DD2434 Machine Learning, Advanced Course Assignment 2

Chia-Hsuan Chou
chchou@kth.se

March 19, 2023

1 Knowing the rules

1.1 Theory

Question 2.1.1 It is mandatory to read the above text. Have you read it?

Yes.

Question 2.1.2 List all your collaborations concerning the problem formulations in this as- signment.

Yes. Cheng HungCheng and I concerns the problem formulations on problem 2.5 and 2.6.

Question 2.1.3 Have you discussed solutions with anybody?

No.

2 Dependencies in a Directed Graphical Model

Question 2.2.4 In the graphical model of Figure 1, is $\mu_k \perp \tau_k$ (not conditioned by anything)?

Yes, μ_k and τ_k are independent to each other, since X^n is unobserved.

Question 2.2.5 In the graphical model of Figure 1, is $\mu_k \perp \tau_k | X_1, ..., X_N$?

No, μ_k and τ_k are not independent to each other when conditioning on X.

Question 2.2.6 In the graphical model of Figure 2, is $\mu \perp \beta^{'}$ (not conditioned by anything) ?

Yes, μ is independent to $\beta^{'}$.

Question 2.2.7 In the graphical model of Figure 2, is $\mu \perp \beta^{'} | X_1, ..., X_N$?

No, μ is not independent to $\beta^{'}$ anymore if we know X^1,....,X^N.

Question 2.2.8 In the graphical model of Figure 2, is $X_n \perp S_n$ (not conditioned by anything) ?

No, X^n and S^n are not independent.

Question 2.2.9 In the graphical model of Figure 2, is $X_n \perp S_n | \mu_k, \tau_k$?

No, X^n and S^n are still not independent even though we know μ_k and τ_k.

3 Likelihood of a tree GM only for E level

Question 2.3.10 implement a dynamic programming algorithm that, for a given T,$Theta$ and β computes $p(\beta|T,\Theta)$.

The implementation of the dynamic programming to find the likelihood of the tree GM is given as following :

```
def calculate_likelihood(tree_topology, theta, beta):
    likelihood = calculate_s(tree_topology,theta,beta,0)
    likelihood = np.dot(theta[0],likelihood)
    return likelihood

def calculate_s(tree, theta, beta, node):

    if isnan(beta[node]):
        children = find_children(tree,node)
        theta1  = theta[children[0]]
        theta2  = theta[children[1]]
        theta1  = np.array(theta1.tolist())
        theta2  = np.array(theta2.tolist())
        s1 = calculate_s(tree,theta,beta,children[0])
        s2 = calculate_s(tree,theta,beta,children[1])
        s1 = np.dot(theta1,s1)
        s2 = np.dot(theta2,s2)
        # print(children[0],s1)
        # print(children[1],s2)
        return s1 * s2

    else:
        k = theta.shape[1]
        s = np.zeros((k,1))
        # print('node',node)
        s[int(beta[node])] = 1
        return s


def find_children(tree_topology,node):
    children = []
    children = np.argwhere(tree_topology == node)
    children = children.reshape((2,))

    return list(children)
```

Question 2.3.11 Apply your algorithm to the graphical model and data provided separately.

The likelihood of tree in different samples is given as follow:

- *small dataset*:
  - sample 1 : 0.008753221441670067
  - sample 2 : 0.0383969250979291
  - sample 3 : 0.009129106859990063
  - sample 4 : 0.0214406975419561
  - sample 5 : 0.011945567814215127
- *medium dataset*:
  - sample 1 : 4.623007229924853e-18
  - sample 2 : 1.8503307902116725e-19
  - sample 3 : 3.8046556943873916e-20
  - sample 4 : 5.3790257640101073e-20
  - sample 5 : 4.308018501901334e-19
- *large dataset*:
  - sample 1 : 2.033319143194999e-74
  - sample 2 : 1.1110716180722111e-76
  - sample 3 : 3.265122472498336e-75
  - sample 4 : 5.609870537950858e-75
  - sample 5 : 9.856642061714724e-77

4 Simple VI

Question 2.4.12 Implement the VI algorithm for the variational distribution in Equation (10.24) in Bishop.

The following codes are the implementation of VI algorithm. Here the dataset is generated in random normal distribution with 0 as mean and 1 as variance.

```
#### generate dataset
def generate_data(mean,var,N):
    Data = np.random.normal(mean,var,N)
    return Data
#### estimate the posterior
def estimate_posterior(mu,muN,LN,tau,aN,bN):
    exponent = - LN * (mu-muN)**2 / 2
    q  = (np.sqrt(LN/(2 * np.pi)) * np.exp(exponent) ) * ((1.0 /gamma(aN)) * (bN**aN) * (tau**(aN-1))
        * np.exp(- bN * tau))
    return q

def VI_algorithm(D,N,L0,mu0,a0,b0,bN,LN,mu,tau):
    iter = 6
    i    = 0
    Post_esti = []
    aN = a0 + (N+1)/2.0
    xbar = np.sum(D) / N
    mu_N = (L0 * mu0 + N * xbar) /(L0 + N)
    ## iteration
    while(i < iter):
        ## mu_N

        ## EXPECT VALUES
        E_tau = aN / bN
        E_mu  = mu_N
        E_mu2 = mu_N**2 + 1 / LN
        LN    = (L0 + N) * E_tau
        bN = b0 + 0.5 * (np.sum(D**2) + L0 * (mu0**2)) - \
                   (np.sum(D) + mu0 * L0) * E_mu + \
                   0.5 * (L0 + N) * E_mu2
       ## store for plotting
        esti = estimate_posterior(mu,mu_N,LN,tau,aN,bN)
        Post_esti.append(esti)
        i = i + 1
    return Post_esti

#### generating dataset
N    = 10
mean = 0
var  = 1
Data = generate_data(mean,var,N)

## some parameters
mu_0 = 1e-16
a_0  = 1e-16
b_0  = 1e-16
L_0  = 1e-16 ## lambda 0
bN   = 0.4
LN   = 0.2

## some tau and some mu
tau = np.linspace(0,4,100)
mu  = np.linspace(-2,2,100)
tauv,muv = np.meshgrid(tau,mu)

## VI algorithm
Pesti = VI_algorithm(Data,N,L_0,mu_0,a_0,b_0,bN,LN,muv,tauv)
```

Question 2.4.13 What is the exact posterior?

According to Bayesian theorem and chain rule, we know that:

$$P(\mu,\tau|D) = \frac{P(D|\mu,\tau)P(\mu|\tau)P(\tau)}{P(D)} \propto P(D|\mu,\tau)P(\mu|\tau)P(\tau)$$

From the book *Bishop*[1] we know that the likelihood function is given by

$$P(D|\mu,\tau) = (\frac{\tau}{2\pi})^{\frac{N}{2}} exp\{-\frac{\tau}{2}\sum_{n=0}^{N}(x_n-\mu)^2\}$$

and the conjugate prior distribution for μ and τ are given by :

$$P(\mu|\tau) = \mathcal{N}(\mu|\mu_0,(\lambda_0\tau)^{-1}) = \sqrt{\frac{\lambda_0\tau}{2\pi}} exp\{-\frac{\lambda_0\tau(\mu-\mu_0)^2}{2}\}$$

and

$$P(\tau) = Gam(\tau|a_0,b_0) = \frac{1}{\Gamma(a_0)} b_0^{a_0}\tau^{a_0-1} exp\{-b_0\tau\}$$

from equation (2.146) and (1.141) defined in *Bishop*. The exact posterior then becomes :

$$\begin{aligned}
P(\mu,\tau|D) &\propto P(D|\mu,\tau)P(\mu|\tau)P(\tau) \\
&= (\frac{\tau}{2\pi})^{\frac{N}{2}} exp\{-\frac{\tau}{2}\sum_{n=0}^{N}(x_n-\mu)^2\}\sqrt{\frac{\lambda_0\tau}{2\pi}} exp\{-\frac{\lambda_0\tau(\mu-\mu_0)^2}{2}\}\frac{1}{\Gamma(a_0)} b_0^{a_0}\tau^{a_0-1} exp\{-b_0\tau\} \\
&= (\frac{\tau}{2\pi})^{\frac{N}{2}} \frac{1}{\Gamma(a_0)}\sqrt{\frac{\lambda_0\tau}{2\pi}} exp\{-\frac{\tau}{2}\sum_{n=0}^{N}(x_n-\mu)^2\} exp\{-\frac{\lambda_0\tau(\mu-\mu_0)^2}{2}\} exp\{-b_0\tau\}
\end{aligned}$$

[1]P470, Pattern Recognition and Machine Learning, Christopher M. Bishop

The following code shows the implementation of the exact posterior.

```
def true_posterior(D,N,mu,tau,mu0,L0,a0,b0):
    ## P(D|mu,tau)
    sumup        = np.sum(D**2) - 2 * np.sum(D) * mu + N * mu**2
    P_Likelihood = (tau / (2 * np.pi))**(N/2) * np.exp(-(tau/2) * sumup)
    ## P(mu | tau)
    sigma        = 1/(L0 * tau)
    exponent     = - (mu0 - mu)**2 / (2 * sigma)
    P_prior      =  1 / (np.sqrt(2 * np.pi * sigma)) * np.exp(exponent)
    ## P(tau)
    P_tau        = (1 /gamma(a0)) * (b0**a0) * (tau**(a0-1)) * np.exp(-b0 * tau)
    return P_Likelihood * P_prior * P_tau
```

Question 2.4.14 Compare the inferred variational distribution with the exact posterior. Run the inference on data points drawn from iid Gaussians. Do this for three interesting cases and visualize the results. Describe the differences.

