Assignment 2

- *Auteur* -
Hamza Benjelloun

22 décembre 2020

0.0.1 Dependencies in a Directed Graphical Model

Question 1

No, because $X_{r,c}$ is observable.

Question 2

No

Question 3 :

A = $\{\mu_{r,c}\}$

Question 4 :

No

Question 5 :

No

Question 6 :

$B = \{Z_n^m; n, m \in [N] \times [M]\} \cup \{C^n; n \in [N]\}$

0.0.2 Likelihood of a Tree Graphical Model

Question 2.2.7

We note $X_{o \cap \downarrow U}$ the observation under the node U.

$$P(\beta/T, \theta) = P(X_{o \cap \downarrow root}/T, \theta) = \sum_{k \in [k]} P(X_{o \cap \downarrow root}/T, \theta, root = k) P(root = k/T, \theta)$$

We note $s(u, i) = P(X_{o \cap \downarrow U}/T, \theta, U = i)$, Then the equation becomes :

$$P(\beta/T, \theta) = \sum_{k \in [k]} P(root = k/T, \theta) s(root, k)$$

Since T is binary let V and W be the two child of U.

$$
\begin{aligned}
s(u,i) &= P(X_{o\cap\downarrow U}/T,\theta,U=i) = P(X_{o\cap\downarrow V}/T,\theta,U=i)P(X_{o\cap\downarrow W}/T,\theta,U=i) \\
&= [\sum_{j\in[K]} P(X_{o\cap\downarrow V},V=j/T,\theta,U=i)][\sum_{k\in[K]} P(X_{o\cap\downarrow W},W=K/T,\theta,U=i)] \\
&= [\sum_{j\in[K]} P(X_{o\cap\downarrow V}/T,\theta,V=j)P(v=j/U=i)][\sum_{k\in[k]} P(X_{o\cap\downarrow W}/T,\theta,W=k)p(W=k/U=i)] \\
&= [\sum_{j\in[K]} p(V=j/U=i)s(V,j)][\sum_{k\in[K]} p(W=k/U=i)s(W,k)]
\end{aligned}
$$

Let's implement a dynamic algorithm to compute this probability :

Listing 1 – Insert code directly in your document

```
def s(name_node, value, tree_topology, theta, beta):
    """
    compute recursivly the probability of observing leaves under the node '
        name_node'
    giving its value
    """
    if(not(isnan(beta[name_node]))):
        # It is a leaf
        return beta[name_node] == value
    else:
        # inner node (apply formula see report)
        # find the names of its child
        for i in range(name_node, len(tree_topology)):
            if tree_topology[i] == name_node:
                children_1 = i
                if(i < len(tree_topology) and tree_topology[i+1] ==
                    name_node):
                    # he has another child
                    children_2 = i + 1
                else:
                    children_2 = None
                break
        proba_1, proba_2 = 0, 1
        K = len(theta[0])
        for i in range(K):
            proba_1 += theta[children_1][value][i]*s(children_1, i,
                tree_topology, theta, beta)
        if(children_2 != None):
            proba_2 = 0
            for i in range(K):
                proba_2 += theta[children_2][value][i]*s(children_2, i,
                    tree_topology, theta, beta)
        return proba_1*proba_2
```

```
def calculate_likelihood(tree_topology, theta, beta):
    print("Calculating the likelihood...")
    likelihood = sum([theta[0][k] * s(0, k, tree_topology, theta, beta)
        for k in range(len(theta[0]))])
    return likelihood
```

Question 2.2.8 :

	Small Tree	Medium Tree	Large Tree
Sample 0	0.016178983188381856	$4.3359985610595595e-18$	$3.287622233372845e-69$
Sample 1	0.015409920999590558	$3.094115541974649e-20$	$1.109451841712131e-66$
Sample 2	0.011368474549717017	$1.050009120509836e-16$	$2.5224240937554143e-68$
Sample 3	0.00864042722338585	$6.585311240142626e-16$	$1.2423555476116806e-66$
Sample 4	0.04091494618599329	$1.4880108219386272e-18$	$3.535477501915256e-69$

