Assignment 2

Includes methods of Directed Graphical Models, Dynamic programming,
Variational Inference and Expectation Maximisation

DD2434 Advanced Machine Learning

Filip Bergentoft, bergento@kth.se

2.1 Dependencies in a Directed Graphical Model

Question 2.1.1: In the graphical model of Figure 1, is $\mu_{r,c} \perp \mu_{r,c+1}$?

Answer: No, $\mu_{r,c}$ and $\mu_{r,c+1}$ are dependent.

Question 2.1.2: In the graphical model of Figure 1,
is $X_{r,c} \perp X_{r,c+1} | \{\mu_{r,c}, \mu_{r,c+1}\}$?

Answer: Yes, $X_{r,c}$ and $X_{r,c+1}$ are d-separated and thus independent.

Question 2.1.3: Give a minimal set of variables A such that $X_{r,c} \perp \mu_0 | A$ in Figure 1

Answer: The minimal set of variables are given by$A = \{\mu_{r,c}, \mu_{r-1,c}, \mu_{r+1,c}, \mu_{r,c-1}, \mu_{r,c+1}\}$

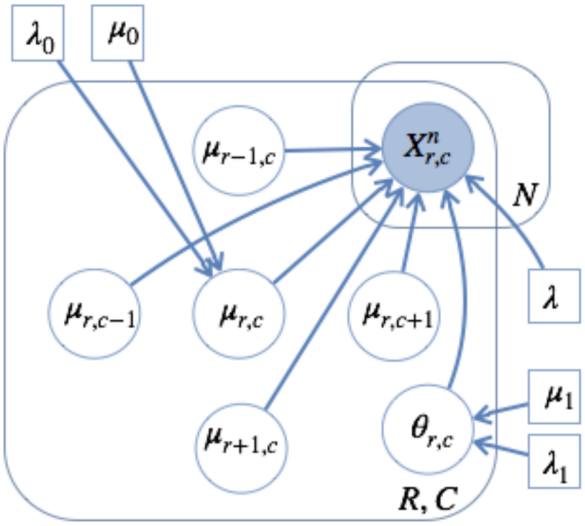

Figure 1

Question 2.1.4: In Figure 2, is $Z \perp X | C$ where $Z = \{Z^n_m : n \in [N], m \in [M]\}$, $X = \{X^n_m : n \in [N], m \in [M]\}$ and $C^n = \{C^n : n \in [N]\}$?

Answer: No, Z and X are not d-separated given C and are thus dependent.

Question 2.1.5: In Figure 2, is $A \perp e | B$ where $A = \{A^k_{i,j} : k \in [K], i, j \in [I]\}$, $e = \{e^k_{i,r} : k \in [K], i \in [I], r \in [R]\}$ and $B = \{Z^n_m : m \in [M], m \text{ odd}\} \cup \{X^n_m : m \in [M], m \text{ even}\}$?

Answer: No, A and e are not d-separated given B and are thus dependent.

Question 2.1.6: In Figure 2, give a minimal set of variables B such that $A \perp X|B$ where $A = \{A^k_{i,j} : k \in [K], i, j \in [I]\}$ and $X = \{X^n_m : n \in [N], m \in [M]\}$

Answer: The minimal set of variables are given by

$$B = \{Z^n_m : n \in [N], m \in [M]\} \cup \{C^n : n \in [N]\}$$

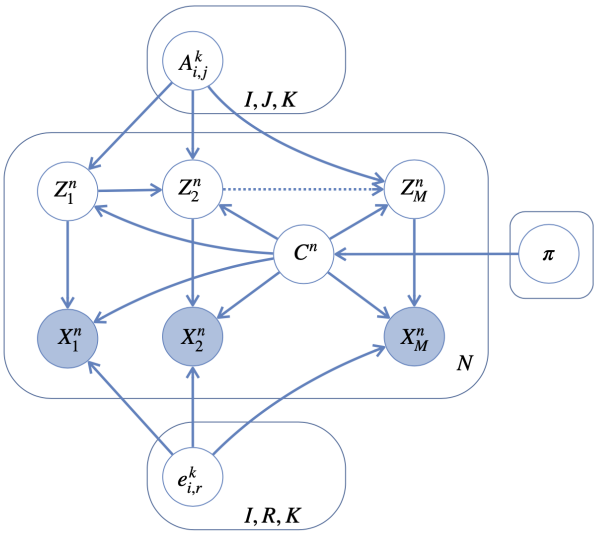

Figure 2

2.2 Likelihood of a Tree Graphical Model

Question 2.2.7: Implement a dynamic programming algorithm that, for a given T, Θ and β computes $p(\beta|T, \Theta)$

T is a binary tree with a vertex set $V(T)$ and a leaf set $L(T)$. For each vertex $v \in V(T)$ there is an associated random variable $X_v \in [K]$ with a corresponding CPD $\theta_v = p(X_v|x_{pa(v)})$ which is a categorical distribution. β is defined as the set of values of all leafs in T such that $\beta = \{x_l : l \in L(T)\}$.

In order to compute $p(\beta|T, \Theta)$ we need to find an expression that can be used for dynamic programming, I.E. splitting up the full problem into smaller subproblems. By looking at the definition of s in equation (1)

$$s(u, i) = p(X_{Observed \cap \downarrow u}|X_u = i) \tag{1}$$

and letting the root node of the tree being denoted by r, one can use that if u is chosen as the root r we get the following expression

$$s(r,i) = p(X_{Observed \cap \downarrow r} | X_r = i) = \left\{ X_{Observed \cap \downarrow r} = \beta \right\} = p(\beta | X_r = i, T, \Theta)$$

We can then marginalise this using Bayes' theorem in the following manner

$$p(\beta|T,\Theta) = \sum_i p(\beta, X_r = i|T,\Theta) = \sum_i p(\beta|X_r = i, T, \Theta) p(X_r = i) \tag{2}$$

$$= \sum_i s(r,i) p(X_r = i) \tag{3}$$

Using that T is a binary tree and thus if v, w are children to a node u then

$$\begin{aligned} s(u,i) &= p(X_{Observed \cap \downarrow u} | X_u = i) \\ &= p(X_{Observed \cap \downarrow v} | X_v = i) p(X_{Observed \cap \downarrow w} | X_w = i) \\ &= \left(\sum_j s(v,j) p(X_v = j | x_u = i) \right) \left(\sum_j s(w,j) p(X_w = j | x_w = i) \right) \end{aligned} \tag{4}$$

A special case is when the node u is a leaf node, then the following holds

$$s(u,i) = \begin{cases} 1, & X_u = i \\ 0, & otherwise \end{cases} \tag{5}$$

Equation (3) can then be computed using dynamic programming by starting at the leaf nodes using equation (5) and then traversing up the nodes in the tree to the root using equation (4) one level at a time and storing the achieved probabilities s along the way.

Question 2.2.8: Apply your algorithm to the graphical model and data provided separately

The following likelihoods were achieved when applying my implementation of the dynamic programming algorithm on the given trees.