In the first case, there are 10 datas in the dataset sampled from the Gaussians distribution with mean equals to 0 and variance equals to 1. a_0, b_0, μ_0 and λ_0 are all set to a small number that close to 0. This case is designed to test whether if the parameter settings are improper, we can still get a proper posterior distribution after several iteration, as described in the book. The initialized b_N is set to be 0.4 and initialize λ_N is set to be 0.2 .

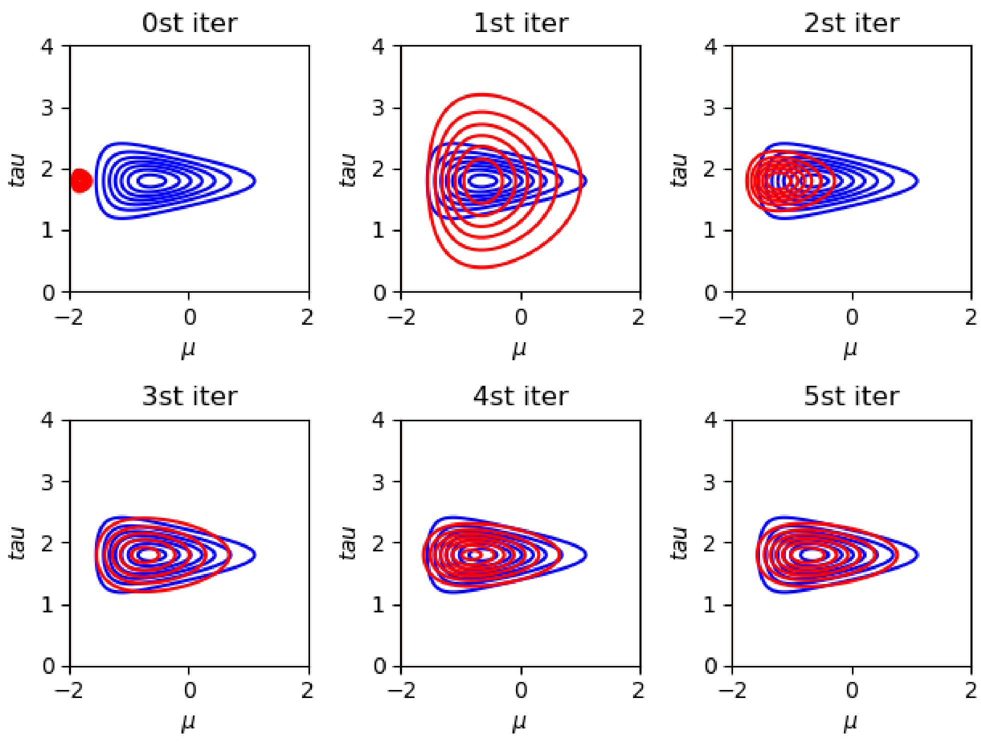

Figure 1: Estimated Posterior in Case 1 in 2.4

From figure 1 we know that although these parameter settings are improper, after 3 iteration times the estimated posterior distribution gradually converges to the exact posterior distribution as described in *Bishop*.

In the second case and the third case, I would like to test if different number of data in the dataset will give us different results. In these two cases, a_0 is set to be 2 ,b_0 is 5, λ_0 is 10, μ_0 is 0.01, initialized b_N is 0.4 and initialized λ_N is 10. The only difference between the second case and the third case is the number of data in dataset. There are 10 datas in the second case, and 50 datas in the third case. Figure 2 shows the comparison of exact posterior and esitmated posterior during iteration in case 2 and figure 3 shows the comparison in case 3.

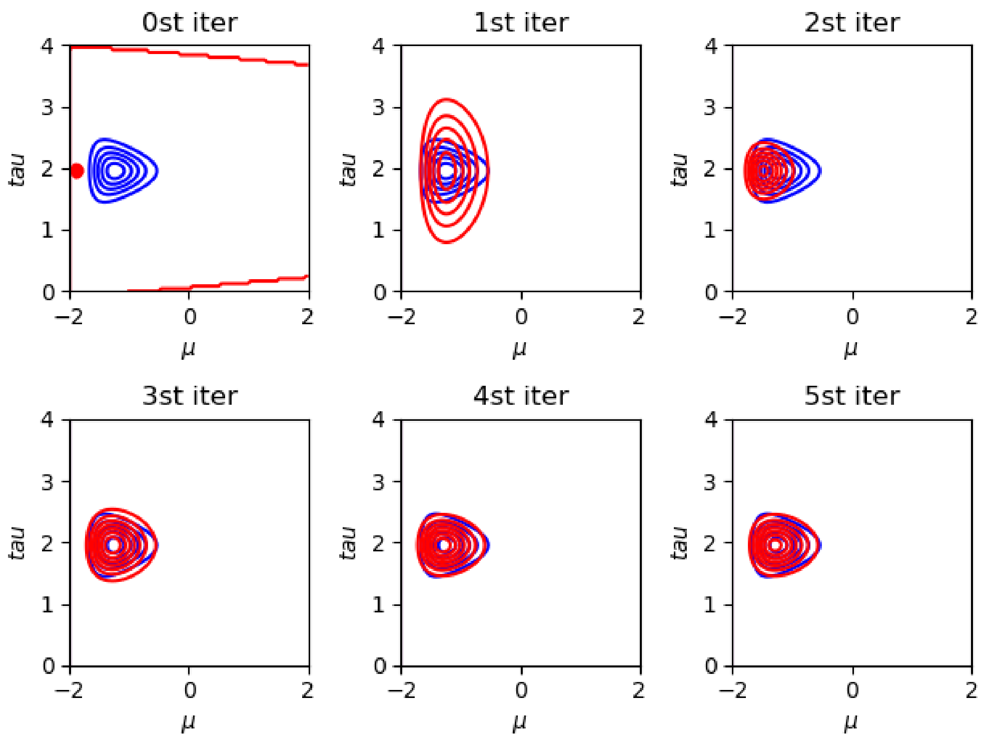

Figure 2: Estimated Posterior in Case 2 in 2.4

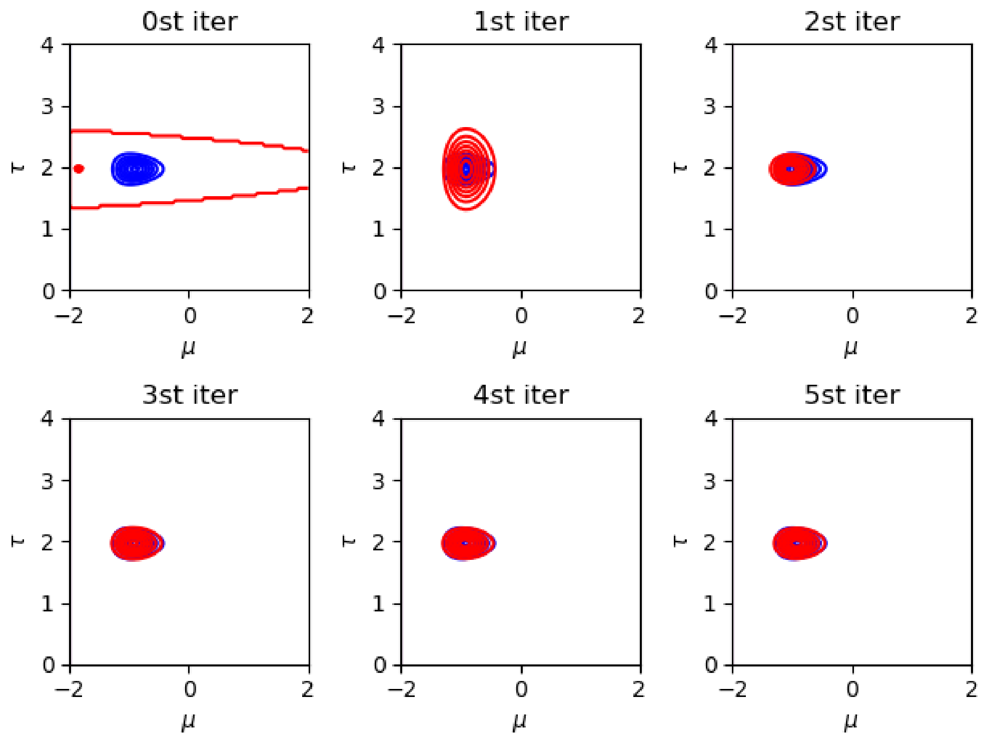

Figure 3: Estimated Posterior in Case 3 in 2.4

From figure 2 and 3, we can see that the density of the exact posterior distribution is tighter in case 3. This makes sense because once we have more datas in the dataset, meaning that we have more information about the dataset, the variance of the posterior distribution will be smaller because we are much confident on the parameter τ and μ we have found . On the other hand, We can see that with the same parameter settings, if there are more datas in the dataset, the estimated posterior will converge to a proper posterior distribution faster. In figure 2, the estimated posterior has similar distribution with the exact posterior distribution after 4 iteration while in figure 3 the estimated posterior converges to a similar distribution after only 2 iteration.

