REPORT

Created by
Evgeny Genov

Date
January 5, 2020

Reference number

Assignment 2

DD2434 Machine Learning. Advanced Course

Knowing the rules

Question 2.1.1

Yes

Question 2.1.2

Sriram Ventatakrishnan

Question 2.1.3

No

Dependencies in a Directed Graphical Model

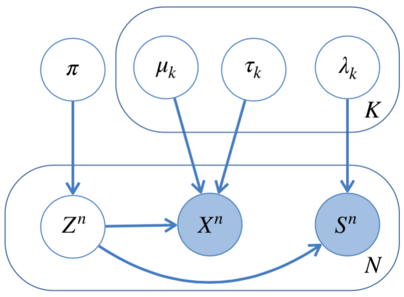

Figure 1: Directed Graphical model

Question 2.2.4

yes

Question 2.2.5

no

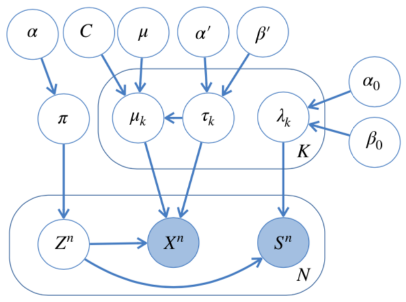

Figure 2: Directed Graphical model

Question 2.2.6

yes

Question 2.2.7

no

Question 2.2.8

no

Question 2.2.9

no

Likelihood of a tree GM

Question 2.3.10

The algorithm outlined hereby estimates the likelihood, of a specific beta sequence given tree topology and conditional probability distributions (CPD) of the tree node values. Beta is a list of assignments to nodes in a probabilistic binary tree structured graphical model. Therefore, probability $p(\beta|\boldsymbol{T}, \boldsymbol{\Theta}$ is a product of the probabilities of assignments of each leaf given the observation X_O where O is the lead set. The assignments are conditionally independent given the observation.

$$p(\beta|\boldsymbol{T}, \boldsymbol{\Theta}) = \prod_u p(X_u = i|X_O) \tag{1}$$

From Bayes theorem we know that the posterior distribution would be proportional to a joint

$$\prod_u p(X_u = i|X_O) \propto p(X_u = i, X_O) \tag{2}$$

Using the product rule

$$p(X_u = i, X_O) = p(X_{o\cap u\uparrow}, X_u = i)p(X_u = i, X_o|X_{o\cap u\uparrow}, X_u = i) \tag{3}$$

After rearranging and simplifying we get

$$p(X_u = i, X_O) = p(X_{o\cap u\uparrow}, X_u = i)p(X_{o\cap u\downarrow}|X_u = i) \tag{4}$$

The two terms form two sub-problems we calculate for each of the node. Firstly, we tackle the conditional probability denoted as

$$s(u, i) = p(X_{o\cap u\downarrow}|X_u = i) \tag{5}$$

We break it down for two children nodes of the node u

$$p(X_{o\cap u\downarrow}|X_u = i) = p(X_{o\cap v\uparrow}|X_u = i)X_{o\cap w\uparrow}|X_u = i) \tag{6}$$

We marginalize one of the terms over its values

$$= \Sigma_j p(X_{o\cap u\uparrow}, X_v = j|X_u = i) \tag{7}$$

Since $X_{o\cap u\uparrow}$ and $X_u = i$ are conditionally independent we use the product rule

$$= \Sigma_j p(X_{o\cap u\uparrow}|X_v = j)p(X_v = j|X_u = i) = \Sigma_j s(v, j)p(X_v = j|X_u = i) \tag{8}$$

The final expression is equivalent to s value of a child node multiplied by a corresponding CPD summed over all possible values j. Henceforth, s sub-problem is concerned with vertices that are induced with smaller-rooted sub-trees, already calculated while traversing the tree. At leaf nodes $s(l, i)$ takes the value

$$s(l, i) = \begin{cases} 1, & if\ l = i \\ 0, & if\ l \neq i \end{cases}$$

The first term in (4) establishes the sub-problem

$$t(u, i) = p(X_{o\cap u\uparrow}, X_u = i) \tag{9}$$

We marginalize over values of parent and sibling vertices

$$= \Sigma_{j,k} p(X_{o\cap u\uparrow}, X_u = i, X_p = j, X_{u'} = k) \tag{10}$$

Using product rule we obtain

$$\Sigma_{j,k} p(X_{o\cap u\uparrow}, X_p = j)p(X_{o\cap u\uparrow}, X_u = i, X_{u'} = k|X_p = j) \tag{11}$$

$$= \Sigma_{j,k} t(p, j)p(X_u = i|X_p = j)p(X_{o\cap u'}, X_{u'} = k|X_p = j) \tag{12}$$

$$= \Sigma_{j,k} t(p, j)p(X_u = i|X_p = j)p(X_{u'} = k|X_p = j)s(u', k) \tag{13}$$

where s is a sub-problem evaluated earlier. Similarly t(p,k) is concerned with smaller sub-trees. The probabilities in the middle are the conditional probabilities retrieved from the vector of CPD. The Python implementation of the DGM tree marginalization is shown in Appendix (2.3)

