Assignment 2 - DD2434 Machine Learning, Advanced Course

August Regnell 970712-9491

19 December, 2020

Code found in Appendix and in the following **Git repo**

Contents

1 Dependencies in a Directed Graphical Model

1.1 Question 2.1.1

No.

1.2 Question 2.1.2

Yes.

1.3 Question 2.1.3

$A = \{\mu_{r,c} \mid r \in [R], c \in [C]\}$

1.4 Question 2.1.4

No.

1.5 Question 2.1.5

No.

1.6 Question 2.1.6

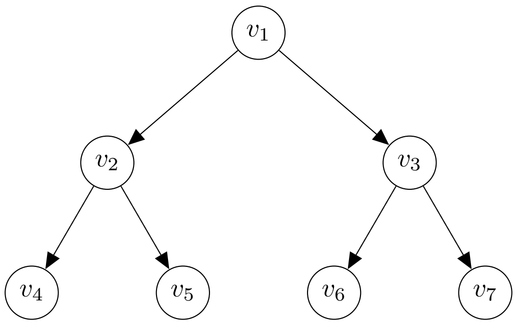
$B = \{Z_m^n \mid n \in [N], m \in [M]\} \cup \{C^n \mid n \in [N]\}$

2 Likelihood of a Tree Graphical Model

2.1 Question 2.2.7

Let T be a binary tree, with vertex set $V(T)$ and leaf set $L(T)$, and consider the graphical model T, Θ described as follows. For each vertex $v \in V(T)$ there is an associated random variable X_v that assumes values in $[K]$. Moreover, for each $v \in V(T)$, the CPD $\theta_v = p(X_v \mid x_{pa(v)})$ is a categorical distribution. Let $\beta = \{x_l : l \in L(T)\}$ be an assignment of values to all the leaves of T.

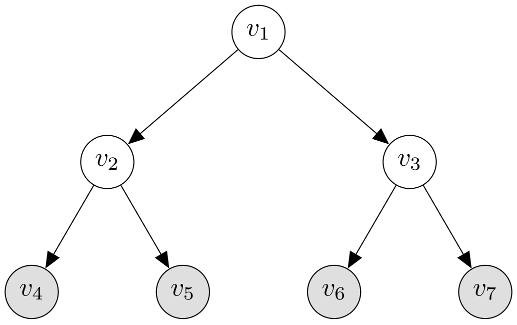
Consider the likelihood of the following small tree consisting of seven vertices $\{v_1, \ldots, v_7\}$ numbered from top to bottom and left to right. (**Note:** the trees provided in the data are not full trees like this example)



Its likelihood is given by:

$$p(T \mid \Theta) = p(v_1)p(v_2 \mid v_1)p(v_3 \mid v_1)p(v_4 \mid v_2)p(v_5 \mid v_2)p(v_6 \mid v_3)p(v_6 \mid v_3)$$

For this tree we have that $\beta = \{x_l : l \in L(T)\} = \{v_4, v_5, v_6, v_7\}$, that is:



Calculating $p(\beta \mid T, \Theta)$ can be done by marginalizing on $V(T) \setminus L(T)$.

$$p(\beta \mid T, \Theta)$$

$$= \sum_{v \in V(T) \setminus L(T)} p(x_{v_1})p(x_{v_2} \mid x_{v_1})p(x_{v_3} \mid x_{v_1})p(X_{v_1} \mid x_{v_2})p(X_{v_1} \mid x_{v_2})p(X_{v_6} \mid x_{v_3})p(X_{v_7} \mid x_{v_3})$$

$$= \sum_{x_1} \sum_{x_2} \sum_{x_3} p(x_{v_1})p(x_{v_2} \mid x_{v_1})p(x_{v_3} \mid x_{v_1})p(X_{v_1} \mid x_{v_2})p(X_{v_1} \mid x_{v_2})p(X_{v_6} \mid x_{v_3})p(X_{v_7} \mid x_{v_3})$$

Generalizing for a larger tree gives:

$$p(\beta \mid T, \Theta) = \sum_{v \in V(T) \setminus L(T)} p(x_{v_1})p(x_{v_2} \mid x_{v_1})p(x_{v_3} \mid x_{v_1}) \cdots \prod_{v \in L(T)} p(X_v \mid x_{pa(v)})$$

$$= \sum_{v_1} \sum_{v_2} \cdots \sum_{v_N} p(x_{v_1})p(x_{v_2} \mid x_{v_1})p(x_{v_3} \mid x_{v_1}) \cdots \prod_{v \in L(T)} p(X_v \mid x_{pa(v)})$$

Calculating the likelihood this way is extremely inefficient. It would in fact be $\mathcal{O}(K^N)$ where $N =\mid V(T) \setminus L(T) \mid$. To make the calculation more efficient we can use dynamic programming and conditional independence to bring the sum inside the product and break the problem into smaller sub-problems.

$$p(\beta \mid T, \Theta) = \sum_{x_{v_1}} p(x_{v_1})\mu(x_{v_1})$$

where

$$\mu(x_v) = \prod_{c \in de(v)} \sum_{x_c} p(X_c = x_c \mid X_v = x_v)\mu(x_c)$$

where $de(v)$ denotes the descendants of v and $\mu(x_v) = \prod_{c \in de(v)} p(X_c = x_c^* \mid X_v = x_v)$ if $de(v) \subseteq \beta$. The $*$ denotes that its value has been observed.

This method is much more efficient than the previous one. It is in fact $\mathcal{O}(NK^2)$. It has been implemented in matrix form in python.

2.2 Question 2.2.8

This implementation was run on the data provided. The results on the different tree sizes and their five respective samples are shown in table 1.

Tree size	1	2	3	4	5
Small	0.016179	0.01541	0.01137	0.0086	0.0409
Medium	$4.336 \cdot 10^{-18}$	$3.094 \cdot 10^{-20}$	$1.050 \cdot 10^{-16}$	$6.585 \cdot 10^{-16}$	$1.488 \cdot 10^{-18}$
Large	$3.288 \cdot 10^{-69}$	$1.109 \cdot 10^{-66}$	$2.522 \cdot 10^{-68}$	$1.242 \cdot 10^{-66}$	$3.535 \cdot 10^{-69}$

Table 1: $p(\beta \mid T, \Theta)$ for the three different tree sizes and their five respective samples.

To ponder the plausibility of these results one can do some rough calculations. Every vertex can take five different values, so if they would be equally likely they would have a probability of 0.2. Using this approximation one can use the respective trees number of leaves (3, 26 and 101) to get a rough approximation of the likelihood. These would be:

$$\left(\frac{1}{5}\right)^3 = 0.008$$

$$\left(\frac{1}{5}\right)^{26} \approx 6.71 \cdot 10^{-19}$$

$$\left(\frac{1}{5}\right)^{101} \approx 2.54 \cdot 10^{-71}$$

We see that these approximations are close to the results.

3 Simple Variational Inference

3.1 Question 2.3.9

The VI algorithm was implemented using python.