5 Mixture of trees with observable variables

Question 2.5.15 Implement this EM algorithm.

```
import argparse
import numpy as np
import matplotlib.pyplot as plt
from math import isnan
from itertools import combinations
import Kruskal_v2 as Kv
import sys
from Tree import Node, Tree, TreeMixture
import dendropy


def calculate_likelihood(sample,topology,theta):

    num_node = len(sample)
    resp     = 1
    #print(len(theta))
    for i in range(num_node):

        ### root
        if isnan(topology[i]):
            resp = resp * theta[0][sample[i]]
        else:
            parent = topology[i]
            psam = sample[int(parent)]
            csam = sample[i]
            CPD  = theta[i]
            resp = resp * CPD[psam][csam]

    return resp

def Compute_qkab(rk,node1,node2,a,b):
    if a == 0 :
        find1 = np.where(node1 == 0)[0]
    else :
        find1 = np.where(node1 == 1)[0]
    if b == 0 :
        find2 = np.where(node2 == 0)[0]
    else :
        find2 = np.where(node2 == 1)[0]
    index = list(set(find1).intersection(set(find2)))
    weight = np.sum(rk[(index)]) / np.sum(rk)
    return weight

def Compute_qka(rk,node1,a):
    if a == 0:
        find1 = np.where(node1 == 0)[0]
    else:
        find1 = np.where(node1 == 1)[0]

    weight = np.sum(rk[find1]) / np.sum(rk)
    return weight

def ComputeWeight(node,r,k,sample):
    weight = 0
    node1 = sample[:,node[0]].reshape((sample.shape[0],1))
    node2 = sample[:,node[1]].reshape((sample.shape[0],1))
    rk    = r[:,k].reshape((sample.shape[0],1))
    for a in range(2):
        for b in range(2):
            qkab = Compute_qkab(rk, node1, node2, a, b)
            qka  = Compute_qka(rk, node1, a)
            qkb  = Compute_qka(rk, node2, b)
            if qka == 0 or qkb == 0 or qkab == 0:
                weight += 0
            else:
                weight += qkab * np.log(qkab/(qka * qkb))
    return weight

def generate_graph(vertex,edge):

    graph = {
        'vertices': [i for i in range(vertex)],
        'edges': {i for i in edge}

    }
    return graph
def Find_Sub_topology(result,parent,tree):
    o = []
    new_result = []
    for (u,v) in result :
        if u == parent:
            if v not in o:
                o.append(v)
            tree[v] = parent
        elif v == parent :
            if u not in o :
                o.append(u)
            tree[u] = parent
        else:
            new_result.append((u,v))
    if len(o) == 0 :
        return new_result,tree
    else :
        for sub_parent in o :
            new_result, tree = Find_Sub_topology(new_result,sub_parent,tree)
        return new_result, tree
def Find_New_topology(result,num_nodes):
    topology = [np.nan] * num_nodes
    new_result = []
    o = []
    ### find first children of 0
    for (u,v,w) in result:

        if u == 0 :
            if v not in o :
                o.append(v)
            topology[v] = 0
        elif v == 0 :
            if u not in o :
                o.append(u)
            topology[u] = 0
        else :
            new_result.append((u,v))
    for subroot in o :
        new_result ,topology = Find_Sub_topology(new_result, subroot, topology)

    return topology
def Compute_theta(r,k,topology,num_node,samples):

    theta = []
    ### the root

    rk   = r[:,k]
    root = [Compute_qka(rk,samples[:,0],0),Compute_qka(rk,samples[:,0],1)]
    root = root/np.sum(root)
    theta.append(root)
    for i in range(1, num_node):
        sub_theta = np.zeros((2,2))
        parent = topology[i]
        for a in range(2): ## parent
            for b in range(2): ## child
                given_p = np.where(samples[:,parent] == a)[0]
                given_c = np.where(samples[:, i] == b)[0]
                index = list(set(given_p).intersection(set(given_c)))
                sub_theta[a][b] = np.sum(rk[index])/np.sum(rk[given_p]) + sys.float_info.epsilon

        sub_theta = sub_theta / np.tile(np.sum(sub_theta, axis = 1).reshape((2, 1)),2)
        ### to fit the data structure from TA = ...

        theta.append([sub_theta[0,:],sub_theta[1,:]])

    return theta
def Compute_likelihood_from_real_tree(mixtree, samples):
    likelihood = 0
    pi = mixtree.pi
    n_samples = samples.shape[0]
    num_clusters = len(pi)
    r = np.zeros((n_samples, num_clusters))
    for n in range(n_samples):
        for c in range(num_clusters):
            topology = mixtree.clusters[c].get_topology_array()
            theta    = mixtree.clusters[c].get_theta_array()
            r[n, c] = calculate_likelihood(samples[n], topology, theta) + sys.float_info.epsilon
    likelihood = np.sum(np.log(np.sum(r * pi, axis=1)))
    return likelihood

def em_algorithm(seed_val, samples, num_clusters, max_num_iter=100):


    # Set the seed
    np.random.seed(seed_val)
    # TODO: Implement EM algorithm here.
    # Start: Example Code Segment. Delete this segment completely before you implement the algorithm.
    #### randomly create trees
    print("Running EM algorithm...")
    loglikelihood = []

    from Tree import TreeMixture
    sieving = 100
    bestlikelihood = 0
    #tm = TreeMixture(num_clusters=num_clusters, num_nodes = samples.shape[1])
    for i in range(sieving):
        tm = TreeMixture(num_clusters=num_clusters, num_nodes = samples.shape[1])
        tm.simulate_pi(seed_val=seed_val)
        tm.simulate_trees(seed_val=seed_val)
        tm.sample_mixtures(num_samples=samples.shape[0], seed_val = seed_val)
        like = Compute_likelihood_from_real_tree(tm, samples)
        if (np.exp(like) > bestlikelihood):
            bestlikelihood = like
            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())

    for k in range(num_clusters):
        topology_list.append(tm.clusters[k].get_topology_array())

        theta_list.append(tm.clusters[k].get_theta_array())



    topology_list = np.array(topology_list)
    theta_list = np.array(theta_list)
    pi         = np.ones((1,num_clusters))
    pi         = pi / np.sum(pi)

    #### start do EM algorithm
    n_samples = samples.shape[0]
    ### responsibility
    ### 1. calculate p(sample|tk,thetak)
    for i in range(max_num_iter):
        r = np.zeros((n_samples,num_clusters))
        for n in range(n_samples):
            for c in range(num_clusters):
                r[n,c] = calculate_likelihood(samples[n],topology_list[c],theta_list[c]) +
                    sys.float_info.epsilon
            #r[n,:] = r[n,:]/ np.sum(r[n,:])

    ### 2. responsibility
        loglikelihood.append(np.sum(np.log(np.sum(r * pi, axis=1))))
        r = pi * (r / np.tile(np.sum(r * pi, axis = 1).reshape((n_samples, 1)),num_clusters))


    ### 3. pi
        pi = np.mean(r, axis=0)
        #pi = pi / np.sum(pi)
    ### 4. compute weight
        NoN = list(combinations([i for i in range(samples.shape[1])],2))
        topology_list = []
        theta_list = []
        for k in range(num_clusters):
            edges = []
            for e in NoN:
                # edge = (node1 , node2)
                nodeWedge = ComputeWeight(e, r, k, samples)
                if nodeWedge != 0:
                    edges.append((e[0], e[1], nodeWedge))
            graphs = generate_graph(samples.shape[1],edges)
            result = Kv.maximum_spanning_tree(graphs)
            new_topology = Find_New_topology(result, samples.shape[1])
            topology_list.append(new_topology)
            new_theta = Compute_theta(r , k , new_topology , samples.shape[1],samples)
            theta_list.append(new_theta)

        theta_list = np.array(theta_list)
        topology_list = np.array(topology_list)
        if abs(loglikelihood[i] - loglikelihood[i-1]) < sys.float_info.epsilon and i != 0:
            loglikelihood = np.array(loglikelihood)
            return loglikelihood, topology_list, theta_list


    return loglikelihood, topology_list, theta_list
```

Question 2.5.16 Apply your algorithm to the provided data and show how well you reconstruct the mixtures. First, compare the real and inferred trees with the unweighted Robinson-Foulds (aka symmetric difference) metric. Do the trees have similar structure (don't worry if the inferred trees don't match with the real trees)? Then, compare the likelihoods of real and inferred mixtures. Finally, simulate more data and analyse the results (try to find some interesting and more challenging cases).

