DD2434 Machine Learning, Advanced Course
Assignment 1

Povel Forsare Källman

Dependencies in a Directed Graphical Model

Question 2.1.1

Yes

Question 2.1.2 I have discussed formulations of problem 2.3 and 2.4 with the members of my project group.

Question 2.1.3

No

Question 2.2.4

$$\mu_k \perp\!\!\!\perp \tau_k \tag{1}$$

Yes

Question 2.2.5

$$\mu_k \perp\!\!\!\perp \tau_k | X^1, \dots X^N \tag{2}$$

No

Question 2.2.6

$$\mu \perp\!\!\!\perp \beta' \tag{3}$$

Yes

Question 2.2.7

$$\mu_{\perp}\perp \beta' | X^1, \dots X^N \tag{4}$$

No

Question 2.2.8

$$X^n \perp\!\!\!\perp S^n \tag{5}$$

No

Question 2.2.9

$$X^n \perp\!\!\!\perp S^n | \mu_k, \tau_k \tag{6}$$

No

Question 2.3.10 The algorithm was implemented using Python and the code can be found in Appendix A.

Question 2.3.11 The graphical models in this task are variations of a binary tree. The structure and the size of the tree varies between the 3 given data sets and each data set consists of 5 samples each. Furthermore, each vertex can assume any discrete value in the range of 0 to 4. Additionally for this task, the entire set of leaves was observed, while every other vertex was unknown. The computational likelihoods for the different models are represented in table 1.

Table 1: Describes the likelihood with the samples in the left column and the trees in the first row.

Sample \Tree	Small tree	Medium tree	Large tree
Sample 0	0.008753221441670067	8.66416414170832e-17	1.2296785012112113e-65
Sample 1	0.0383969250979291	5.394284454090606e-18	1.4347770777980813e-63
Sample 2	0.009129106859990061	8.892415333536359e-18	3.095491016149858e-66
Sample 3	0.0214406975419561	1.1222302136292958e-18	3.4231977224272667e-69
Sample 4	0.011945567814215127	7.58934157249135e-19	4.822393947666207e-67

Question 2.4.12 The VI algorithm was implemented using Python and the code can be found in appendix B.

Question 2.4.13 The exact posterior can be found in the article *Conjugate Bayesian analysis of the Gaussian distribution* by Kevin Murphy.

$$p(\mu, \lambda|D) = \mathcal{NG}(\mu, \lambda|\mu_{n,n}, \alpha_n, \beta_n) \tag{7}$$

