperedge $R$, we have the following chain of equalities:

$$\sum_{R\in\mathcal{R}_k} b_R \pi(\mathbf{d}_R)\delta(\mathbf{z}_R) = \sum_{T\in\mathcal{T}_k} \pi(\mathbf{d}_T)\delta(\mathbf{z}_T) = \sum_{l=1}^{\bar{l}} \sum_{T\subseteq l} \pi(\mathbf{d}_T) = \sum_{l=1}^{\bar{l}} \mathbf{vol}(l)^k$$

where $T \subseteq l$ means that all nodes from $T \in \mathcal{T}_k$ are in cluster $l$, while $\pi(\mathbf{d}_T)$ is the product of all the entries of the vector $\mathbf{d}_T$.
We can rewrite 2.4 keeping in mind that $\boldsymbol{\theta}_R = \mathbf{d}_R$ and applying the expression of the cut function:

$$\begin{aligned}
Q(\mathbf{z}, \Omega, \boldsymbol{\theta}) &= \sum_{R\in\mathcal{R}} \left[a_R \log \Omega(\mathbf{z}_R) - b_R \pi(\boldsymbol{\theta}_R)\Omega(\mathbf{z}_R)\right] \\
&= \sum_{k=1}^{\bar{k}} \sum_{R\in\mathcal{R}_k} \left[a_R \left(\log \omega_{k0} + \delta(\mathbf{z}_R)\beta_k\right) - b_R \pi(\mathbf{d}_R)\left(\omega_{k0} + \delta(\mathbf{z}_R)\gamma_k\beta_k\right)\right] \\
&= \sum_{k=1}^{\bar{k}} \sum_{R\in\mathcal{R}_k} \left[a_R \log \omega_{k0} - b_R\pi(\mathbf{d}_R)\omega_{k0}\right] + \sum_{k=1}^{\bar{k}} \beta_k \left[\sum_{R\in\mathcal{R}_k} \left(a_R\delta(\mathbf{z}_R) - \gamma_k b_R \pi(\mathbf{d}_R)\delta(\mathbf{z}_R)\right)\right] \\
&= \sum_{k=1}^{\bar{k}} \sum_{R\in\mathcal{R}_k} \left[a_R \log \omega_{k0} - b_R\pi(\mathbf{d}_R)\omega_{k0}\right] + \sum_{k=1}^{\bar{k}} \beta_k \left[m_k - \mathbf{cut}_k(\mathbf{z}) - \gamma_k \sum_{l=1}^{\bar{l}} \mathbf{vol}(l)^k\right] \\
&= J(\boldsymbol{\omega}) - \sum_{k=1}^{\bar{k}} \beta_k \left[\mathbf{cut}_k(\mathbf{z}) - \gamma_k \sum_{l=1}^{\bar{l}} \mathbf{vol}(l)^k\right]
\end{aligned}$$

with $J(\boldsymbol{\omega})$ as above. $\square$

Choosing AON affinity function is crucial for the second phase of Louvain algorithm, that works exactly as in the original algorithm, i.e. it builds a reduced network with supernodes that group together nodes belonging to the same community, and weighted hyperedges that span supernodes (we will better explain this phase in the next Chapter, when we talk about our method). In case of a generic affinity function, this phase would be quite different, since we would move entire sets of nodes in the original hypergraph, rather than greedily moving individual nodes; as a consequence, it requires more evaluations, causing the algorithm to slow down.

# Chapter 3

# Multilayer hypergraphs Louvain-like method

In this chapter, we describe our method to detect communities in multilayer hypergraphs. It is a Louvain-like method, thus the general scheme of this algorithm is very similar to Louvain method scheme for single-layer graphs, but it is an extension of it to multilayer hypergraphs (we use formulas provided in [5]). The biggest difference with respect to classic Louvain is the modularity function used, which, in our case, is the AON modularity 2.8. We are also used to dealing with adjacency matrices when applying Louvain method, since they are the most immediate representations of graphs, but working with hypergraphs it is recommended to use incidence matrices.

We make some assumptions before going into the details of the method. First of all, we assume that each layer has the same set of nodes; in addition, our setting does not allow the presence of interlayer edges/hyperedges (i.e. connections between nodes of different layers), but there are only intralayer connections. All the hypergraphs we use are undirected and we are looking for pillar communities, i.e. each node, together with all its counterparts across the layers, belong to the same community, as shown in Figure 1.2. We want a total clustering too, since every node has to belong to one community.
The input of the algorithm is a (possibly weighted) multilayer hypergraph, where the hypergraph of each layer is represented by its incidence matrix. By itself this input is not enough: we also need, for each layer, hyperedges list, their weights, node degrees and the weighted number of hyperedges of each size. Each input is in form of cell arrays, containing as many cells as layers.
As seen before about Louvain method, it is made of two phases, where the first one begins by putting each node in a different community. For each layer, modularity value is computed; then we compute also the average of these modularity values, since average is the quality function we use to evaluate the quality of the partition. At this point, we consider each node $v_i$ and its neighbourhood, which is the set of all possible neighbours of $v_i$, keeping into account all the layers. We recall that a node is a neighbour of another node if they are adjacent or, in other words, they are part of the same hyperedge. For each neighbour $v_j$ of $v_i$ we compute how much the modularity value would change if

we moved $v_i$ from its community to community of $v_j$; variation of modularity value is pretty fast to be computed, since we can use the following formulas to update cut and volume term (the total variation is the sum of these two terms):

$$\Delta c = \sum_{e \in \mathcal{E}_i} \beta_{\bar{s}_i} a_e [\delta(\hat{\mathbf{z}}_e) - \delta(\bar{\mathbf{z}}_e)] \tag{3.1}$$

$$\Delta v = \sum_{k=1}^{\bar{k}} \beta_k \gamma_k \left[ vol_i^k - (vol_i - \bar{d}_i)^k + vol_{C_j}^k - (vol_{C_j} + \bar{d}_i)^k \right] \tag{3.2}$$

