Advanced Machine Learning, Assignment 2

Youssef Taoudi, yousseft@kth.se

December 31, 2019

2.1 Knowing The Rules

Question 1

Yes.

Question 2

No collaborations (aside from discussions in the slack group)

Question 3

No.

2.2 Dependencies in a DGM

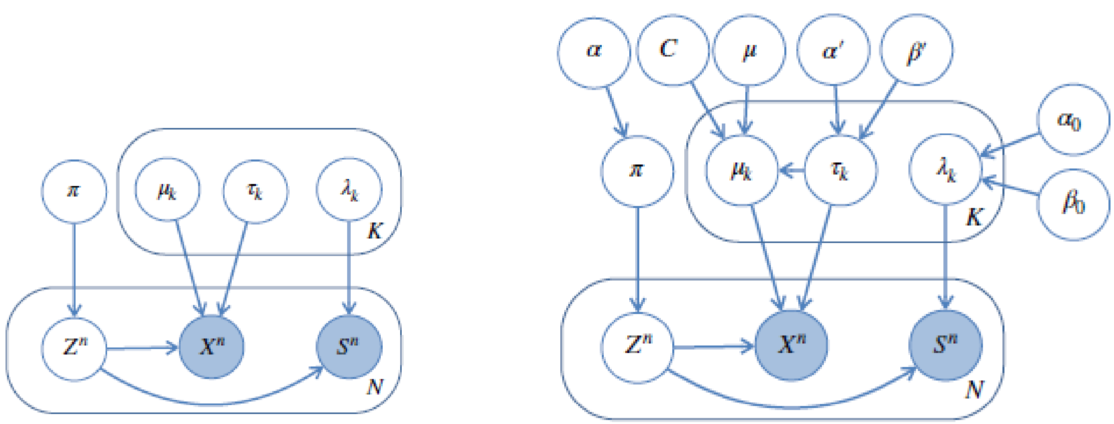

Figure 1: Graphical Models of Figure 1 and Figure 2 from the assignment

Question 4

Yes

Question 5

No

Question 6

Yes

Question 7

No

Question 8

No

Question 9

No

2.3 Tree GM

Question 10

The probability $p(\beta|T, \Theta)$ is the product of the probability of the observed value of each leaf given the observed tree (because all leaves are conditionally independent on the observations).

$$p(\beta|T, \Theta) = \prod_{L} p(X_L = V_L|X_o) \tag{1}$$

Where X_L is a leaf node, V_L is the corresponding value to the node and X_o are all observations in the tree.

$$p(X_L = V_L|X_o) = p(X_L = V_L|X_{o\cap\uparrow L})p(X_L = V_i|X_{o\cap\downarrow L}) \tag{2}$$

In this case $X_{o\cap\downarrow L}$ are all nodes below X_L and $X_{o\cap\uparrow L}$ is the rest. The expression can be rewritten to:

$$p(X_L = V_i|X_o) = s(X_L, V_L)t(X_L, V_L) \tag{3}$$

Where both $s(X_L, V_L)$ and $t(X_L, V_l)$ are subproblems (recursive) that rely on computations in the rest of the tree. $s(u, i)$ takes on different forms depending on if the input node is a root 4 or leaf 5 otherwise it takes the default form 6.

$$s(X_{root}, i) = \sum_{j}^{K} p(X_{root} = i) s(X_{root}, j) \tag{4}$$

$$s(X_L, i) = \begin{cases} 1, & \text{if } V_L = i \\ 0, & \text{otherwise} \end{cases} \tag{5}$$

$$s(X_u, i) = \sum_{j}^{K} p(X_u = i | X_{parent} = j) s(X_{parent}, j) \tag{6}$$

In the case of $t(X_u, i)$, the sibling also has to be taken into account 7.

$$t(X_u, i) = \sum_{j,k}^{K} t(X_{parent}, j) p(X_u = i | X_{parent} = j) p(X_{sibling} = k | X_{parent} = j) s(X_{sibling}, k) \tag{7}$$

Once a given $t(X_u, i)$ or $s(X_u, i)$ has been calculated, it is stored in a directory and can be accessed in constant time if needed without passing through the tree again. Code for the algorithm is included in the appendix A.