To testify my implementation of the EM algorithm, the dataset with 10 nodes, 20 samples and 4clusters are used. Figure 4 is the likelihood and log-likelihood of the mixture model in test dataset.

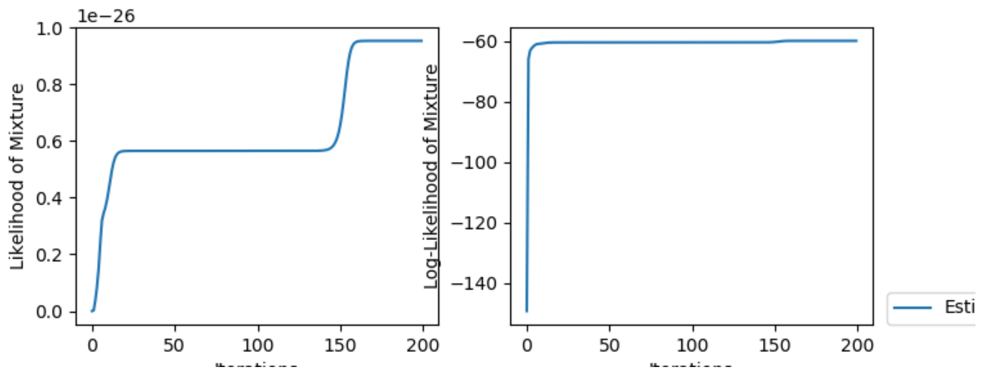

Figure 4: Likelihood of mixture model in Question 2.5.16

The following table 1 is the unweighted Robinson-Foulds metric of real and inferred trees.

	RT 1	RT 2	RT 3	RT 4
IT 1	8	7	9	9
IT 2	6	7	9	7
IT 3	9	6	10	8
IT 4	7	10	12	8

Table 1: RF metric of real and inferred trees in test dataset

After printing out the tree topology, I found that the inferred tree and real tree do not have similar structure. And the following table 2 is the comparison of the likelihood with real and inferred trees.

	Real Trees	Inferred Trees
loglikelihood	-113.143	-59.915
likelihood	7.287e-50	9.537e-27

Table 2: The comparison of likelihood between real trees and inferred trees

From table 2 we can see that given the samples, the inferred trees give us higher likelihood of the Model. This makes sense because we use these samples to do the E-Step and M-Step, after iterations, the algorithm will try to maximize the likelihood function.

Now, two other provided datasets are tested.

- *more nodes* : 20 nodes, 20 samples and 4 clusters.

  Figure 5 is the likelihood and log-likelihood of the mixture model and table 4 is the comparison of likelihood between real trees and inferred trees in this case. and table 3 is the RF distance matric in this case.

	RT 1	RT 2	RT 3	RT 4
IT 1	21	17	21	22
IT 2	19	15	19	18
IT 3	23	17	19	24
IT 4	21	17	21	20

Table 3: RF metric of real and inferred trees in case 2

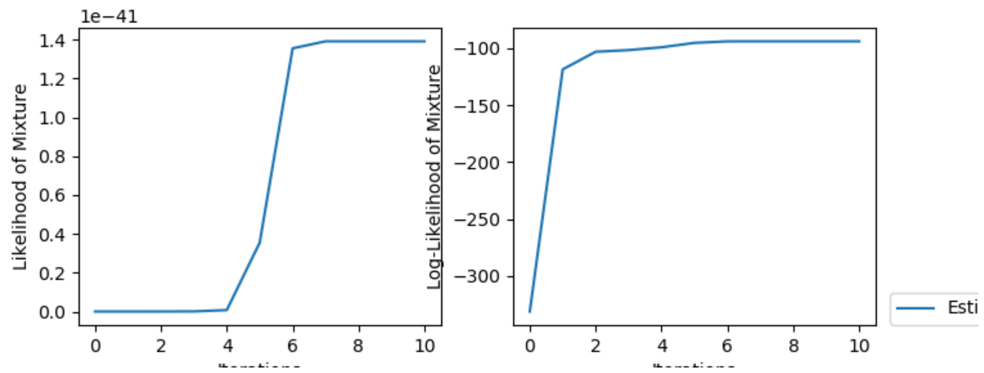

Figure 5: Likelihood of mixture model in case 2

	Real Trees	Inferred Trees
loglikelihood	-286.376	-94.075
likelihood	4.249e-125	1.391e-41

Table 4: The comparison of likelihood between real trees and inferred trees in case 2

- *more samples* : 10 nodes, 50 samples and 4 clusters

  Figure 6 is the likelihood and log-likelihood of the mixture model and table 6 is the comparison of likelihood between real trees and inferred trees in this case. and table 5 is the RF distance matric in this case.

	RT 1	RT 2	RT 3	RT 4
IT 1	8	9	9	9
IT 2	10	7	7	9
IT 3	9	8	12	8
IT 4	7	10	10	8

Table 5: RF metric of real and inferred trees in case 3

	Real Trees	Inferred Trees
loglikelihood	-280.857	-221.693
likelihood	1.061e-122	5.24e-97

Table 6: The comparison of likelihood between real trees and inferred trees in case 3

From table 1, 3 and 5 we can see that, if the number of nodes increases, the distance between true and inferred trees increase. On the other hand, the number of samples has less influence on it. As for the likelihood, once the number of nodes increases, the likelihood decreases. When the number of samples increases, the likelihood of model decrease as well.

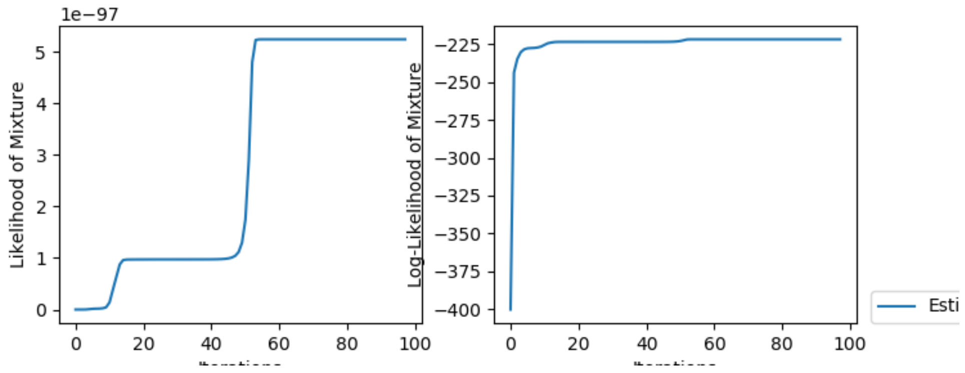

Figure 6: Likelihood of mixture model in case 3

> ***Question 2.5.17*** Simulate new tree mixtures with different number of nodes, samples and clusters. Try to find some interesting cases. Analyse your results as in the previous question.

To verify if my assumption in the previous question is correct, I would like to run my EM Algorithm with a small number of nodes and small number of samples to see if the RF distance will decrease and the likelihood decreases as well.

In the first case, there are 3 clusters, 5 nodes and 10 samples. Table 7 is the RF distance metric in this case and table 8 is the comparison of likelihood between real trees and inferred trees in this case and the log-likelihood is shown in figure 7.

	RT 1	RT 2	RT 3
IT 1	5	6	4
IT 2	3	0	4
IT 3	5	4	4

Table 7: RF metric of real and inferred trees in test case 1

	Real Trees	Inferred Trees
loglikelihood	-41.897	-21.640
likelihood	6.375e-19	4e-10

Table 8: The comparison of likelihood between real trees and inferred trees in test case 1

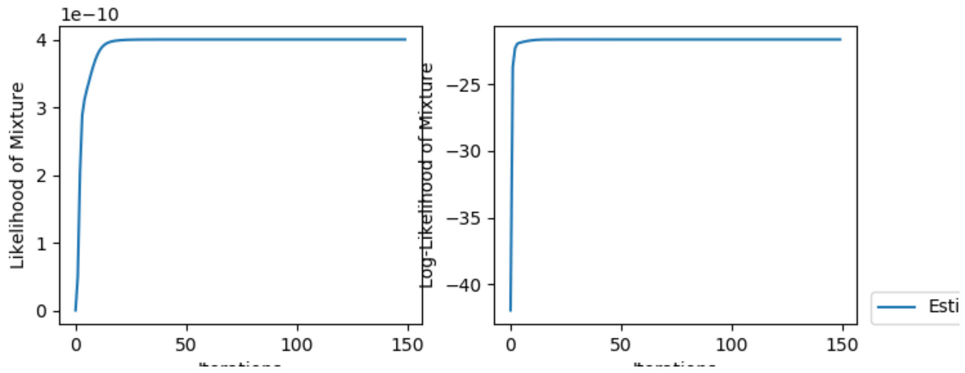

Figure 7: Likelihood of mixture model in test case 1

In this case, I have verified my assumption that when the number of nodes decrease, the RF distance decreases as well (obviously). Also when the number of sample decrease, the likelihood of the mixture model increases.

