ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

ASSIGNMENT 3

NETWORK ANALYTICS MGT - 416

GROUP 06

Gabriel Muret (250754)
Benoit Fontannaz (250809)
Diego Canton (258304)
Emery Sébastien (258565)
Sami Sellami (272658)
Romain Pichard (273060)

7th June 2020

Problem 1

Considering the linear model $x = Bx + e$ and given the following matrix obtained after the ICA step :

$$W = \begin{pmatrix} 0.2 & 0 & 0 & 0 & 0 \\ 0 & 0.6 & 0 & 0.3 & -0.3 \\ -0.2 & 0.3 & -0.5 & 0.1 & 0 \\ -0.8 & 0.4 & 0 & 0 & 0 \\ -0.6 & 0 & 0.6 & 0 & 0 \end{pmatrix}$$

We need to use the remaining steps of the LiNGAM algorithm to find the matrix B.

We can first find the matrix P such as $PW = \tilde{W}$, where P is a permutation matrix and $\tilde{W}$ doesn't have any zeros element on its diagonal.

We find P = $\begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{pmatrix}$ and $\tilde{W} = \begin{pmatrix} 0.2 & 0 & 0 & 0 & 0 \\ -0.8 & 0.4 & 0 & 0 & 0 \\ -0.6 & 0 & 0.6 & 0 & 0 \\ -0.2 & 0.3 & -0.5 & 0.1 & 0 \\ 0 & 0.6 & 0 & 0.3 & 0.3 \end{pmatrix}$

We now want a new matrix $\tilde{W}'$ with ones on the main diagonal. We compute D such as $PDW = \tilde{W}'$.

We find D = $\begin{pmatrix} 5 & 0 & 0 & 0 & 0 \\ 0 & \frac{-10}{3} & 0 & 0 & 0 \\ 0 & 0 & 10 & 0 & 0 \\ 0 & 0 & 0 & 2.5 & 0 \\ 0 & 0 & 0 & 0 & \frac{5}{3} \end{pmatrix}$ and $\tilde{W}' = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ -2 & 1 & 0 & 0 & 0 \\ -1 & 0 & 1 & 0 & 0 \\ -2 & 3 & -5 & 1 & 0 \\ 0 & -2 & 0 & -1 & 1 \end{pmatrix}$

We can then compute an estimate $\hat{B}$ of B as $\hat{B} = I - \tilde{W}'$:

$\hat{B} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 2 & -3 & 5 & 0 & 0 \\ 0 & 2 & 0 & 1 & 0 \end{pmatrix}$ which is equal to B as it has a strictly lower triangular structure.

We can verify that $PDW = I - B$.

If we want to have the matrices P' ans D' such as W = P'D'(I-B) we need to calculate $P'D' = (PD)^{-1}$ and extract P' and D' from $(PD)^{-1}$ which gives us :

$P' = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{pmatrix}$ and $D' = \begin{pmatrix} 0.2 & 0 & 0 & 0 & 0 \\ 0 & 0.4 & 0 & 0 & 0 \\ 0 & 0 & 0.6 & 0 & 0 \\ 0 & 0 & 0 & 0.1 & 0 \\ 0 & 0 & 0 & 0 & -0.3 \end{pmatrix}$

We can then verify that W = P'D'(I-B)

Problem 2

Since the observed nodes are from a jointly Gaussian distribution, we ca use the following method to calculate the distance between nodes :

$d_{ij} = -log(|\rho_{ij}|)$, with $\rho_{ij} = \frac{Cov(X_i,X_j)}{\sqrt{Var(X_i) \cdot Var(X_j)}}$.

Where, $Cov(X_i, X_j) = \Sigma_{ij}$ and $Var(X_i) = \Sigma_{ii}$.

Then, we apply the algorithm seen during the lecture to learn the latent tree.

Here we define the functions to calculate the distances between nodes and the phi matrix.