3.2 Question 2.3.10

We are given the following likelihood function for the data

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

and the following conjugate priors distribution for μ and τ

$$p(\mu \mid \tau) = \mathcal{N}(\mu \mid \mu_0, (\lambda_0\tau)^{-1})$$

$$p(\tau) = \mathrm{Gam}(\tau \mid a_0, b_0)$$

We want to find the exact posterior, $p(\mu, \tau \mid \mathcal{D})$.

$$p(\mu, \tau \mid \mathcal{D}) = \frac{p(\mathcal{D} \mid \mu, \tau)p(\mu, \tau)}{p(\mathcal{D})} \propto p(\mathcal{D} \mid \mu, \tau)p(\mu, \tau) = p(\mathcal{D} \mid \mu, \tau)p(\mu \mid \tau)p(\tau)$$

Taking the logarithm of both sides we get

$$\log(p(\mu, \tau \mid \mathcal{D})) \overset{\pm}{=} \log(p(\mathcal{D} \mid \mu, \tau)) + \log(p(\mu \mid \tau)) + \log(p(\tau))$$

$$\overset{\pm}{=} \frac{N}{2}\log(\tau) - \frac{\tau}{2}\sum_{n=1}^{N}(x_n-\mu)^2 + \frac{1}{2}\log(\tau) - \frac{\lambda_0\tau}{2}(\mu-\mu_0)^2 + (a_0-1)\log(\tau) - b_0\tau$$

$$= (a_0+\frac{N}{2}-1)\log(\tau) - b_0\tau + \frac{1}{2}\log(\tau) - \frac{\tau}{2}\Big(N\mu^2 - 2N\mu\bar{x} + \sum_{n=1}^{N}x_n^2\Big) - \frac{\lambda_0\tau}{2}(\mu^2 - 2\mu\mu_0 + \mu_0^2)$$

$$= (a_0+\frac{N}{2}-1)\log(\tau) - (b_0 + \frac{1}{2}\lambda_0\mu_0^2 + \frac{1}{2}\sum_{n=1}^{N}x_n^2)\tau + \frac{1}{2}\log(\tau) - \frac{\tau}{2}\Big((N+\lambda_0)\mu^2 - 2\mu(N\bar{x}+\lambda_0\mu_0)\Big)$$

$$= \Big(a_0+\frac{N}{2}-1\Big)\log(\tau) - \Big(b_0 + \frac{1}{2}(\lambda_0\mu_0^2 - \frac{(N\bar{x}+\lambda_0\mu_0)^2}{N+\lambda_0} + \sum_{n=1}^{N}x_n^2)\Big)\tau + \frac{1}{2}\log(\tau) - \frac{\tau(N+\lambda_0)}{2}\Big(\mu - \frac{N\bar{x}+\lambda_0\mu_0}{N+\lambda_0}\Big)^2$$

$$= (\hat{a}-1)\log(\tau) - \hat{b}\tau + \frac{1}{2}\log(\tau) - \frac{\tau\hat{\beta}}{2}(\mu-\hat{\mu})^2$$

Notice that from the third to second last expression the square was completed for the last term, leading to adding $\frac{\tau}{2}\frac{(N\bar{x}+\lambda_0\mu_0)^2}{N+\lambda_0}$.

From the last two expressions we see that the exact posterior is *normal-gamma* distributed. That is

$$p(\mu,\tau) = \mathcal{N}(\mu \mid \hat{\mu}, (\hat{\beta}\tau)^{-1})\mathrm{Gam}(\lambda \mid \hat{a}, \hat{b})$$

where

$$\hat{\beta} = N + \lambda_0$$

$$\hat{\mu} = \frac{N\bar{x} + \lambda_0\mu_0}{N + \lambda_0} = \frac{N\bar{x} + \lambda_0\mu_0}{\hat{\beta}}$$

$$\hat{a} = a_0 + \frac{N}{2}$$

$$\hat{b} = b_0 + \frac{1}{2}\Big(\lambda_0\mu_0^2 - \frac{(N\bar{x} + \lambda_0\mu_0)^2}{N + \lambda_0} + \sum_{n=1}^{N} x_n^2\Big) = b_0 + \frac{1}{2}\Big(\lambda_0\mu_0^2 - \hat{\beta}\hat{\mu}^2 + \sum_{n=1}^{N} x_n^2\Big)$$

3.3 Question 2.3.11

Case 1

The first case is the one used in Figure 10.4 in Bishop. That is μ and τ are the parameters for a univariate standard normal distribution. Thus, the data was simulated from a univariate standard normal distribution with 100 data points. Assuming the conjugate priors were relatively well known the parameters shown in table 2 were chosen.

a_0	b_0	μ_0	λ_0
10	10	0	10

Table 2: Chosen parameters of priors for Case 1

These results can be seen in figure 1 and table 3.

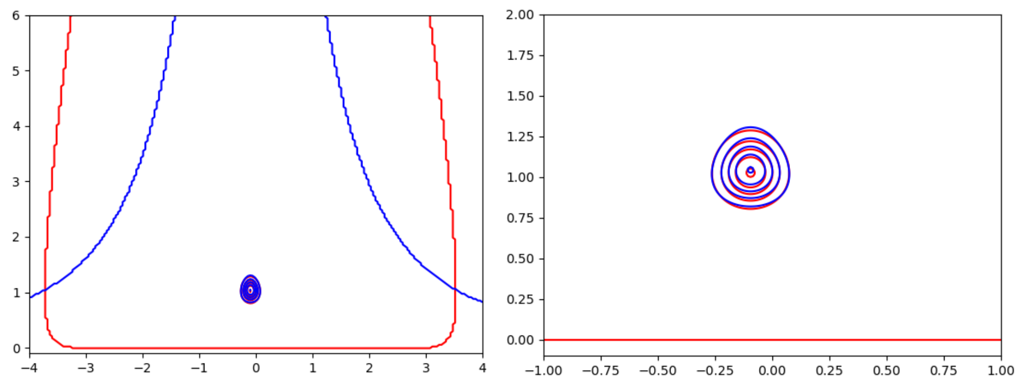


Figure 1: Contour plots of posterior calculated by VI (in red) and by exact posterior (in blue) for Case 1. Horizontal axis μ, vertical axis τ. **Left:** Zoomed out. **Right:** Zoomed in.

Method	a	b	μ	λ/β
VI	60	57.55	-0.09	114.68
Exact	60	57.07	-0.09	110

Table 3: Parameters calculated by VI and exact posterior for Case 2.

We see that the exact posterior is much better at capturing the variance of μ when the precision, τ, is low. This can be seen in the blue contours being wider for smaller τ. However, we see almost no difference in the calculated parameters, which means that the difference is captured in the differing distributions.

Moreover, we also see that the outer contours are wide, which is due to the small amount of data points.

Case 2

We will now increase the number of data points to $n = 10,000$ but keep the distribution we simulate the data from the same, as well as the prior parameters (they can thus be seen in table 2).

The results can be seen in figure 2 and table 4.

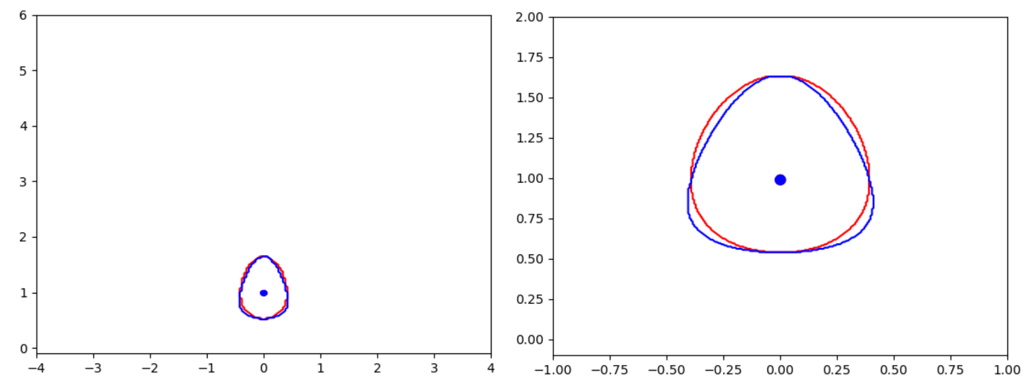


Figure 2: Contour plots of posterior calculated by VI (in red) and by exact posterior (in blue) for Case 2. Horizontal axis μ, vertical axis τ. **Left:** Zoomed out. **Right:** Zoomed in.