Question 2.3.11

If applied to a small-sized tree graphical model, the outputs for likelihood of samples have order of magnitude from e-03 to e-05.

```
  Sample: 0  Beta: [nan nan 1. 3. 0.]
  Likelihood: 9.465633985205277e-05
  Sample: 1  Beta: [nan nan 2. 1. 3.]
  Likelihood: 0.001789988154749095
  Sample: 2  Beta: [nan nan 4. 1. 2.]
  Likelihood: 0.00010151856808122183
  Sample: 3  Beta: [nan nan 4. 1. 3.]
  Likelihood: 0.00058370320850972
  Sample: 4  Beta: [nan nan 3. 1. 2.]
  Likelihood: 0.00017156713522188422
```

For a medium size tree the order of magnitude for likelihoods dramatically reduces and now is in the range from e-33 to 3-37 meaning the probability of a single certain assignment of nodes is very low.

```
Calculating the likelihood...
   Sample: 0 Beta:
  Likelihood: 9.986177128655834e-33
  Sample: 1 Beta:
  Likelihood: 3.3812624952728734e-35
  Sample: 2 Beta:
  Likelihood: 9.573235843321742e-35
  Sample: 3 Beta:
  Likelihood: 1.742574394954778e-36
  Sample: 4 Beta:
  Likelihood: 7.3547178647490496e-37
```

Large-sized tree samples of leaf nodes output very low likelihood value with the order of magnitude in range e-126 to e-137. We can observe that the likelihood decreases with respect to number of nodes in exponential fashion.

```
  Sample: 0  Beta:
  Likelihood: 1.5331874508485792e-130
  Sample: 1  Beta:
  Likelihood: 2.1193011353719957e-126
  Sample: 2  Beta:
  Likelihood: 9.873805079360715e-132
  Sample: 3  Beta:
  Likelihood: 1.1771069219736477e-137
  Sample: 4  Beta:
  Likelihood: 2.3759106019249727e-133
```

Simple VI

Question 2.4.12

We use a conjugate prior of the form

$$p(\mu, \lambda) = \mathcal{N}\left(\mu|\mu_0, (\lambda_0\tau)^{-1}\right) Gam\left(\tau|a_0, b_0\right) \tag{14}$$

A factored variational approximation for the posterior has the form

$$q(\mu, \tau) = q_\mu(\mu)q_\tau(\tau) \tag{15}$$

The optimum parameters can be obtained from the general expression for the optimal solution $q_j(\boldsymbol{Z_j})$ found in Bishop (10.9) [1] Following the steps outlined in [2] Averaging unnormalized log posterior over τ and μ and completing the square and identifying the resulting normal and gamma distributions, we retrieve the expressions for the parameters of Gaussian and Gamma distributions.

$$\mu_N = \frac{\lambda_0\mu_0 + N\overline{x}}{\lambda_0 + N} \tag{16}$$

$$\lambda_N = (\lambda_0 + N)\frac{a_N}{b_N} \tag{17}$$

$$a_N = a_0 + \frac{N+1}{2} \tag{18}$$

$$b_N = b_0 + \lambda_0(\mathbb{E}[\mu^2] + \mu_0^2 - 2\,\mathbb{E}[\mu]\mu_0) + \frac{1}{2}\Sigma_{i=1}^N(x_i^2 + \mathbb{E}[\mu^2] - 2\,\mathbb{E}[\mu]x_i) \tag{19}$$

where the expectation values are computed as following

$$\mathbb{E}_{q(\mu)}[\mu] = \mu_N \tag{20}$$

$$\mathbb{E}_{q(\mu)}[\mu^2] = \frac{1}{\lambda_N} + \mu^2 \tag{21}$$

$$\mathbb{E}_{q(\tau)}[\tau] = \frac{a_N}{b_N} \tag{22}$$

When implementing the code we note that a_N and μ_N are constants, while λ_N and b_N are updated over iterations. Furthermore, we mind that initialization for parameters b_N and λ_N cannot be zero to avoid division by zero. See the implementation code.

```python
#Authored by Evgeny Genov

import numpy as np
from math import pi
import matplotlib.pyplot as plt
from scipy.special import gamma
```

[1] p. 466, Bishop, C. M. (2006). Pattern recognition and machine learning. springer.