In the second case, there are 15 clusters, 10 nodes and 20 samples. I selected this case is because I would like to compare the difference when we have higher number of clusters in the inferred trees. The samples are the same as the first case in question 2.5.16. Compared to table 2, the log-likelihood of inferred trees are most likely the same as this case, as we can see in table 9, and the range of RF distance is most likely the same as well.

	Real Trees	Inferred Trees
loglikelihood	-113.143	-59.915
likelihood	7.287e-50	9.537e-27

Table 9: The comparison of likelihood between real trees and inferred trees in test case 2

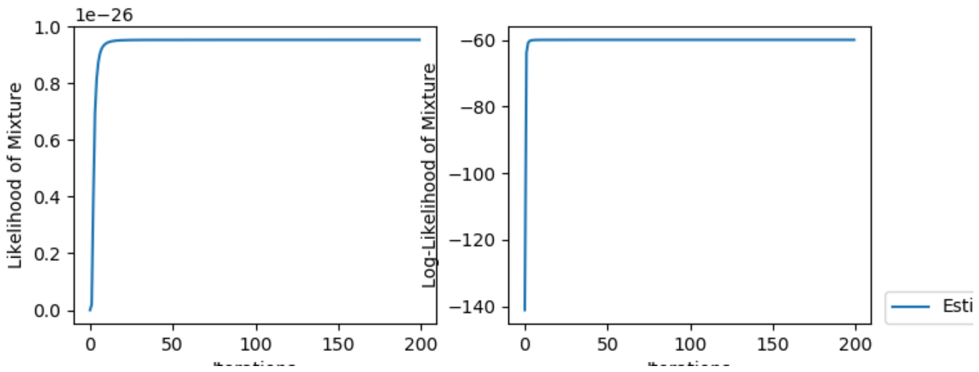

Figure 8: Likelihood of mixture model in test case 2

In the third case, I would like to continuing test different number of clusters in inferred trees and the real trees. In the third case, the number of clusters in inferred trees is smaller than the number of cluster in the real trees. Again, The same samples are used in this case. The number of clusters in inferred trees are 3 in this case.

	Real Trees	Inferred Trees
loglikelihood	-113.143	- -63.394
likelihood	7.287e-50	2.940e-28

Table 10: The comparison of likelihood between real trees and inferred trees in test case 3

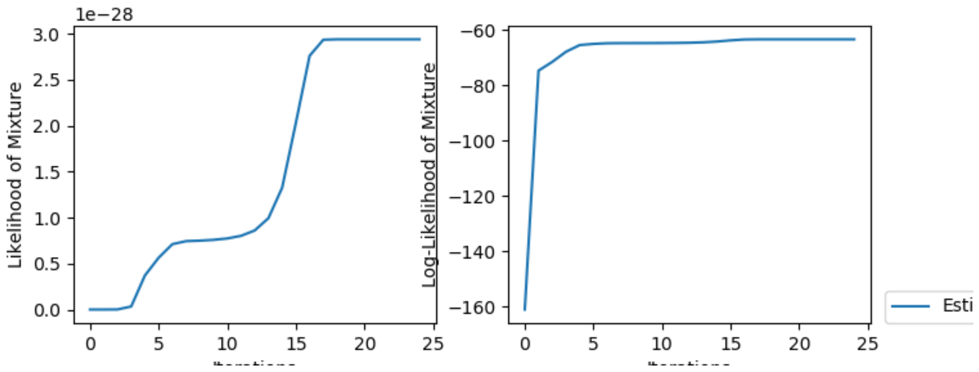

Figure 9: Likelihood of mixture model in test case 3

From table 10 we can see that if the number of clusters in inferred trees is smaller than the number of cluster in the real trees, the likelihood of the inferred trees will decrease. This makes sense since when the number of cluster in inferred tree is smaller, some information of cluster is merged into the other clusters, so Θ can not well represented the actual probabilities in samples. However, when the number of clusters in inferred trees is much larger than the number of cluster in the real trees, the model can give a relative small weight (π in our cases) to those trees that are much different than the real trees, so the likelihood of that mixture model is most likely the same in the case that the number of clusters in referred trees equals to that in real trees. Figure 10 we can see the log-likelihood with different numbers of clusters in inferred models, using the given samples *q_2_5_tm_10node_20sample_4clusters.pkl_samples.txt*.

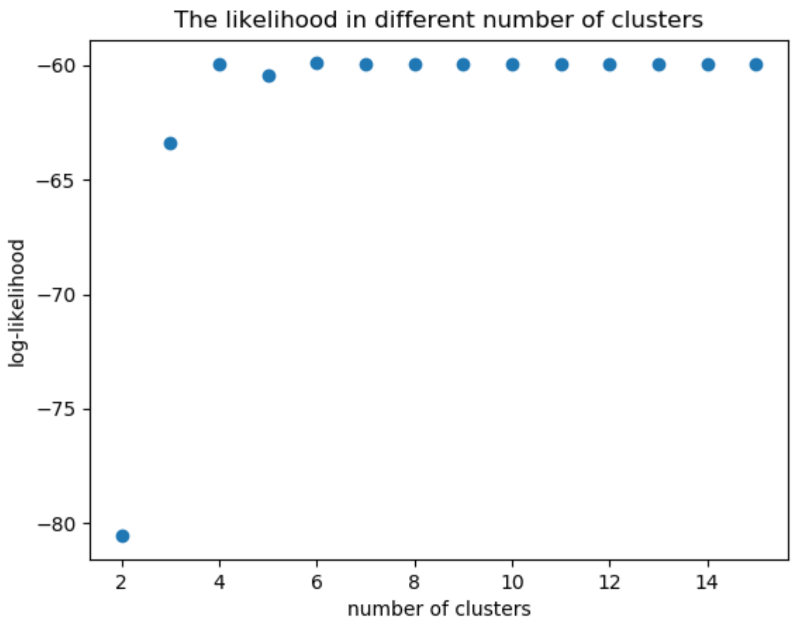

Figure 10: log-Likelihood with different number of clusters in inferred trees

6 Super epientra - EM

Question 2.6.18 Derive an EM algorithm for the model.

1. ***E-STEP***

   In this step, we should calculate the responsibility. the responsibility is defined as follows:

$$r_{n,k} = \frac{\pi_k P(X^n, S^n|Z_k^n)}{P(X^n, S^n)}$$

   where P(X^n,S^n|Z_k^n) is :

$$P(X^n, S^n|Z_k^n) = \mathcal{N}(X^n|\mu_k, \tau_k^{-1}) Poisson(s^n|\lambda_k)$$

   and P(X^n,S^n) is the marginalization of P(X^n,S^n|Z_k^n) :

$$P(X^n, S^n) = \sum_{k=1}^{K} P(X^n, S^n|Z_k^n)$$

   Then the new π becomes:

$$\pi_k = \sum_{n=1}^{N} \frac{r_{n,k}}{N}$$

2. ***M-STEP***

   In M-STEP, we should update μ, τ and λ for each super epicentra.

   μ_k will be :

$$\mu_k^1 = \frac{\sum_{n=0}^{N} r_{n,k} X_1^n}{n_k}$$

$$\mu_k^2 = \frac{\sum_{n=1}^{N} r_{n,k} X_2^n}{n_k}$$

   and τ_k will be :

$$\tau_k^1 = n_k \sum_{n=0}^{N} \frac{1}{r_{n,k}(X_1^n - \mu_k^1)(X_1^n - \mu_k^1)}$$

$$\tau_k^2 = n_k \sum_{n=0}^{N} \frac{1}{r_{n,k}(X_2^n - \mu_k^2)(X_2^n - \mu_k^2)}$$

   and λ_k [2]:

$$\lambda_k = \frac{\sum_{n=1}^{N} r_{n,k} S^n}{n_k}$$

   where:

$$n_k = \sum_{n=1}^{N} r_{n,k}$$

3. ***Log-likelihood***

   After E-STEP and M-STEP, we calculate the log-likelihood of the updated parameters :

$$logP(X, S|\mu, \tau, \pi, \lambda) = \sum_{n=1}^{N} log \sum_{k=1}^{K} r_{n,k}$$

4. ***Convergence***

   We check convergence by calculating the difference between previous likelihood and current likelihood, if the difference is larger than a very small number ϵ, then go back to step 1. If the difference is smaller enough, we stop iterating and return the latest parameters.

