Machine Learning, advanced course

Assignment 2

- *Author* -
Nadir Ait Lahmouch

December 22, 2020

Contents

2.1 Dependencies in a Directed Graphical Model **2**

2.2 Likelihood of a Tree Graphical Model **3**

2.3 Simple Variational Inference **4**

2.4 Mixture of trees with observable variables **7**

Appendix A **12**

Appendix B **14**

Appendix C **17**

2.1 Dependencies in a Directed Graphical Model

Question 2.1.1

No. Because X is observed.

Question 2.1.2

No.

Question 2.1.3

$A = \{\mu_{r,c}\}$.

Question 2.1.4

No.

Question 2.1.5

No.

Question 2.1.6

$B = \{Z_n^m; n, m \in [N] \times [M]\} \cup \{C^n; n \in [N]\}$

2.2 Likelihood of a Tree Graphical Model

Question 2.2.7

The prior can be seen as follows :

$$p(\beta \mid T, \Theta) = \sum_i s(R, i) p\left(X_R = i\right)$$

Where R is the root of the tree and $s(R, i)$ is the smaller. We have:

$$s(v, i) = p\left(x_{\downarrow v \cap \beta} \mid X_v = i\right)$$

Where $\downarrow v$ denotes all nodes below the root of sub-tree v.
Let v be a node in the tree (not a leaf) and let c_1 and c_2 be its children. We then have:

$$s(v, i) = \left(\sum_{j=1}^{K} p\left(X_{c_1} = j \mid X_v = i\right) s(c_1, j)\right)\left(\sum_{j=1}^{K} p\left(X_{c_2} = j \mid X_v = i\right) s(c_2, j)\right)$$

The children are independent given their parent.
But if v is a leaf, then:

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

We can code this function recursively using a hash map to store the small values each time we compute them. To see the code, look at Appendix A.

Question 2.2.8

Here's the results when we apply the code in appendix A to the provided data.

	Small Tree	Medium Tree	Large Tree
Sample 0	0.016178983188381856	4.3359985610595595e − 18	3.287622233372845e − 69
Sample 1	0.015409920999590558	3.094115541974649e − 20	1.109451841712131e − 66
Sample 2	0.011368474549717017	1.050009120509836e − 16	2.5224240937554143e − 68
Sample 3	0.00864042722338585	6.585311240142626e − 16	1.2423555476116806e − 66
Sample 4	0.04091494618599329	1.4880108219386272e − 18	3.535477501915256e − 69

2.3 Simple Variational Inference

Question 2.3.9

From Bishop equation 10.23. We saw that can approximate the exact posterior with a factorized variational approximation to the posterior distribution given by:

$$q(\mu, \tau) = q_\mu(\mu) q_\tau(\tau)$$

With $q_\mu(\mu)$ is a Gaussian $\mathcal{N}(\mu|\mu_N, \lambda_N^{-1})$ and $q_\tau(\tau)$ is a Gamma distribution $Gam(\tau|a_N, b_N)$. These factors are derived by inferring the four hyperparameters $\mu_N, \lambda_N^{-1}, a_N, b_N$ iteratively from the equations (10.26), (10.27), (10.29) and (10.30).
You can find the code in the Appendix B.

Question 2.3.10

Let's find the exact posterior.
Using Bayes equation, the exact posterior can be obtained as the following:

$$\begin{aligned} p(\mu, \tau|D) &= \frac{p(\mu, \tau)p(D|\mu, \tau)}{p(D)} && (1) \\ &= \frac{p(\tau)p(\mu|\tau)p(D|\mu, \tau)}{p(D)} && (2) \\ &\propto p(\tau)p(\mu|\tau)p(D|\mu, \tau) && (3) \end{aligned}$$

And we know from (10.22) and (10.23) that:

$$\begin{gathered} p(\tau) = Gam(\tau|a_0, b_0) \\ p(\mu|\tau) = \mathcal{N}(\mu|\mu_0, (\lambda_0\tau)^{-1}) \\ p(D \mid \mu, \tau) = \left(\tfrac{\tau}{2\pi}\right)^{N/2} \exp\left\{-\tfrac{\tau}{2}\sum_{n=1}^{N}(x_n - \mu)^2\right\} \end{gathered}$$