0.0.3 Simple Variational Inference :

Question 2.3.9

Listing 2 – Insert code directly in your document

```
import numpy as np
from matplotlib.pyplot import contour
from scipy.special import gamma as fct_gamma
from scipy.stats import norm, gamma
import matplotlib.pyplot as plt
"""
    Approximate the posterior probability P(Z/X) using VI
"""


def generate_data(a0, b0, mu0, lambda0, size):
    """
    Generate observations X knowing the priors probabilities P(T) and P(
        mu/T)
    we draw T from a gamma distribution, then we drawn mu from a
        normal distribution
    knowing T, and finally we drawn X_i from a normal distribution
        knowing T and mu.
    """
    if b0 != 0:
      t = np.random.gamma(a0, 1/b0, 1)
    else:
```

```
        t = 1
    if(t == 0):
        t = 10**(-20)
    if lambda0 != 0:
        mu = np.random.normal(mu0, 1/np.sqrt((lambda0 * t)), 1)
    else:
        mu = 0
    return np.random.normal(mu, 1/np.sqrt(t), size)

def compute_posterior_parameter(X, a0, b0, mu0, lambda0):
    N = len(X)
    # simple parameter
    mu_N = (lambda0 * mu0 + sum(X))/(lambda0 + N)
    a_N = a0 + N/2
    # find lambda_N and b_N by an iteratifve approach
    b_N = np.random.random()
    mean_t = a_N/b_N
    iter = 100
    while(iter > 0):
        lambda_N = (lambda0 + N)*mean_t
        mean_mu = mu_N
        mean_mu_square = 1/lambda_N + mu_N**2
        b_N = b0 + 0.5 * ((N + lambda0)*mean_mu_square - 2 *(sum(X) +
            lambda0*mu0) * mean_mu + lambda0*(mu0**2) + sum(X**2))
        mean_t = a_N/b_N
        iter -= 1
    return [mu_N, lambda_N, a_N, b_N]


def normal_gamma(mu, lamda, a, b, mus, ts):
  # the formula of the exact posterior is calculated in the report
  gamma = np.array((b**a)*np.sqrt(lamda) / (fct_gamma(a)*np.sqrt(2*np.
      pi)) * ts**(a-0.5)* np.exp(-b*ts))
  normal = np.array(np.exp(-0.5*lamda*np.dot(ts,((mus-mu)**2).T)))
  return gamma * normal


def normal_dist(mu, t, X):
  return np.array(np.sqrt(t/(2*np.pi)) * np.exp(-0.5 * np.dot(t, np.
      transpose((X - mu)**2))))


def gamma_dist(a, b, X):
  return np.array((1/fct_gamma(a)) * b**a * X**(a-1) * np.exp(-b*X))


def main():
    # choice of parameters
    a0, b0, mu0, lambda0, size = 0, 0, 0, 0, 100
    # generate data
    D = generate_data(a0, b0, mu0, lambda0, size)
```

```
    # compute an approximation of posterior probability
    mu_N, lambda_N, a_N, b_N = compute_posterior_parameter(D, a0, b0,
        mu0, lambda0)
    # plot
    mus = np.linspace(norm(mu_N, 1/np.sqrt(lambda_N)).ppf(0.01), norm(
        mu_N, 1/np.sqrt(lambda_N)).ppf(0.99), 100)
    ts = np.linspace(gamma(a = a_N, scale = 1/b_N, loc = 0).ppf(0.01),
        gamma(a = a_N, scale = 1/b_N, loc = 0).ppf(0.99), 100)
    #
    q_u = normal_dist(mu_N, lambda_N, mus.reshape(len(mus), 1))
    q_t = gamma_dist(a_N, b_N, ts.reshape(len(ts), 1))
    Z = q_u*q_t

    N = len(D)
    mu_exacte = (lambda0 * mu0 + np.sum(D)) / (lambda0 + N)
    lambda_exacte = lambda0 + N
    a_exacte = a0 + N/2
    b_exacte = b0 + 0.5*sum((D-np.mean(D))**2) + (lambda0*N*(np.mean(
        D)-mu0)**2)/(2*(lambda0+N))

    Z_exacte = normal_gamma(mu_exacte, lambda_exacte, a_exacte,
        b_exacte, mus.reshape(len(mus), 1), ts.reshape(len(ts), 1))
    # plot
    X, Y = np.meshgrid(mus, ts)
    contour(X, Y, Z_exacte, 5, colors='red')
    contour(X, Y, Z, 5, colors='green')
    plt.xlabel('mu')
    plt.ylabel('tau')
main()
```