Question 2.6.19 Implement your EM algorithm.

- ***E-STEP***:

```
def _do_estep(self, X, S):
    """
    E-step
    """
    n_samples = self.n_row
    n_cluster = self.n_components
    self.r = np.zeros((n_samples,n_cluster))
    for n in range(n_samples):
        for k in range(n_cluster):
            self.r[n][k] = self.weights[k] * poisson.pmf(S[n], self.rates[k]) *
                multivariate_normal.pdf(X[n],mean = self.means[k],cov = self.covs[k])
            self.r[n][k] += sys.float_info.epsilon
    self.r = self.r / np.tile(np.sum(self.r , axis = 1).reshape((n_samples, 1)),n_cluster)
    ### update weight
    for k in range(n_cluster):
        self.weights[k] = np.sum(self.r[:,k])/n_samples



    return self
```

[2] Brandon Malone, Expectation-Maximization for Estimating Parameters for a Mixture of Poissons, Department of Computer Science, University of Helsinki

- ***M-STEP***:

```
def _do_mstep(self, X, S):
        """M-step, update parameters"""
        n_samples = self.n_row
        ### update mu
        for k in range(self.n_components):
            N = np.sum(self.r[:,k])
            self.means[k][0] = np.sum(self.r[:,k] * X[:,0]) / N
            self.means[k][1] = np.sum(self.r[:,k] * X[:,1]) / N
            self.rates[k] = np.sum(S * self.r[:,k]) / N
            dotx1 = (X[:,0] - self.means[k][0]).reshape(n_samples, 1)
            self.covs[k][0][0] = np.sum(dotx1**2 * self.r[:, k].reshape(n_samples, 1)) / N
            dotx2 = (X[:,1] - self.means[k][1]).reshape(n_samples, 1)
            self.covs[k][1][1] = np.sum(dotx2**2 * self.r[:, k].reshape(n_samples, 1)) / N

        return self
```

- ***log-likelihood***:

```
    def _compute_log_likelihood(self, X, S):
        """compute the log likelihood of the current parameter"""
        n_samples = self.n_row
        for n in range(n_samples):
            for k in range(self.n_components):
                self.r[n][k] = self.weights[k] * poisson.pmf(S[n], self.rates[k])\
                                              * multivariate_normal.pdf(X[n], self.means[k],
                                                  self.covs[k])
        log_likelihood = 0
        log_likelihood = np.sum(np.log(np.sum(self.r, axis=1)))

        return log_likelihood
```

Question 2.6.20 Apply it to the data provided separately, give an account of the success, and provide visualizations for a couple of examples.

- ***case 1: 3 clusters***:

  In this case, the EM algorithm converge after 17 iterations. Figure 11 and 12 show the cluster distribution and contour of case 1 after first iteration and after convergence respectively.

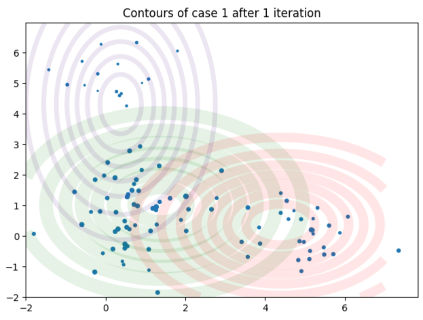

Figure 11: Contour of case 2 after first iteration

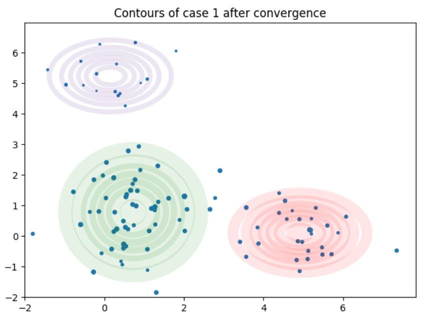

Figure 12: Contour of case 1 after 17th iteration

- ***case 1: 3 clusters***:

  In case 2, the EM-Algorithm converges after 9 iterations. The following figures show the cluster distribution and contour of case 1 after first iteration and after convergence respectively.

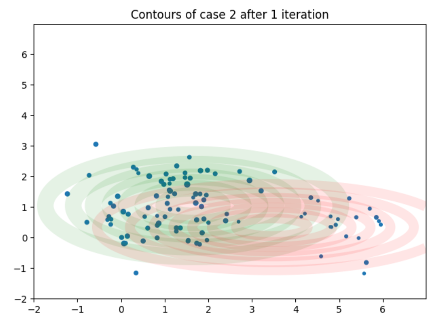

Figure 13: Contour of case 2 after first iteration

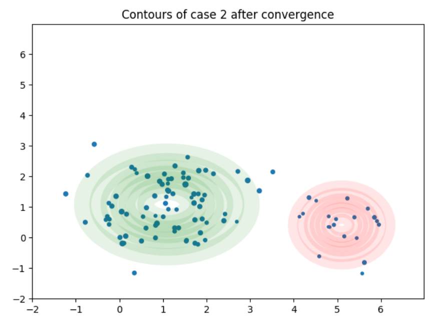

Figure 14: Contour of case 1 after 9th iteration

7 Super epicentra -VI

Question 2.7.21 Derive a VI algorithm that estimates the posterior distribution for this model.

According to the book *Bishop*, the VI assumption in this case can be written as:

$$q(Z,\pi,\mu_k,\tau_k,\lambda_k) = q(z)q(\pi,\mu,\tau)q(\lambda)$$

First of all, let derive q(z) :

$$\begin{aligned}
q(z) &= E_{\pi,\tau,\mu,\lambda}[ln(P(X,S,Z|\pi,\mu,\tau,\lambda)] \\
&= \sum_{n=0}^{N}\sum_{k=0}^{K}(Z_{n,k}(E_{\pi,\tau}[ln(\mathcal{N}(X_n|\mu_K,\tau_k))] + E_\lambda[ln(Poisson(S_n|\lambda_k)]) + E_\pi(ln(\pi_k)) \\
&= \sum_{n=0}^{N}\sum_{k=0}^{K} Z_{n,k}[-log2\pi + E_\tau(ln(\tau_k)) - E[\tau_k]E_\mu[(X_n-\mu_k)(X_n-\mu_k)^T] + E_\lambda[ln(Poisson(S_n|\lambda_k)])E_\pi[ln(\pi_k)] \\
&= \sum_{n=0}^{N}\sum_{k=0}^{K} Z_{n,k} ln\rho_{n,k}
\end{aligned}$$

so $q^*(z)$ becomes:

$$q^*(z) = \prod_{n=0,k=0}^{N,K} \gamma_{n,k}^{Z_{n,k}}$$

where:

$$\gamma_{n,k} = \frac{e^{\rho_{n,k}}}{\sum_{n=0}^{N} e^{\rho_{n,k}}}$$

Now, let's turn to q(π,μ,τ,λ):

$$\begin{aligned}
ln(q(\pi,\mu,\tau,\lambda)) &= E_z[ln(P(X,S|Z,\pi,\mu,\tau,\lambda)] + lnP(Z|\pi) + lnP(\pi) + \sum_{k=0}^{K} lnP(\mu_k,\tau_k) + \sum_{k=0}^{K} lnP(\lambda_k) \\
&= \sum_{n=0}^{N}\sum_{k=0}^{K} E_Z[z_{n,k}]ln(\mathcal{N}(X_n|\mu_k,\tau^{-1}) + \sum_{n=0}^{N}\sum_{k=0}^{K} E_z[z_{n,k}]ln((Poisson(S_n|\lambda_k)) \\
&+ E_z[lnP(Z|\pi)] + lnP(\pi) + \sum_{k=0}^{K} ln(P(\mu_k,\tau_k)) + \sum_{k=1}^{K} ln(P(\lambda_k))
\end{aligned}$$

The equation above can be decomposed to a sum of terms that only include π, terms only include μ and τ and terms only include λ. Therefore, q(π,μ,τ,λ) can be factorized as following:

$$q(\pi,\mu,\tau,\lambda) = q(\pi)\prod_{k=0}^{K} q(\mu_k,\tau_k)q(\lambda_k)$$

Taking the terms that only include π :

$$\begin{aligned} lnq(\pi) &= E_z[lnP(Z|\pi)] + lnP(\pi) \\ &= \sum_{n=0}^{N}\sum_{k=0}^{K} E_z[Z_{n,k}]ln\pi_k + (\alpha_k - 1)ln\pi_k \\ &= \sum_{k=0}^{K} ln\pi_k(\alpha_k - 1 + \sum_{n=0}^{N}\gamma_{n,k}) \end{aligned}$$