The derivation gives:

$$p(\mu,\tau \mid \mathcal{D}) \propto \tau^{N/2} \exp\left[-\frac{\tau}{2}\left(\sum_{n=1}^{N}(x_n-\bar{x})^2+N(\bar{x}-\mu)^2\right)\right]\tau^{a_0-\frac{1}{2}}\exp\left[-b_0\tau\right]\exp\left[-\frac{\lambda\tau\left(\mu-\mu_0\right)^2}{2}\right]$$

$$\propto \tau^{\frac{N}{2}}+a_0-\frac{1}{2}\exp\left[-\tau\left(\frac{1}{2}\sum_{n=1}^{N}(x_n-\bar{x})^2+b_0\right)\right]\exp\left[-\frac{\tau}{2}\left(\lambda_0\left(\mu-\mu_0\right)^2+N(\bar{x}-\mu)^2\right)\right]$$

$$\propto \tau^{\frac{N}{2}+a_0-\frac{1}{2}}\exp\left[-\tau\left(\frac{1}{2}\sum_{n=1}^{N}(x_n-\bar{x})^2+b_0\right)\right]\exp\left[-\frac{\tau}{2}\left((\lambda_0+N)\left(\mu-\frac{\lambda_0\mu_0+N\bar{x}}{\lambda_0+N}\right)^2+\frac{\lambda_0 N\left(\bar{x}-\mu_0\right)^2}{\lambda_0+N}\right)\right]$$

$$\propto \tau^{\frac{N}{2}+a_0-\frac{1}{2}}\exp\left[-\tau\left(\frac{1}{2}\sum_{n=1}^{N}(x_n-\bar{x})^2+b_0+\frac{\lambda_0 N\left(\bar{x}-\mu_0\right)^2}{2\left(\lambda_0+N\right)}\right)\right]\exp\left[-\frac{\tau}{2}\left(\lambda_0+N\right)\left(\mu-\frac{\lambda_0\mu_0+N\bar{x}}{\lambda_0+N}\right)^2\right]$$

We then reach the following result:

$$p(\mu,\tau \mid D) \propto \text{NormalGamma}(\mu_N,\lambda_N,a_N,b_N)$$

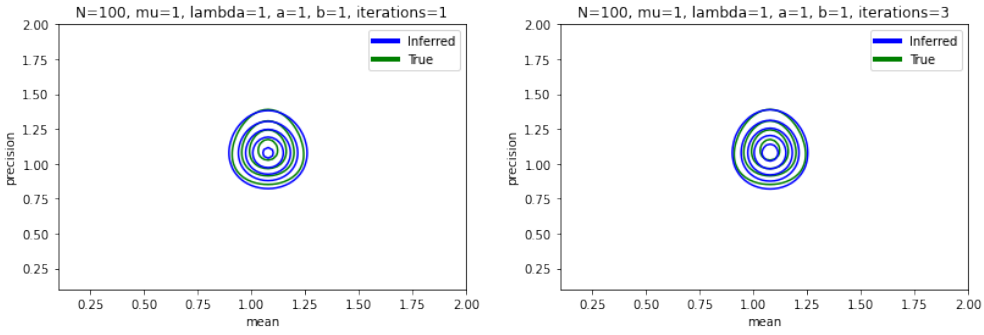
Therefore, $p(\mu,\tau \mid D)$ is Gaussian-gamma, where

$$\mu_N=\tfrac{\lambda_0\mu_0+N\bar{x}}{\lambda_0+N},\quad \lambda_N=\lambda_0+N,\quad a_N=a_0+N/2,$$
$$b_N=b_0+\tfrac{1}{2}(\textstyle\sum_{n=1}^{N}(x_n-\bar{x})^2+\tfrac{\lambda_0 N(\bar{x}-\mu_0)^2}{2(\lambda_0+N)})$$

