DD2434 Machine Learning, Advanced Course - Assignment 2

Hannes Kindbom

2 Report

2.1 Knowing the rules

2.1.1 Question

I have read the instructions for the assignment thoroughly.

2.1.2 Question

This assignment was made completely individually without any collaborators and without discussing with anybody.

2.1.3 Question

See 2.1.2.

2.2 Dependencies in a Directed Graphical Model

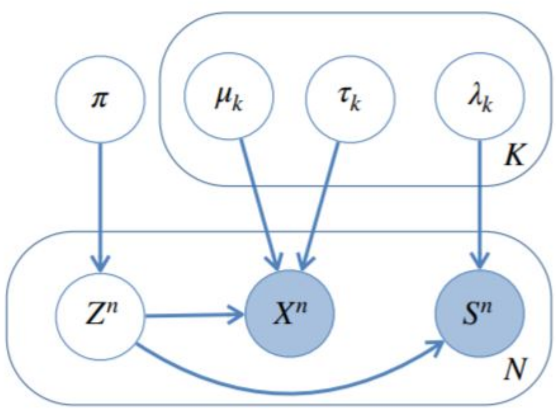

Figure 1: Mixture of components modeling location and strengths of earthquakes associated with a super-epicentra. In the figure, $\mu_k = (\mu_{k,1}, \mu_{k,2})$ and $\tau_k = (\tau_{k,1}, \tau_{k,2})$.

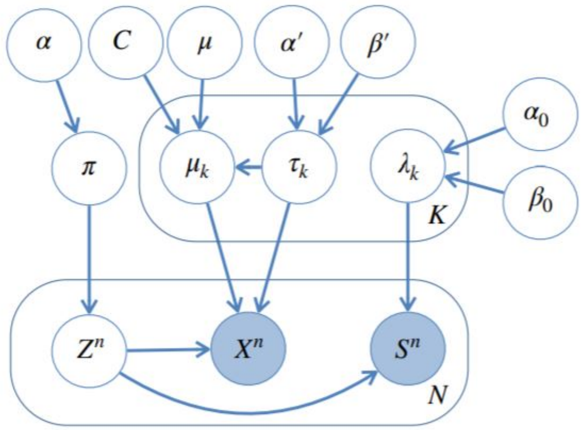

Figure 2: The K super epicentra model with priors.

2.2.4 Question

Yes, i.e. $\mu_k \perp \tau_k$, in figure 1, by d-separation. (see page 375 in Bishop)

2.2.5 Question

No, i.e. $\mu_k \not\perp \tau_k | X^1, ..., X^N$ in figure 1, by d-separation.

2.2.6 Question

Yes, i.e. $\mu \perp \beta'$ in figure 2, by d-separation.

2.2.7 Question

No, i.e. $\mu \not\perp \beta' | X^1, ..., X^N$ in figure 2, by d-separation.

2.2.8 Question

No, i.e. $X^n \not\perp S^n$ in figure 2, by d-separation.

2.2.9 Question

No, i.e. $X^n \not\perp S^n | \mu_k, \tau_k$ in figure 2, by d-separation.

2.3 Likelihood of a tree GM only for E level.

2.3.10 Question

An algorithm for calculating $p(\beta|T, \Theta)$ was implemented. Here, T refers to a rooted binary tree, $\Theta = \{\theta_0, ..., \theta_V\}$ refers to all the CPDs on the tree edges and β denotes a leaf assignment. In particular, $\theta_v = p(X_v|X_{parent(v)} = x_{parent(v)})$, where $parent(v)$ is the parent node of v. Each node is a random variable X_v, which takes one out of K categorical values.