Method	a	b	μ	λ/β
VI	5010	5073.67	0.0011	9884.39
Exact	5010	5073.16	0.0011	10010

Table 4: Parameters calculated by VI and exact posterior for Case 2.

We see that the contour plots now become much tighter, indicating that the increased number of data points significantly has increased the accuracy. In the zoomed picture we also see that most of the probability is contained in the neighborhood of the true parameters, $(0, 1)$. We still see the difference in the shape between the exact and the posterior estimated by the VI algorithm.

Case 3

For the last case we choose a different scenario, were $\mu = 10$ and $\tau = 0.5$. We simultaneously reduce the number of data points to $n = 1,000$ and make a bad guess for the prior parameters (assuming they are not well known). The chosen parameters can be seen in table 5.

a_0	b_0	μ_0	λ_0
2	0.1	5	1

Table 5: Chosen parameters of priors for Case 3

The results can be seen in figure 3 and table 6.

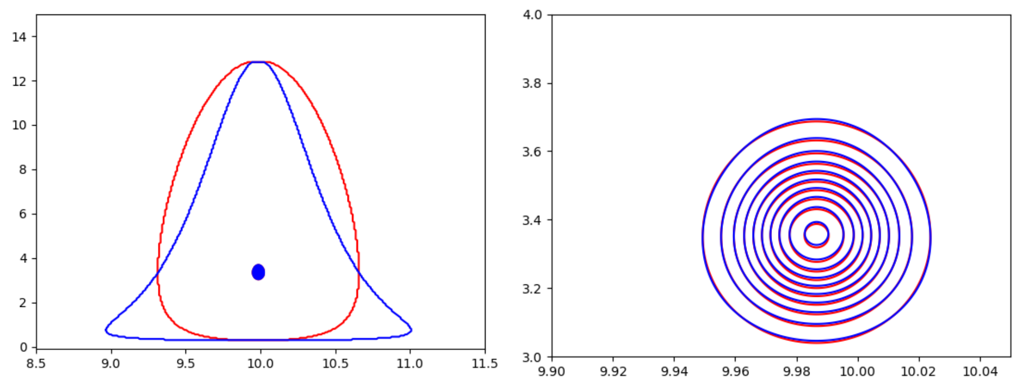


Figure 3: Contour plots of posterior calculated by VI (in red) and by exact posterior (in blue) for Case 3. Horizontal axis μ, vertical axis τ. **Left:** Zoomed out. **Right:** Zoomed in.

Method	a	b	μ	λ/β
VI	502	149.42	9.987	3363.07
Exact	502	149.27	9.987	1001

Table 6: Parameters calculated by VI and exact posterior for Case 3.

Contrary to the other cases, in Case 3 neither of the methods are able to get a good estimate of the true values of τ (they still do well for μ). The contour plots should be centered around $(10, 1/0.5^2) = (10, 4)$ if they had estimated the parameters correctly. This is due to the bad choice of prior parameters (a better choice of prior parameters gives the distribution shown in figure 4). This choice would be equal to believing μ is centered around 5 and τ around $2/0.1 = 20$ (with a variance $\text{Var}(\tau) = 2/0.1^2 = 200$). This shows us that these methods have a hard time when little is known about the prior distributions.

We still see the familiar difference in shape in the zoomed out figure, but the contours start to coincide in the zoomed in figure.

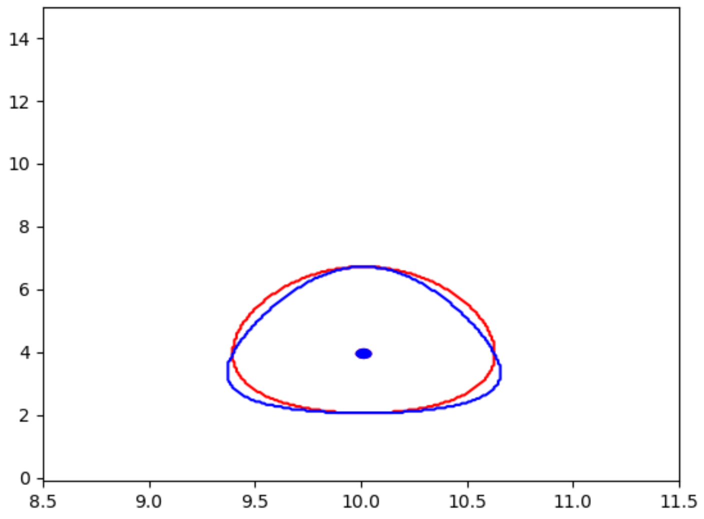


Figure 4: Case 3 but with a better choice of prior parameters.

4 Mixture of trees with observable variables

4.1 Question 2.4.12

The EM algorithm was implemented using python. Sieving was applied in the following way:

1. Choose 100 random seeds and simulate one tree from every seed
2. Run the algorithm for 10 iterations for these 100 trees and choose the 10 seeds that gave the highest likelihood
3. Re-simulate the trees from the 10 best seeds and run the algorithm for 100 iterations
4. Choose the best seed and re-run it to convergence (relative error $< 10^{-6}$) or a maximum of 100 iterations

4.2 Question 2.4.13

The algorithm was run on the provided data. The graphs of the log- and regular likelihood versus the iteration of the algorithm can be seen in figure 5.

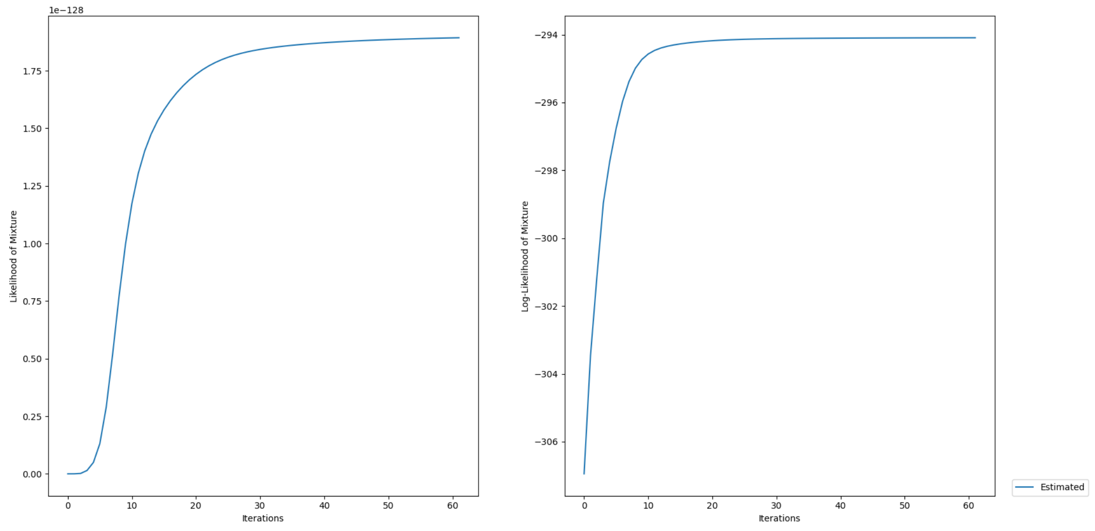


Figure 5: The likelihood of the data given the current (in terms of iteration) tree mixture.

Further, the inferred trees were also compared with the real trees using the unweighted Robinson-Foulds metric and the likelihood. The results can be seen in table 7 and 8 respectively.