Entrée [14]:
```python
import numpy as np
from scipy.io import loadmat
import numpy as np
from scipy.io import loadmat
from collections import defaultdict
import networkx as nx

cov = loadmat("Downloads/Covariance.mat")["O"]
print(cov.shape)
observed_nodes = cov.shape[0]
```
```
(7, 7)
```

Entrée [15]:
```python
def distance(cov, a, b):
    rho = cov[a, b] / np.sqrt(cov[a, a] * cov[b, b])
    return -np.log(np.abs(rho))

def compute_distmat(cov, nb_nodes, observed_nodes, edges):
    distances = np.zeros((nb_nodes, nb_nodes))
    for i in range(nb_nodes):
        for j in range(nb_nodes):
                if i < observed_nodes and j < observed_nodes:
                    if i !=j:
                        distances[i,j] = distance(cov, i, j)
                if i >= observed_nodes:
                    if i != j:
                        childs = list(edges[i])
                        if j in childs:
                            distances[i,j] = distances[j,i] = 0.5 * (distances[childs[0], childs[1]] + phi[childs[0], childs[1], y[2]])
                        else:
                            if j in y:
                                distances[i,j] = distances[j,i] = distances[childs[0],j] - distances[childs[0],i]
                            else:
                                distances[i,j] = distances[j,i] = distances[childs[0], y[2]] - distances[childs[0], i] - distances[j, y[2]]
    return distances

def compute_phimat(distances, nb_nodes):
    phi = np.zeros((nb_nodes, nb_nodes, nb_nodes))
    for i in range(phi.shape[0]):
        for j in range(phi.shape[1]):
            for k in range(phi.shape[2]):
                if i != k and j !=k:
                    phi[i,j,k] = distances[i,k] - distances[j,k]
    return phi
```

We apply the algorithm a first time and find that nodes $\{5, 6\}$ are siblings and node $\{1\}$ is a leaf of node $\{4\}$.

Entrée [16]:
```python
y = list(range(observed_nodes))
y_new = y.copy()
edges = {}
distances = compute_distmat(cov, observed_nodes, observed_nodes, edges)
phi = compute_phimat(distances, observed_nodes)

for i in range(observed_nodes):
    for j in range(observed_nodes):
        if i != j:
            dist = distances[i,j]
            cst = phi[i,j,0]
            c1 = 0
            c2 = 0
            c3 = 0
            for k in range(observed_nodes):
                if k != i and k !=j:
                    if np.isclose(dist, phi[i,j,k]):
                        c1 += 1
                    elif np.isclose(cst, phi[i,j,k]) and not np.isclose(phi[i,j,k], dist) and not np.isclose(phi[i,j,k], -dist):
                        c2 += 1
                    else:
                        c3 + 1
            if c1 == observed_nodes-2:
                edges[j] = [i]
                print(i, 'is a leaf of ', j)
            elif c2 == observed_nodes-2:
                a = True
                for e in list(edges):
                    if np.sum(edges[e]) == i+j:
                        a = False
                if a == True:
                    y_new.append(y_new[-1]+1)
                    edges[y_new[-1]] = [i, j]
                    print(i, "and", j, "are are leaves and siblings")
print('edges :', edges)
print('nodes :', len(y_new))
```
```
4 is a leaf of  1
5 and 6 are are leaves and siblings
edges : {1: [4], 7: [5, 6]}
nodes : 8
```

We apply the algorithm, two more times with the new set of nodes :

Entrée [17]:

```
nb_nodes = len(y_new)
distances = compute_distmat(cov, nb_nodes, observed_nodes, edges)
phi = compute_phimat(distances, nb_nodes)

for i in [1, 7, 2, 3, 0]:
    for j in [1, 7, 2, 3, 0]:
        if i != j:
            dist = distances[i,j]
            cst = phi[i,j,0]
            c1 = 0
            c2 = 0
            c3 = 0
            for k in [1, 7, 2, 3, 0]:
                if k != i and k !=j:
                    if np.isclose(dist, phi[i,j,k]):
                        c1 += 1
                    elif np.isclose(cst, phi[i,j,k]) and not np.isclose(phi[i,j,k], dist) and not np.isclose(phi[i,j,k], -dist):
                        c2 += 1
                    else:
                        c3 + 1
            if c1 == 5-2:
                edges[j] = [i]
                print(i, 'is a leaf of ', j)
            elif c2 == 5-2:
                a = True
                for e in list(edges):
                    if np.sum(edges[e]) == i+j:
                        a = False
                if a == True:
                    y_new.append(y_new[-1]+1)
                    edges[y_new[-1]] = [i, j]
                    print(i, "and", j, "are are leaves and siblings")
print('edges :', edges)
print('nodes :', len(y_new))
```

```
7 is a leaf of  2
edges : {1: [4], 7: [5, 6], 2: [7]}
nodes : 8
```