Tree sample:	0	1	2	3	4
Small tree	0.016	0.015	0.011	0.007	0.041
Medium tree	$4.336 \cdot 10^{-18}$	$3.094 \cdot 10^{-20}$	$1.050 \cdot 10^{-16}$	$6.585 \cdot 10^{-16}$	$1.488 \cdot 10^{-18}$
Large tree	$3.288 \cdot 10^{-69}$	$1.109 \cdot 10^{-66}$	$2.522 \cdot 10^{-68}$	$1.242 \cdot 10^{-66}$	$3.535 \cdot 10^{-69}$

2.3 Simple Variational Inference

Question 2.3.9: Implement the VI algorithm for the variational distribution in Equation (10.24) in Bishop.

The following problem is stated in Bishop. Given a data set $D = \{x_1, ..., x_N\}$ of observed values x that are drawn from a Gaussian distribution we want to infer a posterior distribution using variational inference. The likelihood function is given by

$$p(D|\mu, \tau) = \left(\frac{\tau}{2\pi}\right)^{N/2} \exp\left\{ -\frac{\tau}{2}\sum_{n=1}^{N}(x_n - \mu)^2 \right\}$$

The conjugate prior distributions for μ and τ are given by

$$\begin{aligned} p(\mu|\tau) &= \mathcal{N}\big(\mu; \mu_0, (\lambda_0\tau)^{-1}\big) \\ p(\tau) &= \Gamma(\tau; a_0, b_0) \end{aligned}$$

where $\mu_0, \lambda_0, a_0, b_0$ are hyperparameters. The factorised variational approximation of the posterior is given by

$$q(\mu, \tau) = q_\mu(\mu) q_\tau(\tau)$$

The factorised distributions are computed in Bishop and become

$$\begin{aligned} q_\mu(\mu) &= \mathcal{N}(\mu; \mu_N, \lambda_N^{-1}) \\ q_\tau(\tau) &= \Gamma(\tau; a_N, b_N) \end{aligned}$$

whom are accompanied by the following updating formulas for the parameters.

$$\mu_N = \frac{\lambda_0\mu_0 + N\overline{x}}{\lambda_0 + N} \tag{6}$$

$$\lambda_N = (\lambda_0 + N)\mathbb{E}_\tau[\tau] = (\lambda_0 + N)\frac{a_N}{b_N} \tag{7}$$

$$a_N = a_0 + \frac{N}{2} \tag{8}$$

$$\begin{aligned} b_N &= b_0 + \frac{1}{2}\mathbb{E}_\mu\left[\sum_{n=1}^{N}(x_n - \mu)^2 + \lambda_0(\mu - \mu_0)^2\right] \\ &= b_0 + \frac{1}{2}\left[\sum_{n=1}^{N} x_n^2 + N\mathbb{E}_\mu[\mu^2] - 2N\overline{x}\mathbb{E}_\mu[\mu] + \lambda_0(\mathbb{E}_\mu[\mu^2] + \mu_0^2 - 2\mu_0\mathbb{E}_\mu[\mu])\right] \\ &= \left\{\mathbb{E}_\mu[\mu] = \mu_N,\, \mathbb{E}_\mu[\mu^2] = V_\mu(\mu) + \mathbb{E}_\mu[\mu]^2 = \lambda^{-1} + \mu_N^2\right\} \\ &= b_0 + \frac{1}{2}\left[\sum_{n=1}^{N} x_n^2 + (N + \lambda_0)(\lambda^{-1} + \mu_N^2) - 2\mu_N(N\overline{x} + \mu_0\lambda_0) + \lambda_0\mu_0^2\right] \end{aligned} \tag{9}$$

The VI algorithm is then implemented by updating the parameters in the order of the equations given above.

Question 2.3.10: What is the exact posterior?

The exact posterior can be computed using the likelihood of the data and the priors. Since the two priors for μ and τ are conjugate priors to the likelihood we know that the posterior will be on the form of a Gaussian-Gamma distribution.

$$
\begin{aligned}
p(\mu,\tau|D) &\propto p(D|\mu,\tau)p(\mu|\tau)p(\tau) \\
&\propto \tau^{\frac{N}{2}}\tau^{\frac{1}{2}}\tau^{a_0-1}e^{-b_0\tau}\exp\Bigg\{-\frac{\tau}{2}\Bigg[\sum_{n=1}^{N}(x_n-\mu)^2+\lambda_0(\mu-\mu_0)^2\Bigg]\Bigg\} \\
&\propto \tau^{\frac{N}{2}}\tau^{\frac{1}{2}}\tau^{a_0-1}e^{-b_0\tau}\exp\Bigg\{-\frac{1}{2}\tau(N+\lambda_0)\Bigg(\mu-\frac{N\overline{x}+\lambda_0\mu_0}{N\lambda_0}\Bigg)^2\Bigg\}\times \\
&\quad \exp\Bigg\{-\frac{\tau}{2}\Bigg[-\frac{(N\overline{x}+\lambda_0\mu_0)^2}{N+\lambda_0}+\sum_{n=1}^{N}x_n^2+\lambda_0\mu_0^2\Bigg]\Bigg\} \\
&\propto \tau^{\frac{N}{2}}\tau^{a_0-1}e^{-b_0\tau}\mathcal{N}\Bigg(\mu;\frac{N\overline{x}+\lambda_0\mu_0}{N\lambda_0},\frac{1}{\tau(N+\lambda_0)}\Bigg)\times \\
&\quad \exp\Bigg\{-\frac{\tau}{2}\Bigg[-\frac{(N\overline{x}+\lambda_0\mu_0)^2}{N+\lambda_0}+\sum_{n=1}^{N}x_n^2+\lambda_0\mu_0^2\Bigg]\Bigg\} \\
&\propto \mathcal{N}\Bigg(\mu;\frac{N\overline{x}+\lambda_0\mu_0}{N\lambda_0},\frac{1}{\tau(N+\lambda_0)}\Bigg)\times \\
&\quad \Gamma\Bigg(\tau;a_0+\frac{N}{2},b_0+\frac{1}{2}\Big(\sum_{n=1}^{N}x_n^2+\lambda_0\mu_0^2-\frac{(N\overline{x}+\lambda_0\mu_0)^2}{N+\lambda_0}\Big)\Bigg) \qquad (10)
\end{aligned}
$$

The exact posterior is thus given by the expression in equation (10) where $\Gamma(\tau;\alpha,\beta)$ denotes the gamma distribution with shape α and rate β as parameters.

Question 2.3.11: Compare the inferred variational distribution with the exact posterior. Run the inference on data points drawn from iid Gaussians. Do this for three interesting cases and visualize the results. Describe the differences.

The following plots were obtained when comparing the inferred posterior to the real posterior.