	Real Tree 0	**Real Tree 1**	**Real Tree 2**
Inferred Tree 0	4	5	4
Inferred Tree 1	0	3	4
Inferred Tree 2	2	3	4

Table 7: Robin-Foulds metric between the inferred and real trees.

	Log-likelihood	**Likelihood**
EM	-294.093	1.89298e-128
Real	-311.449	5.48542e-136

Table 8: Values of likelihood for the inferred and real tree mixture.

From table 7 we see that the inferred and real trees are not so similar in terms of structure, with an exception of inferred tree 1 and the real tree 0 which have distance of 0. Note that a value of 5 is large for these trees since the only contain five nodes. Matching the inferred and real trees to get the minimal total distance we get a total distance of 7. Furthermore, looking at the likelihood of the samples given a certain tree mixture we see that the EM algorithm produces a much higher likelihood than the real tree mixture. This is probably due to the low amount of samples, which could lead to overfitting.

4.3 Question 2.4.14

Scenario 1

As a first test, we use the same tree mixture as the previous question but increase the number of samples to $n = 1000$. This, as expected, made the algorithm take much longer to converge. The results can be seen in figure 6. We see that the log-likelihood graph has a similar shape to the previous example with $n = 100$. However, we do get a horizontal line for the likelihood due to python not being able to handle so small numbers (around 10^{-1300}).

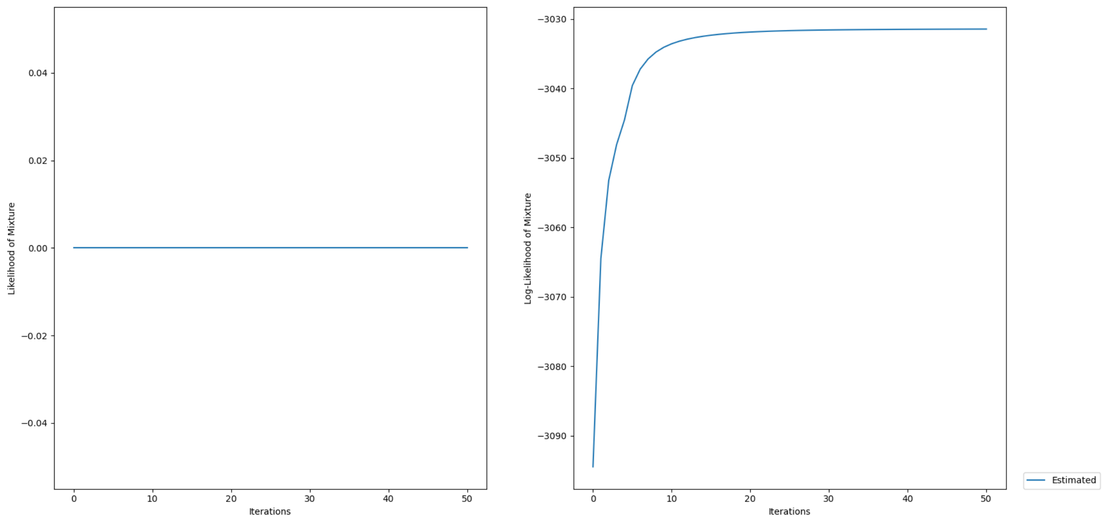


Figure 6: The likelihood of the data given the current (in terms of iteration) tree mixture.

	Real Tree 0	**Real Tree 1**	**Real Tree 2**
Inferred Tree 0	0	3	4
Inferred Tree 1	4	3	0
Inferred Tree 2	4	5	4

Table 9: Robin-Foulds metric between the inferred and real trees.

	Log-likelihood	**Likelihood**
EM	-3031.45	3.4834e-1316
Real	-3052.08	3.1732e-1325

Table 10: Values of likelihood for the inferred and real tree mixture.

We know see that the inferred trees better matches the real tree in structure. Matching in the same way as the previous question gives a minimal total distance of 5, where the pair real tree 1 and inferred tree 2 is the only contributor. This indicates that the higher number of samples leads to a better matching, more precisely since this resulted in two perfect matches (0 distance).

Furthermore, looking at the difference in likelihood we see that the gap in log-likelihood has increased, from ≈ 17 to ≈ 21. However, the number of samples is also ten times more, so this means that the gap has shrunk relatively. This supports the hypothesis that more samples decreased the overfitting.

Scenario 2

For the second scenario, the number of samples was once again reduced to $n = 100$. However, the number of nodes were increased to 7 while the number of clusters reduced to 2. What one first notices is that the algorithm converged after only 10 iterations. This could be due to the lower number of clusters.

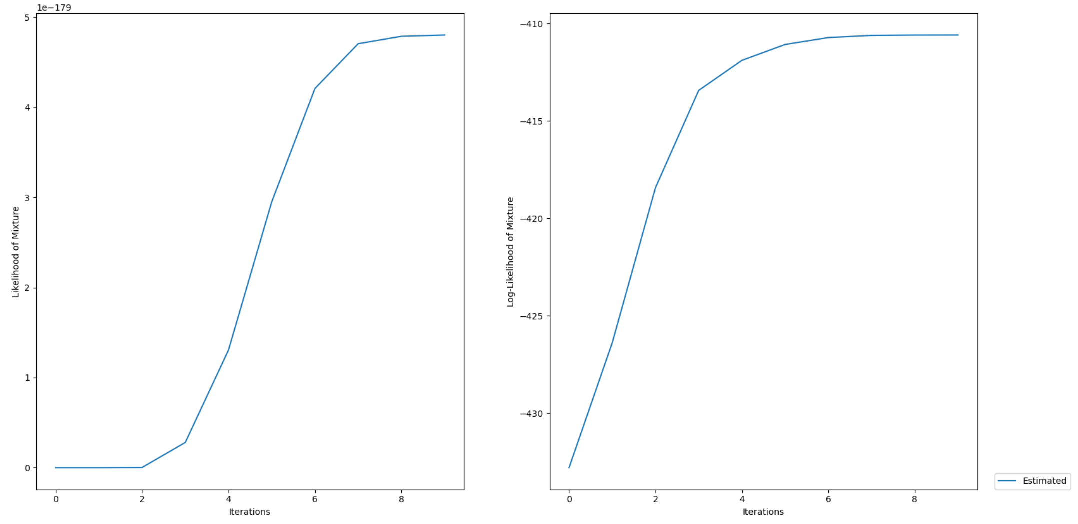


Figure 7: The likelihood of the data given the current (in terms of iteration) tree mixture.

	Real Tree 0	**Real Tree 1**
Inferred Tree 0	3	4
Inferred Tree 1	3	4

Table 11: Robin-Foulds metric between the inferred and real trees.

	Log-likelihood	**Likelihood**
EM	-410.593	4.80378e-179
Real	-424.39	4.89653e-185

Table 12: Values of likelihood for the inferred and real tree mixture.

We first of all see now perfect match in terms of structure between the inferred and real trees. As in question 2.4.13 we see that the minimal total distance is 7. Compared to the same question, this tree mixture has 14 nodes compared to 15. This indicates that the algorithm does about as good for this tree mixture as the tree mixture with 5 nodes and 2 clusters.

Looking at the likelihoods, we first of all see that they are much lower, even though it is the same number of samples as in 2.4.13. This is due to the likelihood of every node in the respective trees are multiplied with each other and the likelihoods of the different trees are added (after multiplying by their π_k). We also see that the difference in likelihood between the inferred and real tree mixture is smaller. This is probably due to less overfitting, due to this tree mixture having less variables to optimize (π and $2 \cdot 7$ nodes).