Entrée [18]:

```
nb_nodes = len(y_new)
distances = compute_distmat(cov, nb_nodes, observed_nodes, edges)
phi = compute_phimat(distances, nb_nodes)

for i in [1, 2, 3, 0]:
    for j in [1, 2, 3, 0]:
        if i != j:
            dist = distances[i,j]
            cst = phi[i,j,0]
            c1 = 0
            c2 = 0
            c3 = 0
            for k in [1, 2, 3, 0]:
                if k != i and k !=j:
                    if np.isclose(dist, phi[i,j,k]):
                        c1 += 1
                    elif np.isclose(cst, phi[i,j,k]) and not np.isclose(phi[i,j,k], dist) and not np.isclose(phi[i,j,k], -dist):
                        c2 += 1
                    else:
                        c3 + 1
            if c1 == 4-2:
                edges[j] = [i]
                print(i, 'is a leaf of ', j)
            elif c2 == 4-2:
                a = True
                for e in list(edges):
                    if np.sum(edges[e]) == i+j:
                        a = False
                if a == True:
                    y_new.append(y_new[-1]+1)
                    edges[y_new[-1]] = [i, j]
                    print(i, "and", j, "are siblings")
print('edges :', edges)
print('nodes :', len(y_new))
```

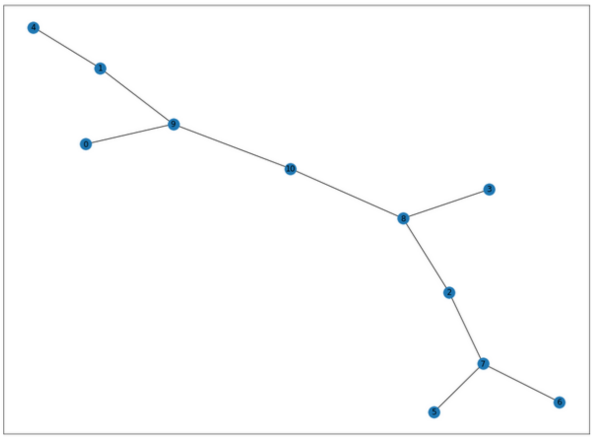
```
2 and 3 are siblings
edges : {1: [4], 7: [5, 6], 2: [7], 8: [2, 3]}
nodes : 9
```

Finally, we link the pieces of trees together to form the full tree :

Entrée [7]:

```
# Connect the 2 remaining nodes
edges[9] = [0, 1]
edges[10] = [8, 9]
print('edges :', edges)

G = nx.from_dict_of_lists(edges)
plt.figure(figsize=(16,12))
nx.draw_networkx(G)
```

```
edges : {1: [4], 7: [5, 6], 2: [7], 8: [2, 3], 9: [0, 1], 10: [8, 9]}
```



Problem 3

I)

Listing 1: Exercice 3.1 Matlab Code

```
clear all;
close all;
load('P1.mat')

T = 20;
Inf = zeros(6);
Sum = 0;

for p = 1:6
    X = XX(p,:,:);
    X = reshape(X, [300,20]);
    for j = 1:6
        if j==p
            break
        end
         Y = XX(j,:,:);
         Y = reshape(Y, [300,20]);
        for k = 1:6
            if (k==j | k==p)
                break
            end
            Z1 = XX(k,:,:);
            Z1 = reshape(Z1, [300,20]);
            for l = 1:6
                if (l==j | l==k | l==p)
                    break
                end
                Z2 = XX(l,:,:);
                Z2 = reshape(Z2, [300,20]);
                for m = 1:6
                    if (m==l | m==p | m==j | m==k)
                        break
                    end
                    Z3 = XX(m,:,:);
                    Z3 = reshape(Z3, [300,20]);
                    for n = 1:6
                        if (n==m | n==p | n==j | n==k | n==l)
                            break
                        end
                        Z4 = XX(n,:,:);
                        Z4 = reshape(Z4, [300,20]);

for i = 2:T
    %DET ZtYt-1
Det1(i) = det(cov([Y(:,1:i),Z1(:,1:i-1),Z2(:,1:i-1),Z3(:,1:i-1),Z4(:,1:i-1)]));
```

```
    %DET  Xt-1Yt-1Zt-1
Det2(i) = det(cov([X(:,1:i-1),Y(:,1:i-1),Z1(:,1:i-1),Z2(:,1:i-1),Z3(:,1:i-1),Z4(
    %DET  Yt-1Zt-1
Det3(i) = det(cov([Y(:,1:i-1),Z1(:,1:i-1),Z2(:,1:i-1),Z3(:,1:i-1),Z4(:,1:i-1)]))
    %DET  Xt-11YtZt-1
Det4(i) = det(cov([X(:,1:i-1),Y(:,1:i),Z1(:,1:i-1),Z2(:,1:i-1),Z3(:,1:i-1),Z4(:,

temp(i) = ((Det1(i)*Det2(i))/(Det3(i)*Det4(i)))
Sum(i) = 0.5*log(temp(i));
Inf(p,j) = Inf(p,j)+Sum(i)

end

                    end
                end
            end
        end
    end
end
```