Question 2.3.11

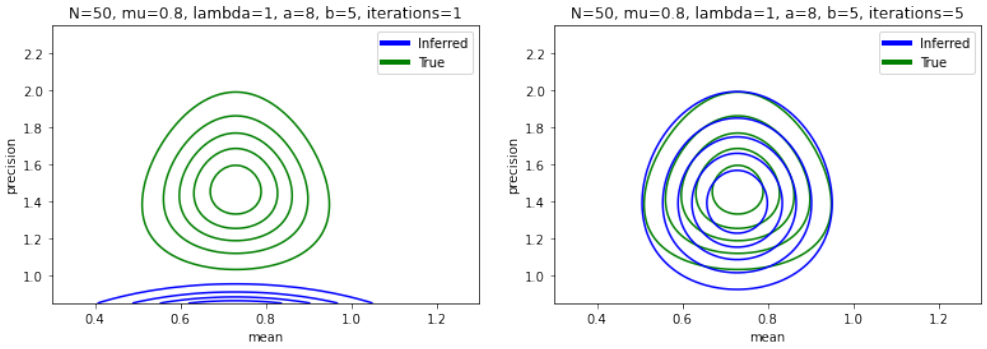
In all the following cases, we set $\mu_0=0,\lambda_0=0,alpha=0,beta=0$ and $Etau=1$.

In the first case, we choose $N=100,\mu=1,\lambda=1,alpha=1,beta=1$:



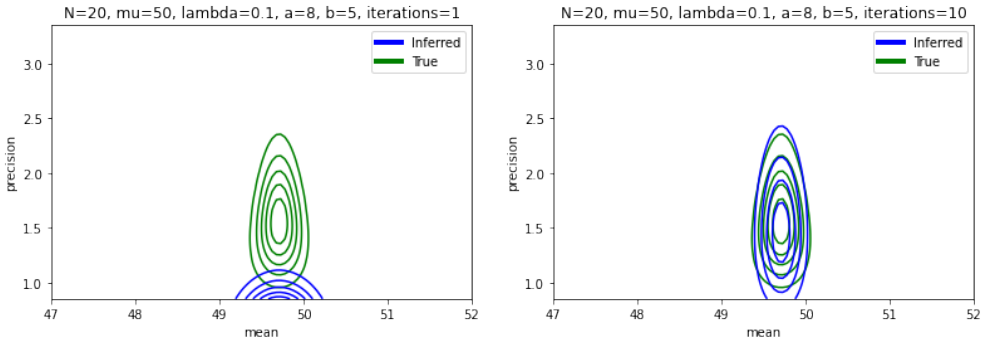
We quickly converge in this first case because the initial value of Etau (a/b) is exactly equal to the value using the hyperparameters of the true data.

Now, for the second case, $N = 50, \mu = 0.8, \lambda = 1, alpha = 8, beta = 5$:



The convergence is now slower when we set the true parameters different that the initial initialization of Etau. Also, we can notice that the shape of the distribution changes when we decrease the number of data points N, the shape is getting triangular when we decrease N.

For the third case, $N = 20, \mu = 50, \lambda = 0.8, alpha = 8, beta = 5$:



We continue to decrease N in this case, an as we said in the previous one, the shape seem to get more and more distorted to a triangular shape when we decrease N. The convergence is slower here too because the initialization parameters in the VI algorithm are way off from the true parameters.

However, and in all the above cases, we notice that they converge to match

the exact posterior with each iteration being better than the previous one.

2.4 Mixture of trees with observable variables

Question 2.4.12

To compute the responsibilities we have this expression:

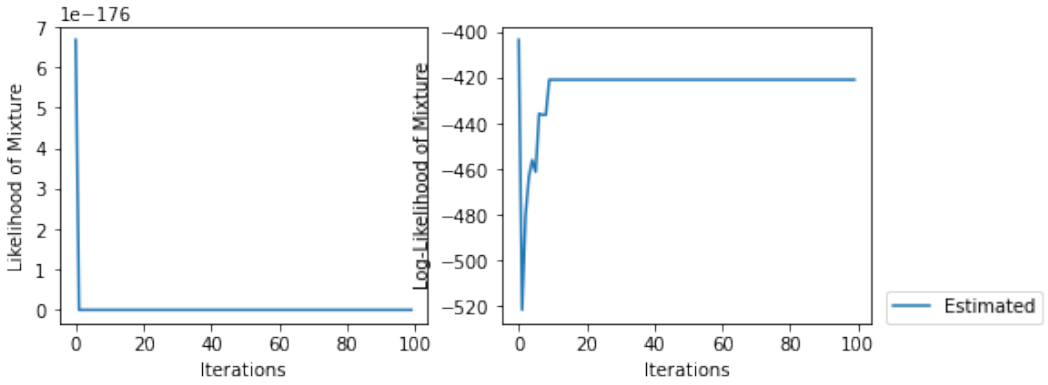
$$r_{n,k} = \frac{\pi_k p\left(x^n \mid T_k, \Theta_k\right)}{p\left(x^n\right)}$$

with $p\left(x^n\right) = \sum_{k=1}^{K} \pi_k p\left(x^n \mid T_k, \Theta_k\right)$.