Scenario 3

For the third and final scenario, the number of samples was once again put at $n = 100$. However, the number of nodes were decreased to 4 while the number of clusters increased to 5. The threshold for convergence was decreased to 10^{-8} to not make the algorithm

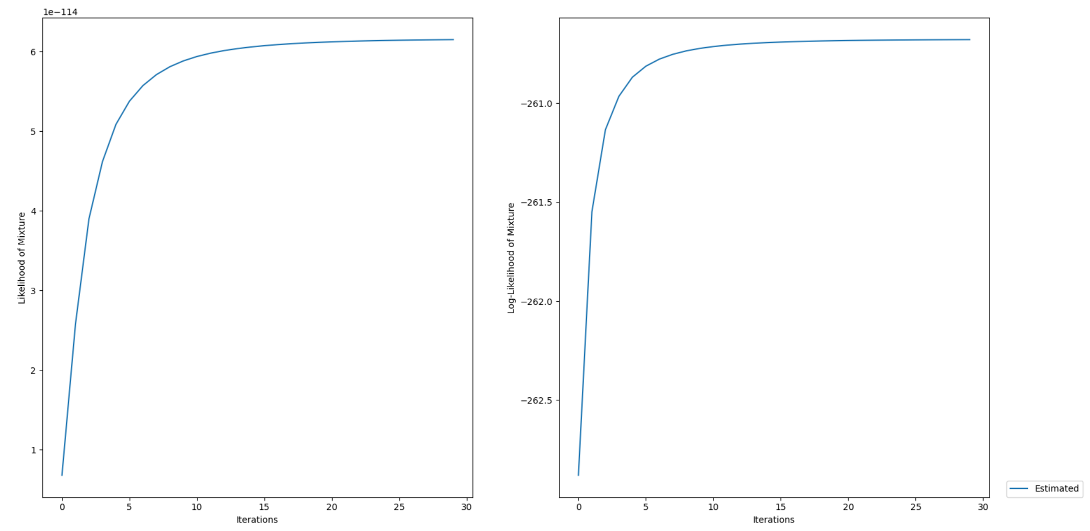


Figure 8: The likelihood of the data given the current (in terms of iteration) tree mixture.

	RT 0	**RT 1**	**RT 2**	**RT 3**	**RT 4**
IT 0	2	2	3	2	0
IT 1	4	4	3	4	2
IT 2	4	4	3	4	2
IT 3	2	2	3	2	0
IT 4	4	4	3	4	2

Table 13: Robin-Foulds metric between the inferred (IT) and real trees (RT).

	Log-likelihood	**Likelihood**
EM	-260.678	6.15237e-114
Real	-267.582	6.17358e-117

Table 14: Values of likelihood for the inferred and real tree mixture.

At first we see that both IT 0 and IT 3 is a perfect match, structure wise, of RT 4. However, choosing one of these when calculating the minimal distance is not the only solution, there are in fact several combinations without these that will grant the minimal value of 13. Moreover, this total distance is higher to the other scenarios, even in the relative case (13 distance with $4 \cdot 5 = 20$ different nodes). We thus conclude that the algorithm had a harder time with these particular structure of tree mixture. It would probably do better with more samples, as was shown with the first scenario.[1]

We also notice that the likelihoods are quite high compared to the other scenarios, they are in fact the highest. This is due to the same reasoning explained in the previous scenario, namely that the number of nodes per tree has the largest influence on the likelihood (if the number of samples is kept constant).

[1]When n was increased to 300 the minimum total distance was reduced to 9

A Appendix: Code

A.1 2.2 - Likelihood of a Tree Graphical Model implementation

```python
import numpy as np
from Tree import Tree
from Tree import Node


def calculate_likelihood(root, beta, k):
    print("Calculating the likelihood...")

    likelihood = recursive(root, beta, k)

    return likelihood


def recursive(node, beta, k):
    if len(node.descendants) == 0: # If the vertex is a leaf
        obs_index = int(beta[int(node.name)])
        theta = np.array(node.cat)
        theta = theta[:,obs_index] # Extract probabilities of the observed vertex given all
        return theta
    else:
        theta = np.array(node.cat)
        descendant_theta = np.ones(k)
        for descendant in node.descendants:
            descendant_theta *= recursive(descendant, beta, k) # Multiply probabilities of
        theta = np.dot(theta, descendant_theta)
        return theta


def main(treeSize):
    print("\nRunning algorithm for " + treeSize + " tree...")
    if treeSize == "small":
        filename = "data/q2_2/q2_2_small_tree.pkl"
    elif treeSize == "medium":
        filename = "data/q2_2/q2_2_medium_tree.pkl"
    else:
        filename = "data/q2_2/q2_2_large_tree.pkl"

    t = Tree()
    t.load_tree(filename)
    # t.print()

    for sample_idx in range(t.num_samples):
        beta = t.filtered_samples[sample_idx]
        sample_likelihood = calculate_likelihood(t.root, beta, t.k)
        print("\tLikelihood for sample " + str(sample_idx) + ": ", sample_likelihood)


def ownTree():
    topology_array = np.array([float('nan'), 0., 0., 1., 1.])
    theta_array = [
        np.array([0.2, 0.8]),
        np.array([[0.9, 0.1], [0.9, 0.1]]),
        np.array([[0.05, 0.95], [0.1, 0.9]]),
        np.array([[0.9, 0.1], [0.9, 0.1]]),
        np.array([[0.1, 0.9], [0.1, 0.9]])
    ]
    t = Tree()
    t.load_tree_from_direct_arrays(topology_array, theta_array)
    t.print()

    t.sample_tree(1)


    beta = t.filtered_samples[0]
    print(beta)
    sample_likelihood = calculate_likelihood(t.root, beta, t.k)
    print("\tLikelihood: ", sample_likelihood)

# ownTree()

if __name__ == "__main__":
    main("small")
    main("medium")
    main("large")
```

A.2 2.3 - Simple Variational Inference implementation

```
import numpy as np
from scipy.stats import norm
from scipy.stats import gamma
import matplotlib.pyplot as plt


def calcMu(a_N, b_N, x, mu0, lambda0):
  N = len(x)                # N
  x_mean = np.mean(x)       # Mean of x
  tau_expected = a_N / b_N  # E[tau]

  mu_N = (lambda0 * mu0 + N * x_mean) / (lambda0 + N)
  lambda_N = (lambda0 + N) * tau_expected

  return mu_N, lambda_N


def calcTau(mu_N, lambda_N, x, a0, b0, mu0, lambda0):
  N = len(x)                                      # N
  x_mean = np.mean(x)                             # Mean of x
  x_square = sum(x ** 2)                          # sum(x^2)
  mu_expected = mu_N                              # E[mu]
  mu_square_expected = 1 / lambda_N + mu_N ** 2   # E[mu^2]

  a_N = a0 + N/2
  b_N = b0 + (1/2) * (x_square + lambda0 * mu0 ** 2) - (mu0 * lambda0 + N * x_mean) * mu_ex

  return a_N, b_N


def getData(n, mu, std_2, seed = 100):
  np.random.seed(seed)
  return np.random.normal(mu, std_2, n) # Drawing from gaussian distrbution


def threshold(a_N, b_N, mu_N, lambda_N, a_N_next, b_N_next, mu_N_next, lambda_N_next, thres
  # Checking if the change in variables are within the threshold
  if abs(a_N_next / a_N - 1) < threshold:
    if abs(b_N_next / b_N - 1) < threshold:
      if abs(mu_N_next / mu_N - 1) < threshold:
        if abs(lambda_N_next / lambda_N - 1) < threshold:
          return False
  return True


def printVIResults(data, iteration):
  labels = ["a", "b", "mu", "lambda"]
  for i in range(len(labels)):
    print(labels[i] + " | From " + str(data[i]) + " to " + str(data[i+4]))
  print("It took " + str(iteration) + " iterations.")


def printTrueResults(data):
  labels = ["a", "b", "mu", "beta"]
  for i in range(len(labels)):
    print(labels[i] + " | " + str(data[i]) )


def q_mu(mus, mu, lambd):
  return norm.pdf(mus, mu, np.sqrt(1 / lambd)) # Gaussian pdf


def q_tau(taus, a, b):
  return gamma.pdf(taus, a, loc = 0, scale = 1 / b) # Gamma pdf


def calcNormGamParam(x, mu_0, lambda_0, a0, b0):
  N = len(x)              # N
  x_mean = np.mean(x)     # Mean of x
  x_square = sum(x ** 2)  # sum(x^2)

  beta = N + lambda_0     # Without tau
  mu = (N * x_mean + lambda_0 * mu_0) / beta

  a = a0 + N / 2
  b = b0 + (lambda_0 * (mu_0 ** 2) - beta * (mu ** 2) + x_square) / 2

  return beta, mu, a, b