Now we specify what is the meaning of every term. With $\mathcal{E}_i$ we denote the set of hyperedges incident to node $v_i$ and $\bar{\mathbf{s}}$ is the hyperedge size vector; in our framework, this subscript is not important, since all $\boldsymbol{\beta}$ entries are equal to 1; it becomes important, instead, when $\boldsymbol{\beta}$ is defined as in 2.9. Then, $\bar{\mathbf{z}}$ is the current clustering, while $\hat{\mathbf{z}}$ is the clustering obtained by moving node $v_i$ in the community of the neighbour $v_j$ (assume that $C_j$ is the community node $v_j$ belongs to). The $\delta$ function takes value 1 if hyperedge $e$ is fully contained in one community, value 0 if it is split. Lastly, $a_e$ has the obvious meaning of weight of hyperedge $e$. We can now focus on the update of volume term: $\bar{d}_i$ is nothing but the weighted degree of node $v_i$ (if $v_i$ is one of the collapsed nodes of the reduced hypergraph, its degree is given by the sum of degrees of nodes contained in the corresponding cluster), $vol_i$ is the volume of the current cluster of $v_i$ and $vol_{C_j}$ is the volume of the community $C_j$ to which we hypothetically move $v_i$. The reason of these formulas is quite easy. We know that cut term counts the number of hyperdges that are split by the current partition, considering their weights. We move $v_i$, which causes the partition to change, therefore it could happen that a hyperedge $e$, that was fully contained in one community, gets split by doing this move; this is the reason why we need the $\delta$ function. Then, we know that volume term deals with node degrees within a community. Obviously, if we change a node's cluster, the volume terms associated with the new and the old partition change too: for example, from current cluster's volume we need to remove $v_i$ degree, since we are moving $v_i$ to another community; on the other hand, we have to add that degree to the volume term related to the cluster in which we want to put the node.
Obviously, new modularity value is given by the sum of the value computed at the beginning and the variation given by the formulas above. After this, the algorithm computes also the variation of quality function value corresponding to the change of community, which is very easy to calculate, since quality function is the average, which is linear with respect to modularity; therefore we take the average of the variation, which can be either positive (this means that it is better to move node $v_i$ from its community) or negative (this means that it is better that $v_i$ stays where it was). Keeping in mind that we are solving a maximization problem, we are interested in increasing modularity function value, so we choose to move $v_i$ to the community $C_j$ that guarantees the highest increase of quality function value, then we update the cluster label vector. Algorithm goes on with this procedure for every node and then starts again from the first one until no further improvements are possible using one move, this means that whenever we try to move a node, we obtain a negative variation of modularity value.

Second phase begins by constructing the reduced network, i.e. for each layer we write the incidence matrix of the reduced hypergraph whose supernodes are the communities. In this network, hyperedges span multiple supernodes and there is a hyperedge between two or more supernodes if there was at least a hyperedge incident to at least one node of each supernode in the original hypergraph. To determine the weights of each hyperedge in the reduced network we have to sum the weights of the corresponding hyperedges in the original hypergraph. When we construct the reduced network, we have to be careful about the way hyperedges collapse: a simple edge of dimension two can derive from other edges of the same type, but also from hyperedges of higher dimensions that span nodes belonging to two different communities. We have always to keep in mind the original structure of the hypergraph, because this is necessary when we reapply phase one of the method and compute modularity value of the reduced network from scratch. Instead of storing hyperedges weights in the classic way, that is associating to each hyperedge its weight, the idea used in our code is associating to each hyperedge a vector of the same length as the maximum dimension of hyperedges with all the entries equal to zero, except for the entry corresponding to the hyperedge's dimension, that contains its weight. When we construct the collapsed network, we sum the vectors related to the hyperedges that collapse in just one hyperedge/edge, perfectly knowing what is the contribution of each hyperdge's size. Let us give an easy example to show how it works: assume to have two hyperedges of size four, each one with weight two, a hyperedge of size three with weight one and an edge with weight two; their associated vectors are respectively [0 0 0 2], [0 0 0 2], [0 0 1 0], [0 2 0 0]. Assume also that all these hyperdges collapse in an edge; its vector will be the sum of the previous one, i.e. [0 2 1 4] and its weight is the sum of the vector components, i.e. 7. Therefore the overall weight of the hyperedge of the reduced network still remains the sum of the weights of the corresponding hyperedges in the original hypergraph, but to apply again the method we need this distinction, since in 2.8 we have a summation over hyperedges' sizes.
We apply first phase of the method to this reduced network, after having saved new matrices, hyperedges lists, weights, node degrees, and repeat the two phases until second phase finds the same partition found in the first one; this means that no improvements are possible for the algorithm, therefore it stops.
See Appendix A for the whole code.

# Chapter 4

# Numerical experiments

In this chapter, we show some numerical results obtained testing our code on synthetic multilayer hypergraphs. In order to generate the hypergraphs, we have used a degree-corrected hypergraph stochastic blockmodel (DCHSBM) proposed in [5], that is an extension of the well known degree-corrected stochastic blockmodel (DCSBM) for simple graphs. The stochastic blockmodel (SBM) is a random graph model that focuses on partitioning the nodes into communities, while DCSBM takes into account also degree heterogeneity within nodes. In the hypergraph setting, the difference is that we have to consider an affinity function that controls the probability of placing a hyperedge at a given node tuple. A code that generates hypergraph according to this model is available at `https://github.com/PhilChodrow/HypergraphModularity`, implemented in Julia language.
In our tests we use a slightly modified version of this generative model, because it takes as inputs number of nodes, number of hyperedges, number of clusters, but it doesn't allow us to decide the size of each cluster, therefore each cluster contains, from time to time, a different number of nodes. This is bad for our setting, because we work with pillar communities, so we have to be sure that each cluster has a fixed size so that, reordering nodes, each node together with all its counterparts belong to the same community; that's why we have changed a bit the part concerning the creation of the clusters. Once we have generated the hypergraphs, we have saved them as Matlab data in order to use them in our code, which is implemented in Matlab and reported in Appendix A.

We started from an available Matlab code, proposed in [18] that already generalizes Louvain method to multilayer graphs using average of modularity values across the layers. We kept its structure, but changed modularity computation and its update, search of neighborhood of each node, construction of reduced network to adapt the algorithm to our case, in particular to the fact that we work with incidence matrices and not adjacency matrices.