Question 11

	Small Tree	**Medium Tree**	**Large Tree**
Sample 1	9.465633985205277e-05	3.0244137624969225e-35	4.148545314103789e-148
Sample 2	0.0017899881547490\u200995	4.0110053722151726e-38	1.2469081008275893e-152
Sample 3	0.00010151856808122183	1.68167184447056e-39	1.0705494297938794e-149
Sample 4	0.00058370320850972	3.9267176420442574e-39	3.180711099819987e-149
Sample 5	0.00017156713522188422	2.3658967438072998e-37	9.768671917063787e-153

2.4 Simple VI

Question 12

Following the derivations from Bishop [1, pp. 470–471] and Murphy[2, pp. 742–744] by inferring two factors $q(\tau)$ and $q(\mu)$ two derive the posterior. The factors are derived by inferring four hyperparameters α_N, β_N, μ_N and λ_N iteratively from equations 8 - 9. Note that since β_N is needed to calculate λ_N and vice versa, the first β_N that is used is a guessed value. Note that $\alpha_0 = \beta_0 = \lambda_0 = \mu_0 = 0$ (Initial guess). An example of how the guessed posterior converges over several iterations can be seen in figure 2 and code can be found in Appendix B.

$$\mu_N = \frac{\lambda_0 \mu_0 + N\bar{x}}{\lambda_0 + N}, \lambda_N = (\lambda_0 + N) \frac{\alpha_N}{\beta_N} \tag{8}$$

$$
\begin{aligned}
&\alpha_N = \alpha_0 + \frac{N+1}{2}, \\
&\beta_N = \beta_0 + \lambda_0 \left(\lambda_N + \mu_N^2 + \mu_0^2 - 2\mu_N\mu_0\right) + \tfrac{1}{2}\textstyle\sum_{i=1}^{N} \left(x_i^2 + \lambda_N + \mu_N^2 - 2\mu_N x_i\right)
\end{aligned}
\tag{9}
$$

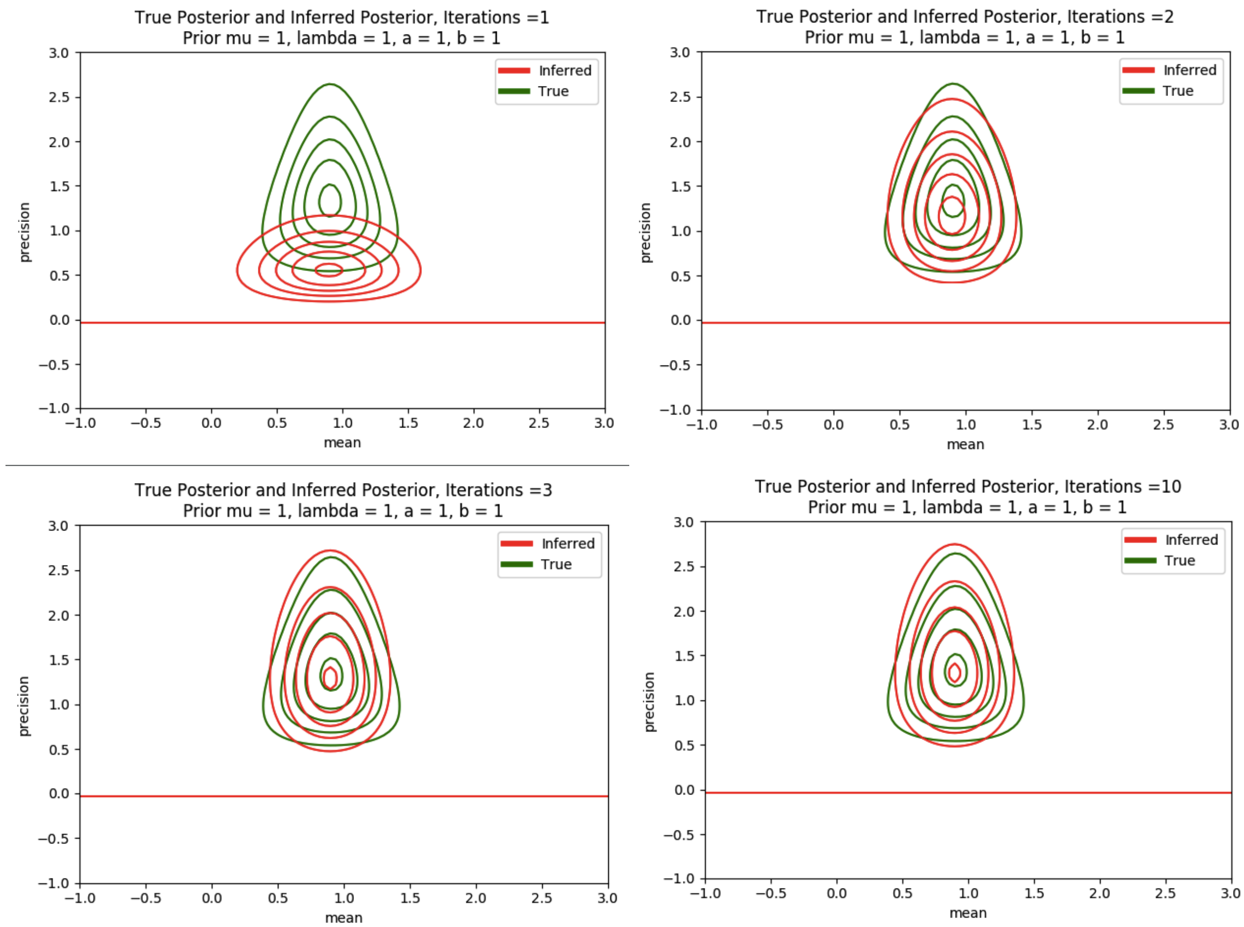

Figure 2: Plot showcasing how the inferred posterior converges after multiple iterations.