[2] p. 743, Murphy, K. P. (2012). Machine learning: a probabilistic perspective. MIT press.

```
# A factorized variational approximation to the posterior as
   expressed in (10.24)
def q(mu, mu_N, lambda_N, tau, a_N, b_N):
   # Gaussian distribution
   q_mu = (lambda_N / (2 * pi))**0.5
   q_mu *= np.exp(-0.5 * np.dot(lambda_N, ((mu - mu_N)**2).T))
   q_tau = gamma(a_N)**(-1) * b_N**a_N
   q_tau *= tau**(a_N-1) * np.exp(-b_N * tau)
   q = q_mu * q_tau
   return q

# expectations to implement parameter updates
def expectations(mu_N, lambda_N, a_N, b_N):
   expected_mu = mu_N
   expected_mu2 = lambda_N ** (-1) * mu_N ** 2
   expected_lambda = a_N / b_N
   return expected_mu, expected_mu2, expected_lambda

# updating scheme for parameters of q_mu and q_tau
def update_parameters(x, lambda_0, mu_0, a_0, b_0, iterations): # x
   - data points
   N = len(x)
   # mu_N and a_N are constants
   mu_N = (lambda_0 * mu_0 + N * np.mean(x)) / (lambda_0 + N)
   a_N = a_0 + (N + 1)/2
   lambda_N = 1
   b_N = 1
   # lambda_N and b_N are updated iteratively
   for i in range(iterations):
      expected_mu, expected_mu2, expected_lambda =
         expectations(mu_N, lambda_N, a_N, b_N)
      b_N = b_0 + lambda_0*(expected_mu2 + mu_0 -
         2*expected_mu*mu_0) + 0.5*np.sum(x**2 + expected_mu2 -
         2*expected_mu*x)
      lambda_N = (lambda_0 + N) * a_N/b_N
   return mu_N, lambda_N, a_N, b_N
# generate random dataset X
# m - mean, p - precision
def generate_data(m, p, N):
   return np.random.normal(m, np.sqrt(p ** (-1)), N)

# generate random data from Gaussian
x = generate_data(0, 1, 100)
# initialize parameters
mu_0, lambda_0, a_0, b_0 = 0,1,0,1
```

```
# number of iterations to converge parameters
iterations = 10
# update parameters
mu_N, lambda_N, a_N, b_N = update_parameters(x, lambda_0, mu_0,
   a_0, b_0, iterations)
# generate mus and taus for conjugate Gaussian, Gamma distributions
mus = np.linspace(-2.0, 2.0, 100)
taus = np.linspace(0, 4.0, 100)
# calculate factored posterior
q_posterior = q(mus[:,None], mu_N, lambda_N, taus[:,None], a_N, b_N)
```

Question 2.4.13

The example given is simple enough to compute the exact posterior analytically. The posterior takes the form of the Gaussian-Gamma distribution. We will use the Bayes theorem for posterior using the conjugate prior distributions $p(\mu|\tau)$, $p(\tau)$ and observed data points $p(\boldsymbol{D}|\mu,\tau)$

$$p(\mu,\tau|D) \propto p(\boldsymbol{D}|\mu,\tau)p(\mu|\tau)p(\tau) \tag{23}$$

The prior Gaussian and Gamma distributions form a constitute Gaussian-gamma conjugate prior distribution

$$p(\mu,\tau|\mu_0,\lambda_0,a_0,b_0) = \frac{b_0^{a_0}\sqrt{\lambda_0}}{\Gamma(a_0)\sqrt{2\pi}}\tau^{a_0-1}e^{-b_0\tau}\tau^{1/2}e^{-\frac{\lambda_0\tau(\mu-\mu_0)^2}{2}} \tag{24}$$

Observed values of x are assumed to be drawn independently from the Gaussian given in (10.21) in Bishop

$$p(\boldsymbol{D}|\mu,\tau) = \left(\frac{\tau}{2\pi}\right)^{N/2} exp\left(-\frac{\tau}{2}\Sigma_{i=1}^{N}(x_N-\mu)^2\right) \tag{25}$$

$$\propto \tau^{N/2}exp\left(-\frac{\tau}{2}\Sigma_{i=1}^{N}(x_i-\overline{x}+\overline{x}-\mu)^2\right) \tag{26}$$

$$\propto \tau^{N/2}exp\left(-\frac{\tau}{2}\Sigma_{i=1}^{N}((x_i-\overline{x})^2+(\overline{x}-\mu)^2\right) \tag{27}$$

$$\propto \tau^{N/2}exp\left(-\frac{\tau}{2}(\lambda(\mu-\mu_0)^2+N(\overline{x}-\mu)^2)\right) \tag{28}$$

where $\overline{x}$ is a mean, and s is a standard deviation $\Sigma_{i=1}^{N}(x_i-\overline{x})^2$. Now, we can apply the Bayes rule (10) and find the posterior

$$p(\mu,\tau|D) \propto p(\boldsymbol{D}|\mu,\tau)p(\mu|\tau)p(\tau) \propto p(\boldsymbol{D}|\mu,\tau)p(\mu,\tau|\mu_0,\lambda_0,a_0,b_0) \tag{29}$$