Dealing with synthetic hypergraphs means that we have control on communities, because, together with incidence matrix and list of hyperdges, the generative model gives also the ground truth, i.e. we know to which community each node belongs. When we run our algorithm, we get as output its predicted partition and we compare it with the

ground truth to evaluate its performance. One way to evaluate it is using the accuracy, that is nothing but the percentage of nodes placed in the right community. To compute accuracy we use the confusion matrix, a sort of matrix that has a row for each community of the ground truth and a column for each community returned by the algorithm. On the diagonal we obtain the number of nodes that are in the same community in both the partitions, while off-diagonal there are those placed in the wrong community, so we compute the number of nodes in the wrong community, subtract this number to the total number of nodes and divide again for the total number of nodes in order to find the percentage of nodes placed in the right community.
Another useful tool to evaluate the output partition of the method is the Normalized Mutual Information (NMI), which is, as the name implies, the normalization of the well known Mutual Information, widely used in the context of community detection to evaluate algorithms performances. As for the accuracy, also NMI value varies between 0 and 1, where 1 means that ground truth partition and predicted partition are the same, while 0 means there is no community structure, but just one community containing all the nodes. If we denote with $Y$ the ground truth partition and with $X$ the partition found by the algorithm, NMI has the following expression:

$$\mathrm{NMI}(Y,X) = \frac{2 \cdot \mathrm{MI}(Y,X)}{\mathrm{H}(Y) + \mathrm{H}(X)} \tag{4.1}$$

where $\mathrm{H}(Y)$ is the entropy of the partition $Y$ and $\mathrm{MI}(Y,X)$ is the Mutual Information between the two partitions, that can be calculated through the following expression:

$$\mathrm{MI}(Y,X) = \mathrm{H}(Y) - \mathrm{H}(Y|X) \tag{4.2}$$

where the second term is the conditional entropy of $Y$ with respect to $X$.
For the whole computation of NMI, that uses again confusion matrix, see the code in Appendix B.

As mentioned before, the code used to generate hypergraphs has some parameters that the user can modify and choose according to the tests he wants to perform. One of those parameters is represented by the vector *pvals*, which is the vector of the within-cluster edge placement probabilities, that has as many components as different sizes of hyperedges. Of course, the higher are these probabilities values, the higher will be the accuracy, since it means there are many hyperedges within the same cluster, making it densely connected internally.
First challenge has been choosing values of probabilities so that results we get are not trivial. The main idea was to mimic in some way the experiments done in [22], choosing same probabilities to see how it works in this case with those values. First of all probabilities are defined in a different way in the two papers: in [22] they use $p_{in}$ and $p_{out}$, that are respectively the probabilities of observing an edge between two nodes if they belong to the same cluster or to different clusters. The relation between these two values is $\frac{p_{in}}{p_{out}} = r$, with $r$ varying between 2 and 3.5. On the other side, in [5] $p_{in}$ and $p_{out}$ have more or less the same meaning, but they are not linked to each other by that relation, actually $p_{out} = 1 - p_{in}$. By equating the two expressions of $p_{out}$, we get, after a few simple calculations, that $p_{in} = \frac{r}{r+1}$, therefore an option could be to replace each

probability of the generative hypergraphs model with this value, varying $r$ parameter. This turned out not to be a good idea, since the presence of hyperedges of higher order makes it easier for the algorithm to find the right partition. Indeed, varying $r$ value, already on a single layer hypergraph, led to accuracy and NMI equal to 1, which means that the algorithm detect communities too easily.
At the same time, in [5] they don't consider same probability for each hyperedge size and we follow their choice. Noted that high probabilities related to hyperedges of size strictly bigger than 2 leads to too easy communities to be detected, we reduce these values to $p_{in} = \frac{1}{n^2}$. It remains as before the probability related to edges. Therefore the input vector *pvals* has the following form:

$$pvals = \left[\frac{r}{r+1}, \frac{1}{n^2}, \frac{1}{n^2}\right] \tag{4.3}$$

with $r$ varying between 1 and 2 with step 0.2 (the case with $r = 3.5$ performs all too well already on a single layer).

We tested the method on multilayer hypergraphs with two and three layers (all informative layers), hyperedges of sizes two, three and four, and what we expect is that the accuracy increases increasing the number of layers, since there is more information deriving from the layers. To show how much the presence of two informative layers can improve the performance with respect to just one informative layer, we compare accuracy and NMI obtained also testing the method on a single layer.
We have considered hypergraphs of 500 nodes and 5000 hyperedges with four clusters of equal size, that is four clusters, each one of 125 nodes.

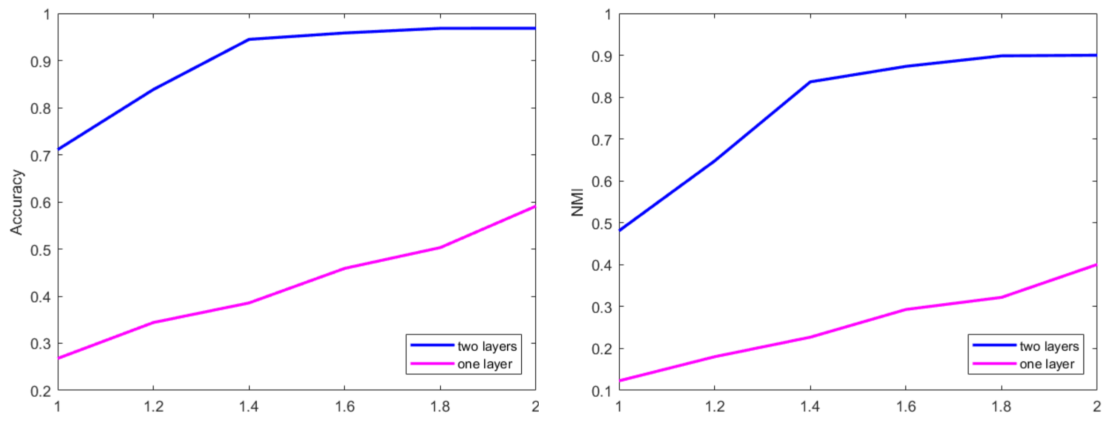


Figure 4.1: On the left: in blue average values of accuracy computed after 10 runs related to 2-layers hypergraphs, generated through DCHSBM with parameter $r \in \{1.0, 1.2, 1.4, 1.6, 1.8, 2.0\}$. In magenta average values of accuracy computed after treating each hypergraph as a single-layer hypergraph.
On the right: average values of NMI referred to the same tests on the left.

In Figure 4.1 we want to focus on the huge difference between accuracy and NMI values when we run the method on a multilayer hypergraph and on a single-layer hypergraph. For each value of $r \in \{1.0, 1.2, 1.4, 1.6, 1.8, 2.0\}$ we have generated 20 hypergraphs according to the DCHSBM, with probability vector as in 4.3. We have then considered 10 multilayer (two layers) hypergraphs and run the code 10 times, computing at the end the average of accuracy and NMI values. After this, we have considered each hypergraph as a single-layer hypergraph and we have run the code for each of them, taking again average values of accuracy and NMI. It's straightforward to understand that probabilities' values are really low in this example, so it is hard for the method to find the right partition having just one layer. We notice, indeed, that the highest accuracy value for $r = 2.0$ is still under 60%. The same happens to NMI values, which are even worse. Things change radically when we have two layers. Recall that we are dealing with informative layers, so having two layers means having more information. This results in a far better performance in the multilayer case. Note that already for $r = 1.4$ average accuracy value is almost 95%.