Question 13

The exact posterior, $P(\mu, \tau|D) = P(\mu|\tau)P(\tau)P(D|\mu, \tau)$. $P(\mu|\tau)$ is Gaussian while $P(\tau)$ is Gamma distributed, meaning the conjugate priors will result in a Normal-Gamma distribution which is normalized by the likelihood $P(\mu, \tau|D)$ to form the exact posterior. [2, p. 742]. Examples of exact posteriors can be showcased in figures 2 and 3 where the exact posterior are drawn in green. Note, $p(\mu|\tau) \sim \mathcal{N}(\mu_0, (\lambda\tau)^{-1})$ and $p(\tau) \sim Gamma(\alpha, \beta)$.

Question 14

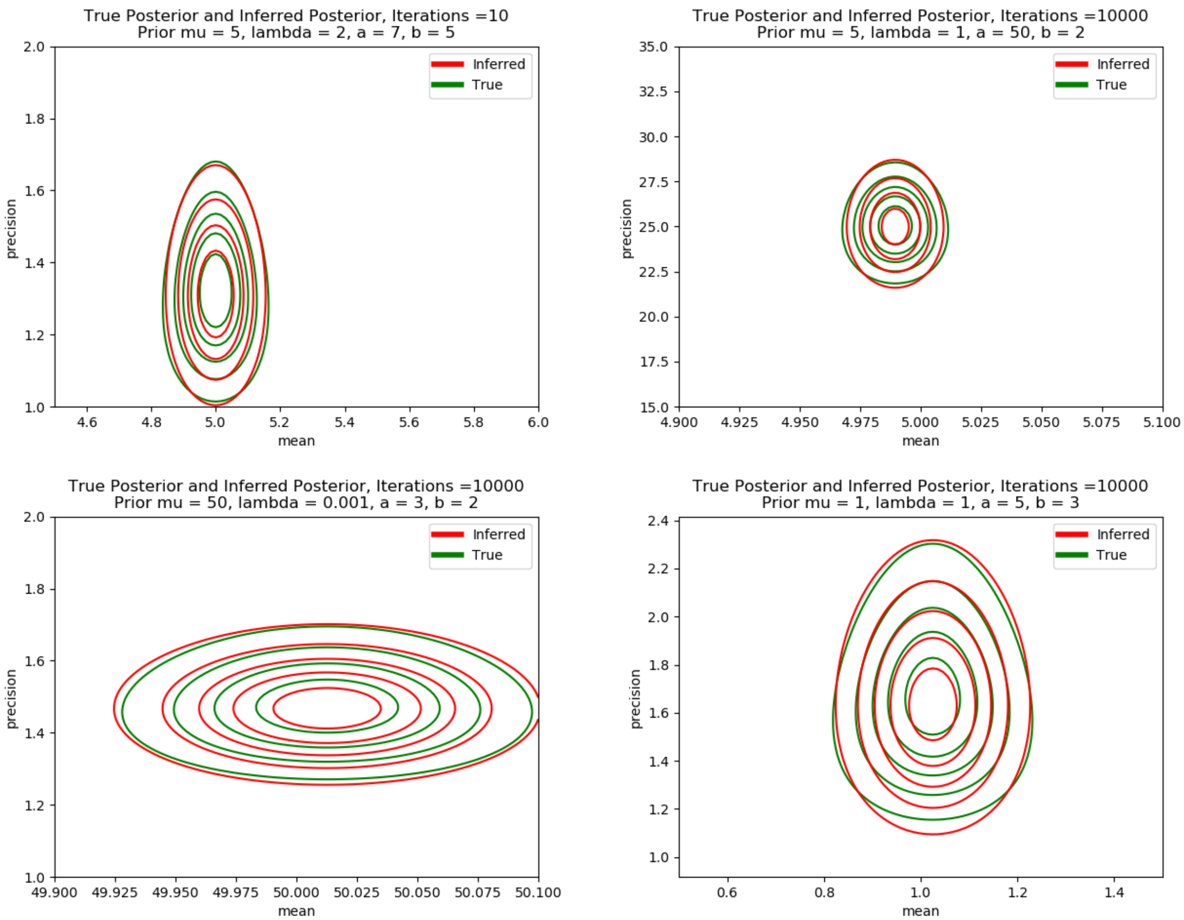

Figure 3: Plots of the true posterior compared to the inferred posterior for 4 different cases with different values on the hyperparameters.