The algorithm in brief is described as follows: Starting a recursion from the root and calculating $s(v,i) = p(x_{\downarrow v \cap \beta}|X_v = i)$, where $\downarrow v$ denotes all nodes below the root of sub-tree v. Furthermore, at leaf l:

$$s(l,i) = \begin{cases} 0 \text{ if } \mathrm{x}_l \neq i \\ 1 \text{ if } \mathrm{x}_l = i \end{cases} \tag{1}$$

and at any other node v:

$$s(v,i) = \Big(\sum_{j \in 1,\ldots,K} p(X_u = j|X_v = i)s(u,j)\Big)\Big(\sum_{j \in 1,\ldots,K} p(X_w = j|X_v = i)s(w,j)\Big) \tag{2}$$

and the final result (at root r) is given by (3):

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

The python code for this algorithm can be seen in appendix.

2.3.11 Question

The algorithm in section 2.3.10 was then applied to three different trees with varying size and the likelihood for each β sample was reported in the following table:

Tree: sample	**Likelihood**, $p(\beta\|T,\Theta)$
small: 0	0.00875
small: 1	0.03840
small: 2	0.00913
small: 3	0.02144
small: 4	0.01195
medium: 0	$4.623 * 10^{-18}$
medium: 1	$1.850 * 10^{-19}$
medium: 2	$3.805 * 10^{-20}$
medium: 3	$5.379 * 10^{-20}$
medium: 4	$4.308 * 10^{-19}$
large: 0	$2.033 * 10^{-74}$
large: 1	$1.111 * 10^{-76}$
large: 2	$3.265 * 10^{-75}$
large: 3	$5.610 * 10^{-75}$
large: 4	$9.857 * 10^{-77}$

2.4 Simple VI

2.4.12 Question

Synthetic data $x_1, ..., x_N$ was first sampled from a Gaussian distribution with mean μ and precision τ. The following Gaussian-gamma conjugate prior was thereafter introduced:

$$p(\mu|\tau) = N(\mu|\mu_0, (\lambda_0\tau)^{-1}) \tag{4}$$

$$p(\tau) = Gam(\tau|a_0, b_0) \tag{5}$$

These two were thereafter used to compute $q(\mu, \tau)$, a factorized variational approximation to the posterior distribution:[1]:

$$q(\mu, \tau) = q_\mu(\mu)q_\tau(\tau) \tag{6}$$

where $q_\mu(\mu)$ is Gaussian and $q_\tau(\tau)$ is Gamma distributed.[2] The Python code in the appendix shows the algorithm for computing $q(\mu, \tau)$ iteratively.

2.4.13 Question

In this case, the exact posterior distribution could be obtained as the following:

$$p(\mu, \tau|D) \propto N(\mu|\mu_N, (\lambda_N\tau)^{-1}) \times Gam(\tau|a_N, b_N) \tag{7}$$

I.e. $p(\mu, \tau|D)$ is Gaussian-gamma, where

$$\mu_N = \frac{\lambda_0\mu_0 + N\bar{x}}{\lambda_0 + N}, \quad \lambda_N = \lambda_0 + N, \quad a_N = a_0 + N/2, \text{ and}$$

$$b_N = b_0 + \frac{1}{2}\sum_{n=1}^{N}(x_n - \bar{x})^2 + \frac{\lambda_0 N(\bar{x} - \mu_0)^2}{2(\lambda_0 + N)}$$

2.4.14 Question

The iteratively computed $q(\mu, \tau)$ was compared to the exact posterior in (7) for three different cases. An initial guess of $E[\tau] = 1$ was used throughout all tests and the parameters for the prior were set to the following in all cases:

$$\lambda_0 = 1 \quad \mu_0 = 0 \quad a_0 = 1 \quad b_0 = 1$$

In the first case were $N = 300$ $\mu = 0.5$ and $\tau = 0.7$ and the exact posterior in this case is shown in figure 3. The exact posterior may be compared to the approximations in figure 4. It is easy to observe that the variance in both the approximations and the exact posterior is lower than in case two and three. One major explanation for this is that the number of samples N is larger in case one. Another observation is that the approximated posterior seem to converge faster than in the other two cases, due to reasonable inital guesses on for instance $E[\tau]$.