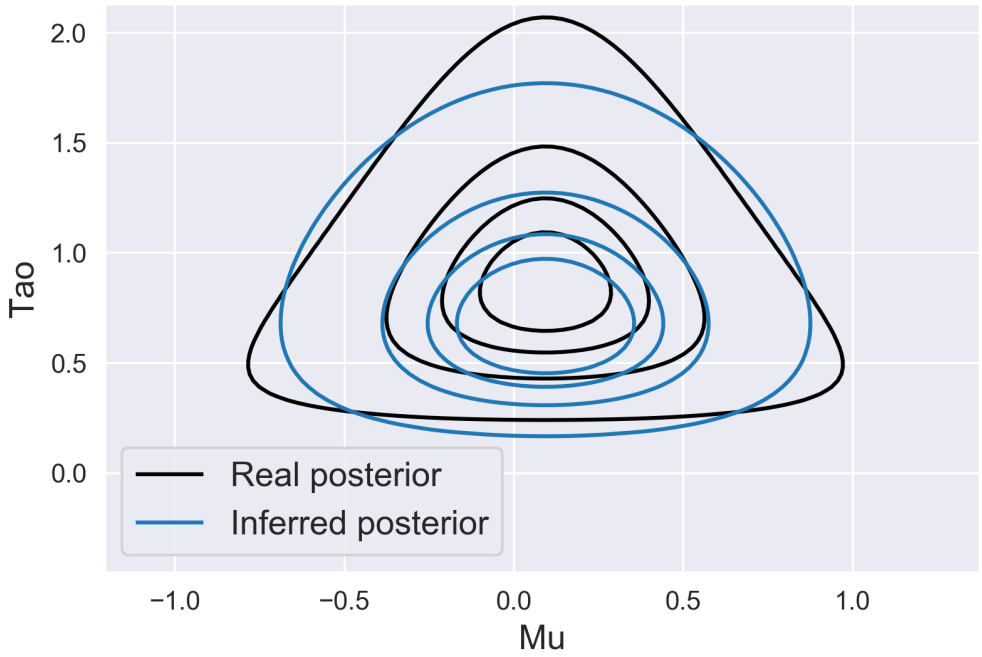

Figure 3: 10 data points drawn from $\mathcal{N}(0, 1)$, all hyperparameters set to 0

In Figure 3 one can see a similar behaviour to the book as the hyperparameters are set the same. The real target for the parameters are $(\mu, \tau) = (0, 1)$ which the real posterior seem to be a bit closer to but considering the low number of data points both distributions perform quite well. When comparing the shape of the the two posteriors one can see that the real posterior is closer to a triangle than the inferred posterior but in general the inferred posterior is close to the real posterior. Note that the uncertainty is about the same for both μ and τ.

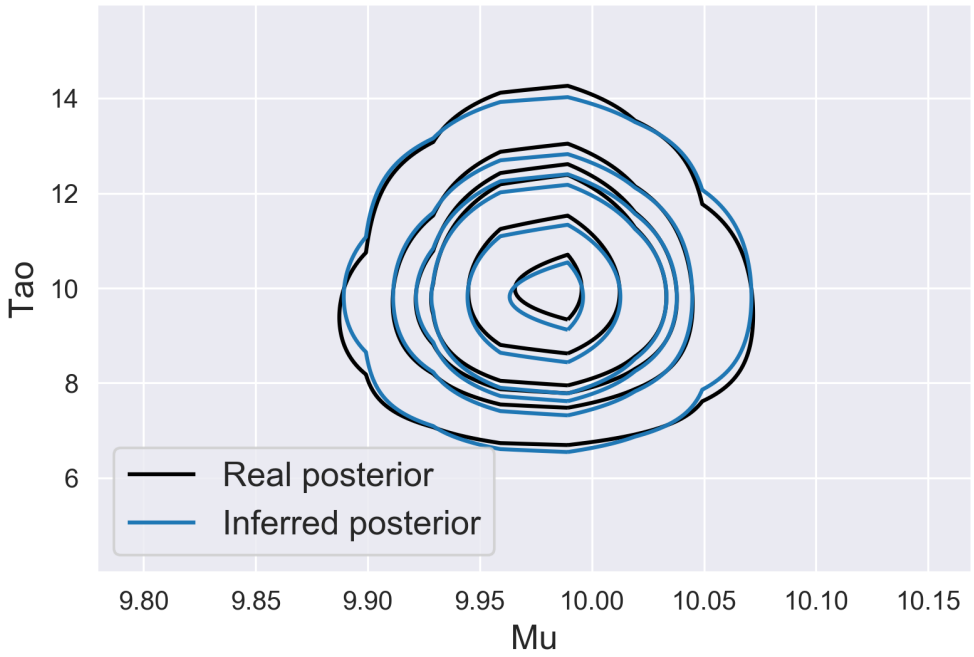

Figure 4: 100 data points drawn from $\mathcal{N}(10, \frac{1}{10})$, all hyperparameters set to 0

In Figure 4 the number of data points has increased to 100 and the distribution they are generated from has changed to $\mathcal{N}(10, \frac{1}{10})$. Noticing that the axis scales are different here the conclusions from Figure 3 still holds except that the uncertainty in τ is almost two orders of magnitude larger than the uncertainty for μ.

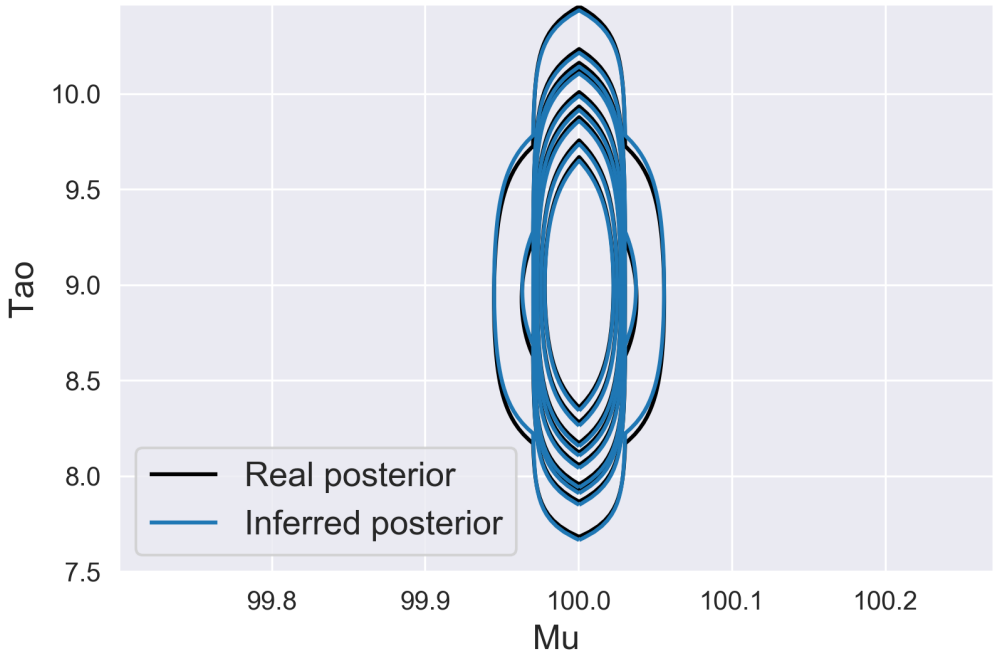

Figure 5: 1000 data points drawn from $\mathcal{N}(100, \frac{1}{100})$, all hyperparameters set to 0