$$\propto \tau^{N/2}exp\left(-\frac{\tau}{2}(\lambda(\mu-\mu_0)^2+N(\overline{x}-\mu)^2)\right)\tau^{a_0+N/2-1}e^{-b_0\tau}e^{-\frac{\lambda_0\tau(\mu-\mu_0)^2}{2}} \tag{30}$$

$$\propto \tau^{N/2}exp\left(-\frac{\tau}{2}(\lambda(\mu-\mu_0)^2+N(\overline{x}-\mu)^2)\right)\tau^{a_0+N/2-1}e^{-b_0\tau}exp\left(-\tau(b_0+\frac{1}{2}\Sigma(x_i-\overline{x})^2)\right) \tag{31}$$

Here we need to identify the general forms of Gaussian and Gamma distributions. We get involved with the first term of observed data and rearrange the exponential with quadratic μ completing the square

$$\lambda(\mu-\mu_0)^2 + N(\overline{x}-\mu)^2 = (\lambda+N)\mu^2 - 2(\lambda\mu_0 + N\overline{x})\mu + \lambda\mu_0^2 + N\overline{x}^2 \tag{32}$$

$$= (\lambda_0+N)\left(mu - \frac{\lambda\mu_0 - N\overline{x}}{\lambda+N}\right)^2 + \frac{\lambda N(\mu_0-\overline{x})^2}{\lambda+N} \tag{33}$$

Plugging in the new term

$$\tau^{1/2} exp\left(-\frac{\tau(\lambda_0+N)}{2}\left(\mu - \frac{\lambda_0\mu_0 - N\overline{x}}{\lambda_0+N}\right)^2\right) \tag{34}$$

$$\times\tau^{a_0+N/2-1} exp\left(-\tau\left(b_0 + \frac{1}{2}\Sigma(x_i-\overline{x})^2 + \frac{\lambda_0 N(\mu_0-\overline{x})^2}{\lambda_0+N}\right)\right) \tag{35}$$

The first term reveals the factorized approximation of Gaussian form

$$q_\mu(\mu) = \mathcal{N}\left(\mu|\frac{\lambda_0\mu_0 + N\overline{x}}{\lambda_0+N}, \frac{1}{\tau(\lambda_0+N)}\right) \tag{36}$$

The accuate expressions for mean and precision are

$$\mu = \frac{\lambda_0\mu_0 + N\overline{x}}{\lambda_0+N} \tag{37}$$

$$\lambda = \lambda_0 + N \tag{38}$$

The term in (22) makes the expression for Gamma with the parameters

$$a_N = a_0 + N/2 \tag{39}$$

$$b_N = b_0 + \frac{1}{2}\Sigma_{i=1}^N(x_i-\overline{x})^2 + \frac{\lambda_0 N(\mu_0-\overline{x})^2}{\lambda_0+N} \tag{40}$$

Question 2.4.14

We add the plotting output functionality to the algorithm displayed in the solution to 2.4.12 and compare it with the same plots for exact posteriors evaluated in 2.4.13. The code implementation is included in the appendix.

Firstly, we explore how the information gain from prior affects the output of the Variational Bayes. Figure 3 demonstrates the iteration of variation inference for mean and precision for a prior initialized with all zeros in parameters. b_N and $lambda_N$ were initialized with 0.1 instead to avoid division by zero. We can see that in the beginning the prior is far off the exact posterior but with further iterations the approximation becomes more accurate. After 7th iteration no noticeable difference is observed and the approximation seems to converge.