$$\mu_n = \frac{k_0\mu_0 + n\bar{x}}{k_0 + n} \tag{8}$$

$$k_n = k_0 + n \tag{9}$$

$$\alpha_n = \alpha_0 + n/2 \tag{10}$$

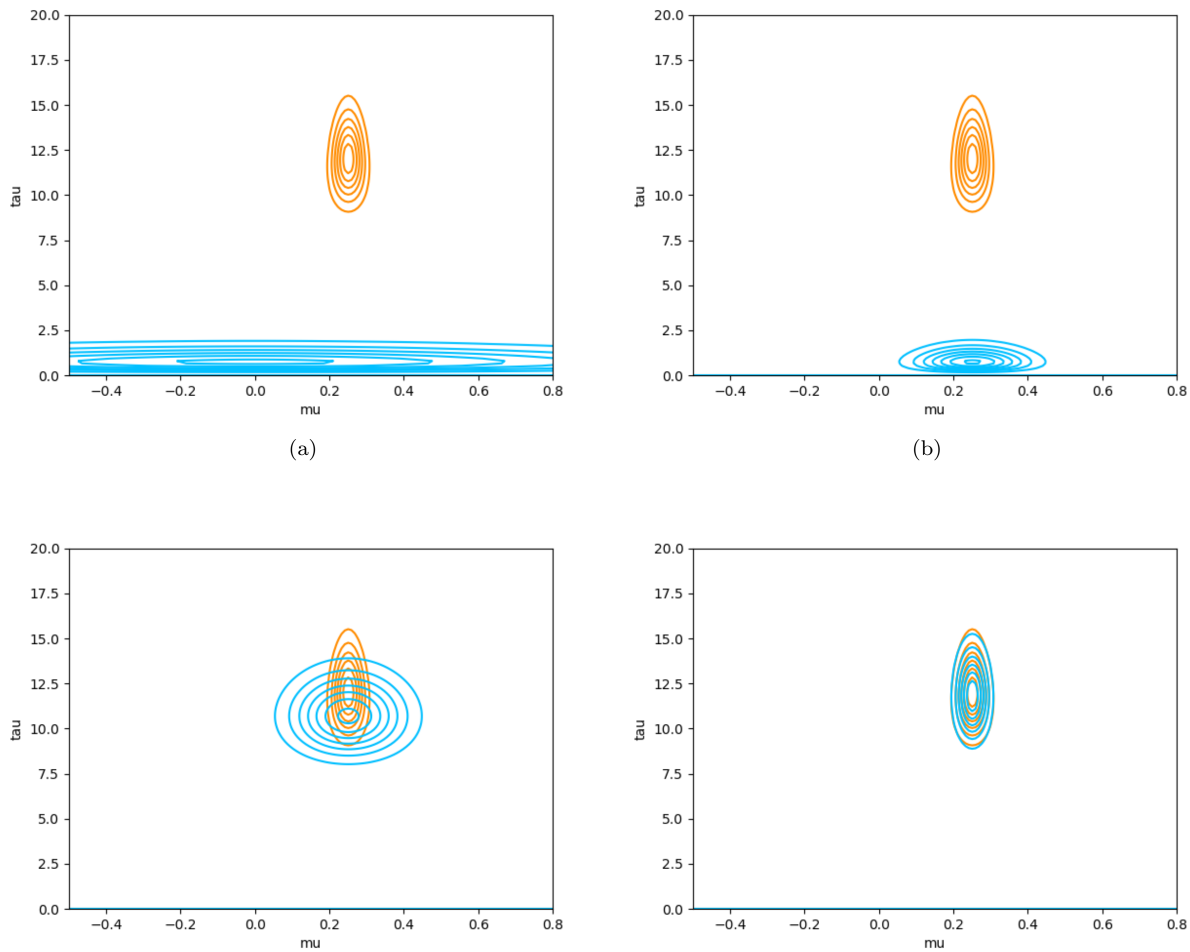
$$\beta_n = \beta_0 + \frac{1}{2}\sum_{i=1}^{n}(x_i - \bar{x})^2 + \frac{k_0 n(\bar{x} - \mu_0)^2}{2(k_0 + n)} \tag{11}$$

Question 2.4.14 For the first case, 100 data points were sampled from a univariate Gaussian distribution, while the second case only sampled 5 data points. However, the same initialization values were used for both cases. The first case is shown in figure 1 and the second case is shown in figure 2. When increasing the amount of sampled data points N, we can see that both posteriors will converge towards the maximum-likelihood solution, which in our case is a Gaussian both for the estimated and the exact posterior. Therefore, we can see that the contours of the plot in case 1 takes a rounder shape, *closer* to a Gaussian distribution and more compact than for the second case. Furthermore, we can see that the expected posterior managed to converge for both cases, even though the second case only used 5 samples.