[1] C.M. Bishop. Pattern recognition and machine learning. 2nd ed. 2006. p.470
[2] C.M. Bishop. Pattern recognition and machine learning. 2nd ed. 2006. p.471

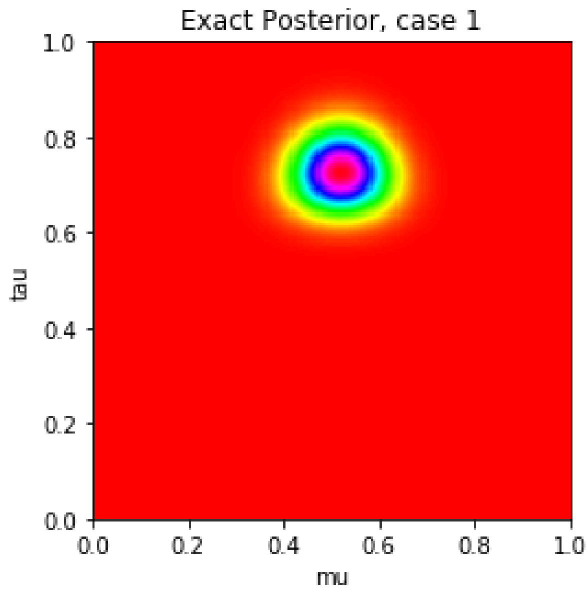

Figure 3: The exact posterior in case 1

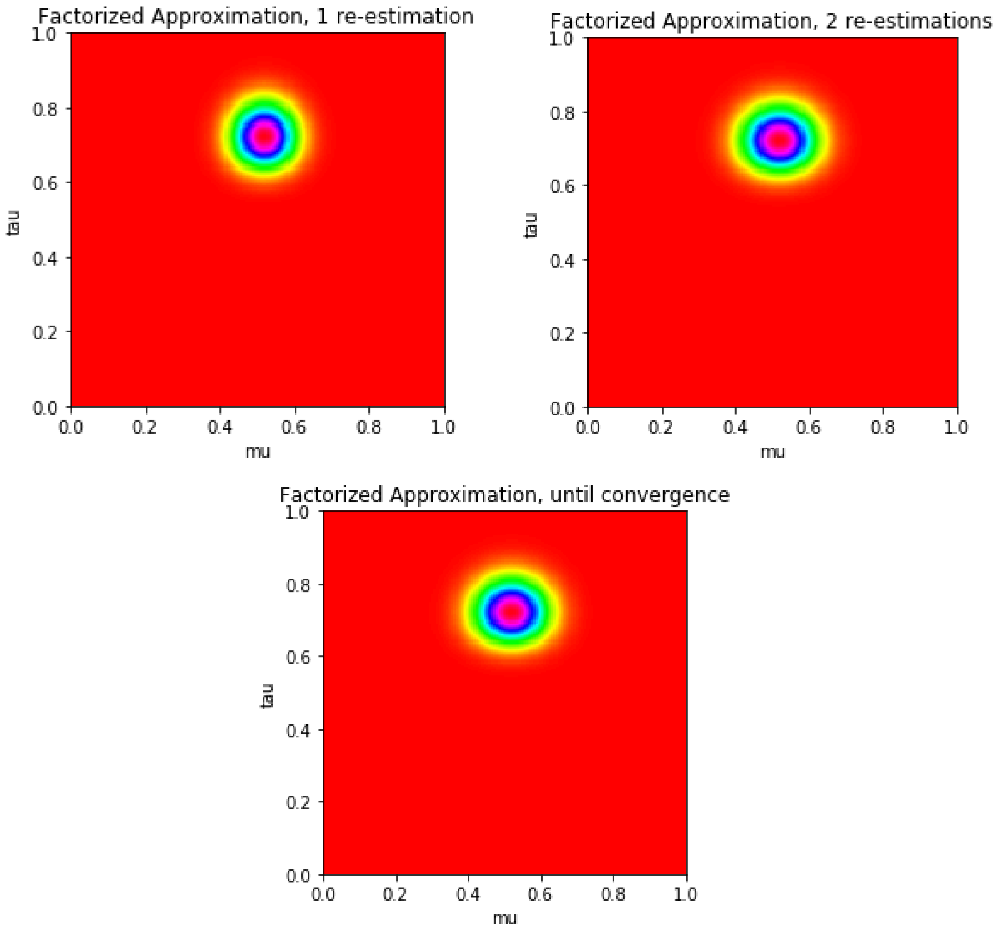

Figure 4: An illustration of the variational inference of τ and μ for 300 samples in case 1.

In the second case were $N = 100$ $\mu = 0.8$ and $\tau = 0.3$ and the exact posterior in this case is shown in figure 5. The exact posterior may be compared to the approximations in figure 6. The difference between the exact and approximated posterior after one re-estimation of parameters, is larger than in case one. This may be explained by the fact that the initial guesses were more off in this case.