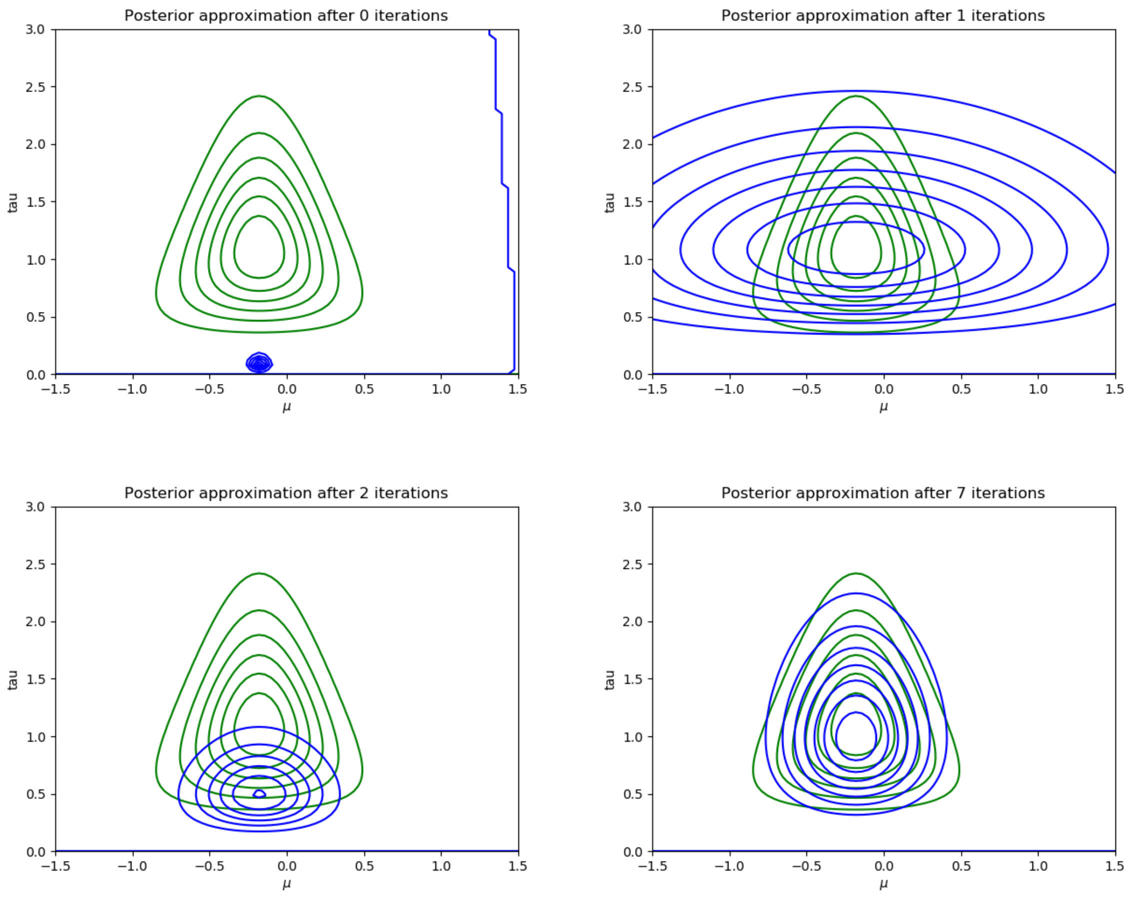

Figure 3: Illustration of variational inference for the mean μ and precision tau of a univariate Gaussian distribution. Sample size - 10 points. Non-informative prior. Contour of the true posterior is shown in green. Contour of the VI approximation is shown in blue.

Next figure, Figure 4, gives an example for a prior located near values of the parameters for the exact posterior. In that case the information gain from the prior is more tangible and convergence happens faster. Here the convergence is done at 3rd iteration with no subsequent visible change. Another possible test would be having the parameters of the prior be equal to the parameters of the exact posterior, then the VI approximation would stay constant over iterations.