We now repeat the same numerical experiments done before in presence of three informative layers. Following the comments made about the previous example, we expect to find an even better accuracy in this case.

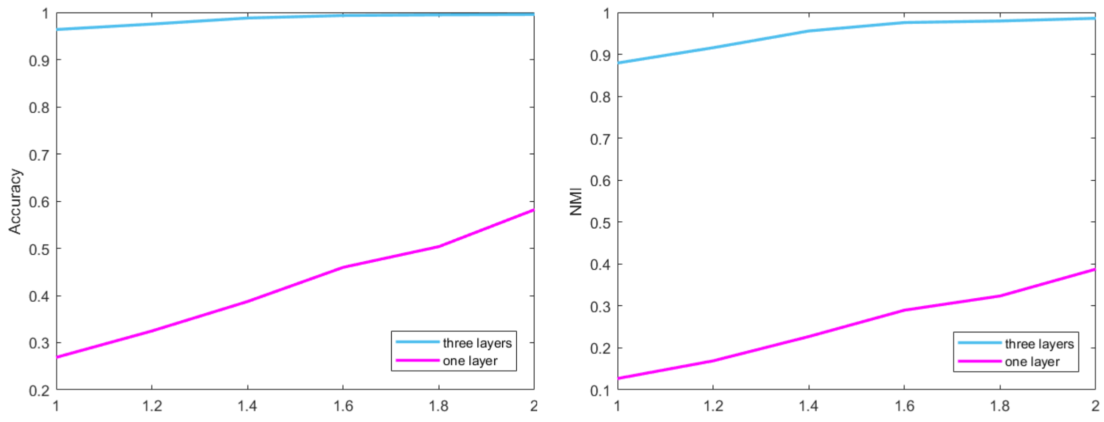


Figure 4.2: On the left: in light blue average values of accuracy computed after 10 runs related to 3-layers hypergraphs, generated through DCHSBM with parameter $r \in \{1.0, 1.2, 1.4, 1.6, 1.8, 2.0\}$. In magenta average values of accuracy computed after treating each hypergraph as a single-layer hypergraph.
On the right: average values of NMI referred to the same tests on the left.

Figure 4.2 confirms our expectation: even if within-cluster edge placement probabilities are low, when we have three layers we reach an almost total accuracy. NMI average values are near 1 too.

As said before, in our case parameter $r = 3.5$ leads to good results also for single-layer hypergraphs, so let's see what happens when $r \in \{3.0, 3.1, 3.2, 3.3, 3.4, 3.5\}$.

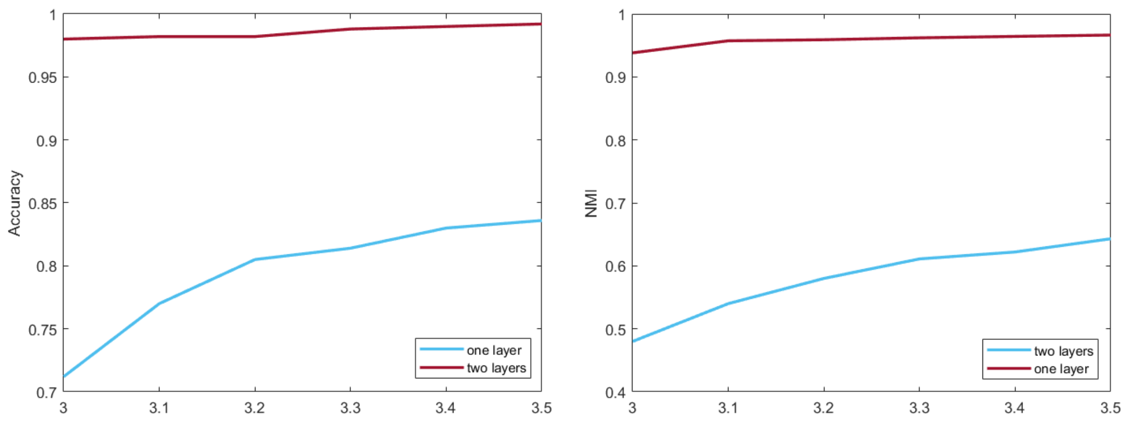


Figure 4.3: On the left: in burgundy average values of accuracy computed after 10 runs related to 2-layers hypergraphs, generated through DCHSBM with parameter $r \in \{3.0, 3.1, 3.2, 3.3, 3.4, 3.5\}$. In light blue average values of accuracy computed after treating each hypergraph as a single-layer hypergraph.
On the right: average values of NMI referred to the same tests on the left.

In Figure 4.3 we can notice that in the worst case, corresponding to single-layer hypergraph with $r = 3.0$, accuracy is anyway higher than 70%, which is not so bad considering that within-cluster hyperedges probabilities are still low. NMI, instead, is characterized by lower values, but this is perfectly in line with what happened in the other examples. Really good is the performance on 2-layers hypergraphs: these values of $r$ are able to reach values of accuracy and NMI very close to those obtained in Figure 4.2 using lower values of $r$ and three layers.
This shows that by increasing even slightly the probabilities related to the edges, while leaving the others unchanged, there is a significant improvement in the perfomance of the algorithm.