The rest of the calculations are direct. You can find the code in Appendix C.

Question 2.4.13

We only have on set of data, the file **q2_4_tree_mixture.pkl**. Here's the result for it:



And this is the table of the Robinson-Foulds metric of the inferred vs true trees:

inferred trees \true trees	0	1	2
0	0	3	4
1	5	5	4
2	4	5	4

In the table above, the columns are the true trees and the rows are the inferred trees. From our observations, the inferred and true trees don't have a similar structure.
Let's now compare the likelihoods. Our results are:

True likelihood = -311.449480
Likelihood of inferred mixtures = -420.839474

The high difference between the two values approve what we said above about the inferred and true trees not having a similar structure.

Question 2.4.14

I simulated new tree mixtures by controlling the parameters N the number of samples, the number of clusters and the number of nodes. For 100, 500, 1000 samples, I computed the difference between the likelihoods of real and inferred mixtures for 3, 6, 10 clusters and 3, 7, 15 nodes.
100 samples:

nodes \clusters	3	6	10
3	22.63	2.85	2.31
7	268.10	157.00	102.45
15	681.68	536.40	411.80

500 samples:

nodes \clusters	3	6	10
3	124.16	3.76	5.71
7	1372.8	1084.94	891.26
15	3970.25	3471.23	3130.10

1000 samples:

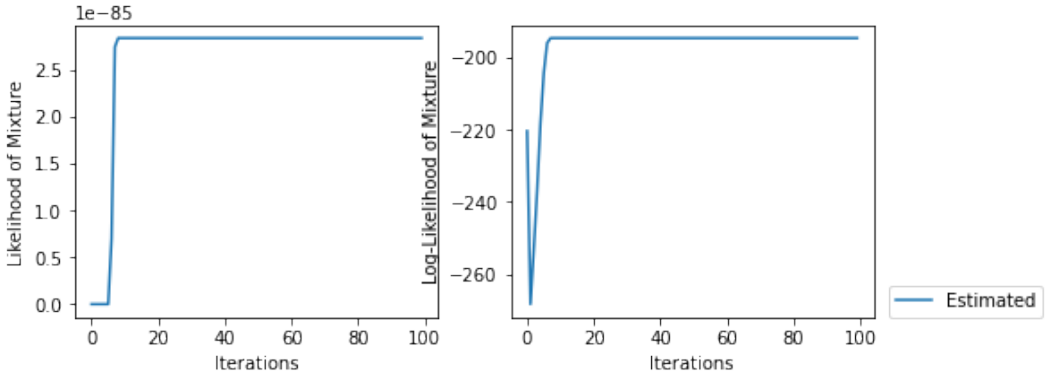
nodes \clusters	3	6	10
3	243.79	2.17	5.95
7	2753.25	2120.99	1607.94
15	8122.24	7285.28	6995.45

It seems from the tables above that the difference increases with the number of nodes and that's normal as the true tree gets more complex.
However, it seems like the performance of the EM algorithm doesn't improve when when we increase the number of samples.

Two interesting cases are:

1st case : 100 samples, 10 clusters, 3 nodes
2nd case : 1000 samples, 6 clusters, 3 nodes

First case: 100 samples, 10 clusters, 3 nodes



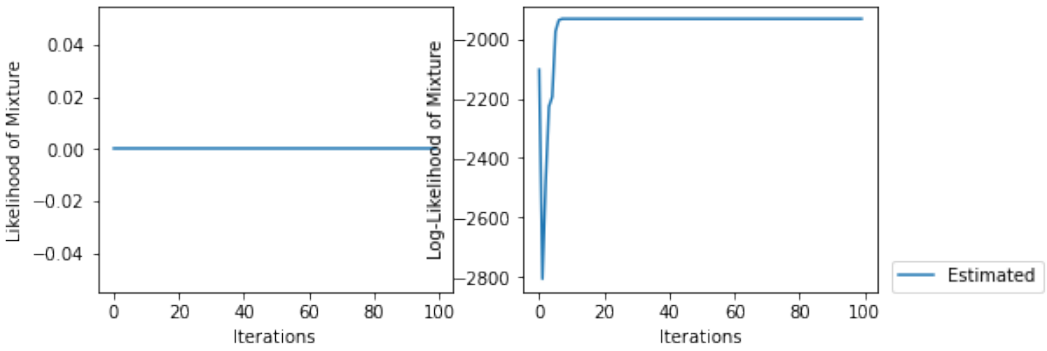
Here's difference between the true and inferred trees using the RF metric:

inferred trees \true trees	0	1	2	3	4	5	6	7	8	9
0	2	2	0	0	2	0	2	2	0	2
1	0	0	2	2	0	2	0	0	2	0
2	2	2	0	0	2	0	2	2	0	2
3	2	2	0	0	2	0	2	2	0	2
4	2	2	0	0	2	0	2	2	0	2
5	2	2	0	0	2	0	2	2	0	2
6	2	2	0	0	2	0	2	2	0	2
7	2	2	0	0	2	0	2	2	0	2
8	2	2	0	0	2	0	2	2	0	2
9	0	0	2	2	0	2	0	0	2	0

Let's now compare the likelihoods:

True likelihood = -196.988982
Likelihood of inferred mixtures = -194.677510

Second case: 1000 samples, 6 clusters, 3 nodes



And here's difference between the true and inferred trees using the RF metric:

inferred trees \true trees	0	1	2	3	4	5
0	2	2	2	2	2	2
1	2	2	0	0	2	0
2	0	0	2	2	0	2
3	0	0	2	2	0	2
4	2	2	0	0	2	0
5	2	2	0	0	2	0

Let's now compare the likelihoods:

True likelihood = -1932.803153
Likelihood of inferred mixtures = -1930.627744

Compared to the given data, this two simulations lead to a similar structure of the trees (the RF distances are less important). Also, there's way less difference between the true likelihood and the one of inferred mixtures. And the reason for that is, the true tree isn't complex (only 3 nodes) so it's easier to infer in this case.

Appendix A

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


def childrenOf(node, topology):
    """
    This function return the children to the node
    given as parameter
    """
    children = []
    for index, parent in enumerate(topology):
        if parent == node:
            children.append(index)
    return children



def CPD(theta, node, k, parent_cat=None):
    """
    This function return the value of CPD
    given the node, its value and the parent node
    """
    if parent_cat is None:
        return theta[node][k]
    else:
        return theta[node][parent_cat][k]
```

```
def calculate_likelihood(tree_topology, theta, beta):
    print("----------------------------------------")
    print("Calculating the likelihood...")
    s_dict = defaultdict(dict)
    #-------------------------------------------------
    def S(node, k, children):
      """
      Compute the small value for a node
      """
      if s_dict[node].get(k) is not None:
        return s_dict[node].get(k)
      if len(children) == 0:
        if beta[node] == k:
          s_dict[node][k] = 1
          return 1
        else:
          s_dict[node][k] = 0
          return 0

      children_1 = 0
      children_2 = 0

      N = len(theta[0])
      for j in range(N):
        children_1 += S(children[0], j, childrenOf(children[0], tree_topology))
                    * CPD(theta, children[0], j, k)
        children_2 += S(children[1], j, childrenOf(children[1], tree_topology))
                    * CPD(theta, children[1], j, k)

      s_final = children_1 * children_2
      s_dict[node][k] = s_final

      return s_final
      #-------------------------------------------------
    likelihood = 0
    N = len(theta[0])
    for k in range(N):
      likelihood += S(0, k, childrenOf(0 , tree_topology)) * CPD(theta, 0, k)
    return likelihood
```

Appendix B