Question 2.3.10 :

We have :

$$P(\mu,\tau/D) = \frac{P(D)P(\mu,\tau)}{P(X)} \propto P(D/\mu,\tau)P(\mu,\tau) = P(D/\mu,\tau)P(\tau)P(\mu/\tau)$$

We note $\bar{x}$ the empirical mean of X and s^2 the empirical variance.

$$\begin{aligned} P(D/\mu,\tau) &= (\frac{\tau}{2\pi})^{\frac{N}{2}} \exp{(\frac{-\tau}{2}\sum_{n=1}^{N}(x_n-\mu)^2)} = (\frac{\tau}{2\pi})^{\frac{N}{2}} \exp{(\frac{-\tau}{2}\sum_{n=1}^{N}((x_n-\bar{x})-(\mu-\bar{x}))^2)} \\ &= (\frac{\tau}{2\pi})^{\frac{N}{2}} \exp{(\frac{-\tau}{2}(\sum_{n=1}^{N}(x_n-\bar{x})^2 + N(\mu-\bar{x})^2 - 2(\mu-\bar{x})(n\bar{x}-\sum_{n=1}^{N}x_n)))} \\ &= (\frac{\tau}{2\pi})^{\frac{N}{2}} \exp{(\frac{-\tau}{2}(Ns^2 + N(\mu-\bar{x})^2)} \end{aligned}$$

Then the posterior probability becomes :

$$
\begin{aligned}
P(\mu,\tau/D) &\propto (\frac{\tau}{2\pi})^{\frac{N}{2}} \exp{(\frac{-\tau}{2}(Ns^2+N(\mu-\bar{x})^2)} \times (\frac{\tau}{2\pi})^{\frac{1}{2}} \exp{(\frac{-\tau}{2}(\mu-\mu_0))} \\
&\quad \times \tau^{a_0-1} \exp{(-b_0\tau)} \\
&\propto \tau^{\frac{N-1}{2}+a_0} \exp{(-b_0\tau)} \exp{(\frac{-\tau}{2}(Ns^2+N(\mu-\bar{x})^2) - \frac{\tau\lambda_0}{2}(\mu-\mu_0))} \\
&\propto \tau^{\frac{N-1}{2}+a_0} \exp{(-(\frac{1}{2}Ns^2+b_0)\tau)} \exp{(\frac{-\tau N}{2}(\mu-\bar{x})^2) - \frac{\tau\lambda_0}{2}(\mu-\mu_0))}
\end{aligned}
$$

We have : $N(\mu-\bar{x})^2+\lambda_0(\mu-\mu_0)^2$ is a quadratic equation of u then it can be factorized in format $a(u-b)^2+c$ to show a normal distribution. We have :

$$
\begin{aligned}
N(\mu-\bar{x})^2+\lambda_0(\mu-\mu_0)^2 &= (\lambda_0+N)\mu^2 - 2(\lambda_0\mu_0+N\bar{x})\mu + \lambda_0\mu_0^2 + N\bar{x}^2 \\
&= (\lambda_0+N)\left(\mu - \frac{\lambda_0\mu_0+N\bar{x}}{\lambda_0+N}\right)^2 + \frac{\lambda_0 N(\bar{x}-\mu_0)^2}{\lambda_0+N}
\end{aligned}
$$