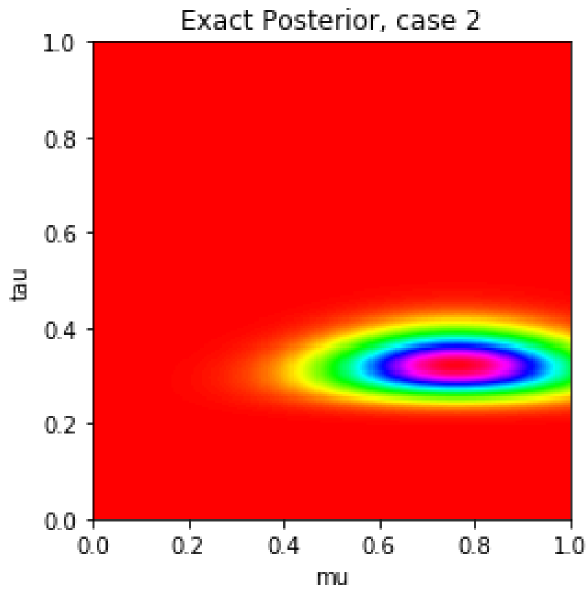

Figure 5: The exact posterior in case 2

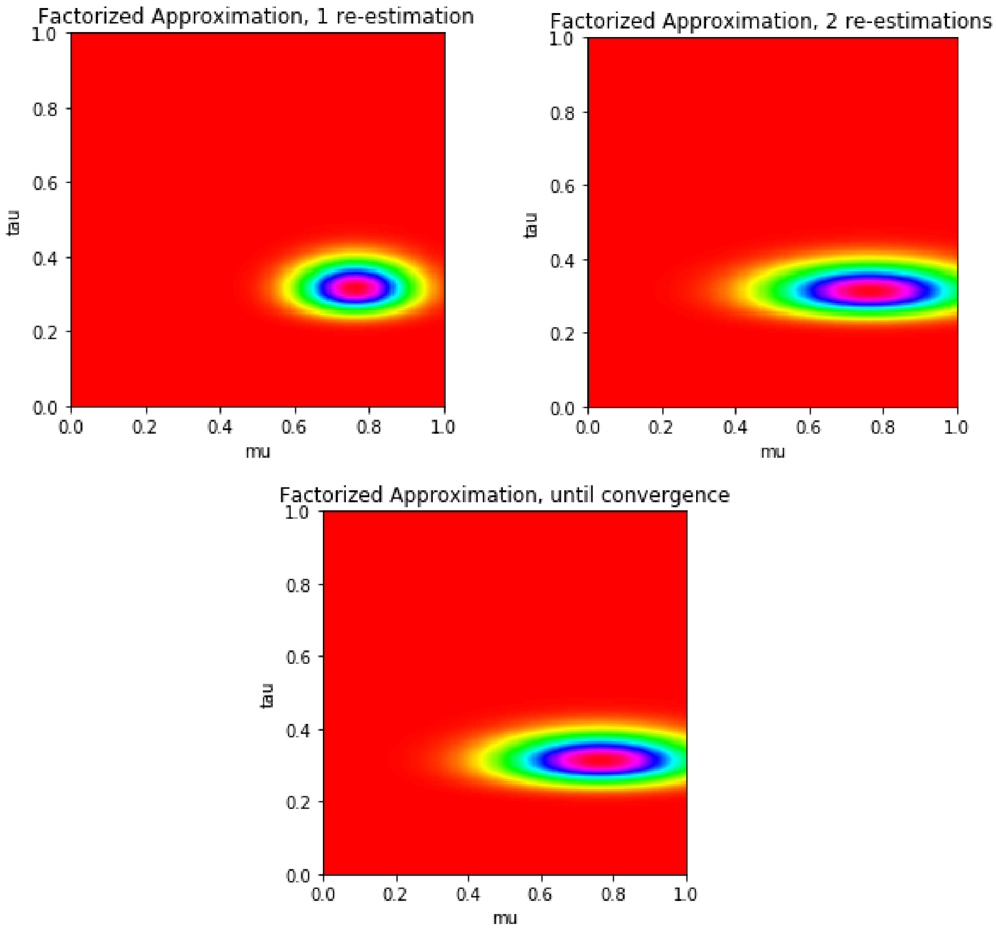

Figure 6: An illustration of the variational inference of τ and μ for 100 samples in case 2.

In the third case were $N = 30$ $\mu = 0.2$ and $\tau = 0.5$ and the exact posterior in this case is shown in figure 7. The exact posterior may be compared to the approximations in figure 8. The consequence of the smaller dataset is illustrated by the larger variance in the posterior. As in the other cases, the approximation gets closer after each iteration.