```
import numpy as np
"""
For the functions below, X is our data
and N is the length ouf this data
"""

def calc_an(a0, N):
  return a0 + N/2

def calc_bn(b0, e_mu):
  return b0 + e_mu

def calc_mun(lambda_0, mu_0, N, X):
  return (lambda_0*mu_0 + N*np.average(X))/(lambda_0 + N)

def calc_lambdan(lambda_0, Etau, N):
  return (lambda_0 + N)*Etau

def e_mu(lambda_n, mu_n, X, mu_0, lambda_0):
  # E(mu**2) = Var(mu) +(E(mu))**2
  e_mu2 = 1/lambda_n + mu_n**2
  #E(cA + B) = cE(A) + E(B)
  factor_1 = np.sum(X**2 - 2*X*mu_n + e_mu2)
  factor_2 = lambda_0 * (e_mu2 - 2 * mu_n * mu_0 + mu_0**2)
  return (1/2)*(factor_1+factor_2)


def variational_inference(iterations, EtauG, lambda_0, mu_0, a0, b0, X):
  N = len(X)
  Etau = EtauG
  for i in range(iterations):
    mu_n = calc_mun(lambda_0, mu_0, N, X)
    lambda_n = calc_lambdan(lambda_0, Etau, N)
    a_n = calc_an(a0, N)
    b_n = calc_bn(b0, e_mu(lambda_n, mu_n, X, mu_0, lambda_0))
    Etau = a_n/b_n
  return mu_n, lambda_n, a_n, b_n
```

Generate the data, compute densities and plot functions

```
import numpy as np
import matplotlib.pyplot as plt
import scipy.stats as stats
from matplotlib.lines import Line2D
from mpl_toolkits.mplot3d import Axes3D


# Generate a random gaussian distribution
def gen_data(mean, sd, N):
  return np.random.normal(mean, sd, N)

# Densities functions to compute the probabilities
def pdf_norm(x, mean, precision):
    return stats.norm.pdf(x, mean, np.sqrt(1/precision))

def pdf_gamma(tau, a, b):
    return stats.gamma.pdf(tau, a, loc=0, scale=(1 / b))


def likelihood(X, mu, tau):
    """
    Compute the true likelihood
    """
    N = len(X)
    D = (tau / (2 * np.pi)) ** (N / 2) * np.exp(-(tau / 2)
        * np.sum((X - mu) ** 2))
    return D


def plot_function(X, vi_parameters=None):
    if vi_parameters != None:
      m_i, lamb_i, a_i, b_i = vi_parameters[0],
      vi_parameters[1], vi_parameters[2], vi_parameters[3]

    mus = np.linspace(mean - 0.5, mean + 0.5, 100)
    taus = np.linspace(precision - 1, precision + 1, 100)
    M, T = np.meshgrid(mus, taus, indexing="ij")
    C = np.zeros_like(M)
```

```
if vi_parameters != None:
    #Plot the inferred posterior
  for i in range(C.shape[0]):
    for j in range(C.shape[1]):
      C[i][j] = pdf_norm(mus[i], m_i, lamb_i) * pdf_gamma(taus[j], a_i, b_i)
  plt.contour(M, T, C, 5, colors='blue')
else:
    #Plot the exact posterior
  for i in range(C.shape[0]):
    for j in range(C.shape[1]):
      C[i][j] = pdf_norm(mus[i], mean, lamb*taus[j])
              * pdf_gamma(taus[j], a, b)
              * likelihood(X, mus[i], taus[j])
  plt.contour(M, T, Z, 5, colors='green')
```

Appendix C

Some helper functions

```
import numpy as np
import matplotlib.pyplot as plt
from Kruskal_v1 import Graph
from Tree import TreeMixture, Tree, Node
import sys
import dendropy
epsilon = sys.float_info.epsilon


def compute_r(N, num_clusters, samples, tree, num_nodes, epsilon, loglikelihood):
    """
    Function to compute the responsabilities
    """
  r = np.ones((N, num_clusters))
  for n, x in enumerate(samples):
    for k, ct in enumerate(tree.clusters):
      r[n,k] *= tree.pi[k]
      ct_nodes_tovisit = [ct.root]
      num_visited_nodes = 1
      while num_visited_nodes <= num_nodes:
        curr_node = ct_nodes_tovisit.pop(0)
        ct_nodes_tovisit = ct_nodes_tovisit + curr_node.descendants
        if num_visited_nodes == 1:
          root_cat = x[int(curr_node.name)]
          r[n,k] *= curr_node.cat[root_cat]
        else:
          parent_cat = x[int(curr_node.ancestor.name)]
          node_cat = x[int(curr_node.name)]
          r[n,k] *= curr_node.cat[parent_cat][node_cat]
        num_visited_nodes +=1
  r += epsilon
  M = np.reshape(np.sum(r, axis=1), (N, 1))
  loglikelihood.append(np.sum(np.log(M)))
  r_denom = np.repeat(M, num_clusters, axis=1)
  r /= r_denom
```