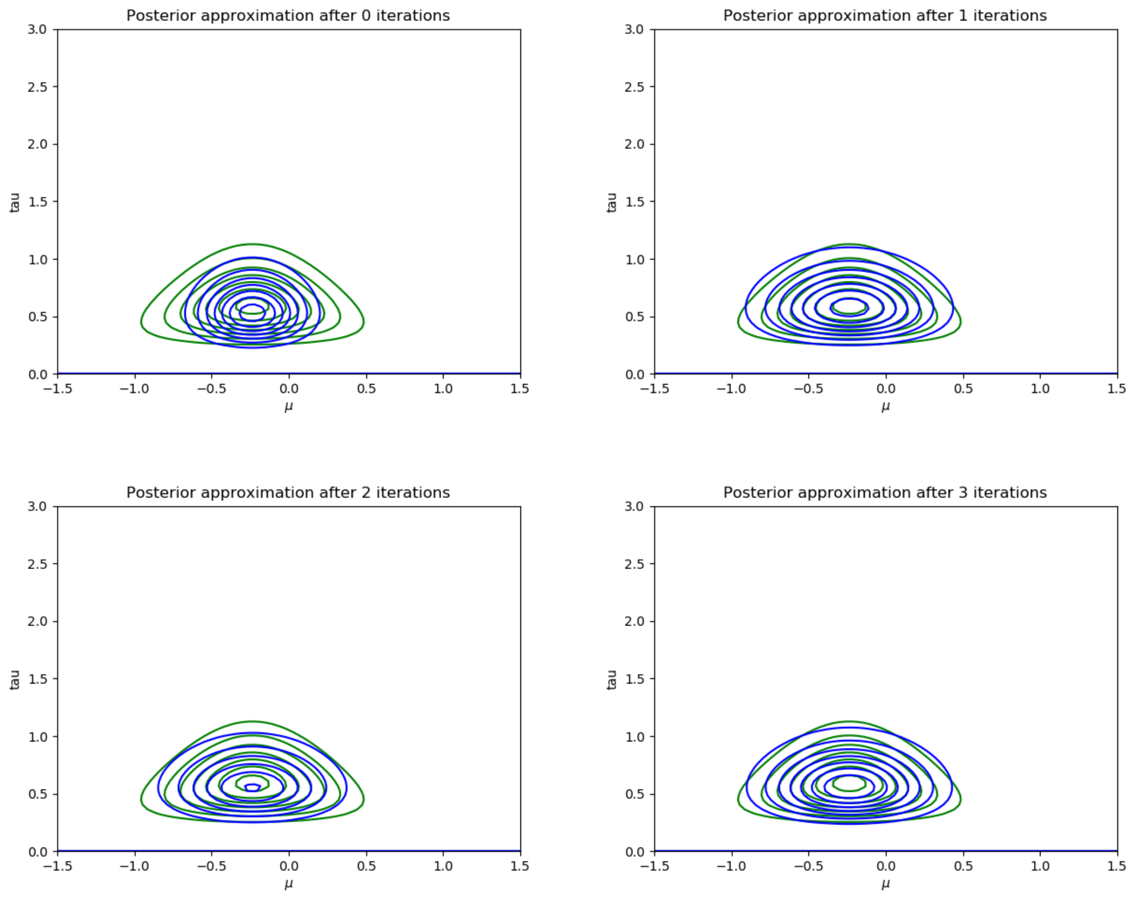

Figure 4: Illustration of variational inference for the mean μ and precision tau of a univariate Gaussian distribution. Prior near the exact posterior calculated analytically. Contour of the true posterior is shown in green. Contour of the VI approximation is shown in blue.

Figure 5 illustrates VI approximations for more points sampled from a univariate Gaussian. Sample size - 10 points. The prior has parameters at zero similar to the configuration in first example. The accuracy and speed of convergence improve in correspondence to change in number of data points from 10 to 100. At 4th iteration the approximation converges.

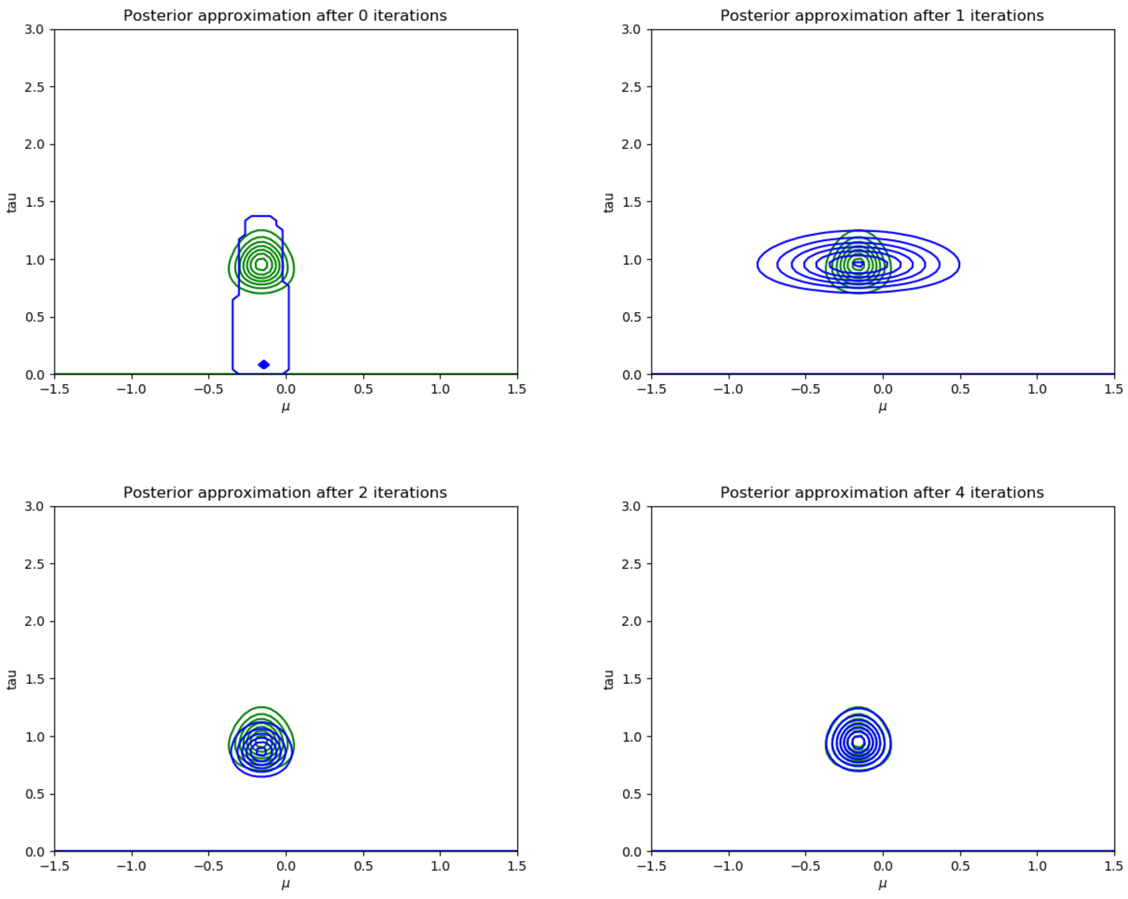

Figure 5: Illustration of variational inference for the mean μ and precision tau of a univariate Gaussian distribution. Larger sample size - 100 points. Non-informative prior. Contour of the true posterior is shown in green. Contour of the VI approximation is shown in blue.