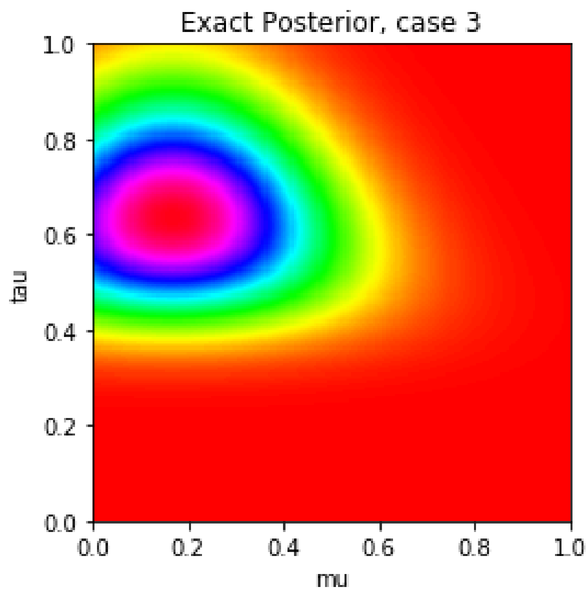

Figure 7: The exact posterior in case 3

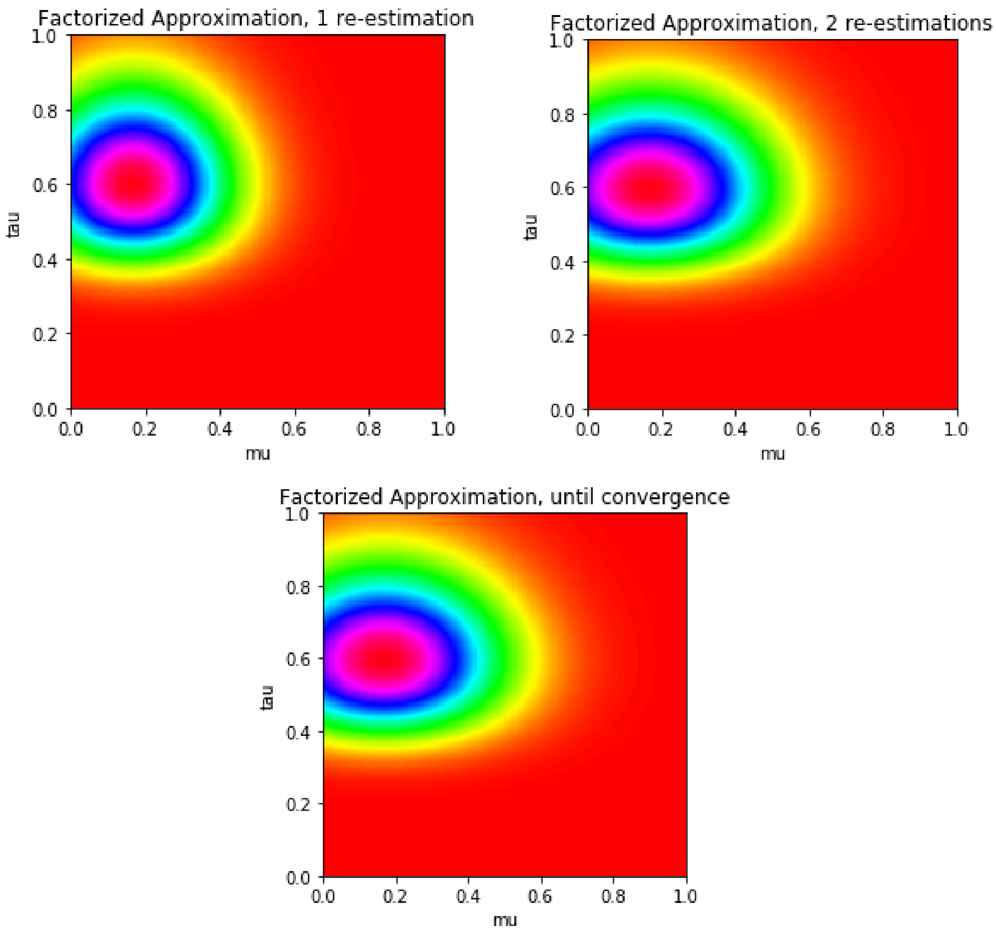

Figure 8: An illustration of the variational inference of τ and μ for 30 samples in case 3.