```

```

def relativeErrors(vi_results, true_results):
  labels = ["a", "b", "mu"]
  for i in range(len(labels)):
    print(labels[i] + " | " + str(round(100*abs(vi_results[i]/true_results[i]-1),2)) + "%"


def vi_algorithm(a_N, b_N, mu_N, lambda_N, x, mu0, lambda0, a0, b0):
  thresh = 1e-7

  running = True
  iteration = 0

  # Running algorithm
  while running:
    #Calculating new parameters
    mu_N_next, lambda_N_next = calcMu(a_N, b_N, x, mu0, lambda0)
    a_N_next, b_N_next = calcTau(mu_N, lambda_N, x, a0, b0, mu0, lambda0)

    #Checking if threshold is satisfied
    running = threshold(a_N, b_N, mu_N, lambda_N, a_N_next, b_N_next, mu_N_next, lambda_N_n

    # Saving parameters
    a_N, b_N, mu_N, lambda_N = a_N_next, b_N_next, mu_N_next, lambda_N_next
    iteration += 1

  return a_N, b_N, mu_N, lambda_N, iteration


def pTrue(x, y, beta, mu, a, b):
  return norm.pdf(x, mu, np.sqrt(1 / (beta * y))) * gamma.pdf(y, a, loc = 0, scale = 1 / b)


def plotResults(a, b, mu, precision, center, std_2, exact = False):
  # Getting interval that fits data
  mus = np.linspace(-4,4,300) * (std_2 ** (0.7)) + center
  taus = np.linspace(-0.1,6,200) * std_2 ** (-2)
  # mus = np.linspace(8.5,11.5,300)
  # taus = np.linspace(-0.1,15,300)
  # mus = np.linspace(9.9,10.05,300)
  # taus = np.linspace(3,4,300)

  # For the exact posterior
  if exact:
    color = "blue"
    Ms, Ts = np.meshgrid(mus, taus, indexing="ij")
    Z = np.zeros_like(Ms)

    for i in range(Z.shape[0]):
      for j in range(Z.shape[1]):
          Z[i][j] = pTrue(mus[i], taus[j], precision, mu, a, b)

  # The posterior calculated by the VI algorithm
  else:
    color = "red"
    q_mus = q_mu(mus, mu, precision)
    q_taus = q_tau(taus, a, b)
    Ms, Ts = np.meshgrid(mus, taus, indexing="ij")

    Z = np.outer(q_mus, q_taus)

  # Plotting the contour
  plt.contour(Ms, Ts, Z, 10, colors = color)


def main():
  # Data attributes
  n = 10000
  center = -2
  std = 30

  print("1. Simulating guassian data with\nmu =", center, "| sigma =", std, "| n =", n)

  x = getData(n, center, std, 1021)

  # -------------------------------------------------------------------------------------

  a0, b0 = 0.01, 9
  mu0, lambda0 = -2, 10

  print("\n2. Setting prior parameters\na0 =", a0, "| b0 =", b0, "| mu0 =", mu0, "| lambda0

  # ------------------------------------------------------------------------------------------

  a_start, b_start = 1e-9, 1e-9
  mu_start, lambda_start = 1e-9, 1e-9

  print("\n3. Setting start values\na_start =", a_start, "| b_start =", b_start, "| mu_star

  a_N, b_N = a_start, b_start
  mu_N, lambda_N = mu_start, lambda_start

  # ------------------------------------------------------------------------------------------

  print("\n4. Running VI algorithm...")

  a_N, b_N, mu_N, lambda_N, iteration = vi_algorithm(a_start, b_start, mu_start, lambda_sta

  printVIResults([a_start, b_start, mu_start, lambda_start, a_N, b_N, mu_N, lambda_N], iter

  # ------------------------------------------------------------------------------------------

  print("\n5. Calculating true posterior...")

  beta, mu, a, b = calcNormGamParam(x, mu0, lambda0, a0, b0)

  printTrueResults([a, b, mu, beta])

  # ------------------------------------------------------------------------------------------

  print("\n6. Calculate relative errors")

  relativeErrors([a_N, b_N, mu_N], [a, b, mu])

  # ------------------------------------------------------------------------------------------

  print("\n7. Plot results")

  plotResults(a_N, b_N, mu_N, lambda_N, center, std)

  plotResults(a, b, mu, beta, center, std, True)

  plt.show()
```

```


main()
```

A.3 2.4 - Mixture of trees with observable variables implementation

```python
import numpy as np
import matplotlib.pyplot as plt
import Kruskal_v1 as kr
import Kruskal_v2 as kr2
import queue
from Tree import TreeMixture
from Tree import Tree
import sys
import random
from Phylogeny import tree_to_newick_rec
from tabulate import tabulate
import dendropy


def save_results(loglikelihood, topology_array, theta_array, filename):

    likelihood_filename = filename + "_em_loglikelihood.npy"
    topology_array_filename = filename + "_em_topology.npy"
    theta_array_filename = filename + "_em_theta.npy"
    print("Saving log-likelihood to ", likelihood_filename, ", topology_array to: ", topolo
          ", theta_array to: ", theta_array_filename, "...")
    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=100, debugging = False):

    # Set the seeds
    np.random.seed(seed_val)
    num_start_tms = 100
    seeds = np.array(np.random.ranf((num_start_tms,)) * 1e5).astype(int)


    # Run the algorithm
    print("Running EM algorithm...")


    # Creating intital tms
    print("\nCreating initial", num_start_tms, "tree mixtures...")
    tree_mixtures = create_initial_tms(seeds, num_clusters, samples)

    if debugging:
        em_one_dimension(tree_mixtures[0], samples, num_clusters, 10)

        return

    # First step of sieving
    print("\nRunning 10 iterations for the first", num_start_tms, "tree mixtures...\n")
    loglikelihoods = []
    j = 0
    for tm in tree_mixtures:
        loglikelihoods.append(em_one_dimension(tm, samples, num_clusters, 10))
        j += 1
        if j % 10 == 0:
            print(str(j) + " of the first", num_start_tms, "tree mixtures done")


    # Choosing 10 best tree mixtures
    num_best_trees = 10
    print("\nChoosing the", num_best_trees, "best tree mixtures")
    indices = np.argsort(-np.array(loglikelihoods))
    sieved_indices = indices[:num_best_trees]
    best_seeds = seeds[sieved_indices]


    # Recreating best tms
    print("\nRecreating the", num_best_trees, "best tree mixtures and running them for", ma
    tree_mixtures = create_initial_tms(best_seeds, num_clusters, samples)


    # Running 100 iterations for the best trees
    loglikelihoods = []
    j = 0
    for tm in tree_mixtures:
        loglikelihoods.append(em_one_dimension(tm, samples, num_clusters, 100))
        j += 1
        print(str(j) + " of the first", num_best_trees, "tree mixtures done")