```
  return r

def find_indexes(samples,s,t,a,b):
    """
    Finds the indexes where X_s = a and X_t = b
    """
  cond = samples[:,(s,t)]==[a,b]
  indexes = []
  for i,e in enumerate(cond):
    if e[0] == e[1] and e[0] == True:
      indexes.append(i)
  return indexes

def compute_I(r, num_nodes, num_clusters, samples):
    """
    Compute and return information and the distribution q.
    We will use q to update the CDPs.
    """
  denom = np.sum(r, axis=0)
  q = np.zeros((num_nodes, num_nodes, 2, 2, num_clusters)) # (s, t, a, b, k)
  for s in range(num_nodes):
       for t in range(num_nodes):
          for a in range(2):
              for b in range(2):
                  indexes = find_indexes(samples,s,t,a,b)
                  # We add an epsilon to avoid zero errors
                  q[s, t, a, b] = (np.sum(r[indexes], axis=0) / denom) + epsilon

  q_s = np.zeros((num_nodes, 2, num_clusters))
  for s in range(num_nodes):
      for a in range(2):
          indexes = np.where(samples[:, s]==a)
          q_s[s,a] = (np.sum(r[indexes], axis=0) / denom) + epsilon

  I = np.zeros((num_nodes, num_nodes, num_clusters))
  for s in range(num_nodes):
      for t in range(num_nodes):
           for a in range(2):
               for b in range(2):
                  I[s,t] += q[s,t,a,b] * np.log(q[s,t,a,b] / (q_s[s,a]*q_s[t,b]))
```

```
  return I, q, q_s


def construct_MST(edges, num_nodes):
    """
    Constructing the maximum spanning tree using
    its topology_array
    We then first compute the topology_array and
    construct it in the end.
    """
  topology_array = np.zeros(num_nodes)
  topology_array[0] = np.nan
  visited_nodes = [0]
  num_visited_nodes = 1
  while num_visited_nodes <= num_nodes:
    curr_node = visited_nodes.pop(0)
    childs = []
    to_remove = []
    for e in edges:
      if e[0] == curr_node:
        childs.append(e[1])
        to_remove.append(e)
      elif e[1] == curr_node:
        childs.append(e[0])
        to_remove.append(e)
    for e in to_remove:
      edges.remove(e)

    for child in childs:
      topology_array[int(child)] = curr_node
      visited_nodes.append(int(child))
    num_visited_nodes += 1

  tree = Tree()
  tree.load_tree_from_direct_arrays(topology_array)
  tree.k = 2
  tree.alpha = [1.0] * 2
  return tree
```

```
def update_CPD(mst, num_nodes, q, q_s, k):
    """
    mst is our maximum spanning tree
    """
  visited_nodes = [mst.root]
  num_visited_nodes = 1

  while num_visited_nodes <= num_nodes:
    curr_node = visited_nodes.pop(0)
    visited_nodes = visited_nodes + curr_node.descendants
    if num_visited_nodes == 1:
      curr_node.cat = q_s[int(curr_node.name),:,k].tolist()
    else:
      cat = q[int(curr_node.ancestor.name),int(curr_node.name),:,:,k]
      curr_node.cat = [cat[0], cat[1]]
    num_visited_nodes += 1


def em_steps(tree, samples, num_clusters, max_num_iter=100):
    """
    em algorithm helper that uses all the functions above
    """
  N = np.size(samples, 0)
  num_nodes = np.size(samples, 1)
  loglikelihood = []

  for i in range(max_num_iter):
    # Step 1-----------------------------------------------
    r = compute_r(N, num_clusters, samples, tree,
                  num_nodes, epsilon, loglikelihood)
    # Step 2-----------------------------------------------
    tree.pi = np.mean(r, axis=0)
    # Step 3-----------------------------------------------
    I, q, q_s = compute_I(r, num_nodes, num_clusters, samples)
    clusters = []
    for k in range(num_clusters):
      g = Graph(num_nodes)
      for s in range(num_nodes):
        for t in range(s+1, num_nodes):
          g.addEdge(s, t, I[s, t ,k])
```