Therefore :

$$
\begin{aligned}
P(\mu,\tau/D) \propto \tau^{\frac{N-1}{2}+a_0} &\exp{(-(\frac{1}{2}Ns^2+b_0+\frac{\lambda_0 N(\bar{x}-\mu_0)^2}{\lambda_0+N})\tau)} \\
&\times \exp{(-\frac{\tau}{2}(\lambda_0+N)(\mu-\frac{\lambda_0\mu_0+N\bar{x}}{\lambda_0+N})^2)}
\end{aligned}
$$

Conclusion :
$P(\mu,\tau/D)$ is a Normal-gamma distribution.

$$
\begin{aligned}
p(\mu,\tau/D) &= NormalGamma(\mu',\lambda',a',b') \\
\mu' &= \frac{\lambda_0\mu_0+N\bar{x}}{\lambda_0+N} \\
\lambda' &= \lambda_0+N \\
a' &= a_0+\frac{N}{2} \\
b' &= b_0+\frac{1}{2}\left(Ns^2+\frac{\lambda_0 N(\bar{x}-\mu_0)^2}{\lambda_0+N}\right).
\end{aligned}
$$

Question 2.3.11 :

From the previous question we see that the difference between the exact and the approximate posterior is b and λ Let's try our algorithm on interesting cases :

In the following cases we represent the exact posterior in red and the approximate posterior in green.

Case where $\mu_0 = 0, \lambda_0 = 0, a_0 = 0, b_0 = 0$:

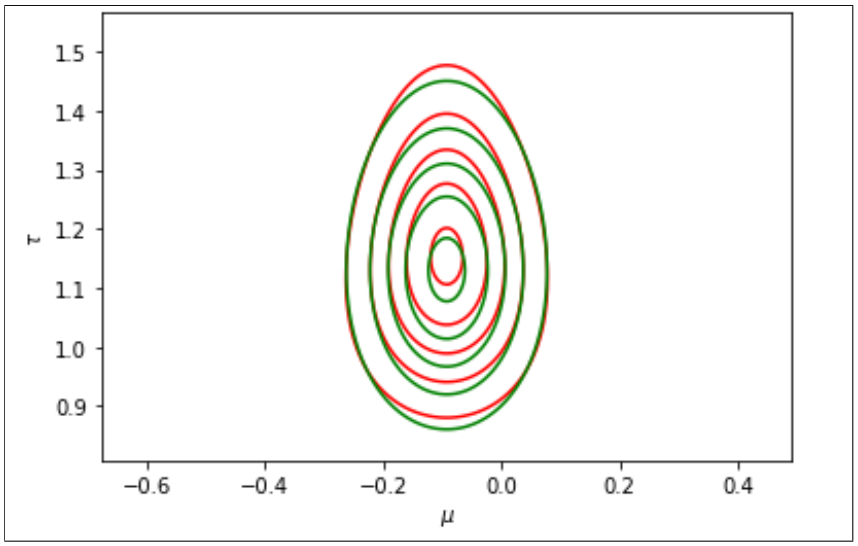

FIGURE 1 –

This case correspond to non informative priors, we don't have any information about μ and τ. In my case the convergence was fast (7 iteration needed). We notice that the mean of μ is always correct.