The code above has been used in order to generate the Mutual Information matrix, but without success.

II)

```python
import numpy as np
import scipy.io
```

```python
mat = scipy.io.loadmat('P1.mat')
XX = np.array(mat['XX'])
shape = XX.shape
```

```python
print(shape)
```

```
(6, 300, 20)
```

```python
X = np.zeros((shape[0], (shape[2]-1)*shape[1]))
Y = np.zeros((shape[0], (shape[2]-1)*shape[1]))
```

```python
for trial in range(shape[1]):
    X[:, trial*(shape[2]-1):trial*(shape[2]-1)+(shape[2]-1)] = XX[:, trial, 0:shape[2]-1]
    Y[:, trial*(shape[2]-1):trial*(shape[2]-1)+(shape[2]-1)] = XX[:, trial, 1:shape[2]]
```

A*X=Y

X'*A'=Y'

```python
x = np.linalg.lstsq(X.T, Y.T)[0].T
```

```
/home/benoit/.local/lib/python3.6/site-packages/ipykernel_launcher.py:1: FutureWarning: `rcond` parameter will chan
ge to the default of machine precision times ``max(M, N)`` where M and N are the input matrix dimensions.
To use the future default and silence this warning we advise to pass `rcond=None`, to keep using the old, explicitl
y pass `rcond=-1`.
  """Entry point for launching an IPython kernel.
```

```python
print(x)
```

```
[[-2.84205926e-03  5.21725094e-03  5.63848517e-01  5.59803736e-03
  -8.36412520e-03 -1.17374773e-03]
 [ 5.33086571e-01 -4.00984322e-03 -2.45224442e-03  5.90555109e-01
  -8.97519745e-03 -5.80217625e-01]
 [-4.42104963e-01  5.74967953e-01 -9.90657735e-03  9.37722623e-03
   5.45924724e-01  4.04094139e-04]
 [-1.19492450e-02  5.84956211e-01 -4.59547876e-01 -5.72627244e-01
  -1.03400327e-02 -5.63366613e-01]
 [-6.92832168e-03  2.93093571e-04  2.77957066e-03 -6.78352185e-03
   1.31477088e-02 -5.10300522e-01]
 [-5.04089750e-01 -3.38558253e-04  5.32719798e-01  5.41530851e-01
   1.28581951e-02 -1.04333503e-03]]
```

Problem 4

In this problem, we would like to implement the Chow-Liu algorithm for the given data set P2. The Chow-Liu algorithm is composed of 3 different phases. First, we have to compute the weight (the mutual information shared between every nodes). After we will have to maximize the total mutual information of the tree. To finish we will give orientations to our edges in the tree.

1. Mutual information computation:

As our variables are Gaussian, we can compute the mutual information with the co-variance matrices using the following formula:

$$I(X;Y) = 1/2 * log(abs((K_X * K_Y)/K_{XY})$$

where K_X and K_y are the determinant of the co-variance matrices of X and Y and K_{XY} is the determinant of the co-variance matrix XY.

This leads to the following implementation :

```
# mutual information computation

I = np.zeros((data.shape[0], data.shape[0]))

for i in range(data.shape[0]):
    for j in range(i,data.shape[0]): #only working with on one half of the matrix to save computation time
            X = data[i,:]
            Y = data[j,:]
            XY = np.column_stack((X,Y)).T
            K_x = np.cov(X)
            K_y = np.cov(Y)
            K_xy = np.linalg.det(np.cov(XY))

            I[i,j] = 0.5 * np.log(np.absolute((K_x * K_y) / K_xy))
            I[j,i] = 0.5 * np.log(np.absolute((K_x * K_y) / K_xy))
```