For the equation above, we know that $q(\pi)$ is a Dir(α^*), where α^* is updated to be :

$$\alpha_k^* = \alpha_k \sum_{n=0}^{N}\gamma_{n,k}$$

Now, Let's take a look on terms that only include λ_k:

$$\begin{aligned} lnq(\lambda) &= E_z[z_{n,k}]ln((Poisson(S_n|\lambda_k)) + \sum_{k=1}^{K} ln(P(\lambda_k)) \\ &= \sum_{n=0}^{N}\sum_{k=0}^{K} E_z[Z_{n,k}][(-\lambda_k) + S_n ln(\lambda_k) - ln(S_n!)] \\ &+ (\alpha_0 - 1)ln(\lambda_k) + \alpha_0 ln(\beta_0) - \beta_0\lambda_k + const \end{aligned}$$

For the equation above, we know that $q(\lambda)$ is a Gam($\lambda_k|\alpha_0^*$,β_0^*), where α_0^* is updated to be :

$$\alpha_0^* = \alpha_0 + \sum_{n=0}^{N}\gamma_{n,k}S_n$$

and β_0^* is updated to be :

$$\beta_0^* = \beta_0 + \sum_{n=0}^{N}\gamma_{n,k}$$

Then, we turn to terms that contains μ and τ:

$$\begin{aligned} lnq^*(\mu_k,\tau_k) &= lnq^*(\mu_k|\tau_k)q^*(\tau_k) \\ &= E_Z[z_{n,k}]ln(\mathcal{N}(X_n|\mu_k,\tau^{-1}) + \sum_{k=0}^{K} ln(P(\mu_k,\tau_k)) \\ &= \sum_{n=0}^{N}\gamma_{n,k}(ln\tau_k - \frac{\tau_k(X_n-\mu_k)^2}{2}) + ln\tau_k - \frac{C\tau_k}{2}(\mu_k-\mu)^2 + (\alpha^{'} - 1)ln\tau_k - \beta^{'}\tau_k \end{aligned}$$

From the equation above, we know that $q^*(\tau_k)$ is a Gam($\tau_k|\alpha^{'*}$,$\beta^{'*}$), and $q^*(\mu_k|\tau_k)$ is a $\mathcal{N}(\mu_k|\mu^*$,$\tau^*)$ where in Gam($\tau_k|\alpha^{'*}$,$\beta^{'*}$) $\alpha^{'*}$ can be updated to

$$\alpha_k^{'*} = \alpha_k^{'} + \sum_{n=0}^{N}\gamma_{n,k}$$

and $\beta^{'*}$ is updated to be

$$\beta^{'*} = \beta_k^{'} + \sum_{n=0}^{N}\gamma_{n,k}X_n^2 + \frac{C\mu^2}{2}$$

and in $\mathcal{N}(\mu_k|\mu^*$,$\tau^*)$, μ^* becomes:

$$\mu^* = \frac{C\tau_k\mu + \tau_k\sum_{n=0}^{N}\gamma_{n,k}X_n}{\tau^*}$$

and τ^* becomes:

$$\tau^* = C\tau_k + \tau_k\sum_{n=0}^{N}\gamma_{n,k}X_n$$

8 Sampling from a tree GM

Question 2.8.22 Derive these algorithms.

- ***DP Algorithm***:

  Here we would like to find a odd-sum output that all the sum of all leaf nodes is an odd number. Then we can formulate the problem into this form:

$$P(\sum_{l=0}^{L(T)} X_l = odd) = \sum_{k=0}^{K} S(X_{root} = k)$$

  where s(X_{root} = k) is the sum of the leave under root, given root is any possible k. To find s(X_{root} = k), we can then separate this problem into two different sub-problem:

  1. if s(X_{root} = k) is odd, then the children of root, let's say, A and B, will either be s($X_A|X_{root}$ = k) is odd and s($X_B|X_{root}$ = k) is even or s($X_A|X_{root}$ = k) is even and s($X_B|X_{root}$ = k) is odd.
  2. Moving on, if s($X_A|X_{root}$ = k) is even, then the the children of X_A, let's say, $X_{A_{c1}}$ and $X_{A_{c2}}$, will both s($X_{A_{c1}}|X_A$ = k) and s($X_{A_{c2}}|X_A$ = k) are odd or both s($X_{A_{c1}}|X_A$ = k) and s($X_{A_{c2}}|X_A$ = k) are even.

  Then we can find the conditional probability of sum of the leave under a node is an odd :

$$\begin{aligned} s(X_A = odd, k) &= P(X_A = odd|X_A = k) \\ &= (\sum_{i=0}^{K} s(X_{A_{c1}} = odd, i)P(X_{A_{c1}} = i|X_A = k))(\sum_{i=0}^{K} s(X_{A_{c2}} = even, i)P(X_{A_{c2}} = i|X_A = k)) \\ &+ (\sum_{i=0}^{K} s(X_{A_{c1}} = even, i)P(X_{A_{c1}} = i|X_A = k))(\sum_{i=0}^{K} s(X_{A_{c2}} = odd, i)P(X_{A_{c2}} = i|X_A = k)) \end{aligned}$$

  Given the definition of q , we can rewrite s($X_{node} = odd$,k) into a following form:

$$\begin{aligned} s(X_A = odd, k) &= P(X_A = odd|X_{root} = k) \\ &= (\sum_{i=0}^{K} Cs(X_{A_{c1}} = odd, i)P(X_{A_{c1}} = i|X_{root} = k))(\sum_{i=0}^{K} s(X_{A_{c2}} = even, i))P(X_{A_{c2}} = i|X_{root} = k)) \\ &+ (\sum_{i=0}^{K} (s(X_{A_{c1}} = even, i))P(X_{A_{c1}} = i|X_{root} = k))(\sum_{i=0}^{K} (Cs(X_{A_{c2}} = odd, i))P(X_{A_{c2}} = i|X_{root} = k)) \end{aligned}$$

  where $s(X_{A_{c1}} = even, i)$ is 1 - $s(X_{A_{c1}} = odd, i)$ and $s(X_{A_{c2}} = even, i)$ is 1 - $s(X_{A_{c2}} = odd, i)$.

  If $s(X_A = even, k)$ then the the conditional probability of sum of the leave under a node is an even can be written as :

$$\begin{aligned} P(X_A = even, K) &= P(X_A = even|X_{root} = k) \\ &= (\sum_{i=0}^{K} (s(X_{A_{c1}} = even, i))P(X_{A_{c1}} = i|X_{root} = k))(\sum_{i=0}^{K} (s(X_{A_{c2}} = even, i))P(X_{A_{c2}} = i|X_{root} = k)) \\ &+ (\sum_{i=0}^{K} (s(X_{A_{c1}} = odd, i))P(X_{A_{c1}} = i|X_{root} = k))(\sum_{i=0}^{K} (s(X_{A_{c2}} = odd, i))P(X_{A_{c2}} = i|X_{root} = k)) \end{aligned}$$

  If A_{c_1} is a leaf, then $s(A_{c_1}, k)$ will be :

$$s(A_{c_1}, X_A) = P(A_{c_1} = odd|X_A = k) = \begin{cases} 0 \ if \ X_E = even \\ 1 \ if \ X_E = odd \end{cases}$$

- ***Sampling Algorithm*** For sampling, I did the iteration from root to leaves. First of all, I sample the root with the distribution that is calculated as below:

$$P(X_{root} = k) = s(root = odd, k)P(X_{root} = k)$$

Later on, for each children of the root, we will have two distribution of sampling of the children because child 1 can be either odd or even. If we say child 1 is odd, then child 2 is even, and vice versa. Therefore, to give an equal probability for both possible cases, I set the two possible cases in equal probabilities, that is 0.5 for each.

Let's say if child 1 is odd and child 2 is even, then the distribution for sampling child 1 will be :

$$P(X_{A_{c1}} = i, s(X_{Ac1} = even)) = Cs(X_{Ac1} = odd, i)P(X_{Ac1} = i|X_{root} = X_0)$$

and child 2 will be :

$$P(X_{A_{c1}} = i, s(X_{Ac2} = even)) = s(X_{Ac2} = even, i)P(X_{Ac2} = i|X_{root} = X_0)$$

Where X_0 is what we have sampled for the root and C is the constant.

After we have sampled the children of the root, then we are going to sample the children of the child 1 and child 2, the process is the same, let's say the children of the current node is A and B, we will set equal probabilities among possible cases for A and B, then do the same iteration as above.