```

```python
    # Choosing the best tree mixture
    print("\nChoosing the best tree mixtures and running it until convergence\n")
    indices = np.argsort(-np.array(loglikelihoods))
    best_seed = best_seeds[indices[0]]
    tree_mixtures = create_initial_tms([best_seed], num_clusters, samples)
    threshold = 1e-6

    tm, loglikelihoods = em_one_dimension_final(tree_mixtures[0], samples, num_clusters, ma

    print("\n The final loglikelihood was", loglikelihoods[-1])

    topology_list, theta_list = get_arrays(tm, num_clusters)

    return loglikelihoods, topology_list, theta_list, tm.pi


def get_arrays(tm, num_clusters):
    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())

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

    return topology_list, theta_list


def create_initial_tms(seeds, num_clusters, samples):
    tree_mixtures = []
    for seed in seeds:
        tm = TreeMixture(num_clusters=num_clusters, num_nodes=samples.shape[1])
        tm.simulate_pi(seed_val=seed)
        tm.simulate_trees(seed_val=seed)
        tree_mixtures.append(tm)

    return tree_mixtures


def em_one_dimension_final(tm, samples, num_clusters, max_num_iterations, threshold):
    d = tm.clusters[0].k
    loglikelihoods = []
    iteration = 0

    for _ in range(max_num_iterations):
        # Step 0 - Create new tree mixture
        tm_new = TreeMixture(num_clusters=num_clusters, num_nodes=samples.shape[1])

        # Step 1 - Calculate responsibilities
        r_new = responsibilities(tm, samples)

        # Step 2 - Calculate pi'
        tm_new.pi = new_pi(r_new)

        # Step 3 - Create G_k's
        tm_new.clusters = create_new_tms2(r_new, samples, num_clusters, d)

        iteration += 1
        loglikelihoods.append(log_likelihood(tm_new, samples)[0])  # Calculating loglikelih

        tm = tm_new

        if iteration > 1:
            if loglikelihoods[-2] > loglikelihoods[-1]:
                print("!! The likelihood decreased !!") # Debugging
            if abs(loglikelihoods[-1] / loglikelihoods[-2] - 1) < threshold:
                break

    if iteration < max_num_iterations:
        print("The algorithm converged after", iteration, "iterations")
    else:
        print("The algorithm did not converge after", iteration, "iterations")

    return tm, loglikelihoods


def em_one_dimension(tm, samples, num_clusters, max_num_iterations):
    d = tm.clusters[0].k

    for _ in range(max_num_iterations):
        # Step 0 - Create new tree mixture
        tm_new = TreeMixture(num_clusters=num_clusters, num_nodes=samples.shape[1])

        # Step 1 - Calculate responsibilities
        r_new = responsibilities(tm, samples)

        # Step 2 - Calculate pi'
        tm_new.pi = new_pi(r_new)

        # Step 3 - Create G_k's
        tm_new.clusters = create_new_tms2(r_new, samples, num_clusters, d) # Adding the tre

        tm = tm_new

    loglikelihood = log_likelihood(tm_new, samples) # Calculating loglikelihood

    return loglikelihood[0]


def create_new_tms(r_new, samples, num_clusters, d):
    clusters_new = []
    num_vertices = samples.shape[1]
    G_ks = [kr.Graph(num_vertices) for i in range(num_clusters)] # Creating a graph for eve
    for (k, G_k) in enumerate(G_ks):
        root_cdf = np.array([q_root(k, a, r_new, samples) for a in range(d)]) # Calculating

        for s in range(num_vertices):
            for t in range(s):
                G_k.addEdge(t, s, mutual_information(k, t, s, r_new, samples, d)) # Assign

        # print(G_k.graph)
        max_G = G_k.maximum_spanning_tree() # Calculating the maximum spanning tree

        clusters_new.append(create_tree(max_G, k, r_new, samples, d, root_cdf)) # Adding th

    return clusters_new


def create_new_tms2(r_new, samples, num_clusters, d):
    clusters_new = []
    num_vertices = samples.shape[1]
    G_ks = [set() for i in range(num_clusters)] # Creating a graph for every cluster
    for (k, G_k) in enumerate(G_ks):
        root_cdf = np.array([q_root(k, a, r_new, samples) for a in range(d)]) # Calculating

        for s in range(num_vertices):
            for t in range(s):
                G_k.add((t, s, mutual_information(k, t, s, r_new, samples, d))) # Assign va

        vertices = list(range(num_vertices))
        graph = {
            'vertices': vertices,
            'edges': G_k
        }

        result = kr2.maximum_spanning_tree(graph) # Calculating the maximum spanning tree
        max_G = np.array([np.array([edge[0], edge[1]]) for edge in result])

        clusters_new.append(create_tree(max_G, k, r_new, samples, d, root_cdf)) # Adding th

    return clusters_new


def q_root(k, a, r, samples):
    return q_single(k, 0, a, r, samples)


def log_likelihood(tm, samples):
    return sum(np.log(responsibilities(tm, samples, True)))


def nan_function(x):
    if x == 0:
        return float('nan')
    else:
        return x


def create_tree(G, k, r, samples, d, root_cdf):
    topology_array, index_array = topology_index(G, d)

    theta_array = t_array(k, topology_array, index_array, r, samples, d)
    theta_array.insert(0, root_cdf)

    t = Tree()
    t.load_tree_from_direct_arrays(np.array(topology_array), theta_array)

    return t


def t_array(k, topology_array, index_array, r, samples, d):
    theta_array = []
    for s in range(1,len(topology_array)):
        t = topology_array[s] # Index of parent
        theta = np.array([[q_conditional(k, s, t, a, b, r, samples) for a in range(d)] for
        theta_array.append(theta)

    return theta_array


def topology_index(G, d):
    topology_array = [float('nan')]
    index_array = []
    visited_rows = []

    q = queue.Queue()
    q.put(0)
    i = 0

    while not q.empty():
        x = q.get()

        index_1, index_2 = np.where(G == x)
        index_array.append(x)
        for j in range(len(index_1)):
            row = index_1[j]
            descendant = int(G[row, (index_2[j] + 1) % 2])
            if row not in visited_rows:
                visited_rows.append(row)
                topology_array.append(i)
                q.put(descendant)
        i += 1

    return topology_array, index_array


def mutual_information(k, s, t, r, samples, d):
    information = 0

    for a in range(d):
        q_a = q_single(k, s, a, r, samples)
        for b in range(d):
            q_b = q_single(k, t, b, r, samples)
            q_ab = q_joint(k, s, t, a, b, r, samples)
            if q_ab != 0:
                information += (q_ab + sys.float_info.epsilon) * np.log(q_ab / (q_a * q_b )

    return information


def q_conditional(k, s, t, a, b, r, samples):
    q = q_joint(k, s, t, a, b, r, samples) / q_single(k, t, b, r, samples)

    return q


def q_single(k, s, a, r, samples):
    denominator = sum(r[:,k]) # Sum of r_n,k over N

    column = samples[:,s]

    indices = np.where(column == a)[0]

    numerator = sum(r[indices,k])

    q = numerator / denominator

    return q


def q_joint(k, s, t, a, b, r, samples):
    denominator = sum(r[:,k]) # Sum of r_n,k over N