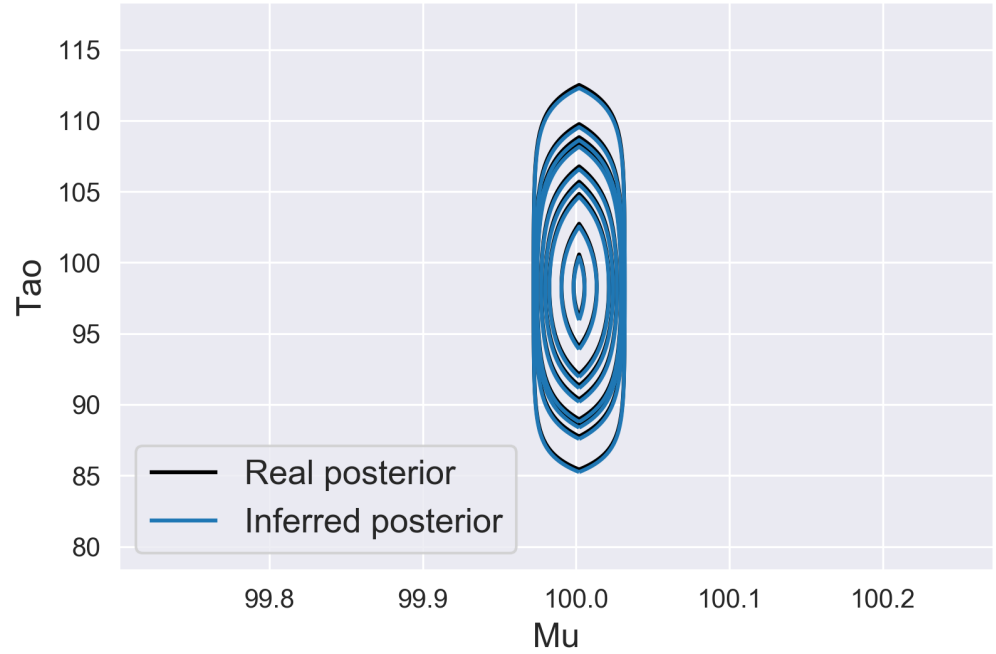

Figure 6: 1000 data points drawn from $\mathcal{N}(100, \frac{1}{100})$, all hyperparameters set to 0 except $\mu_0 = 100$

In Figures 5 and 6 the effect of the hyperparameter μ_0 is enlightened. In both examples 1000 data points are drawn from $\mathcal{N}(100, \frac{1}{100})$. In Figure 5 when $\mu_0 = 0$ both posteriors are centred around $\tau = 9$, very far from the real value of $\tau = 100$ whilst in Figure 6 both posteriors are centred around $\tau = 98$, very close to the real value. This shows how important the hyperparameters can be if one does not have a sufficient amount of data. In addition to this conclusion one can also conclude that the inferred posterior is very similar to the real posterior in this example.

2.4 Mixture of trees with observable variables

Question 2.4.12: Implement this EM algorithm.

Note: Due to the unclear instructions of point 4 in the EM-algorithm I discussed with Ludvig Doeser if the problem formulation meant that the node names should carry over to the new tree T'_k or not.

The EM algorithm was implemented as instructed using the provided Tree package and sieving was implemented in the following manner.

Algorithm 1: EM algorithm with sieving

Input: Data samples
Output: Tree mixture
1 Create a set of 100 Tree Mixtures $\{TM_i\}_{i=1}^{1}00$
2 For each Tree Mixture TM_i do Expectation Maximisation for 10 iterations
3 Continue Expectation Maximisation for 90 iterations on the 10 Tree Mixtures with the highest likelihood
4 Return the Tree Mixture with the highest likelihood as the inferred mixture

Question 2.4.13: Apply your algorithm to the provided data and show how well you reconstruct the mixtures. First, compare the real and inferred trees with the unweighted Robinson-Foulds (aka symmetric difference) metric. Do the trees have similar structure? Then, compare the likelihoods of real and inferred mixtures.

The following result was achieved when applying the EM-algorithm to the provided data.

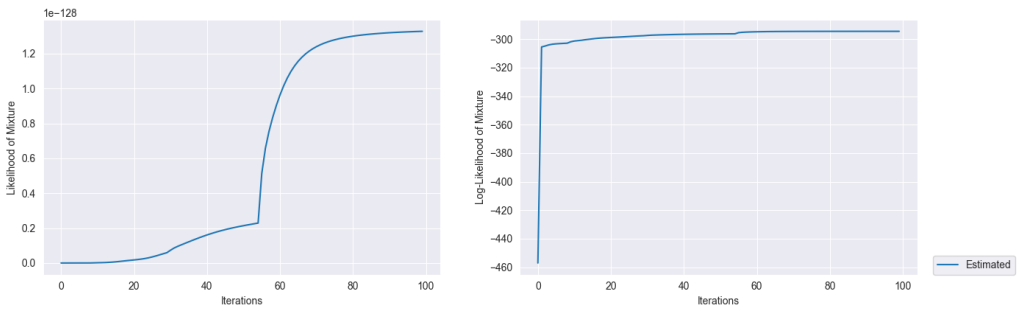

Figure 7: Convergence of EM

	Inferred tree 0	Inferred tree 1	Inferred tree 2
Real tree 0	4	6	4
Real tree 1	3	5	5
Real tree 2	4	4	4

Table 1: RF-metric between real and inferred mixture

Judging by Figure 7 the likelihood is monotonically increasing which is a good indication that the EM-algorithm is working. The Robinson-Foulds metric was computed between the two mixtures which gave the following result.

the trees had the following topology arrays

Real tree 0	*nan*	0	0	1	3
Real tree 1	*nan*	0	0	0	3
Real tree 2	*nan*	0	0	0	0
Inferred tree 0	*nan*	4	0	2	3
Inferred tree 1	*nan*	2	0	4	2
Inferred tree 2	*nan*	0	0	1	2

Table 2: Topology arrays of the different trees

Judging by Tables 1 and 2 the real trees and the inferred trees are not very similar in structure. The following likelihoods were obtained on the provided data.

Mixture	LogLikelihood
Real Mixture	-311.4
Inferred Mixture	-294.2

Table 3: LogLikelihoods of the mixtures

I believe that the inferred mixture achieves a higher loglikelihood due to being more specified on the 100 provided samples than the real mixture and is thus overfitted to the data.

Question 2.4.14: Simulate new tree mixtures with different number of nodes, samples and clusters. Try to find some interesting cases. Analyse your results as in the previous question.

Simulated tree mixture 1: The first tree mixture simulated from was made up of 2 clusters having 10 nodes each. 100 samples was generated from the tree mixture and used to train the inferred mixture, this yielded the following results.