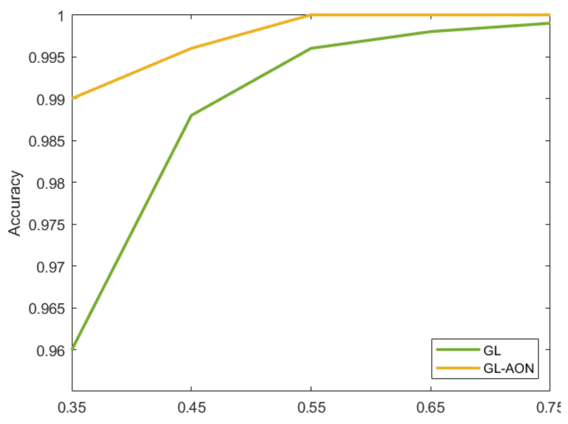
We want to give a further comparison. Not knowing exactly how to deal with hypergraphs, we can think about replacing hyperedges with cliques, turning a hypergraph in a simple graph. At this point, we can apply one of the many algorithms now present in literature to detect communities in these "approximated" (multilayer) graphs.
First of all, we recall how to make this replacement. In [1] we understand how to go from the incidence matrix $I$ of a graph to its adjacency matrix $A$. In case of simple graphs, the adjacency matrix can be obtained from the following expression:

$$A = II^T - D, \tag{4.4}$$

where $D$ is the diagonal matrix whose diagonal entries are the nodes degrees. Although the most suitable matrices for describing hypergraphs are incidence matrices, we can also write a kind of adjacency matrix here using the same expression, with diagonal entries of matrix $D$ that now are the number of hyperedges a vertex belongs to. If the hypergraph is weighted, we write another diagonal matrix $W$, having hyperedges weights as diagonal entries, and expression 4.4 becomes:

$$A = IWI^T - D. \tag{4.5}$$

We considered 2-layers hypergraphs with 500 nodes, 5000 hyperedges and four clusters of 125 nodes; we changed within-cluster edge placement probabilities, because with probability equal to $\frac{1}{n^2}$, Generalized Louvain had very low accuracy; in this case $pvals = [t, 0.2, 0.2]$ with $t \in \{0.35, 0.45, 0.55, 0.65, 0.75\}$. We applied our method to the multilayer hypergraphs, then we replaced each hypergraph with its corresponding clique expansion graph and applied Generalized Louvain to these multilayer graphs.



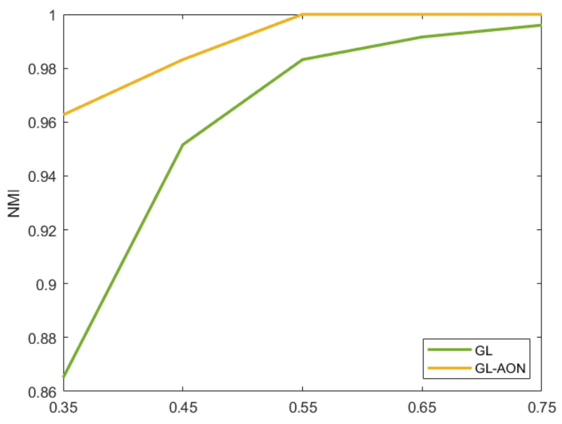


Figure 4.4: On the left: in yellow average values of accuracy computed after 10 runs related to 2-layers hypergraphs, generated through DCHSBM, $pvals = [t, 0.2, 0.2]$ with $t \in \{0.35, 0.45, 0.55, 0.65, 0.75\}$. In green average values of accuracy computed applying Generalized Louvain after replacing each hypergraph with its clique expansion graph. On the right: average values of NMI referred to the same tests on the left.

In Figure 4.4 we note that accuracy and NMI values are higher applying our method instead of Generalized Louvain. It is also a faster procedure: if we have multilayer hypergraphs, we can directly apply our method, instead of using 4.4 to find adjacency matrices and apply Generalized Louvain, which turns out to be less accurate. This is a very important result, because it proves that using hypergraphs is more advantageous than using the corresponding graphs with only pairwise interactions.

# Chapter 5

# Conclusions

In this thesis project we presented a method for community detection in multilayer hypergraphs. Over the past few years many methods to extract communities from multilayer graphs have been developed, but literature is still lacking with regard to multilayer hypergraphs. One of the most common approaches is to replace hypergraphs with their clique expansion graphs in order to apply already existing methods; another technique is projecting the hypergraph to a weighted graph. In both cases we lose hypergraph structure and with this project we want to show that we can develop ad hoc algorithms also for these networks, without the need of approximating them to simple graphs, causing a possible loss of information.

The proposed algorithm is based on the Generalized Louvain method, that extends classic Louvain algorithm to multilayer graphs, using average of modularity across the layers. In order to keep hypergraph structure, we used a different modularity function, called All-Or-Nothing (AON) modularity, whose expression derives from application of maximum likelihood inference.

We implemented our code in Matlab and then tested it on synthetic multilayer hypergraphs, with two and three layers, generated using the degree-corrected hypergraph stochastic blockmodel (DCHSBM). We tested the method on single-layer hypergraphs too, to highlight how much accuracy and NMI values increase adding informative layers. The last numerical experiment was a comparison between accuracy of our method and accuracy of Generalized Louvain method tested on multilayer hypergraphs where each hypergraph is replaced by the corresponding clique expansion graph. Results show that our method performs better.

This work is just the beginning of the research in the field of community detection in multilayer hypergraphs. It would be interesting to understand how the output changes choosing different values of $\beta$ and $\gamma$ parameters or how to deal with the presence of noisy layers. In this latter case, it would be better to use a liner combination of average and sampled variance of modularity values across the layers and, for this aim, it is necessary to find a formula to compute variation of the part related to the variance. Lastly, it would be nice to exploit the multiobjective nature of the problem developing, also in this case, a filter type method based on the concept of Pareto dominance.