As seen in figure 3 the converged posterior takes different shapes depending on hyperparameters but it seems to match the exact posterior well most of the time.

Appendices

A: TreeGM Algorithm

```
# Calculating s values for the tree and storing them
def s_root(tree_topology, theta, beta):
    prob = 0
    s_dict = defaultdict(dict)

    def S(u, j, children):
        if s_dict[u].get(j) is not None:
            return s_dict[u][j]
        if len(children) < 1:
            if beta.astype(int)[u] == j:
                s_dict[u][j] = 1
                return 1
            else:
                s_dict[u][j] = 0
                return 0
        result = np.zeros(len(children))
        for child_nr, child in enumerate(children):
            for category in range(0, len(theta[0])):
                result[child_nr] += S(child, category, find_children(child,
                    tree_topology, beta)) * CPD(theta, child,
                                                                                  c



        s_result = np.prod(result)
        s_dict[u][j] = s_result
        return s_result

    for i, th in enumerate(theta[0]):
        prob += S(0, i, find_children(0, tree_topology, beta)) * CPD(theta, 0, i)
    return s_dict

def calculate_likelihood(tree_topology, theta, beta):
    """
    This function calculates the likelihood of a sample of leaves.
    :param: tree_topology: A tree topology. Type: numpy array. Dimensions:
        (num_nodes, )
    :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]
    :return: likelihood: The likelihood of beta. Type: float.
    """
```

```
    s_dict = s_root(tree_topology, theta, beta)
    t_dict = defaultdict(dict)

    likelihood = 1

    """Recursively calculates t(u,i) if it has not already been calculated"""
    def t(u, i, parent, sibling):
        if t_dict[u].get(i) is not None: # If it has already been calculated
            return t_dict[u][i]

        if np.isnan(parent): # If root
            return CPD(theta, u, i) * s_dict[u][i]
        if sibling is None: # If no siblings
            result = 0
            for j in range(0, len(theta[0])):
                result += CPD(theta, u, i, j) * t(parent, j, tree_topology[parent],
                                                  find_sibling(parent, tree_topology))
                t_dict[u][i] = result
            return result

        parent = int(parent)
        result = 0
        for j in range(0, len(theta[0])):
            for k in range(0, len(theta[0])):
                result += CPD(theta, u, i, j) * CPD(theta, sibling, k, j) * \
                s_dict[sibling][k] * t(parent, j,tree_topology[parent],
                    find_sibling(parent,tree_topology))
        t_dict[u][i] = result
        return result

    for leaf, cat in enumerate(beta):
        if not np.isnan(cat):
            return t(leaf, cat, int(tree_topology[leaf]),
                          find_sibling(leaf, tree_topology)) *s_dict[leaf][cat]
    return likelihood
```

B: Variational Inference Algorithm

```
def expected_mu(lamb0, X, mu0, mu_n, lamb_n):
    E_mu2 = lamb_n ** (-1) + mu_n ** 2
    square_sum = np.sum((X ** 2) - (2 * X * mu_n) + E_mu2)
    return (1 / 2 * square_sum) + lamb0 * ((mu0 ** 2) - (2 * mu0 * mu_n) + E_mu2)

def approx_a(a0, n):
    return a0 + ((n + 1) / 2)
```

```
#Note, argument l0 = Lambda_0, NOT TEN
def approx_mu(l0, m0, X, n):
    return (l0 * m0 + n * np.average(X)) / (l0 + n)

def approx_lambda(l0, a_n, b_n, n):
    return (l0 + n) * (a_n / b_n)

def approx_b(m0, m_n, l_n, l0, b0):
    return b0 + expected_mu(l0, X, m0, m_n, l_n)

def VariationalInference(mu0, lamb0, a0, b0, X, iterations):
    i = 0
    la = 1 #Initial guess
    be = 1 #Initial guess
    mu = mu0
    al = a0
    while i < iterations:
        al = approx_a(a0, N)
        mu = approx_mu(lamb0, mu0, X, N)
        be = approx_b(mu0, mu, la, lamb0, b0)
        la = approx_lambda(lamb0, al, be, N)
        i += 1
        if i == iterations:
            return mu, la, al, be
```

References

[1] Christopher M. Bishop. *Pattern recognition and machine learning*. en. Information science and statistics. New York: Springer, 2006. ISBN: 978-0-387-31073-2.

[2] Kevin P. Murphy. *Machine learning: a probabilistic perspective*. en. Adaptive computation and machine learning series. Cambridge, MA: MIT Press, 2012. ISBN: 978-0-262-01802-9.