	Inferred tree 0	Inferred tree 1
Simulated tree 0	6	10
Simulated tree 1	8	10

Table 4: RF-metric between simulated and inferred mixture

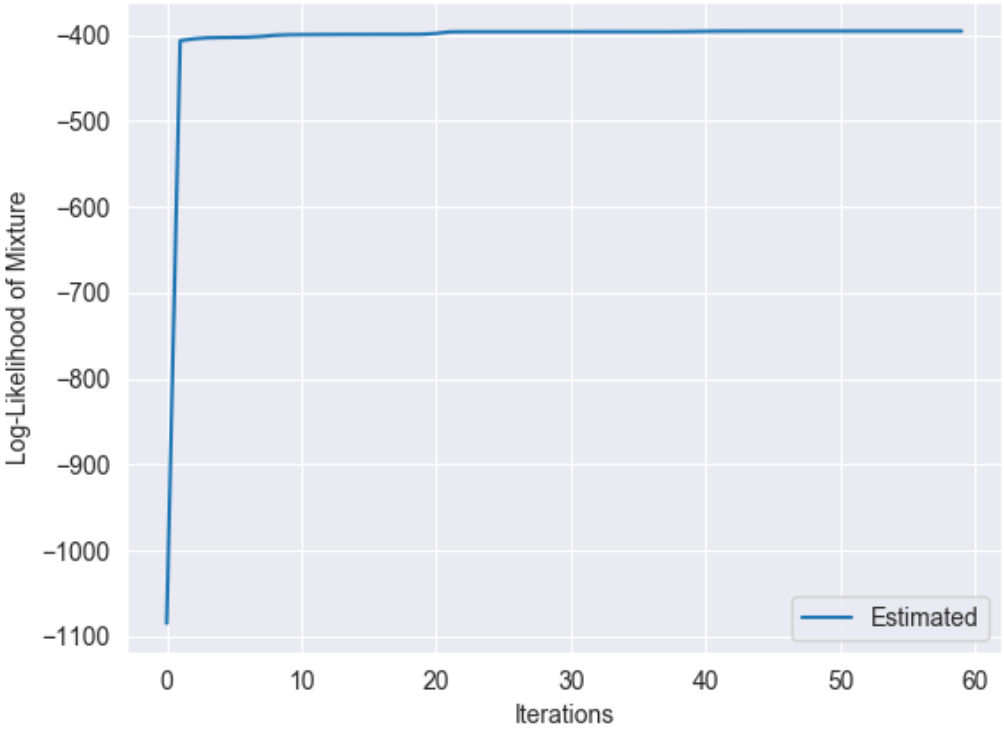

Figure 8: Convergence of EM simulated tree mixture 1

The likelihood was omitted from Figure 8 due to the low values.

Mixture	LogLikelihood
Simulated Mixture 1	−426.3
Inferred Mixture	−395.6

Table 5: LogLikelihoods of the mixtures

The results are similar to the previous results both when it comes to the RF-metric and the loglikelihoods. The RF-metrics are about half of the maximum value of 20 and the loglikelihood for the inferred mixture is greater than that of the simulated mixture.

Simulated tree mixture 2: The second tree mixture simulated from was made up of 2 clusters having 3 nodes each. 10000 samples was generated from the tree mixture and used to train the inferred mixture in order to see if the samples and inferred mixture will be more similar.

	Inferred tree 0	Inferred tree 1
Simulated tree 0	0	2
Simulated tree 1	2	2

Table 6: RF-metric between simulated and inferred mixture

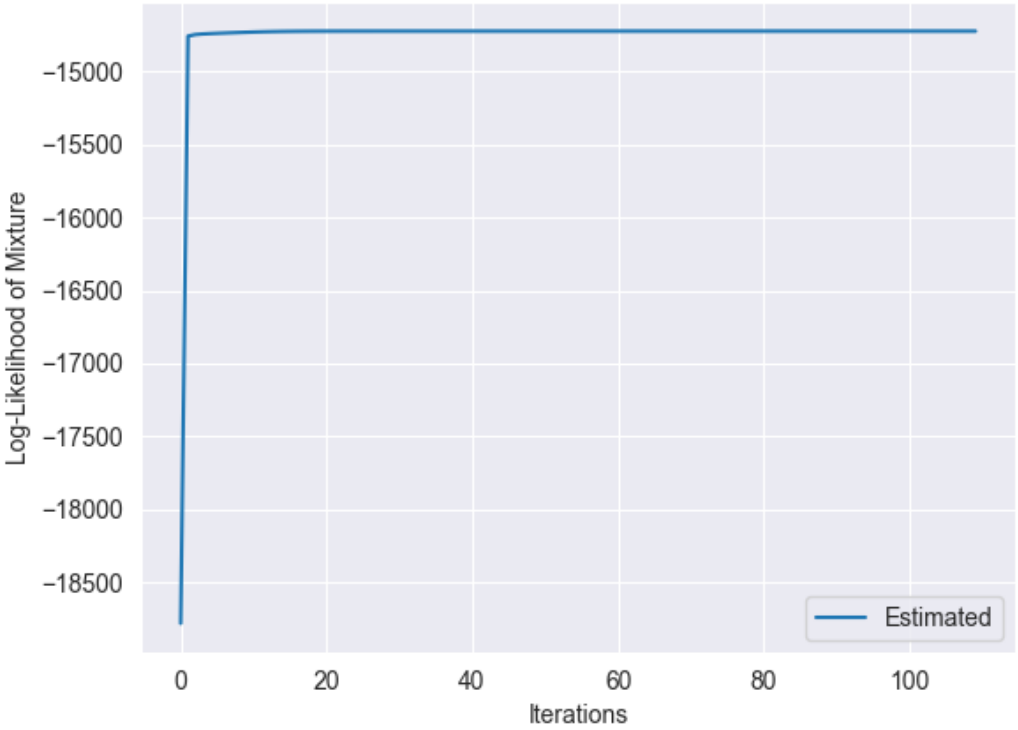

Figure 9: Convergence of EM simulated tree mixture 2

Mixture	LogLikelihood
Simulated Mixture 2	−14731.6
Inferred Mixture	−14726.1

Table 7: LogLikelihoods of the mixtures

Again the results are similar as the loglikelihood for the inferred mixture is greater than the loglikelihood for the mixture sampled from. However in this example one of the inferred trees has the same structure as one of the trees that are simulated from. However it is unclear if this is just a coincidence due to the low number of nodes or if it is due to the increased number of training samples. If more time was given this could be further investigated.