# Appendix A

# Code of The Generalized Louvain method with AON affinity function

Code A.1: Generalized Louvain with AON modularity and average as quality function to evaluate partition

```
% Generalized Louvain with AON affinity function
% Communities index by size
% Inputs :
% M : incidence matrices
% e2n : hyperedges
% a : weight of hyperedges
% a_dim : weight of hyperedges stored according to their size
% theta : degree of nodes
% m : number of hyperedges of each size
% z : 1 = Recursive computation
%   : 0 = Just one level computation
%
% Outputs :
% COMTY, structure with the following information
% for each level i :
%   COMTY.COM{i} : vector of community partition
%   COMTY.SIZE{i} : vector of community sizes
%   COMTY.MOD{i} : vector of modularities of clustering on the layers
%   COMTY.Average(i) : average of modularity on the layers
%   COMTY.Niter(i) : number of iteration before convergence

function [COMTY,ending] = GL_AON(M,e2n,a,a_dim,theta,m,z)

if nargin < 6
  error('not enough argument');
end

if nargin < 7
  z = 1;
end

S = size(M{1});
N = S(1); %number of nodes
```

```
k = length(M);
hyp_sizes = cell(k,1);
k_max = cell(k,1);
for kk = 1:k
    hyp_sizes{kk} = sum(M{kk},1);
    k_max{kk} = max(hyp_sizes{kk});
end

%Trivial case
if N==1 || sum(cellfun(@isempty,hyp_sizes))==k
  ending = 1;
  COMTY = 0;
  return;
end

ending = 0;

Niter = 0; %number of iterations

COM = 1:S(1); %initial partition where each node is a community

Q = zeros(k,1); %modularity
vol=zeros(k,length(COM)); %volume term of AON modularity
bet_k=cell(k,1); %beta parameter
gam_k=cell(k,1); %gamma parameter
a_R=cell(k,1); %weights in cut term

for s=1:k
    [Q_,vol_,bet_k_,gam_k_,a_R_] = compute_AONmodularity(N,M{s},e2n{s
       },a{s},a_dim{s},theta{s},m{s},COM);%modularity of layer s
    Q(s)=Q_;
    vol(s,:)=vol_;
    bet_k{s}=bet_k_;
    gam_k{s}=gam_k_;
    a_R{s}=a_R_;
end
Average = sum(Q)/k; %average of modularity

NBc = cell(N,1);
%Neighbourhood
for j=1:N
    for s=1:k
        edges_ind = find(M{s}(j,:)); %hyperedges that contain node j
        for ind = edges_ind
            edge_vec = cell2mat(e2n{s}(ind))';
            NBc{j} = [NBc{j} edge_vec];
        end
    end
    NBc{j} = NBc{j}(NBc{j}~=j); %remove j from its neighbourhood
    NBc{j} = unique(NBc{j}); %neighbourhood of node j
end

gain = 1;
while (gain == 1)  %no increase of average possible moving one node
    gain = 0;
    for i=1:N
        Ci = COM(i); %community of node i
        NB = NBc{i}; %neighbourhood of node i
        G = zeros(1,N); %gain vector
        best_increase = -inf;
        Cnew = Ci;
        for j=1:length(NB) %consider each neighbour j of node i
            Cj = COM(NB(j)); %community of j
            if (G(Cj) == 0) %if we haven't already considered the community of node j
                COM_new = COM; %new list of communities
                COM_new(i) = Cj; %new list of communities with node i in community of j
                if (Ci==Cj)==1
                    Delta_Q = zeros(k,1);
                else
                    Delta_Q = zeros(k,1);
                    for kk = 1:k
                        Delta_Q(kk) = change_AONmod(i,COM_new,COM,theta{kk},bet_k{kk},gam_k{kk},M{kk},e2n{kk},a{kk},a_dim{kk},hyp_sizes{kk},k_max{kk}); %variation of modularity value moving node i
                    end

                    G(Cj) = sum(Delta_Q)/k; %average gain
                    if G(Cj) > best_increase %if positive gain - average increases
                        best_increase = G(Cj); %gain average
                        Q_t = Delta_Q; %gain of modularity
                        Cnew_t = Cj; %new community of node i
                    end
                end
            end
        end
        if best_increase > 0 %if best increase is positive
            Cnew = Cnew_t; %new community of node i
            Q = Q + Q_t; %modularity
            Average = Average + best_increase; %average
        end
        COM(i) = Cnew; %insert node i in the new community
        if (Cnew ~= Ci) %no increase of average possible moving one node
            gain = 1;
        end
    end
    Niter = Niter + 1;
end
Niter = Niter - 1;
[COM] = reindex_com(COM); %reindex the communities by size
%Output
COMTY.COM{1} = COM;
COMTY.MOD{1} = Q';
COMTY.Average(1) = Average;
COMTY.Niter(1) = Niter;
```

```

%Perform second phase of the method
if (z == 1)
  %matrix of the reduced network
  Mnew = M;
  theta_ = theta;
  a_ = a;
  a_dim_ = a_dim;
  e2n_ = e2n;
  COMcur = COM; %current communities
  COMfull = COM; %communities in the original graph
  j = 2; %number of pass (step1+step2)
  while 1
    Mold = Mnew;
    th_old = theta_;
    a_old = a_;
    a_dim_old = a_dim_;
    e2n_old = e2n_;
    S2 = size(Mold{1});
    Nnode = S2(1);
    COMu = unique(COMcur);
    ind_com = sparse(length(COMu),Nnode);
    ind_com_full = sparse(length(COMu),N);
    for ii = 1:length(COMu)
        ind = find(COMfull==ii);
        ind_com_full(ii,1:length(ind)) = ind;
    end
    Mnew = {};
    theta_ = {};
    a_ = {};
    a_dim_ = {};
    for s = 1:k
        M5 = []; %matrix where we sum the rows of the nodes in the
            same community
        for ii = 1:length(COMu)
            ind = find(COMcur==ii);
            ind_com(ii,1:length(ind)) = ind;
            if length(ind)==1
                M5(ii,:) = Mold{s}(ind,:);
                theta_{s}(ii) = th_old{s}(ind);
            else
                M5(ii,:) = sum(Mold{s}(ind,:));
                theta_{s}(ii) = sum(th_old{s}(ind));
            end
            M5(M5~=0)=1; %M5 binary matrix
        end
        M3 = M5; %matrix with hyperedges between communities
        M5s = sum(M5,1);
        in = find(M5s > 1);
        M3 = M5(:,in); %remove hyperedges within the same community,
            they are useless for the reduced network
        a_old{s} = a_old{s}(in);
        e2n_old{s} = e2n_old{s}(in);
        a_dim_old{s} = a_dim_old{s}(in);
        indices = {};
```

```matlab
        for ss = 1:size(M3,2)
            indices{ss} = find(ismember(M3.',M3(:,ss).','rows')'); % find repetitions of each column
        end
        Indices = cellfun(@str2num,unique(cellfun(@num2str,indices,'uni',0),'stable'),'uni',0);
        a_dim_{s} = {};
        for t = 1:length(Indices)
            a_dim_{s}{t} = zeros(k_max{s},1);
            for tt = 1:length(Indices{t})
                a_dim_{s}{t} = a_dim_{s}{t} + a_dim_old{s}{Indices{t}(tt)};
            end
        end
        M4 = [];
        for jj = 1:length(Indices)
            M4(:,jj) = M3(:,Indices{jj}(1)); %keep just one of the repeated columns
            a_{s}(jj) = sum(a_old{s}(Indices{jj}));
        end
        %update hyperedges list
        e2n{s} = {};
        for r = 1:size(M4,2)
            e2n{s}{r} = find(M4(:,r));
        end
        Mnew{s} = M4; %matrix of the reduced hypergraph
    end
    [COMt,e] = GL_AON(Mnew,e2n,a_,a_dim_,theta_,m,0); %apply the first phase on this reduced network
    if (e ~= 1)
      COMfull = sparse(1,N);
      COMcur = COMt.COM{1};
      for p=1:length(COMu)
        ind1 = ind_com_full(p,:);
        COMfull(ind1(ind1>0)) = COMcur(p);
      end
      [COMfull2] = reindex_com(COMfull); %reindex the communitites
      %Output
      COMTY.COM{j} = COMfull2;
      COMTY.MOD{j} = COMt.MOD{1};
      COMTY.Average(j) = COMt.Average(1);
      COMTY.Niter(j) = COMt.Niter;
      Ind = (COMfull2 == COMTY.COM{j-1});
      if (sum(Ind) == length(Ind)) %no changes
        return;
      end
    else
      return;
    end
    j = j + 1;
  end
end
end

% Re-index community partition by size
```

```
function [C] = reindex_com(COMold)

C = sparse(1,length(COMold));
COMolds=sort(COMold);
COMu=COMolds([true;diff(COMolds(:))>0]);
S = sparse(1,length(COMu));
for l=1:length(COMu)
    S(l) = length(COMold(COMold==COMu(l)));
end
[Ss INDs] = sort(S,'descend');
for l=1:length(COMu)
    C(COMold==COMu(INDs(l))) = l;
end
end
```