3 Appendix

Question 2.3 - Code

```python
#This file was built upon the the solution template for question 2.3 in DD2434 - Assignment 2.

import numpy as np
from Tree import Tree
import math

def calculate_s(theta, beta, tree, node):
    """
    This function calculates "smaller" part in the recursive formula (see report).
    :param: theta: CPD of the tree. Type: numpy array. Dimensions: (num_nodes, K)
    :param: beta: A list of node assignments. Type: numpy array. Dimensions: (num_nodes, )
                Note: Inner nodes are assigned to np.nan. The leaves have values in [K]
    :param: tree: The tree object
    :param: node: The current node to calculate for. Type: int.
    :return: s: The smaller value for the node. Type: numpy array. Dimensions: (K, )
    """
    nr_categories = theta[0].size
    s = np.zeros(nr_categories)

    # if leaf
    if not math.isnan(beta[node]):
        beta_value = beta[node].astype(int)
        s[beta_value] = 1
        return s

    # if node is parent
    nodes_children = tree.get_children_array_of(node)
    factors = []
    for child in np.nditer(nodes_children):
        #converting to suitable format
        child_CPD = get_matrix(theta[child])

        s_child = calculate_s(theta, beta, tree, child)
        factors.append(child_CPD.dot(s_child))
```

```python

    if len(factors) > 1:
        s = np.multiply(factors[0], factors[1])  # elementwise multiplication
    else:
        s = factors[0]
    return s

def calculate_likelihood(theta, beta, tree):
    """
    This function calculates the likelihood of a sample of leaves.
    :param: theta: CPD of the tree. Type: numpy array. Dimensions: (num_nodes, K)
    :param: beta: A list of node assignments. Type: numpy array. Dimensions: (num_nodes, )
                Note: Inner nodes are assigned to np.nan. The leaves have values in [K]
    :param: tree: The tree object
    :return: likelihood: The likelihood of beta. Type: float.
    """

    print("Calculating the likelihood...")
    s_root = calculate_s(theta, beta, tree, 0)

    likelihood = theta[0].dot(s_root)

    return likelihood


def get_matrix(mat):
    """
    This function converts a np.array of arrays to a format which we can calculate with.
    :param: mat: Type: numpy array. Dimensions: (K, K)

    :return: matrix: A non-nested matrix of "standard format". Type: numpy array. Dimensions: (K, K)
    """
    matrix = np.zeros((mat.size, mat.size))
    for index, array in enumerate(mat):
        matrix[index] = array
    return matrix
```

```
def test_prob_sum(tree):