Appendix

Question 2.3 Implementation

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


# probability of child with assigned value cat with a parrent with
   assigned value parent_cat
def CPD(theta, node, cat, parent_cat):
   if parent_cat is None:
      return theta[node][cat]
   else:
      return theta[node][int(parent_cat)][int(cat)]

# non-nan values are assignments for leaves. nan values are inner
   nodes
def find_leaves(beta):
   leaves = []
   for index, value in enumerate(beta):
      if not np.isnan(value):
         leaves.append(index)
   return leaves

# if the parent value of a node checked in topology equals given
   node, then the checked node is a child of given node
def find_children(p, topology):
   children = []
   for index, parent in enumerate(topology):
      if parent == p:
         children.append(index)
   return children

# if the parent is the same but the node has a different index
def find_sibling(u, topology):
   for index, parent in enumerate(topology):
      if parent == topology[u] and u != index:
         return index
   return None

def calculate_likelihood(tree_topology, theta, beta):
   # initialize dictionaries for s and t
```

```
likelihood = 0
S_VALUES = defaultdict(dict)
t_VALUES= defaultdict(dict)

def find_s(tree_topology, theta, beta):
   def subproblem_S(u, j, children):
      if S_VALUES[u].get(j) is not None: # If it has already
         been calculated
         return S_VALUES[u][j]
      # Visit the vertices from leaves to root
      if len(children) < 1:   # identify leaves
         if int(beta[u]) == j:
            S_VALUES[u][j] = 1
            return 1
         else:
            S_VALUES[u][j] = 0
            return 0
      subsolution = np.zeros(len(children))
      # run down the tree solving the subproblem for children of
         children recursively
      for index, child in enumerate(children):
         for category in range(0, len(theta[0])):
            subsolution[index] += subproblem_S(child, category,
               find_children(child, tree_topology)) * CPD(theta,
               child,category, j)
      s_subsolution = np.prod(subsolution)
      S_VALUES[u][j] = s_subsolution
      return s_subsolution
   # Start from root and find run the subproblem for children of
      the root node
   for i in range(len(theta[0])):
      subproblem_S(0, i, find_children(0, tree_topology))
   return S_VALUES

S_VALUES = find_s(tree_topology, theta, beta)
# Do the same for t, now syblings are taken into consideration
def subproblem_t(u, i, parent, sibling):
   if t_VALUES[u].get(i) is not None: # If it has already been
      calculated
      return t_VALUES[u][i]

   if np.isnan(parent): # identify the root
      return CPD(theta, u, i, None) * S_VALUES[u][i]
   subsolution = 0
   if sibling is None: # simplified if there is no siblings
      for j in range(0, len(theta[0])):
```

```
                subsolution += CPD(theta, u, i, j) * t(parent, j,
                    tree_topology[parent],subproblem_sibling(parent,
                    tree_topology))
                t_VALUES[u][i] = subsolution
            return subsolution
        parent = int(parent)
        for j in range(0, len(theta[0])):
            for k in range(0, len(theta[0])):
                subsolution += CPD(theta, u, i, j) * CPD(theta,
                    sibling, k, j) * S_VALUES[sibling][k] *
                    subproblem_t(parent,j,tree_topology[parent],find_sibling(parent,t
        t_VALUES[u][i] = subsolution
        return subsolution

    # Expressing the joint
    for leaf, cat in enumerate(beta):
        if not np.isnan(cat):
            likelihood = subproblem_t(leaf, cat,
                int(tree_topology[leaf]),find_sibling(leaf,
                tree_topology)) * S_VALUES[leaf][cat]
    print("Calculating the likelihood...")
    #likelihood = np.random.rand()

    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" #
        "data/q2_3_medium_tree.pkl", "data/q2_3_large_tree.pkl"
    t = Tree()
    t.load_tree(filename)
    t.print()

    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()
```

Question 2.4 Implementation

```
#Authored by Evgeny Genov

import numpy as np
from math import pi, sqrt
import matplotlib.pyplot as plt
from scipy.special import gamma