Code A.2: Function that computes AON modularity from scratch in line 62 of A.1

```
% Function that computes AON modularity of a given partition
% Inputs:
% n : number of nodes
% I : incidence matrix of hypergraph
% R : hyperedges list
% a : hyperedges weights
% a_dim : weight of hyperedges stored according to their size
% theta : node degrees
% m : number of hyperedges of each size
% z : label vector
%
% Outputs:
% Q : modularity
% vol : volume term
% bet_k : beta parameter
% gam_k : gamma parameter
% a_R : weights according to hyperedges size

function [Q,vol,bet_k,gam_k,a_R] = compute_AONmodularity(n,I,R,a,a_dim,theta,m,z)

hyp_sizes = sum(I,1); %sizes of hyperedges
k_min = min(hyp_sizes); %min size of hyperedges
k_max = max(hyp_sizes); %max size of hyperedges

l = unique(z); %clusters without repetitions
vol_H = sum(theta); %sum of all node degrees
bet_k = ones(length(m),1); %beta parameter
gam_k = zeros(length(m),1); %gamma parameter
cut = zeros(length(m),1); %number of hyperedges that contain nodes in distinct cluster
vol = zeros(length(l),1); %volume term
a_R_cut = zeros(k_max,1); %a_R weights in cut term

%Compute volume term
for h = 1:length(l)
```

```
    ind = find(z==l(h));
    vol(h) = sum(theta(ind));
end

%Compute all the other terms for AON modularity
vol_k = zeros(length(m),1);
a_R = zeros(k_max,1);
for k = 1:length(m)
    for r = 1:length(R)
        R_r_vec = cell2mat(R(r));
        if all(z(R_r_vec)==z(R_r_vec(1))) %if all the nodes spanned by
            the hyperedge are in the same community
            cut(k) = cut(k) + 0;
        else
            cut(k) = cut(k) + a_dim{r}(k);
        end
    end
    gam_k(k) = m(k)/(vol_H)^k;
    vol_k(k) = sum(vol.^k); %volume term
end

%Modularity
Q = -sum(bet_k.*(cut+gam_k.*vol_k));

end
```

Code A.3: Function that computes variation of AON modularity value when we move node i from its community to another community in line 104 of A.1

```
% Function that computes variation of AON modularity value when we
   move a node from its community to that of one of its neighbours
% Inputs:
% node : node to move
% z_new : new label vector
% z : old label vector
% theta : node degrees
% bet_k : beta parameter
% gam_k : gamma parameter
% I : incidence matrix
% R : hyperedges list
% a : weight of hyperedges
% a_dim : weight of hyperedges stored according to their size
% hyp_sizes : sizes of hyperdges
% k_max : max size of hyperedges
%
% Outputs:
% Delta_Q : change in modularity

function Delta_Q = change_AONmod(node,z_new,z,theta,bet_k,gam_k,I,R,a,
   a_dim,hyp_sizes,k_max)

%Evaluation of Delta_v
```

```matlab
l = unique(z);
vol = zeros(length(l),1);
for h = 1:length(l)
    ind = find(z==l(h));
    vol(h) = sum(theta(ind));
end
vol_i = vol(l==z(node));
d_i = theta(node);
v_A = vol(l==z_new(node));
Delta_v = 0;
for k1 = 1:length(gam_k)
    Delta_v = Delta_v + bet_k(k1)*gam_k(k1)*(vol_i^k1-(vol_i-d_i)^k1
       ...
        +v_A^k1-(v_A+d_i)^k1);
end

%Evaluation of Delta_c
Delta_c = 0;
a_new = zeros(length(a),1);
a_old = zeros(length(a),1);
edges_inc_ind = find(I(node,:)); %index of hyperedges incident to node
     i
for ind4=edges_inc_ind
    edge_vec = R{ind4};
    if all(z_new(edge_vec)==z_new(edge_vec(1)))
        a_new(ind4) = sum(a_dim{ind4});
    else
        a_new(ind4) = 0;
    end
    if all(z(edge_vec)==z(edge_vec(1)))
        a_old(ind4) = sum(a_dim{ind4});
    else
        a_old(ind4) = 0;
    end
    Delta_c = Delta_c + bet_k(hyp_sizes(ind4))*(a_new(ind4)-a_old(ind4
       ));
end
Delta_Q = Delta_v + Delta_c; %change in modularity

end
```

# Appendix B

# Codes for computation of accuracy and NMI to evaluate partitions

Code B.1: Function that counts the number of nodes in the wrong community to determine accuracy of the method

```
% Function counts number of nodes in the wrong community
% Inputs: E vector ground truth communities
%         P vector predicted communities
% Output: n number of nodes in the wrong community

function [n] = wrong(E,P)

[T] = confusion_matrix(E,P); %confusion matrix

n=0; %counts nodes in wrong community
while ~isempty(T)
    M = max(max(T)); %maximum value of T
    [x,y] = find(T==M); %indices of M in T
    %use x(1) and y(1) because maybe more entries correspond to
        maximum value
    n = n + sum(T(x(1),:)) - T(x(1),y(1));
    n = n + sum(T(:,y(1))) - T(x(1),y(1));
    T(x(1),:) = [];
    T(:,y(1)) = [];
end

end
```