For the third case, the variance for the data points x was increased, while the rest of the initial parameters were equal to the first case. From the plots we can see that this resulted in decreased variance for τ. We can further derive this result with the help of (12) and (13). From (12) it becomes clear that increasing the variance for x will result in a larger b_N, which in return will result in a smaller variance for τ from (13).

$$b_N = b_0 + \frac{1}{2}\,\mathbb{E}_\mu \left[\sum_{n=1}^{N} (x_n - \mu)^2 + \lambda_0(\mu - \mu_0)^2 \right] \tag{12}$$

$$var[\tau] = \frac{a_N}{b_N^2} \tag{13}$$



(a) (b)

(c) (d)

Figure 1: Contours for the estimated posterior are shown in blue, while the contours of the true posterior is shown in orange. (a) Shows the initial factorized approximation $q_\mu(\mu)q_\tau(\tau)$. (b) Shows the factorized approximation $q_\mu(\mu)q_\tau(\tau)$ after re-estimating $q_\mu(\mu)$. (c) Shows the factorized approximation $q_\mu(\mu)q_\tau(\tau)$ after re-estimating $q_\tau(\tau)$. (d) Shows an optimal factorized approximation.

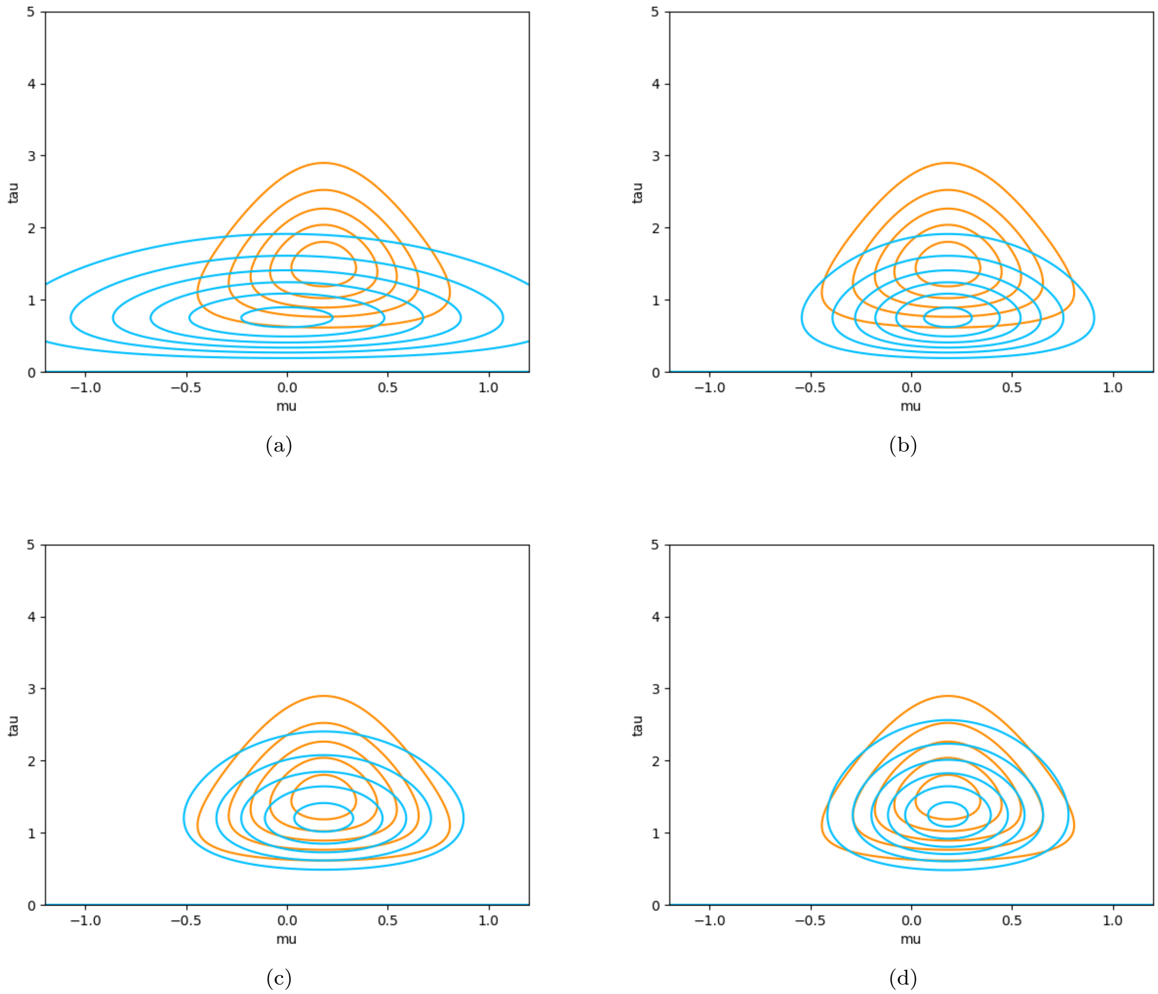


Figure 2: Contours for the estimated posterior are shown in blue, while the contours of the true posterior is shown in orange. (a) Shows the initial factorized approximation $q_\mu(\mu)q_\tau(\tau)$. (b) Shows the factorized approximation $q_\mu(\mu)q_\tau(\tau)$ after re-estimating $q_\mu(\mu)$. (c) Shows the factorized approximation $q_\mu(\mu)q_\tau(\tau)$ after re-estimating $q_\tau(\tau)$. (d) Shows an optimal factorized approximation.