Appdendix Code

Problem 2.2

```
from os import DirEntry
import numpy as np
from Tree import Tree
from Tree import Node
import os


def calculate_likelihood(tree_topology, theta, beta):


    # tree_topology: index corresponds to node,
    # element corresponds to parent of that node, nan := root node
    # K defines the number of nodes in the tree
    K = np.asarray(theta).shape[1]
    number_of_nodes = np.asarray(theta).shape[0]
    node_ids = range(number_of_nodes-1,-1,-1)
    print(tree_topology)
    # Matrix of probabilities, each row coresponds to a node
    S = np.empty((number_of_nodes,K))

    # Walking backwards in the nodes
    for node_id in node_ids:

        # If leaf, do this:
        if node_id not in tree_topology:

            leaf_value = np.int(beta[node_id])
            s = np.zeros((K))
            # Setting the probability of the observed variable to 1
            s[leaf_value] = 1
            S[node_id,:] = s

        # If not leaf, do this:
        else:
            # Using the structure of tree_topology to find the node's children
            children_node_id = np.where(tree_topology == node_id)[0]
            s = np.ones(K)

            for child_node_id in children_node_id:
                theta_child = np.stack(theta[child_node_id],axis=0)
                s_child = S[child_node_id,:]
                prob_child = np.matmul(theta_child, s_child)
                s = np.multiply(s, prob_child)
            S[node_id,:] = s

    s_root = S[0,:]
    theta_root = np.stack(theta[0],axis=0)
    likelihood = np.dot(s_root, theta_root)
    # End: Example Code Segment

    return likelihood


def main():
    print("\n1. Load tree data from file and print it\n")
    directory = '/Users/filipbergentoft/Desktop/Github/DD2434/Assignment 2/2_2/'

    filename = directory + "data/q2_2/q2_2_small_tree.pkl"
    t = Tree()
    t.load_tree(filename)
    #t.print()
    #t.print_topology()
    print("K of the tree: ", t.k, "\talphabet: ", np.arange(t.k))

    print("\n2. Calculate likelihood of each FILTERED sample\n")

    for sample_idx in range(5):
        beta = t.filtered_samples[sample_idx]
        print("\n\tSample: ", sample_idx, "\tBeta: ", beta)
        sample_likelihood = calculate_likelihood(t.get_topology_array(),
         t.get_theta_array(), beta)
        print("\tLikelihood: ", sample_likelihood)


if __name__ == "__main__":
    main()
```

Problem 2.3

```
import numpy as np
import seaborn as sbs
import matplotlib.pyplot as plt
import seaborn as sns
import scipy.stats as stats
```

```
# Setting hyperparameters

def inference_model(data, tol, a_0, b_0, lambda_0, mu_0):

    data_mean = np.mean(data)
    data_sum_squared = np.dot(data,data)

    # Initializing parameters
    old_parameters = 0
    new_parameters = 1
    N = np.size(data)
    E_tao = 1
    mu_N = 0

    while np.linalg.norm(new_parameters - old_parameters) > tol:


        mu_N, lambda_N = VI_mu_update(data,
                                      data_mean, N, lambda_0, mu_0, E_tao)
        a_N, b_N = VI_tao_update(data, data_mean,
                                      N, lambda_0, mu_0, a_0, b_0, mu_N, lambda_N)
        E_tao = a_N/b_N
        old_parameters = new_parameters
        new_parameters = np.array([mu_N, E_tao])

    return mu_N, lambda_N, a_N, b_N


def VI_mu_update(data, data_mean, N, lambda_0, mu_0, E_tao):
    """
    Parameter: data: Nx1 vector of observed variables
    Parameter: data_mean: mean of data vector
    Parameter: lamda_0: hyperparameter; Precision for prior on mu
    Parameter: mu_0: hyperparameter; Mean for prior on mu
    Parameter: E_tao: Expected value of tao
    """

    # Variational distriution for mu - Gaussian
    mu_N = (lambda_0*mu_0 + N*data_mean) / (lambda_0 + N)
    lambda_N = (lambda_0 + N)*E_tao

    return mu_N, lambda_N

def VI_tao_update(data, data_mean, N,
                  lambda_0, mu_0, a_0, b_0, mu_N, lambda_N):

    a_N = a_0 + N/2

    first_term = np.dot(data,data)\
         - 2*N*mu_N*data_mean + N*(1/lambda_N + mu_N**2)
    second_term = lambda_0*((1/lambda_N + mu_N**2) - 2*mu_0*mu_N + mu_0**2)
    b_N = b_0 + 0.5*(first_term + second_term)

    return a_N, b_N

def true_posterior():
    pass


def real_posterior(data, mu, tao, a_0, b_0, lambda_0, mu_0):
    N = np.size(data)
    data_mean = np.mean(data)

    a_post = a_0 + N/2
    b_post = b_0 + 0.5 * (np.dot(data,data) \
        + lambda_0*mu_0**2 - (N*data_mean + lambda_0*mu_0)**2/(N+lambda_0))

    mu_post = ( N * data_mean + lambda_0*mu_0 )/(N + lambda_0)
    tao_post = tao * (N + lambda_0)
    std_post = np.sqrt(1/tao_post)

    gamma_likalihood = stats.gamma.pdf(tao, a_post, loc=0, scale=1/b_post)
    gauss_likelihood = stats.norm.pdf(mu,mu_post,std_post)

    return gamma_likalihood * gauss_likelihood

def inferred_posterior(mu, tao, mu_N, lambda_N, a_N, b_N):
    std_N = np.sqrt(1/lambda_N)

    gamma_likelihood = stats.gamma.pdf(tao, a_N, loc=0, scale=1/b_N)
    gauss_likelihood = stats.norm.pdf(mu,mu_N,std_N)

    return gamma_likelihood * gauss_likelihood


# Constructing data
real_mu = 100
real_tao = 100
N = 1000
sigma = np.sqrt(1/real_tao)
data = np.random.normal(real_mu, sigma, N)

# Hyperparameters
a_0 = 0
b_0 = 0
```

```
lambda_0 = 0
mu_0 = 100

mu_N, lambda_N, a_N, b_N = inference_model(data,
     tol = 1e-30, a_0=a_0,b_0=b_0, lambda_0=lambda_0, mu_0=mu_0)

E_tao = a_N/b_N


plot_range_x = 0.3
plot_range_y = 20
mu_axis = np.arange(mu_N-plot_range_x, mu_N + plot_range_x, 0.03)
tao_axis = np.arange(E_tao-plot_range_y, E_tao + plot_range_y, 0.03)

nx = np.size(mu_axis)
ny = np.size(tao_axis)

[X,Y] = np.meshgrid(mu_axis, tao_axis)
Z_real = np.asarray([[real_posterior(data,
 mu, tao, a_0, b_0, lambda_0, mu_0) for tao in tao_axis] for mu in mu_axis])
Z_inferred = np.asarray([[inferred_posterior(mu,
 tao, mu_N, lambda_N, a_N, b_N) for tao in tao_axis] for mu in mu_axis])

sns.set_style('darkgrid')
fig, ax = plt.subplots(1, 1)
# plots contour lines

cntr1 = ax.contour(X, Y, Z_real.T, colors='k',
     levels=[0.1,0.5,0.8,1,2,3,4,7,10])
cntr2 = ax.contour(X, Y, Z_inferred.T, colors='C0',
     levels=[0.1,0.5,0.8,1,2,3,4,7,10])
h1,_ = cntr1.legend_elements()
h2,_ = cntr2.legend_elements()

ax.legend([h1[0], h2[0]],
     ['Real posterior', 'Inferred posterior'], fontsize = 13, loc=3)

plt.xlabel('Mu', fontsize=13)
plt.ylabel('Tao', fontsize=13)
plt.savefig('VI_plot_N_1000-mu_100-tau_100-mu0_100', dpi=300)
plt.show()
```