Code B.2: Function that calculates the confusion matrix mentioned in Chapter 4

```
% Function calculates the confusion matrix
% Inputs: E vector expected communities
%         P vector predicted communities
% Output: T confusion matrix

function [T] = confusion_matrix(E,P)
```

```
[E]=reindex_com(E); %reindex community in E
[P]=reindex_com(P); %reindex community in P

%columns ground truth communities - rows predicted communities
Es=sort(E);
Eu=Es([true;diff(Es(:))>0]);
Ps=sort(P);
Pu=Ps([true;diff(Ps(:))>0]);
T = zeros(length(Pu),length(Eu));

for i = Eu
    Pi = P(E==i);
    for j = Pu
        T(j,i) = sum(Pi==j);
    end
end

end
```

Code B.3: Function that computes NMI mentioned in Chapter 4 to evaluate output partition of the method

```
% Normalized Mutual Information (NMI)
% Input: T confusion matrix
% Output: n Normalized Mutual Information

function[n]=NMI(T)

%H(C) entropy of C cluster labels
HC=0;
for i=1:size(T,1)
    HC = HC + sum(T(i,:)) * log2(sum(T(i,:))/sum(sum(T)));
end
%H(Y) entropy of class labels
HY=0;
for j=1:size(T,2)
    HY = HY + sum(T(:,j)) * log2(sum(T(:,j))/sum(sum(T)));
end
%I(Y;C) Mutual Information between Y and C
IYC=0;
for i=1:size(T,1)
    for j=1:size(T,2)
        if T(i,j)~= 0
            IYC = IYC + T(i,j) * log2((T(i,j)*sum(sum(T)))/(sum(T(i,:)
                )*sum(T(:,j))));
        end
    end
end
%NMI
n = -(2*IYC)/(HC+HY);
end
```

# Bibliography

[1] Federico Battiston, Giulia Cencetti, Iacopo Iacopini, Vito Latora, Maxime Lucas, Alice Patania, Jean-Gabriel Young, and Giovanni Petri. Networks beyond pairwise interactions: structure and dynamics. *Physics Reports*, 874:1–92, 2020.

[2] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of communities in large networks. *Journal of statistical mechanics: theory and experiment*, 2008(10):P10008, 2008.

[3] Timoteo Carletti, Duccio Fanelli, and Renaud Lambiotte. Random walks and community detection in hypergraphs. *Journal of Physics: Complexity*, 2(1):015011, 2021.

[4] Jingchun Chen and Bo Yuan. Detecting functional modules in the yeast protein–protein interaction network. *Bioinformatics*, 22(18):2283–2290, 2006.

[5] Philip S Chodrow, Nate Veldt, and Austin R Benson. Generative hypergraph clustering: From blockmodels to modularity. *Science Advances*, 7(28):eabh1303, 2021.

[6] Manlio De Domenico, Vincenzo Nicosia, Alexandre Arenas, and Vito Latora. Structural reducibility of multilayer networks. *Nature communications*, 6(1):6864, 2015.

[7] Xiaowen Dong, Pascal Frossard, Pierre Vandergheynst, and Nikolai Nefedov. Clustering with multi-layer graphs: A spectral perspective. *IEEE Transactions on Signal Processing*, 60(11):5820–5831, 2012.

[8] Santo Fortunato. Community detection in graphs. *Physics reports*, 486(3-5):75–174, 2010.

[9] Elad Ganmor, Ronen Segev, and Elad Schneidman. Sparse low-order interaction network underlies a highly correlated and learnable neural population code. *Proceedings of the National Academy of sciences*, 108(23):9679–9684, 2011.

[10] Debarghya Ghoshdastidar and Ambedkar Dukkipati. Spectral clustering using multilinear svd: Analysis, approximations and applications. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 2015.

[11] Bogumił Kamiński, Valérie Poulin, Paweł Prałat, Przemysław Szufel, and François Théberge. Clustering via hypergraph modularity. *PloS one*, 14(11):e0224307, 2019.

[12] Jungeun Kim and Jae-Gil Lee. Community detection in multi-layer graphs: A survey. *ACM SIGMOD Record*, 44(3):37–48, 2015.

[13] Mikko Kivelä, Alex Arenas, Marc Barthelemy, James P Gleeson, Yamir Moreno, and Mason A Porter. Multilayer networks. *Journal of complex networks*, 2(3):203–271, 2014.

[14] P Krishna Reddy, Masaru Kitsuregawa, P Sreekanth, and S Srinivasa Rao. A graph based approach to extract a neighborhood customer community for collaborative filtering. In *Databases in Networked Information Systems: Second International Workshop, DNIS 2002 Aizu, Japan, December 16–18, 2002 Proceedings 2*, pages 188–200. Springer, 2002.

[15] Andrea Lancichinetti and Santo Fortunato. Consensus clustering in complex networks. *Scientific reports*, 2(1):336, 2012.

[16] Jonathan M Levine, Jordi Bascompte, Peter B Adler, and Stefano Allesina. Beyond pairwise mechanisms of species coexistence in complex communities. *Nature*, 546(7656):56–64, 2017.

[17] Matteo Magnani, Obaida Hanteer, Roberto Interdonato, Luca Rossi, and Andrea Tagarelli. Community detection in multiplex networks. *ACM Computing Surveys (CSUR)*, 54(3):1–35, 2021.

[18] 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, 2010.

[19] Mark EJ Newman. Modularity and community structure in networks. *Proceedings of the national academy of sciences*, 103(23):8577–8582, 2006.

[20] Mark EJ Newman and Michelle Girvan. Finding and evaluating community structure in networks. *Physical review E*, 69(2):026113, 2004.

[21] Wei Tang, Zhengdong Lu, and Inderjit S Dhillon. Clustering with multiple graphs. In *2009 Ninth IEEE International Conference on Data Mining*, pages 1016–1021. IEEE, 2009.

[22] Sara Venturini, Andrea Cristofari, Francesco Rinaldi, and Francesco Tudisco. A variance-aware multiobjective louvain-like method for community detection in multiplex networks. *Journal of Complex Networks*, 10(6):cnac048, 2022.

[23] Chaoqi Yang, Ruijie Wang, Shuochao Yao, and Tarek Abdelzaher. Hypergraph learning with line expansion. *arXiv preprint arXiv:2005.04843*, 2020.

[24] Zhiping Zeng, Jianyong Wang, Lizhu Zhou, and George Karypis. Coherent closed quasi-clique discovery from large dense graph databases. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 797–802, 2006.

[25] Yaoming Zhen and Junhui Wang. Community detection in general hypergraph via graph embedding. *Journal of the American Statistical Association*, pages 1–10, 2022.