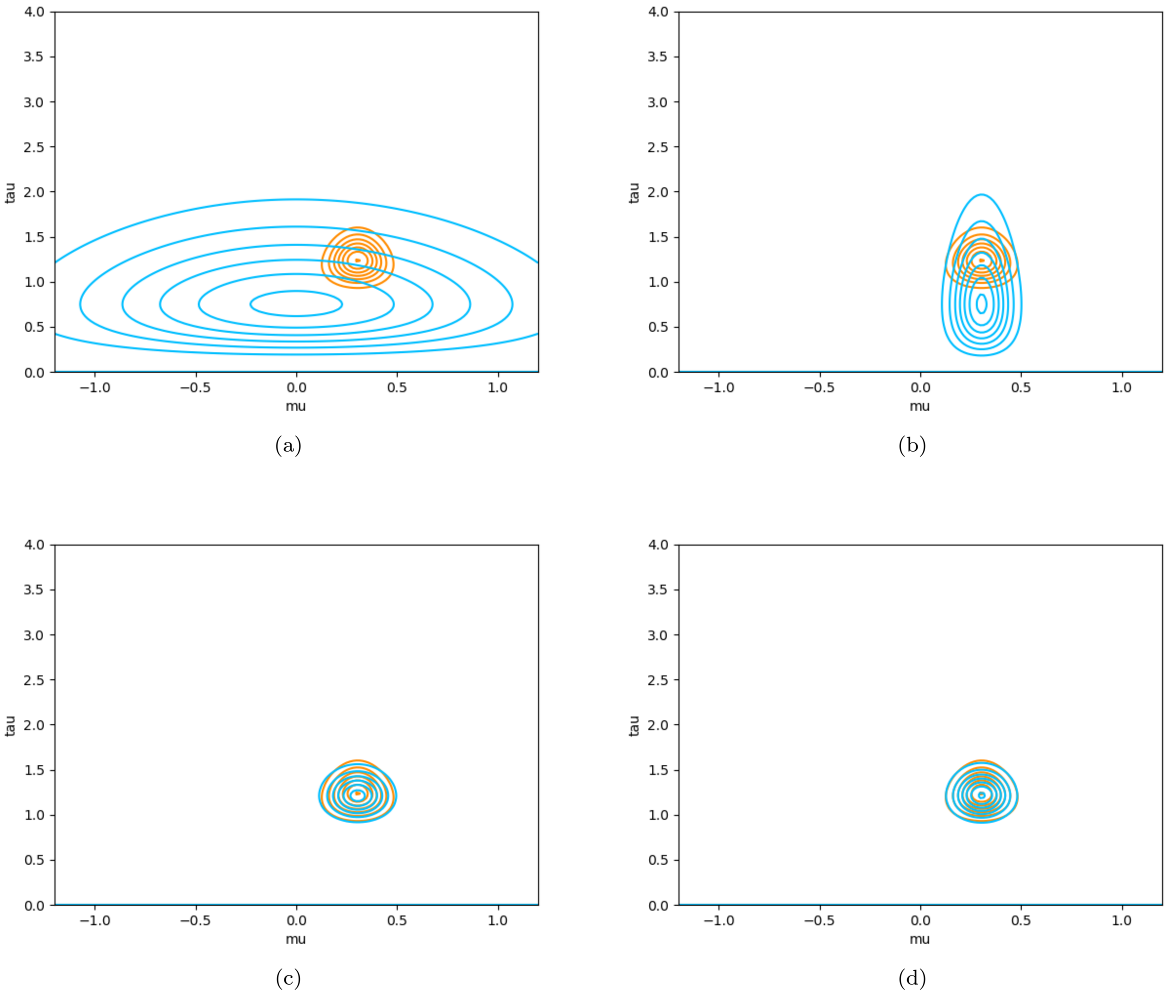


Figure 3: Contours for the estimated posterior are shown in blue, while the contours of the true posterior is shown in orange. (a) Shows the initial factorized approximation $q_\mu(\mu)q_\tau(\tau)$. (b) Shows the factorized approximation $q_\mu(\mu)q_\tau(\tau)$ after re-estimating $q_\mu(\mu)$. (c) Shows the factorized approximation $q_\mu(\mu)q_\tau(\tau)$ after re-estimating $q_\tau(\tau)$. (d) Shows an optimal factorized approximation.

References

[1] Kaare Brandt Petersen and Michael Syskind Pedersen. (2012). *The Matrix Cookbook*

[2] Michael E. Tipping and Christopher M. Bishop. (2002). *Probabilistic Principal Component Analysis*

[3] Christopher M. Bishop. (2006). *Pattern Recognition and Machine Learning*

[4] Kevin Murphy. (2007). Conjugate Bayesian analysis of the Gaussian distribution.

Appendices

Appendix A

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


def compute_subprob(node, i, theta, beta, k, memory_cache):
    if str(int(node.name))+str(i)+str(beta[int(node.name)]) in memory_cache:
        return memory_cache[str(int(node.name))+str(i)+str(beta[int(node.name)])]

    if len(node.descendants) == 0:
        if beta[int(node.name)] == i:
            value = 1
        else:
            value = 0

    else:
        left_child = node.descendants[0]
        right_child = node.descendants[1]
        left=0
        right=0

        for j in range(k):
            left += theta[int(left_child.name)][i][j] * compute_subprob(left_child, j, theta, beta, k, memory_cache)
            right += theta[int(right_child.name)][i][j] * compute_subprob(right_child, j, theta, beta, k, memory_cache)
        value = left * right
    memory_cache[str(int(node.name))+str(i)+str(beta[int(node.name)])] = value

    return value


def calculate_likelihood(tree_topology, theta, beta):
    memory_cache={}

    t = Tree()
    t.load_tree_from_direct_arrays(tree_topology, theta)

    k = len(theta[0])
    likelihood = 0
    for i in range(k):
        likelihood += compute_subprob(t.root, i, theta, beta, k, memory_cache) * theta[0][i]
        memory_cache.clear()

    return likelihood


def main():
    print("Hello World!")
    print("This file is the solution template for question 2.3.")

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

    # filename = "data/q2_3_small_tree.pkl"
    # filename = "data/q2_3_medium_tree.pkl"
    filename = "data/q2_3_large_tree.pkl"


    t = Tree()
    t.load_tree(filename)


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

```
    # These filtered samples already available in the tree object.
    # Alternatively, if you want, you can load them from corresponding .txt or .npy files

    for sample_idx in range(t.num_samples):
        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()
```