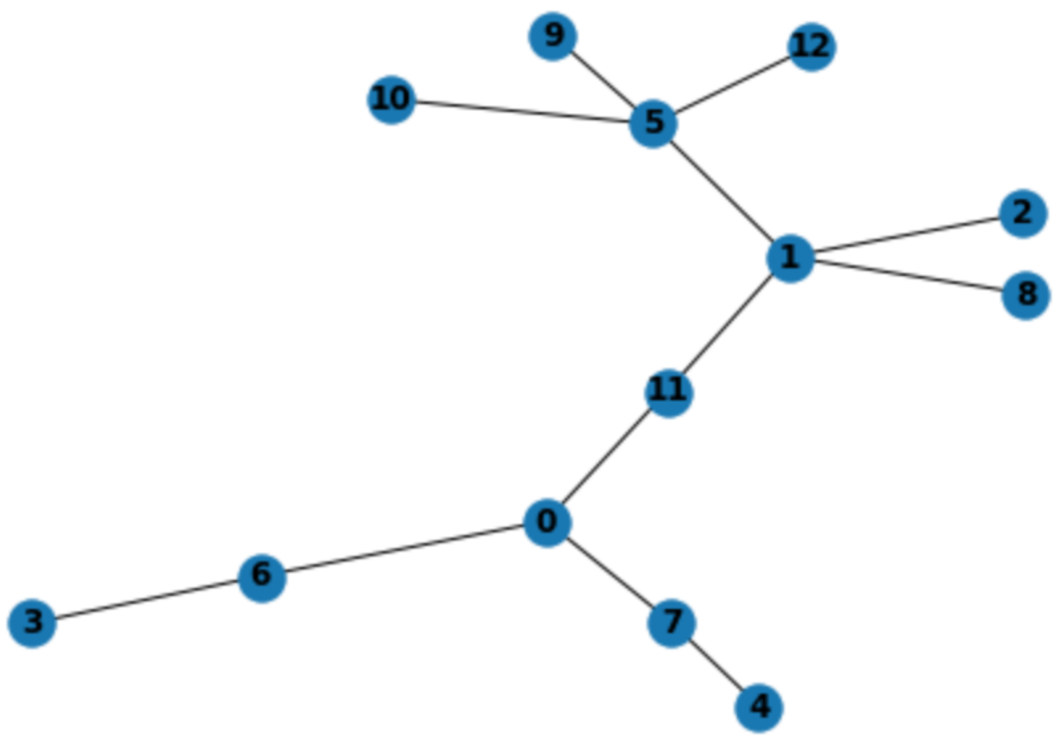
2. Total mutual information maximization:

In this part we decided to order all pairs of nodes by their mutual information I(X;Y). The first step for our algorithm is to select the link with higher mutual information in order to choose the first 2 nodes in the tree and the first edge. After the algorithm will look for the relation between a new nodes that is not in the current graph and a nodes that is already in the graph with the highest mutual information. This operation is repeated until the tree condition is respected.

This part of the algorithm is implemented as follow:

```
for i in range(data.shape[0]): # removing the infinite value on the diagonal
    I[i,i] = 0

G = nx.Graph()
X,Y = find_2D_index(np.argmax(I))
nodes = [X,Y]
G.add_nodes_from((X,Y))
G.add_edge(X,Y)

I[find_2D_index(np.argmax(I))] = 0 # we remove the value in order to forget it because it destroy the tree

n_iter = data.shape[0]**2
n=0

weight = I[0,:]
for i in range(1,I.shape[0]): # we create a single vector "weight" compsed with the value of matrix I
    weight = np.concatenate((weight,I[i,:]))
values = np.sort(weight)[::-1]

while n < n_iter: # we iterate to find the solution by testing combinaison in order to build the tree
    n += 1
    for i in range(len(weight)): # we go through all vecteur if a valid node isn't find before
        max_val = values[i]
        ind = np.where(weight == max_val)
        X,Y = find_2D_index(ind[0][0])

        if X not in nodes or Y not in nodes:
            if X in nodes and Y not in nodes:
                G.add_nodes_from((X,Y))
                G.add_edge(X,Y)
                nodes.append(X)
                nodes.append(Y)
                nodes = list(set(nodes))
                break

            if Y in nodes and X not in nodes:
                G.add_nodes_from((X,Y))
                G.add_edge(X,Y)
                nodes.append(X)
                nodes.append(Y)
                nodes = list(set(nodes))
                break
```

It produces the following skeleton:



Appendices