    """
    This function calculates the sum of the probabilities for all possible node assignments (betas) and prints it.
    :param: tree: Type: object.
    """
    all_possible_betas = tree.generate_all_possible_betas()

    prob_sum = 0
    for beta_sample in all_possible_betas:
        sample_likelihood = calculate_likelihood(tree.get_theta_array(), beta_sample, tree)
        prob_sum += sample_likelihood

    print("Probability sum: ", prob_sum)

def generate_tree(filename):
    """
    This function generates a probabilistic binary tree, some filtered samples and then saves it.
    :param: filename: Type: string.
    """
    tree = Tree()
    tree.create_random_binary_tree(10, 6, 6)
    tree.sample_tree(2)
    tree.save_tree(filename)

def main():

    print("\n1. Load tree data from file and print it\n")

    filename = {"small_test":"data/q2_3_small_test_tree.pkl", "small":"data/q2_3_small_tree.pkl", "medium": "data/q2_3_medium_t
    tree = Tree()
    tree.load_tree(filename["small"])
    tree.print()

    print("tree topology: ", tree.get_topology_array())

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


    #Testing if probability of all possible beta values sums to 1
    test_prob_sum(tree)

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

if __name__ == "__main__":

    main()
```

```
    def generate_all_possible_betas(self):
        """ This function generates and returns all possible betas as a numpy array. Each row is one possible beta. """
        nr_cat = self.k
        sample_beta = self.filtered_samples[0]
        leaf_idxs = np.argwhere(np.isfinite(sample_beta)).flatten()
        nr_leaves = leaf_idxs.shape[0]

        cat_list = range(nr_cat)
        #generate all possible betas (cartesian product)
        all_possible_betas_raw = [p for p in itertools.product(cat_list, repeat=nr_leaves)]

        all_possible_betas = np.zeros((len(all_possible_betas_raw), sample_beta.shape[0]))
        all_possible_betas[:] = np.nan
        for possible_beta_idx, possible_beta in enumerate(all_possible_betas_raw):
            for index, beta_value in enumerate(possible_beta):
                all_possible_betas[possible_beta_idx, leaf_idxs[index]] = beta_value

        return all_possible_betas
```

```
    def get_children_array_of(self, parent_id):
        """ This function returns the children of a given perent as a numpy array. """

        topology_array = self.get_topology_array()
        children_array = np.where(topology_array == parent_id)

        return children_array[0]
```