Appendix B

```python
import numpy as np
from numpy import sqrt,pi,vectorize,exp
from scipy.special import gamma
from scipy import stats
import math
import matplotlib.pyplot as plt


# Samples from distributions
def gammaGaussian(mu,tau):
    term1 = np.power(true_b,true_a)*sqrt(true_lambda)
    term2 = gamma(true_a)*sqrt(2*pi)
    term3 = np.power(tau,(true_a-0.5))*np.exp(-true_b*tau)
    term4 = np.exp(-0.5*true_lambda*tau*np.power(mu-true_mu,2))
    return (term1/term2)*term3*term4


# plot function
def plotFunction(mu_N, lambda_N, a_N, b_N, k, s):
    mu = np.linspace(-1.2,1.2,150)
    tau = np.linspace(0,4,150)
    MU, TAU = np.meshgrid(mu,tau, indexing="ij")

    q_mu = stats.norm(mu_N, np.sqrt(1/lambda_N))
    q_tau = stats.gamma(a_N, loc=0, scale=1/b_N)

    T = np.zeros_like(MU)
    Z = np.zeros_like(MU)
    for i in range(Z.shape[0]):
        for j in range(Z.shape[1]):
            Z[i][j] = q_mu.pdf(mu[i]) * q_tau.pdf(tau[j])
            T[i][j] = gammaGaussian(mu[i],tau[j])

    plt.plot()
    plt.xlabel('mu')
    plt.ylabel('tau')
    plt.contour(MU,TAU,T,colors='darkorange')
    plt.contour(MU,TAU,Z, colors='deepskyblue')
    plt.savefig('case3_' + str(k) + s + '_Q2411.png')
    plt.clf()
    #plt.show()



# computing true parameters
def computeTrueParameters():
    true_mu = (lambda_0*mu_0 + N*E_mu)/(lambda_0 + N)
    true_lambda = lambda_0 + N
    true_a = a_0 + N/2
    true_b = b_0 + 0.5*np.sum(np.square(x-np.mean(x))) + (lambda_0*N*np.square(np.mean(x)-
    mu_0)) / (2*(lambda_0+N))
    return true_mu, true_lambda, true_a, true_b



# computing parameters
def compute_mu_N():
    return (lambda_0*mu_0 + np.sum(x))/(lambda_0+N)

def compute_lambda_N(E_tau):
    return (lambda_0 + N)*E_tau

def compute_a_N():
    return a_0 + (N/2)

def compute_b_N(E_mu, E_mu2):
```

```
    return b_0 + 0.5*lambda_0 * (E_mu2 + mu_0**2 - 2*mu_N*mu_0) + 0.5* np.sum(x**2 + E_mu2 -
     2*mu_N*x)



# global variables
mu = 0.25
sigma = 0.1
N = 100

np.random.seed(1)
x = np.random.normal(mu, sigma, N)
E_mu = np.mean(x)

# initial guess for E[tau] + initilizations
E_tau=1
lambda_0=2
mu_0=0
a_0=4
b_0=4


# initalization for plots, just so we can plot before updating q_mu and q_tau
a_N=a_0
b_N=b_0
mu_N=mu_0
lambda_N=lambda_0

# Computing true parameters
true_mu, true_lambda, true_a, true_b = computeTrueParameters()
print('true_mu ', true_mu)
print('true_lambda', true_lambda)
print('true_a', true_a)
print('true_b', true_b)
print('\n')



# iterations
maxIter = 5
for i in range(maxIter):

    plotFunction(mu_N, lambda_N, a_N, b_N, i,'first')       # plot before updating q_mu
    mu_N = compute_mu_N()
    lambda_N = compute_lambda_N(E_tau)

    E_mu = mu_N
    E_mu2 = 1./lambda_N + mu_N**2

    plotFunction(mu_N, lambda_N, a_N, b_N, i,'second')      # plot before updating q_tau
    a_N = compute_a_N()
    b_N = compute_b_N(E_mu, E_mu2)

    E_tau = a_N/b_N

#plotFunction(mu_N, lambda_N, a_N, b_N, i,'first')
print('iterative mean', mu_N)
print('iterative lambda', lambda_N)
print('iterative a_N', a_N)
print('iterative b_N', b_N)
```