Sorbonne Université
2023–2024

M1 Mathématiques
MU4MA056 : Programmation en C++

**TP noté 1 : Modèles à blocks stochastiques**

IMPORTANT : à l'issue du TP, chaque groupe envoie par email trois fichiers `sbm.hpp` , `sbm.cpp` et `test_sbm.cpp` . **Il est impératif de mettre en commentaire, dans tous les fichiers, les nom, prénom et n° d'étudiant de chacun des membres du groupe.**

**Problème :** Considerons un ensemble de $n$ sommets (dits aussi noeuds) $\{1,\ldots,n\}$ qui peuvent être reliés entre eux par des arêtes. On appelle cela un graphe et on peut le représenter à travers sa matrice d'adjacence $A \in \mathcal{M}(n,n,\{0,1\})$ telle que $A_{i,j}=1$ s'il existe une arête qui va du sommet $i$ au sommet $j$, sinon $A_{i,j}=0$. De plus $A_{i,i}=0$ pour tout $i=1\ldots,n$ (un sommet n'a pas d'arête avec lui même).

Un graphe est dit orienté (ou dirigé) si les arêtes sont à sens unique (si le noeud $i$ est lié au noeud $j$, le noeud $j$ n'est pas nécessairement lié au noeud $i$), sinon il est dit non orienté (ou non dirigé), et dans ce cas la matrice d'adjacence est symmétrique.

Un modèle à blocs stochastiques (SBM – Stochastic Block Model) est un modèle qui permet de générer des graphes où l'on suppose que les noeuds appartiennent à $K$ blocs. L'appartenance aux blocs peut être décrite par un vecteur de variables latentes $\mathbf{Z}=(Z_1,\ldots,Z_n)\in\{1,\ldots,K\}^n$ tel que $Z_i=k$ si le noeud $i$ appartient au bloc $k$.

Prenons par exemple une matrice $A=\begin{pmatrix}0&1&1\\1&0&0\\1&0&0\end{pmatrix}$ et un vecteur de variables latentes $\mathbf{Z}=(0,0,1)$, une représentation graphique de ce graphe dirigé est donné en Figure 1.


FIGURE 1 – Graphe dirigé avec trois sommets, correspondant à la matrice d'adjacence $A$ et aux variables latentes $\mathbf{Z}$ (le jaune correspond au bloc 0 et le blanc au bloc 1).

Un SBM avec $K$ blocs est associé à un couple de paramètres $(\boldsymbol{\pi},\boldsymbol{\gamma})$, tels que $\boldsymbol{\pi}=(\pi_1,\ldots,\pi_K)\in(0,1)^K$ sont les proportions des blocs avec $\sum_{k=1}^K \pi_k = 1$, et $\boldsymbol{\gamma}=(\gamma_{k,\ell})_{k,\ell}\in(0,1)^{K\times K}$ est la matrice de connection. Plus précisement $\mathbb{P}(Z_i=k)=\pi_k$ pour tout $k\in\{1,\ldots,K\}$ et $i\in\{1,\ldots,n\}$. Conditionnellement à l'appartenance des noeuds $\mathbf{Z}$, la matrice d'adjacence observée $A=(A_{i,j})_{1\le i,j\le n}\in\{0,1\}^{n\times n}$ vérifie

$$A|\mathbf{Z} = \otimes_{i\neq j} A_{i,j}|Z_j,Z_j = \otimes_{i\neq j}\mathcal{B}(\gamma_{Z_i,Z_j}) \tag{1}$$

avec $\mathcal{B}(\cdot)$ distribution de Bernoulli. En d'autres mots, si le noeud $i$ appartient au bloc $k$ et le noeud $j$ appartient au bloc $\ell$ ($Z_i=k, Z_j=\ell$), la probabilité qu'il y ait une arête entre le noeud $i$ et le noeud $j$ est égale à $\gamma_{k,\ell}$ ($\mathbb{P}(A_{ij}=1|Z_i=k,Z_j=\ell)=\gamma_{k,\ell}$).

Nous allons utiliser la bibliothèque `Eigen` , plus précisément `Eigen/Dense` pour définir les matrices et les vecteurs qui interviennent dans un SBM.