# A factorized variational approximation to the posterior as
    expressed in (10.24)
def q(a_N, b_N, mu_N, lambda_N, mu, tau):
   # Gaussian distribution
   q_mu = (lambda_N/(2*pi))**(0.5)
   q_mu *= np.exp(-0.5*np.dot(lambda_N,((mu-mu_N)**2).T))
   # Gamma distribution
   q_tau = (1/gamma(a_N))*(b_N**a_N * tau**(a_N-1) *
      np.exp(-b_N*tau))
   q = q_mu*q_tau
   return(q)

# Compute q for true parameters
def exactPost(mu, mu_T, lambda_T, tau, a_T, b_T):
   post = ((b_T**a_T)*sqrt(lambda_T))/(gamma(a_T)*sqrt(2*pi))
   post = post*(tau**(a_T-0.5))*np.exp(-b_T*tau)
   post = post*np.exp(-0.5*(lambda_T*np.dot(tau,((mu-mu_T)**2).T)))
   return(post)

# expectations to implement parameter updates
def expectations(mu_N, lambda_N, a_N, b_N):
   expected_mu = mu_N
   expected_mu2 = (1/lambda_N) + mu_N**2
   expected_tau = a_N/b_N
   return(expected_mu, expected_mu2, expected_tau)

# updating scheme for parameters of q_mu and q_tau
```

```
def update_parameters(x, lambda_0, mu_0, a_0, b_0, iterations, mu,
    tau): # x - data points
  N = len(x)
  x_mean = (1/N)*sum(x)
  # mu_N and a_N are constants
  mu_N = (lambda_0*mu_0 + N*x_mean)/(lambda_0 + N)
  a_N = a_0 + (N+1)/2
   # declare parameters of an exact posterior
  a_T, b_T, mu_T, lambda_T = trueParameters(x,a_0,b_0,mu_0,lambda_0)
  # Initalized values before they get updated
  # INITALIZE b_N AND lambda_N HERE
  b_N = 0.1
  lambda_N = 0.1

  # lambda_N and b_N are updated iteratively
  for i in range(iterations):
    expected_mu, expected_mu2, expected_tau = expectations(mu_N,
        lambda_N, a_N, b_N)
    lambda_N = (lambda_0+N)*expected_tau
    b_N = b_0-expected_mu*(lambda_0*mu_0+sum(x))
    b_N =
        b_N+0.5*(expected_mu2*(lambda_0+N)+lambda_0*mu_0**2+sum(x**2))

      # calculate factored posterior
    q_posterior = q(a_N,b_N,mu_N,lambda_N,mu[:,None],tau[:,None])

      # calculate exact posterior
    exact = exactPost(mu[:,None], mu_T, lambda_T, tau[:,None],
        a_T, b_T)

    # Plot the posterior approximation and the exact one in the
        same canvas
    plotexact(mu, tau, exact)
    colour = 'b'
    if i == iterations-1:
      colour = 'r'

    plotPost(mu, tau, q_posterior, colour, i)
# generate random dataset X
# m - mean, p - precision
def generate_data(m, p, N):
   return np.random.normal(m, np.sqrt(p ** (-1)), N)

# build the canvas and plot the posterior
def plotPost(mu, tau, q_posterior, colour, i):
  muGrid, tauGrid = np.meshgrid(mu, tau)
```

```
    plt.contour(muGrid, tauGrid, q_posterior, colors = colour)
    plt.title('Posterior approximation after '+str(i)+' iterations')
    plt.axis([-1.5,1.5,0,3])
    plt.xlabel('$\mu$')
    plt.ylabel('tau')
    plt.savefig('./plots/morepoints_iteration_'+str(i)+'.png')
    plt.clf()


# parameters of of q_mu and q_tau for an exact posterior found
    analytically
def trueParameters(x, a_0, b_0, mu_0, lambda_0):
    N = len(x)
    x_mean = (1/N)*sum(x)
    mu_T = (lambda_0*mu_0 + N*x_mean)/(lambda_0 + N)
    lambdaT = lambda_0 + N
    aT = a_0 + N/2
    b_T = b_0 + 0.5*sum((x-x_mean)**2)
    b_T = b_T + (lambda_0*N*(x_mean-mu_0)**2)/(2*(lambda_0+N))
    return(aT, b_T, mu_T, lambdaT)


# Plot the exact posterior
def plotexact(mu, tau, exact):
    muGrid, tauGrid = np.meshgrid(mu, tau)
    plt.contour(muGrid, tauGrid, exact, colors = 'g')

# generate random data from Gaussian
x = generate_data(0, 1, 100)

# initialize parameters
a_0 = 0
b_0 = 0
mu_0 = 0
lambda_0 = 0


# generate mus and taus for conjugate Gaussian, Gamma distributions
mu = np.linspace(-2,2,100)
tau = np.linspace(0,4,100)

# update parameters and plot factored posterior over iterations
    paired with the exact posterior
update_parameters(x, lambda_0, mu_0, a_0, b_0, iterations, mu, tau)
```