Problem 2.4

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import dendropy
from Tree import TreeMixture
from Tree import Tree
from Kruskal_v2 import maximum_spanning_tree
from help_functions import q_joint, q_marginal, I_element, mutual_information, \
    create_ordered_pairs, create_ordered_nodes, create_tree_attributes,\
         create_graph, sample_likelihood, create_tree_attributes1


def save_results(loglikelihood, topology_array, theta_array, filename):
    """ This function saves the log-likelihood vs iteration values,
        the final tree structure and theta array to corresponding numpy arrays. """

    likelihood_filename = filename + "_em_loglikelihood.npy"
    topology_array_filename = filename + "_em_topology.npy"
    theta_array_filename = filename + "_em_theta.npy"
    np.save(likelihood_filename, loglikelihood)
    np.save(topology_array_filename, topology_array)
    np.save(theta_array_filename, theta_array)

def em_algorithm(seed_val, samples, num_clusters, max_num_iter, tm=None):

    num_samples = samples.shape[0]
    num_nodes = samples.shape[1]

    loglikelihood = []
    if tm is None:
        tm = TreeMixture(num_clusters=num_clusters, num_nodes=num_nodes)
        tm.simulate_pi(None)
        tm.simulate_trees(None)
        #samples = tm.samples

    for iter_ in range(max_num_iter):
        # 1. Compute responsibilities for all trees
        sample_likelihoods = np.array([[sample_likelihood(tm.clusters[ii], samples[jj,:]\
             , tm.pi[ii]) for ii in range(num_clusters)] for jj in range(num_samples)])
        sum_over_trees_likelihoods = np.reshape(np.sum(sample_likelihoods, axis = 1),(num_samples,1))
        Responsibilities = np.divide(sample_likelihoods,sum_over_trees_likelihoods)
        # Computing loglikelihood
        ll = np.sum(np.log(np.sum(sample_likelihoods,axis=1)),axis=None)
        loglikelihood.append(ll)

        tm.loglikelihood.append(ll)
        # 2. Updating pi for all trees
        tm.pi = np.sum(Responsibilities,axis=0)/num_samples
        vertices = list(range(num_nodes))
```

```
            # 3. Updating each tree
            for i in range(num_clusters):
                tree = tm.clusters[i]
                responsibilities = Responsibilities[:,i]
                # Creating the symmetric mutual information matrix
                mutual_information_matrix = np.asarray([[mutual_information(responsibilities, samples, s_idx, t_idx) \
                    for s_idx in vertices] for t_idx in vertices])
                # Computing the graph
                graph = create_graph(num_nodes, responsibilities, samples, mutual_information_matrix, vertices)
                # Finding the maximum spanning tree
                MST = maximum_spanning_tree(graph)
                # Choosing the root as 0
                root_name = 0
                # Finding the order of nodes in the tree
                ordered_nodes, I_sum_tree = create_ordered_nodes(MST, root_name)
                # Getting attributes for tree to enable update
                topology_array, theta_array = create_tree_attributes1(ordered_nodes,
                 root_name, samples, responsibilities, num_nodes)
                # Updating the tree
                tree.load_tree_from_direct_arrays(topology_array, theta_array)

        # ------------------------------------------
        topology_list = []
        theta_list = []
        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 = np.array(loglikelihood)
        topology_list = np.array(topology_list)
        theta_list = np.array(theta_list)

        return loglikelihood, topology_list, theta_list, tm

    def sieving(n_first_mixtures, n_second_mixtures, n_first_iterations,n_second_iterations, samples, num_clusters):

        mixtures = []
        mixtures_loglikelihoods_verification = []

        for i in range(n_first_mixtures):

            # Training the model for 10 iterations
            tm = em_algorithm(seed_val=None, samples=samples,
             num_clusters=num_clusters, max_num_iter=n_first_iterations, tm=None)[-1]
            mixtures.append(tm)
            # Computing the loglikelihood on the verification samples
            loglikelihood_verification = mixture_likelihood(tm, samples)
            mixtures_loglikelihoods_verification.append(loglikelihood_verification)

        mixtures_loglikelihoods_verification = np.asarray(mixtures_loglikelihoods_verification)
        idx_best_mixtures = np.argsort(mixtures_loglikelihoods_verification)[0:10]

        second_mixtures = []
        second_mixtures_loglikelihoods = []

        for i in range(n_second_mixtures):
            print('Optimizing second mixture number: ', i)
            tm_idx = idx_best_mixtures[i]
            tm = mixtures[tm_idx]
            tm = em_algorithm(seed_val=None, samples=samples,
             num_clusters=num_clusters, max_num_iter=n_second_iterations, tm=tm)[-1]
            second_mixtures.append(tm)
            loglikelihood_verification = mixture_likelihood(tm, samples)
            second_mixtures_loglikelihoods.append(loglikelihood_verification)

        best_idx = np.argmax(second_mixtures_loglikelihoods)
        best_tm = second_mixtures[best_idx]
        print(second_mixtures_loglikelihoods)

        topology_list = []
        theta_list = []
        for i in range(num_clusters):
            topology_list.append(best_tm.clusters[i].get_topology_array())
            theta_list.append(best_tm.clusters[i].get_theta_array())

        #loglikelihood = np.array(loglikelihood)
        topology_list = np.array(topology_list)
        theta_list = np.array(theta_list)

        return best_tm



    def mixture_likelihood(mixture, samples):
        num_clusters = mixture.pi.shape[0]
        num_samples = samples.shape[0]
        sample_likelihoods = np.array([[sample_likelihood(mixture.clusters[ii], samples[jj,:]\
            , mixture.pi[ii]) for ii in range(num_clusters)] for jj in range(num_samples)])

        # Computing loglikelihood
        ll = np.sum(np.log(np.sum(sample_likelihoods,axis=1)),axis=None)

        return ll

    def RF_comparison(inferred_tm, real_tm):
```

```
    tns = dendropy.TaxonNamespace()

    dendr_inferred_trees = []
    dendr_real_trees = []
    for inferred_tree in inferred_tm.clusters:
        inferred_tree = dendropy.Tree.get(data=inferred_tree.newick, schema="newick", taxon_namespace=tns)
        dendr_inferred_trees.append(inferred_tree)

    for real_tree in real_tm.clusters:
        real_tree = dendropy.Tree.get(data=real_tree.newick, schema="newick", taxon_namespace=tns)
        dendr_real_trees.append(real_tree)

    #x-dir: inferred trees, y-dir: real trees
    RF_matrix = [[dendropy.calculate.treecompare.symmetric_difference(inferred_tree, real_tree)\
         for inferred_tree in dendr_inferred_trees] for real_tree in dendr_real_trees]

    return np.asarray(RF_matrix)

def main():

    num_clusters = 3
    new_tm = TreeMixture(num_clusters=num_clusters, num_nodes=3)
    new_tm.simulate_pi(None)
    new_tm.simulate_trees(None)
    new_tm.sample_mixtures(100)
    new_samples = new_tm.samples
    #samples = tm.samples
    seed_val = None
    directory = '/Users/filipbergentoft/Desktop/Github/DD2434/Assignment 2/2_4/'
    sample_filename = directory + "data/q2_4/q2_4_tree_mixture.pkl_samples.txt"
    output_filename = directory + "data/q2_4/q2_4_own_results"
    real_values_filename = directory + "data/q2_4/q2_4_tree_mixture.pkl"


    samples = np.loadtxt(sample_filename, delimiter="\t", dtype=np.int32)
    np.random.shuffle(samples)

    best_tm = sieving(n_first_mixtures=50, n_second_mixtures=10,
     n_first_iterations=10, n_second_iterations=100, samples=samples, num_clusters=num_clusters)