Case where the size of the sample is very small (size = 10)

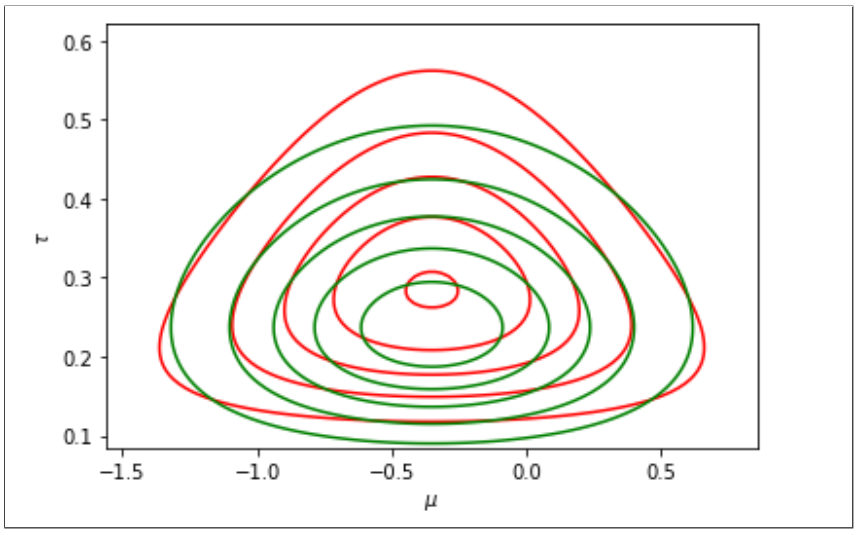

FIGURE 2 –

In this case the mean of μ is correct, that is because we assign the correct value to μ_N in the first iteration and we don't change it. However the mean of τ is not so good, because it is inferred using data D, and it is very small.

case where all parameters different to 0 :

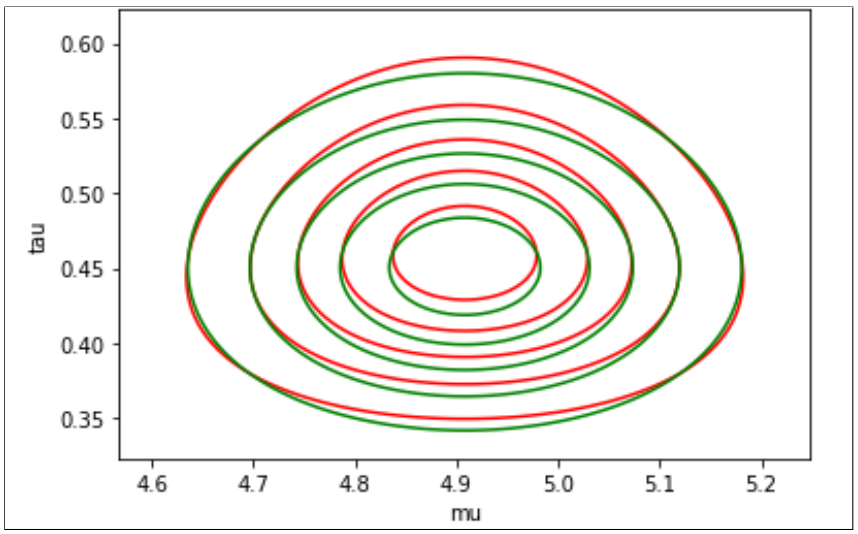

FIGURE 3 – $a_0 = 1, b_0 = 4, \mu_0 = 5, \lambda_0 = 2, size = 100$

The convergence is quite good.

0.0.4 Mixture of trees with observable variables

Question 2.4.12 :