Nous allons donc ajouter à notre fichier d'entête les includes nécessaires et les définitions utiles suivants :

```
#include <Eigen/Dense>
#include <Eigen/StdVector>
```

```
using MatDouble = Eigen::Matrix<double, Eigen::Dynamic, Eigen::Dynamic>;
using VecDouble = Eigen::Vector<double, Eigen::Dynamic>;
using MatInt = Eigen::Matrix<int, Eigen::Dynamic, Eigen::Dynamic>;
using VecInt = Eigen::Vector<int, Eigen::Dynamic>;
```

**Un mémo sur le fonctionnement de cette bibliothèque est fourni en fin de sujet.**

**Implémentation en C++.** Nous imposons la classe suivante dans un fichier `sbm.hpp` :

```
class Graph{
private:
        MatInt m_adj; // of size nxn
        bool m_directed; // for directed or undirected graphs
        VecInt m_Z; //latent variables
};
```

***Attention :*** **On veillera à étiqueter `const` toutes les méthodes et les arguments nécessaires.**

**1.** Utiliser le fichier `sbm.hpp` donné sur Moodle et ajouter les `include` nécessaires. Préparer également un fichier `sbm.cpp` avec les `include` nécessaires.

**2.** Écrire un constructeur de la classe `Graph` qui prend en argument une matrice d'adjacence `A` (de taille $n \times n$), un booléen `directed` et un vecteur d'entiers (au sens Eigen) `Z`, et remplit la matrice `m_adj` avec le contenu de `A`, définit si le graphe est dirigé ou pas à partir du booléen et remplit les variables latentes avec le contenu de `Z`.

**3.** Ajouter comme valeur par défaut de `directed` `true` et comme valeur par défaut de `Z` le vecteur vide. Dans la définition du constructeur, si la dimension de `Z` est nulle, définir les variables latentes comme un vecteur de taille égale au nombre de colonnes (ou de lignes) de la matrice d'adjacence, rempli de zéros.

**4.** Écrire un constructeur de la classe `Graph` qui prend en argument un flux d'entrée `std::istream&` vers un fichier de la forme suivante :

```
n
directed
A_11        A_12 ...      A_1n
A_21        A_22 ...      A_2n
...
A_n1        A_n2 ...      A_nn
Z_1         Z_2       ... Z_n
```

où la dernière ligne est optionnelle. Ce constructeur redimensionne la matrice `m_adj` pour que ce soit une matrice de taille `n x n`, définit si le graphe est dirigé ou pas, remplit les lignes de `m_adj` et attribue un unique bloc par défaut au graphe. Si le flux n'a pas atteint la fin du fichier (c-à-d si la dernière ligne contient les variables latentes) remplir le vecteur de variables latentes `m_Z`. On pourra utiliser le méthode `bool eof()` d'un flux qui renvoie `true` si le flux a atteint la fin du fichier et `false` sinon.

**5.** Ajouter les accesseurs aux champs privés de la classe.

**6.** Ajouter une méthode qui renvoie le nombre de noeuds du graphe.

**7.** Ajouter une méthode qui renvoie le nombre de blocs du graphe. Cette quantité est calculée à partir du vecteur de variables latentes, en supposant qu'il n'y a pas de blocs vides.

On pourra utiliser la fonction `max_element` de la bibliothèque `algorithm` qui prend en entrée un itérateur vers le début et un itérateur vers la fin d'un conteneur et renvoie un itérateur vers le plus grand élément. Si plusieurs éléments sont équivalents à l'élément le plus grand, l'itérateur renvoie au premier de ces éléments.

**8.** Écrire le code de l'opérateur d'écriture `<<` pour un objet de la classe `Graph` .

**9.** Écrire un programme complet `test_sbm.cpp` qui teste les 2 constructeurs : créer d'abord une matrice `A` de taille $3 \times 3$ avec les éléments suivants :

$$\begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix}$$

et un vecteur de variables latentes `Z` $= (0, 0, 1)$. Construire deux objets `simple_graph_directed` et `simple_graph_undirected` de type `Graph` à partir de la matrice `A` et du vecteur `Z` , un dirigé et l'autre non dirigé. Tester ensuite le constructeur par flux, en créant un objet `f_graph` de type `Graph` à partir du fichier `graph.dat` . Afficher les trois graphes.

Nous supposons maintenant que la matrice de connection est de la forme $\gamma_{k,\ell} = \alpha^{k\ell}$ avec $\alpha \in [0, 1]$. Nous imposons la classe suivante dans le même fichier `sbm.hpp` :

```
class SBM{
private:
        std::vector<double> m_pi; // block proportions
        double m_alpha; // Bernoulli param for connectivity
        bool m_directed; //directed or undirected
};
```

**10.** Recopier ce code dans le fichier `sbm.hpp` et ajouter les `include` nécessaires. Ajouter le constructeur naturel de la classe `SBM` qui prend en argument un vecteur de probabilités `pi` , un réel `alpha` et un booléen.

**11.** Ajouter un accesseur `get_K` au nombre de blocs du `SBM` .

Nous rappelons que, en utilisant la bibliothèque `random` du standard C++11 et la bibliothèque `ctime` de la STL, nous pouvons créer et initialiser un générateur de nombres aléatoires

```
std::mt19937_64 G(time(nullptr));
```

Se rappeler du parallel entre un générateur de nombres aléatoires et un espace de probabilités, et placer correctement dans le code la création de `G` .