    columns = samples[:,[s,t]]

    indices = np.unique(np.where((columns == [a,b]).all(axis=1))[0])


    numerator = sum(r[indices,k])

    q = numerator / denominator

    return q


def responsibilities(tm, samples, log_prob = False):
    n_clusters = len(tm.clusters)
    n_samples = len(samples)

    p_matrix = np.zeros((n_samples, n_clusters))

    for i in range(n_samples): # Calculate probabilities of every sample for every cluster
        p_matrix[i,:] = [responsibility(tree.root, samples[i]) for tree in tm.clusters]

    r_unscaled = p_matrix * tm.pi # Multiply by categorical

    probabilities = r_unscaled.sum(axis=1).reshape((n_samples,1)) # Get normalizing factors

    if log_prob:
        return probabilities

    r = r_unscaled * probabilities ** (-1) # Normalize r

    return r


def new_pi(r):
    N = r.shape[0]
    return r.sum(axis=0) / N


def responsibility(node, sample):
    if node.ancestor == None:
        p = node.cat[sample[int(node.name)]] # Value of node
    else:
        p =  node.cat[sample[int(node.ancestor.name)]][sample[int(node.name)]] # Value of a
    if len(node.descendants) > 0:
        for descendant in node.descendants:
            p *= responsibility(descendant, sample)
    return p


def rf_analysis(real_values_filename, output_filename, num_clusters):
    print("\n4.1.1 Loading ground truth trees from Newick files:\n")

    # If you want to compare two trees, make sure you specify the same Taxon Namespace!
    tns = dendropy.TaxonNamespace()

    realTrees = []
    for i in range(num_clusters):
        filename = real_values_filename + "_tree_" + str(i) + "_newick.txt"
        with open(filename, 'r') as input_file:
            newick_str = input_file.read()
        realTrees.append(dendropy.Tree.get(data=newick_str, schema="newick", taxon_namespac


    print("\n4.1.2 Loading inferred trees")
    filename = output_filename + "_em_topology.npy"  # This is the result you have.
    topology_list = np.load(filename)

    inferredTrees = []
    for i in range(num_clusters):
        rt = Tree()
        rt.load_tree_from_direct_arrays(topology_list[i])
        inferredTrees.append(dendropy.Tree.get(data=rt.newick, schema="newick", taxon_names

    print("\n4.1.3 Compare trees and print Robinson-Foulds (RF) distance:\n")
    for i in range(num_clusters):
        print("\tt" +str(i) + " vs inferred trees")
        for j in range(num_clusters):
            print("\tRF distance: \t", dendropy.calculate.treecompare.symmetric_difference(


def true_log_likelihood(real_values_filename, sample_filename, num_clusters):
    samples = np.loadtxt(sample_filename, delimiter="\t", dtype=np.int32)

    tm = TreeMixture(num_clusters=num_clusters, num_nodes=samples.shape[1])

    tm.load_mixture(real_values_filename)

    topology_array, theta_array = get_arrays(tm, num_clusters)

    print("\nStructure of the true trees:")
    for i in range(num_clusters):
        print("\n\tCluster: ", i)
        print("Pi: ", tm.pi[i])
        print("\tTopology: \t", topology_array[i])
        print("\tTheta: \t", theta_array[i])

    return log_likelihood(tm, samples)


def create_real_trees1():
    # Create tree for scenario 1
    filename = "data/q2_4/more_samples"

    tm = TreeMixture(num_clusters=3, num_nodes=5)
    tm.load_mixture("data/q2_4/q2_4_tree_mixture.pkl")
    tm.samples = list()
    tm.sample_assignments = list()

    tm.sample_mixtures(1000, 1337)

    tm.save_mixture(filename)


def create_real_trees2():
    # Create tree for scenario 2
    filename = "data/q2_4/more_nodes_less_clusters.pkl"

    tm = TreeMixture(num_clusters=2, num_nodes=7)

    tm.simulate_pi(seed_val=1337)
    tm.simulate_trees(seed_val=1337)

    tm.sample_mixtures(100, 1337)

    tm.save_mixture(filename, True)


def create_real_trees3():
    # Create tree for scenario 3
    filename = "data/q2_4/less_nodes_more_clusters.pkl"

    tm = TreeMixture(num_clusters=5, num_nodes=4)

    tm.simulate_pi(seed_val=1337)
    tm.simulate_trees(seed_val=1337)

    tm.sample_mixtures(100, 1337)

    tm.save_mixture(filename, True)


def main(scenario = "normal"):
    seed_val = 123412567

    if scenario == "normal":
        sample_filename = "data/q2_4/q2_4_tree_mixture.pkl_samples.txt"
        output_filename = "q2_4_results.txt"
        real_values_filename = "data/q2_4/q2_4_tree_mixture.pkl"
        num_clusters = 3
    elif scenario == "more samples":
        sample_filename = "data/q2_4/more_samples_samples.txt"
        output_filename = "q2_4_1_results.txt"
        real_values_filename = "data/q2_4/q2_4_tree_mixture.pkl"
        num_clusters = 3
    elif scenario == "ml":
        sample_filename = "data/q2_4/more_nodes_less_clusters.pkl_samples.txt"
        output_filename = "q2_4_2_results.txt"
        real_values_filename = "data/q2_4/more_nodes_less_clusters.pkl"
        num_clusters = 2
    elif scenario == "lm":
        sample_filename = "data/q2_4/less_nodes_more_clusters.pkl_samples.txt"
        output_filename = "q2_4_3_results.txt"
        real_values_filename = "data/q2_4/less_nodes_more_clusters.pkl"
        num_clusters = 5

    print("\n1. Load samples from txt file.\n")

    samples = np.loadtxt(sample_filename, delimiter="\t", dtype=np.int32)
    num_samples, num_nodes = samples.shape
    print("\tnum_samples: ", num_samples, "\tnum_nodes: ", num_nodes)
    print("\tSamples: \n", samples)

    print("\n2. Run EM Algorithm.\n")

    loglikelihood, topology_array, theta_array, pi = em_algorithm(seed_val, samples, num_cl

    print("\n3. Save, print and plot the results.\n")

    save_results(loglikelihood, topology_array, theta_array, output_filename)

    for i in range(num_clusters):
        print("\n\tCluster: ", i)
        print("Pi: ", pi[i])
        print("\tTopology: \t", topology_array[i])
        print("\tTheta: \t", theta_array[i])

    plt.figure(figsize=(8, 3))
    plt.subplot(121)
    plt.plot(np.exp(loglikelihood), label='Estimated')
    plt.ylabel("Likelihood of Mixture")
    plt.xlabel("Iterations")
    plt.subplot(122)
    plt.plot(loglikelihood, label='Estimated')
    plt.ylabel("Log-Likelihood of Mixture")
    plt.xlabel("Iterations")
    plt.legend(loc=(1.04, 0))
    plt.show()

    if real_values_filename != "":
        print("\n4. Retrieve real results and compare.\n")
        print("\tComparing the results with real values...")

        print("\t4.1. Make the Robinson-Foulds distance analysis.\n")
        rf_analysis(real_values_filename, output_filename, num_clusters)

        print("\n\t4.2. Make the likelihood comparison.\n")
        true_likelihood = true_log_likelihood(real_values_filename, sample_filename, num_cl

        data = [("EM", loglikelihood[-1], np.exp(loglikelihood[-1])),
                ("True", true_likelihood, np.exp(true_likelihood))]

        headers = ["","Loglikelihood","Likelihood"]

        print(tabulate(data, headers=headers))


# create_real_trees1()


# create_real_trees2()


# create_real_trees3()


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