Question 2.8.23 Implement your bottom up DP algorithm for the probability of generating an odd sum output.

```python
def tree_DP(tree_topology, theta,node):
    # TODO: Implement algorithm for dynamic programming
    #likelihood_odd1 = calculate_s(tree_topology, theta, node, 1)
    c = 1.2
    likelihood_odd = calculate_s(tree_topology, theta, node, 0)
    #likelihood = likelihood_odd + likelihood_odd1
    return  likelihood_odd / np.sum(likelihood_odd)

def calculate_s(tree_topology, theta, node,even):
    children = find_children(tree_topology, node)
    c = 10
    ### if current node is even and it is leave
    k = theta.shape[1]
    if len(children) == 0:
        s = np.zeros((k, 1))
        if even == 1:
            for i in range(k):
                if i % 2 == 0:
                    s[i] = 1
                else:
                    s[i] = 0
        else:
            for i in range(k):
                if i % 2 == 1:
                    s[i] = c
                else:
                    s[i] = 0
        return s / np.sum(s)
    else:

        theta1 = theta[children[0]]
        theta2 = theta[children[1]]
        theta1 = np.array(theta1.tolist())
        theta2 = np.array(theta2.tolist())
        if even == 1: ## 2 for even or 2 for odd
            s1 = calculate_s(tree_topology, theta, children[0], 0)
            s2 = calculate_s(tree_topology, theta, children[1], 0)
            likelihood = (1 - np.dot(theta1, s1)) * (1 - np.dot(theta2, s2))
            s1 = calculate_s(tree_topology, theta, children[0], 0)
            s2 = calculate_s(tree_topology, theta, children[1], 0)
            likelihood += c * np.dot(theta1, s1) * c * np.dot(theta2, s2)
        else:  ## 1 even 1 odd
            s1 = calculate_s(tree_topology, theta, children[0], 0)
            s2 = calculate_s(tree_topology, theta, children[1], 0)
            likelihood = c * np.dot(theta1, s1) * (1 - np.dot(theta2, s2))
            s1 = calculate_s(tree_topology, theta, children[0], 0)
            s2 = calculate_s(tree_topology, theta, children[1], 0)
            likelihood += (1 - np.dot(theta1, s1)) * c * np.dot(theta2, s2)
        likelihood = likelihood / np.sum(likelihood)
        return likelihood


def find_children(tree_topology, node):
    children = []
    children = np.argwhere(tree_topology == node)
    if len(children) == 0:
        return []
    else:
        children = children.reshape((2,))

        return list(children)
```

Question 2.8.24 Implement your sampling algorithm.

```
def odd_sum_sampling(topology, theta):
    num_nodes = len(topology)
    samples = [0] * num_nodes
    filtered_samples = {}
    cur_sample = []
    visit_list = [0]
    here = []
    c = 1
    while len(visit_list) != 0:
        cur_node = visit_list[0]
        children = find_children(topology, cur_node)
        visit_list = visit_list[1:] + children
        par_node = topology[cur_node]
        if len(children) == 0:
            filtered_samples[str(cur_node)] = samples[int(cur_node)]
        else:
            if cur_node == 0:
                a = tree_DP(topology,theta,0)
                # b = np.array(theta[0].reshape(5,1))
                cat = a * np.array(theta[0]).reshape(1,5)
                cat = cat[0]

                cat = cat / np.sum(cat)

                cur_sample = np.random.choice(np.arange(5), p = cat.tolist())
                samples[int(cur_node)] = cur_sample
            else:
                cur_sample = samples[int(cur_node)]
            if cur_sample % 2 == 0: ### even
                test = np.random.randint(2)
                if test == 1:

                    cat1 = c * calculate_s(topology, theta, children[0], 0) *
                        theta[children[0]][cur_sample].reshape(5,1)
                    cat2 = c * calculate_s(topology, theta, children[1], 0) *
                        theta[children[1]][cur_sample].reshape(5,1)
                else:
                    cat1 = (1 - calculate_s(topology, theta, children[0], 0)) *
                        theta[children[0]][cur_sample].reshape(5,1)
                    cat2 = (1 - calculate_s(topology, theta, children[1], 0)) *
                        theta[children[1]][cur_sample].reshape(5,1)
                cat1 = cat1.reshape(1, 5) / np.sum(cat1)
                cat2 = cat2.reshape(1, 5) / np.sum(cat2)
                cat1 = cat1[0]
                cat2 = cat2[0]
                cur_sample = np.random.choice(np.arange(5), p = cat1.tolist())
                samples[int(children[0])] = cur_sample
                cur_sample = np.random.choice(np.arange(5), p = cat2.tolist())
                samples[int(children[1])] = cur_sample
            else:
                test = np.random.randint(2)
                if test == 1:
                    cat1 = c * calculate_s(topology, theta, children[0], 0) *
                        theta[children[0]][cur_sample].reshape(5,1)
                    cat2 = (1 - calculate_s(topology, theta, children[1], 0)) *
                        theta[children[1]][cur_sample].reshape(5,1)
                else:
                    cat1 = (1 - calculate_s(topology, theta, children[0], 0)) *
                        theta[children[0]][cur_sample].reshape(5,1)
                    cat2 = c * calculate_s(topology, theta, children[1], 0) *
                        theta[children[1]][cur_sample].reshape(5,1)
                cat1 = cat1.reshape(1, 5) / np.sum(cat1)
                cat2 = cat2.reshape(1, 5) / np.sum(cat2)
                cat1 = cat1[0]
                cat2 = cat2[0]
                cur_sample = np.random.choice(np.arange(5), p = cat1.tolist())
                samples[int(children[0])] = cur_sample
                cur_sample = np.random.choice(np.arange(5), p=cat2.tolist())
                samples[int(children[1])] = cur_sample
        return filtered_samples
```

Question 2.8.25 Apply your algorithm to the graphical model and data provided separately

The result will become:

```
leaf_samples {'7': 3, '19': 1, '20': 4, '12': 1, '9': 0, '17': 1, '18': 1, '15': 1, '16': 0, '13': 3,
    '14': 2}
sum:  17
leaf_samples {'7': 0, '19': 1, '20': 1, '12': 4, '9': 3, '17': 1, '18': 0, '15': 0, '16': 3, '13': 4,
    '14': 3}
sum:  20
leaf_samples {'7': 1, '19': 3, '20': 4, '12': 3, '9': 2, '17': 3, '18': 3, '15': 3, '16': 0, '13': 2,
    '14': 4}
sum:  28
leaf_samples {'7': 2, '19': 2, '20': 1, '12': 3, '9': 3, '17': 4, '18': 1, '15': 1, '16': 1, '13': 3,
    '14': 3}
sum:  24
leaf_samples {'7': 0, '19': 1, '20': 1, '12': 0, '9': 1, '17': 0, '18': 1, '15': 1, '16': 1, '13': 3,
    '14': 4}
sum:  13
leaf_samples {'7': 4, '19': 0, '20': 4, '12': 1, '9': 4, '17': 0, '18': 2, '15': 1, '16': 0, '13': 0,
    '14': 3}
sum:  19
leaf_samples {'7': 1, '19': 1, '20': 1, '12': 4, '9': 1, '17': 4, '18': 3, '15': 1, '16': 1, '13': 1,
    '14': 1}
sum:  19
leaf_samples {'7': 1, '19': 3, '20': 3, '12': 1, '9': 2, '17': 2, '18': 2, '15': 3, '16': 1, '13': 2,
    '14': 1}
sum:  21
leaf_samples {'7': 3, '19': 1, '20': 0, '12': 0, '9': 1, '17': 4, '18': 1, '15': 4, '16': 4, '13': 0,
    '14': 2}
sum:  20
leaf_samples {'7': 3, '19': 3, '20': 1, '12': 1, '9': 1, '17': 3, '18': 1, '15': 1, '16': 0, '13': 1,
    '14': 3}
sum:  18
leaf_samples {'7': 0, '19': 1, '20': 1, '12': 4, '9': 3, '17': 3, '18': 4, '15': 1, '16': 1, '13': 0,
    '14': 1}
sum:  19
```

```
leaf_samples {'7': 2, '19': 1, '20': 1, '12': 1, '9': 4, '17': 0, '18': 1, '15': 1, '16': 3, '13': 1,
    '14': 3}
sum:   18
leaf_samples {'7': 2, '19': 3, '20': 3, '12': 4, '9': 0, '17': 4, '18': 2, '15': 1, '16': 0, '13': 4,
    '14': 0}
sum:   23
leaf_samples {'7': 3, '19': 2, '20': 4, '12': 1, '9': 1, '17': 3, '18': 3, '15': 4, '16': 1, '13': 3,
    '14': 4}
sum:   29
leaf_samples {'7': 3, '19': 3, '20': 1, '12': 1, '9': 3, '17': 3, '18': 0, '15': 0, '16': 0, '13': 0,
    '14': 1}
sum:   15
leaf_samples {'7': 4, '19': 4, '20': 3, '12': 1, '9': 3, '17': 3, '18': 2, '15': 1, '16': 1, '13': 3,
    '14': 1}
sum:   26
odd_sum_ratio: 0.492
sample ratio: 0.277
node ratio: {'7': 0.44000000000000034, '19': 0.35000000000000026, '20': 0.44500000000000034, '12':
    0.6540000000000005, '9': 0.43700000000000033, '17': 0.5580000000000004, '18':
    0.43600000000000033, '15': 0.19500000000000015, '16': 0.5200000000000004, '13':
    0.6350000000000005, '14': 0.4040000000000003}
```

9 Failing components VI

Question 2.9.26 Derive a VI algorithm that estimates the posterior distribution for this model