Question 2.4 - Code

Generate synthetic dataset

```
np.random.seed(1338)
N = 30 # number of samples
mu = 0.2
tau = 0.5 #precision

X_range = np.linspace(0, 1.0, num=N)
X = np.random.normal(mu, 1/tau**0.5, N)
X_mean = np.mean(X)
```

```python
#Assuming prior P(mu|tau) to be N(mu_0, (lambda_0*tau)^-1)
# and prior p(tau) to be Gamma(a_0, b_0)

#help functions
def get_mu_N(X_mean, N):
    mu_N = (lambda_0*mu_0 + N*X_mean)/(lambda_0 + N)
    return mu_N

def get_lambda_N(E_tau, N):
    lambda_N = (lambda_0 + N)*E_tau
    return lambda_N

def get_a_N(N):
    a_N = a_0 + N/2
    return a_N

def get_b_N(X, E_mu, Var_mu):
    b_N = b_0 + 0.5*get_E_mu_expression(X, E_mu, Var_mu)
    return b_N

def get_E_mu_expression(X, E_mu, Var_mu):
    E_mu_2 = Var_mu + E_mu**2
    term1 = np.power(X, 2) - 2*X*E_mu + E_mu_2
    summed = np.sum(term1)
    term2 = lambda_0*E_mu_2 - 2*E_mu*mu_0 + mu_0**2
    E_mu_expression = summed + term2
    return E_mu_expression
```

Iteratively compute variational distribution

```python
#Prior parameters
lambda_0 = 1
mu_0 = 0
a_0 = 1
b_0 = 1
```

```python
def re_estimate_param(iterations, E_tau_guess):

    #initial guess
    E_tau = E_tau_guess

    for i in range(iterations):
        mu_N = get_mu_N(X_mean, N)
        lambda_N = get_lambda_N(E_tau, N)

        a_N = get_a_N(N)
        b_N = get_b_N(X, mu_N, 1/lambda_N)

        #re-estimate
        E_tau = a_N/b_N

    return lambda_N, mu_N, a_N, b_N
```

Visualizing the result

```
pixels = 200
#q_mu is gaussian and q_tau is gamma(a,b)
q_mu_prec, q_mu_mean, q_tau_a, q_tau_b = re_estimate_param(iterations = 10, E_tau_guess = 1)

mu_range = np.linspace(0, 1.0, num=pixels)
tau_range = np.linspace(0, 1.0, num=pixels)

X, Y = np.meshgrid(tau_range, mu_range)
N, M = len(X), len(Y)
Z = np.zeros((N, M))
for i,(x,y) in enumerate(product(tau_range,mu_range)):
    pos = np.hstack((x, y))
    tau = pos[0]
    mu = pos[1]
    Z[np.unravel_index(i, (N,M))] =  norm(q_mu_mean, 1/q_mu_prec**0.5).pdf(mu)*gamma.pdf(tau, q_tau_a, scale=1/q_ta

im = plt.imshow(Z,cmap='hsv', origin='lower', extent=(0, 1, 0, 1)) #extent = (left, right, bottom, top)
ax = plt.gca()
ax.grid(False)
plt.title("Factorized Approximation, until convergence")
plt.xlabel('mu')
plt.ylabel('tau')
plt.show()
```

Comparing with exact posterior

```
post_mu = (lambda_0*mu_0 + N*X_mean)/(lambda_0 + N)
post_lambda = lambda_0 + N
post_a = a_0 + N/2
post_b = b_0 + 0.5*np.sum(np.power(X - X_mean, 2)) + (lambda_0*N*(X_mean - mu_0)**2)/(2*(lambda_0 + N))

post_mu_range = np.linspace(0, 1.0, num=pixels)
post_tau_range = np.linspace(0, 1.0, num=pixels)

X_post, Y_post = np.meshgrid(post_tau_range, post_mu_range)
post_N, post_M = len(X_post), len(Y_post)
Z_post = np.zeros((post_N, post_M))
for i,(x,y) in enumerate(product(post_tau_range,post_mu_range)):
    pos = np.hstack((x, y))
    tau = pos[0]
    mu = pos[1]
    tau_sample = gamma.pdf(tau, post_a, scale=1/post_b)
    Z_post[np.unravel_index(i, (post_N,post_M))] =  norm(post_mu, 1/(post_lambda*tau)**0.5).pdf(mu)*tau_sample

im = plt.imshow(Z_post,cmap='hsv', origin='lower', extent=(0, 1, 0, 1))
ax = plt.gca()
ax.grid(False)
plt.title("Exact Posterior, case 3")
plt.xlabel('mu')
plt.ylabel('tau')
plt.show()
```