Listing 3 – Insert code directly in your document

```
def compute_q(samples, r):
    # for a given responsabilities r, a node X_s, a node X_t and values a and
        b compute q
    num_clusters = r.shape[1]
    num_nodes = samples.shape[1]
    q = 0
    Q = np.zeros((num_clusters, num_nodes, num_nodes, 2, 2))
    for k in range(num_clusters):
        for s in range(num_nodes):
            for t in range(num_nodes):
                for a in [0, 1]:
                    for b in [0, 1]:
                        q = 0
                        for n, x in enumerate(samples):
                            if x[s] == a and x[t] == b:
                                q += r[n, k]
                        Q[k, s, t, a, b] = q/np.sum([r[n, k] for n in range(len
                            (samples))])
    return Q
```

```
def compute_I(Q):
    num_clusters = Q.shape[0]
    num_nodes = Q.shape[1]
    matrix_I = np.ones((num_clusters, num_nodes, num_nodes))
    for k in range(num_clusters):
        for s in range(num_nodes):
            for t in range(num_nodes):
                I = 0
                for a in range(2):
                    for b in range(2):
                        I += Q[k, s, t, a, b] * np.log(Q[k, s, t, a, b]/Q[k, s,
                            s, a, a] / Q[k, t, t, b, b])
                matrix_I[k, s, t] = I
    return matrix_I


def em_algorithm(seed_val, samples, num_clusters, max_num_iter=100):

    # Set the seed
    np.random.seed(seed_val)

    # TODO: Implement EM algorithm here.

    # Start: Example Code Segment. Delete this segment completely before
        you implement the algorithm.
    print("Running EM algorithm...")
    loglikelihood = []
    from Tree import TreeMixture
    # initialize thetas and the tree randomly
    tm = TreeMixture(num_clusters=num_clusters, num_nodes=samples
        .shape[1])
    tm.simulate_pi(seed_val=seed_val)
    tm.simulate_trees(seed_val=seed_val)
    #
    topology_list = []
    theta_list = []
    num_samples = np.size(samples, 0)
    num_nodes = np.size(samples, 1)

    loglikelihood = []
    for i in range(num_clusters):
        topology_list.append(tm.clusters[i].get_topology_array())
        theta_list.append(tm.clusters[i].get_theta_array())
    loglikelihood = []
    for iter in range(max_num_iter):
        # step 1: compute responsabilities
        responsabilities = np.ones((len(samples), num_clusters))
        for n, x in enumerate(samples):
            for k in range(num_clusters):
                responsabilities[n, k] *= tm.pi[k]
                root = tm.clusters[k].root
```

```
            visit_list = [root]
            # BFS
            while len(visit_list) != 0:
                cur_node = visit_list[0]
                visit_list = visit_list[1:] + cur_node.
                    descendants
                par_node = cur_node.ancestor

                if par_node is None:
                    cat = cur_node.cat[x[int(cur_node.name)]]
                else:
                    cat = cur_node.cat[x[int(par_node.name)]][x[int(
                        cur_node.name)]]

                responsabilities[n, k] *= cat
    responsabilities += sys.float_info.min

    marginal = np.reshape(np.sum(responsabilities, axis=1), (len(
        samples), 1))
    loglikelihood.append(np.sum(np.log(marginal)))
    for n in range(len(samples)):
        #normalize
        responsabilities[n] = responsabilities[n]/np.sum(
            responsabilities[n])
    # step 2: compute pi_k
    for k in range(num_clusters):
        tm.pi[k] = np.mean([responsabilities[i][k] for i in range(len(
            samples))])
    # step 3: construct the graph
    Q = compute_q(samples, responsabilities) + sys.float_info.
        min
    I = compute_I(Q)
    num_clusters = responsabilities.shape[1]
    num_nodes = samples.shape[1]



    graphs = []
    trees = []
    num_nodes = samples.shape[1]
    for k in range(num_clusters):
        graph = Graph(num_nodes)
        for s in range(num_nodes):
            for t in range(s+1, num_nodes):
                graph.addEdge(s, t, I[k, s, t])
        graphs.append(graph)
        # step 4: construct a tree from a Graph
        # tree topology
        edges = graph.maximum_spanning_tree()
        edges_parcouru = []
        tree_topology = num_nodes*[np.nan]
```

```
            parents = [0]
            while(len(parents) != 0):
                parent = parents[0]
                for i, edge in enumerate(edges):
                    if edge not in edges_parcouru:
                        if(parent in edge):
                            if(edge[0] == parent):
                                child = edge[1]
                            else:
                                child = edge[0]
                            edges_parcouru.append(edge)
                            tree_topology[child] = parent
                            parents.append(child)
                parents.pop(0)
            tree = Tree()
            tree.load_tree_from_direct_arrays(np.array(
                tree_topology))
            # step 5:
            root = tree.root
            visit_list = [root]
            # BFS
            while len(visit_list) != 0:
                cur_node = visit_list[0]
                visit_list = visit_list[1:] + cur_node.descendants
                par_node = cur_node.ancestor

                if cur_node == root:
                    cur_node.cat = [Q[k, int(cur_node.name), int(cur_node
                        .name), 0, 0], Q[k, int(cur_node.name), int(cur_node.
                        name), 1, 1]]
                else:
                    cur_node.cat = [np.array([Q[k, int(cur_node.ancestor.
                        name), int(cur_node.name), 0, 0],
                                              Q[k, int(cur_node.ancestor.
                                                  name), int(cur_node.name
                                                  ), 0, 1]]),
                                    np.array([Q[k, int(cur_node.ancestor.
                                        name), int(cur_node.name), 1, 0],
                                              Q[k, int(cur_node.ancestor.
                                                  name), int(cur_node.name
                                                  ), 1, 1]])]



            trees.append(tree)
        tm.clusters = trees

    ###
    topology_list = []
    theta_list = []
    for t in tm.clusters:
```

```
        topology_list.append(t.get_topology_array())
        theta_list.append(t.get_theta_array())
    loglikelihood = np.array(loglikelihood)
    topology_list = np.array(topology_list)
    theta_list = np.array(theta_list)

    return loglikelihood, topology_list, theta_list, tm
```