    real_tm = TreeMixture(num_clusters = 3, num_nodes = 5)
    real_tm.load_mixture(real_values_filename)

    print('best tree',mixture_likelihood(best_tm, samples))
    print('best tree',best_tm.pi)
    print('real tree',mixture_likelihood(real_tm, samples))
    print('real tree',real_tm.pi)


    print(RF_comparison(best_tm, real_tm))
    for tree in new_tm.clusters:
        print('Real tree topology')
        print(tree.get_topology_array())

    for tree in best_tm.clusters:
        print('Inferred tree topology')
        print(tree.get_topology_array())


    sns.set_style('darkgrid')
    """
    plt.subplot(121)
    plt.plot(np.exp(best_tm.loglikelihood), label='Estimated')
    plt.ylabel("Likelihood of Mixture")
    plt.xlabel("Iterations")
    plt.subplot(122)
    """
    plt.plot(best_tm.loglikelihood, label='Estimated')
    plt.ylabel("Log-Likelihood of Mixture")
    plt.xlabel("Iterations")
    plt.legend()
    plt.show()


if __name__ == "__main__":
    main()

import numpy as np

def q_joint(responsibilities, s_vec, t_vec, a, b):
    numerator = sum([r for (r,s,t) in zip(responsibilities, s_vec, t_vec) if s == a and t == b])
    denominator = np.sum(responsibilities) + 2e-16

    return numerator/denominator

def q_marginal(responsibilities, s_vec, a):
    numerator = np.sum(responsibilities[s_vec == a])
    denominator = np.sum(responsibilities)

    return numerator/denominator

def I_element(responsibilities, s_vec, t_vec, a, b):
    factor1 = q_joint(responsibilities, s_vec, t_vec, a, b)
    if factor1 <= 0:
```

```
        return 0
    q_s = q_marginal(responsibilities, s_vec, a)
    q_t = q_marginal(responsibilities, t_vec, b)

    factor2 = np.log(factor1/(q_s*q_t))
    if factor1 <= 0:
        return 0
    else:
        return factor1*factor2

def mutual_information(responsibilities, samples, s_idx, t_idx):
    # Going from node s to node t
    s_vec = samples[:,s_idx]
    t_vec = samples[:,t_idx]
    I_matrix = [[I_element(responsibilities, s_vec, t_vec, a, b) for a in [0,1]] for b in [0,1]]

    return np.sum(I_matrix, axis=None)

def create_graph(num_nodes, responsibilities, samples, mutual_information_matrix, vertices):
    #vertices = [x for x in range(num_nodes)]
    """Maybe dont add edges from same node to same node"""
    edges = set()
    [[edges.add((s_idx,t_idx,mutual_information_matrix[s_idx, t_idx])) \
        for s_idx in vertices] for t_idx in vertices]
    graph = {
    'vertices': vertices,
    'edges': edges
    }
    return graph


def create_ordered_pairs(ordered_nodes,parent_node_name, pairs):

    mask = np.where(np.sum(pairs == parent_node_name,axis=1) == 1)
    parent_children_pairs = pairs[mask,:]
    # Removing the connections from contention
    pairs = np.delete(pairs, mask,axis=0)
    children_names = parent_children_pairs[parent_children_pairs != parent_node_name]


    # Add children to ordered pairs
    if len(children_names) is 0:
        pass
    else:
        for child_name in children_names:
            ordered_nodes.append([parent_node_name,child_name])

    # Go recursive
    if len(children_names) is 0:
        pass
    else:
        for child_name in children_names:
            create_ordered_pairs(ordered_nodes, child_name, pairs)

def create_ordered_nodes(MST, root_name):
    # Finding the node pairs
    pairs = []
    I_sum = 0
    for pair in MST:
        pairs.append(pair[0:2])
        I_sum += pair[2]
    pairs = np.asarray(pairs)
    #Adding root to ordered list of nodes
    ordered_nodes = []
    ordered_nodes.append([np.nan,root_name])

    # creating ordered pairs of nodes
    create_ordered_pairs(ordered_nodes,root_name, pairs)
    return np.asarray(ordered_nodes), I_sum

def create_tree_attributes(ordered_nodes, root_name, samples, responsibilities, num_nodes):
        # Create theta matrices
    all_theta = []
    topology_array = []

    # Handling the root as a special case
    s_vec = samples[:,root_name]
    theta_root = np.asarray([q_marginal(responsibilities, s_vec, 0), q_marginal(responsibilities, s_vec, 1)])
    all_theta.append(theta_root)
    topology_array.append(np.nan)
    for j in range(1,num_nodes):
        # s is the parent
        #
        s = ordered_nodes[j,0].astype(int)
        s_vec = samples[:,s]
        t = ordered_nodes[j,1].astype(int)
        t_vec = samples[:,t]

        theta = np.asarray([[q_joint(responsibilities, s_vec, t_vec, a,
         b)/q_marginal(responsibilities, s_vec, a) for b in [0,1]] for a in [0,1]])
        all_theta.append(theta)

        parent_idx = np.where(ordered_nodes[:,1]==s)[0][0]
        topology_array.append(parent_idx)
    topology_array = np.asarray(topology_array)#.astype(int)
```

```
    topology_array[0] = float('nan')

    return topology_array, all_theta


def sample_likelihood(tree, sample, pi_tree):

    likelihood = np.nan
    tree_topology = tree.get_topology_array().astype(int)
    all_theta = np.array(tree.get_theta_array())
    for node_idx in range(tree.num_nodes):
        # If root do:
        if node_idx == 0:
            theta = all_theta[node_idx]
            node_value = sample[node_idx]
            likelihood = pi_tree*theta[node_value]
        # If not root do:
        else:
            theta = np.stack(all_theta[node_idx],axis=1)
            parent_idx = tree_topology[node_idx]
            parent_value = sample[parent_idx]
            node_value = sample[node_idx]
            likelihood *= theta[node_value,parent_value]

    return likelihood

def create_tree_attributes1(ordered_nodes, root_name, samples, responsibilities, num_nodes):

    # Initializing the new topology array and setting root parent to nan
    topology_array = np.zeros(num_nodes)
    topology_array[0] = float('nan')


    # Initializing the new theta array and setting root theta
    theta_array = list(range(num_nodes))
    root_samples = samples[:,root_name]
    theta_root = np.asarray([q_marginal(responsibilities, root_samples, 0),
     q_marginal(responsibilities, root_samples, 1)])
    theta_array[0]= theta_root

    # Iterating to set the remaining topology_array and theta_array
    for j in range(1, num_nodes):
        child_idx = ordered_nodes[j,1].astype(int)
        parent_idx = ordered_nodes[j,0].astype(int)
        topology_array[child_idx] = parent_idx

        child_samples = samples[:,child_idx]
        parent_samples = samples[:,parent_idx]
        theta = np.asarray([[q_joint(responsibilities, parent_samples, child_samples, a, b)\
            /q_marginal(responsibilities, parent_samples, a) for b in [0,1]] for a in [0,1]])
        theta_array[child_idx] = theta


    return topology_array, theta_array
```