**12.** Ajouter une méthode `generate_graph` qui prend en argument un entier `n` et une référence vers un générateur de nombres aléatoires de type `mt19937_64 G` et qui renvoie un objet de la classe `Graph`, avec `n` sommets, construit à partir du `SBM` comme il suit :

- Tire les variables latentes $\mathbf{Z} = (Z_1, \ldots, Z_n) \in \{1, \ldots, K\}^n$ en utilisant les probabilités `m_pi`. On pourra utiliser la distribution `discrete_distribution` qui a un constructeur avec la signature suivante

```
discrete_distribution( InputIt first, InputIt last );
```

  où `first, last` sont les itérateurs vers le début et la fin de la plage d'éléments définissant les nombres à utiliser comme poids et qui génère des entiers aléatoires sur l'ensemble $\{0, \ldots, n-1\}$ où $n$ est la taille du vecteur de poids.
- Tire la matrice d'adjacence $A$ selon la loi donnée en (1). On pourra utiliser la distribution `bernoulli_distribution` qui se construit à partir de son paramètre $\gamma$.
- Renvoie le graphe construit à partir de $\mathbf{Z}$ et $A$, dirigé comme le `SBM`.

**13.** Dans le fichier `test_SBM.cpp`, construire un SBM `my_SBM` dirigé à 3 blocs avec $\alpha = 0.5$, $\boldsymbol{\pi} = (0.5, 0.3, 0.2)$ et générer un graphe à $n = 10$ sommets à partir de `my_SBM`.

Nous allons utiliser l'encodage *one-hot* des variables latentes pour pouvoir calculer les statistiques d'un graphe. Cet encodage consiste à encoder une variable à $K$ états sur $K$ bits dont un seul prend la valeur 1, le numéro du bit valant 1 étant le numéro de l'état pris par la variable. Concrètement $Z_i = (Z_{i,1}, \ldots, Z_{i,K})$ tels que $Z_{i,k} = 0$ pour tout $k \neq Z_i$ et $Z_{i,Z_i} = 1$.

**14.** Ajouter à la classe `Graph` une méthode `one_hot_Z` qui renvoie une matrice de taille $n \times K$ où chaque ligne $i$ correspond à l'encodage *one-hot* de $Z_i$.

**15.** Afficher l'encodage *one-hot* des variables latentes de `f_graph`. Vous devrez obtenir la sortie suivante :

```
0 1 0
1 0 0
0 0 1
0 1 0
0 0 1
0 0 1
1 0 0
0 0 1
0 1 0
0 0 1
```

Nous allons calculer les statistiques d'un graphe qui permettent d'obtenir le nombre de sommets dans le bloc $k$, le nombre d'arêtes entre les blocs $k$ et $\ell$ et le nombre d'arêtes manquantes entre les blocs $k$ et $\ell$ qui, dans le cas dirigé, s'écrivent :

$$s_k := \sum_{i=1}^{n} Z_{i,k}, \quad a_{k,\ell} := \sum_{i \neq j} Z_{i,k} Z_{j,\ell} A_{i,j}, \quad b_{k,\ell} := \sum_{i \neq j} Z_{i,k} Z_{j,\ell} (1 - A_{i,j}) \tag{2}$$

Dans le cas non dirigé, les éléments sur les diagonales $a_{k,k}$ et $b_{k,k}$ sont comptés 2 fois, il faudr[a] donc les diviser par 2.

**16.** Ajouter à la classe `Graph` une méthode `count_statistics_s` qui renvoie le vecte d'entiers $s = (s_1, \ldots, s_K)$ et les deux méthodes `count_statistics_a` et `count_statistics_` qui renvoient les matrices de taille $K \times K$ correspondantes.

**17.** Vérifier que les statistiques de `f_graph` sont égales à

$$s = (2,3,5), \quad (a_{k,\ell})_{k,\ell} = \begin{pmatrix} 2 & 6 & 10 \\ 6 & 1 & 5 \\ 10 & 6 & 0 \end{pmatrix}, \quad (b_{k,\ell})_{k,\ell} = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 5 & 10 \\ 0 & 9 & 20 \end{pmatrix},$$

de `simple_graph_directed`

$$s = (2,1), \quad (a_{k,\ell})_{k,\ell} = \begin{pmatrix} 2 & 1 \\ 1 & 0 \end{pmatrix}, \quad (b_{k,\ell})_{k,\ell} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

et de `simple_graph_undirected`

$$s = (2,1), \quad (a_{k,\ell})_{k,\ell} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}, \quad (b_{k,\ell})_{k,\ell} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}.$$