Question 2.4.13 :

Let's apply EM algorithm implemented in the previous question on the provided data.

We obtain the following results :

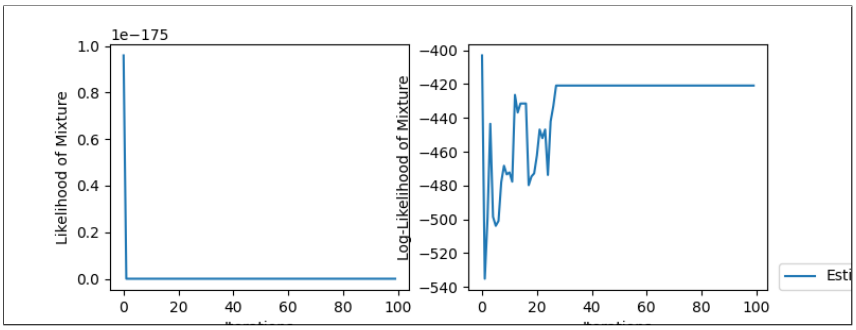

FIGURE 4 –

And the comparison between the different trees provide :

$$\begin{pmatrix} 4 & 5 & 4 \\ 0 & 3 & 4 \\ 4 & 5 & 4 \end{pmatrix}$$

The element (i, j) represents the distance between the ith EM tree and jth real tree.

Real and inferred trees have different structure. The RF distance helps us find the best mapping between the real and EM trees.

The real log likelihood is -311.449480 and the EM log likelihood is -420.839474. The difference is very big, because the EM converge to a local maxima. To solve this problem we should start using different seeds and choose the one that give the best result.

Question 2.4.14 :

Let's simulate new tree mixtures by changing the number of samples, clusters and nodes and compute the difference between the real and the inferred likelihood.

Case where the size of samples = 100

cluster/node	3	10
3	39.11	370
12	1.87	133

The best case is : num_clusters = 12 and num_node = 3

Case where the size of samples = 500

cluster/node	3	10
3	214.5	2404.84
12	4.19	2066.23

The best case is : num_clusters = 12 and num_node = 3

It appears that the difference increases with the number of nodes, and it is normal because the tree becomes more complex.

EM doesn't improve when we increase the number of samples.