```
        # Step 4-----------------------------------------------
          mst_edges = g.maximum_spanning_tree()
          mst = construct_MST(mst_edges, num_nodes)
        # Step 5-----------------------------------------------
          update_CPD(mst, num_nodes, q, q_s, k)
        #-----------------------------------------------------
          clusters.append(mst)

        tree.clusters = clusters

    return loglikelihood, tree
```

EM algorithm

```
def save_results(loglikelihood, topology_array, theta_array, filename):
    """ This function saves the log-likelihood vs iteration values,
        the final tree structure and theta array to corresponding numpy arrays.

    likelihood_filename = filename + "_em_loglikelihood.npy"
    topology_array_filename = filename + "_em_topology.npy"
    theta_array_filename = filename + "_em_theta.npy"
    print("Saving log-likelihood to ", likelihood_filename, ", topology_array to:
          ", theta_array to: ", theta_array_filename, "...")
    np.save(likelihood_filename, loglikelihood)
    np.save(topology_array_filename, topology_array)
    np.save(theta_array_filename, theta_array)


def em_algorithm(seed_val, samples, num_clusters, max_num_iter=100):
    """
    This function is for the EM algorithm.
    :param seed_val: Seed value for reproducibility. Type: int
    :param samples: Observed x values. Type: numpy array.
    Dimensions: (num_samples, num_nodes)
    :param num_clusters: Number of clusters. Type: int
    :param max_num_iter: Maximum number of EM iterations.
    Type: int
    :return: loglikelihood: Array of
    log-likelihood of each EM iteration. Type: numpy array.
```

```
    Dimensions: (num_iterations, )
    Note: num_iterations does not have to be equal to max_num_iter.
    :return: topology_list: A list of tree topologies.
    Type: numpy array. Dimensions: (num_clusters, num_nodes)
    :return: theta_list: A list of tree CPDs.
    Type: numpy array. Dimensions: (num_clusters, num_nodes, 2)

    This is a suggested template. Feel free to code
    however you want.
    """

    print("====================> Running EM algorithm...")
    sieving = 10
    num_samples = np.size(samples, 0)
    num_nodes = np.size(samples, 1)

    np.random.seed(seed_val)
    seeds = np.random.randint(0, 99999, sieving)
    logs = []
    tms = []
    for seed in seeds:
        np.random.seed(seed)
        tm = TreeMixture(num_clusters=num_clusters, num_nodes=num_nodes)
        tm.simulate_pi(seed_val=seed)
        tm.simulate_trees(seed_val=seed)
        tm_loglikelihood, tm = em_steps(tm, samples,
                                        num_clusters, max_num_iter=10)
        logs.append(tm_loglikelihood[-1])
        tms.append(tm)

    print("=====================> Sieving finished")
    max_seed_index = logs.index(max(logs))
    seed = seeds[max_seed_index]
    tm = TreeMixture(num_clusters=num_clusters, num_nodes=num_nodes)
    tm.simulate_pi(seed_val=seed)
    tm.simulate_trees(seed_val=seed)
    loglikelihood, tm = em_steps(tm, samples,
                                 num_clusters, max_num_iter=max_num_iter)

    print("=====================> EM finished")
```

```
    topology_list = []
    theta_list = []
    for t in tm.clusters:
        topology_list.append(t.get_topology_array())
        theta_list.append(t.get_theta_array())
    loglikelihood = np.array(loglikelihood)
    topology_list = np.array(topology_list)

    return loglikelihood, topology_list, theta_list, tm
```

Functions to compute the RF distances and the true likelihood

```
def compute_RF(true_tree, inferred_tree):
  num_samples, num_nodes = true_tree.samples.shape
  num_clusters = true_tree.num_clusters
  tns = dendropy.TaxonNamespace()
  for k in range(num_clusters):
      for i in range(num_clusters):
          t_k = inferred_tree.clusters[k].get_tree_newick()
          t_k = dendropy.Tree.get(data=t_k.newick,
                schema="newick", taxon_namespace=tns)
          t_i = true_tree.clusters[i].get_tree_newick()
          t_i = dendropy.Tree.get(data=t_i.newick,
                schema="newick", taxon_namespace=tns)
          print(dendropy.calculate.treecompare.symmetric_difference(t_k, t_i),
               end=" ")


def true_likelihood(true_tree, samples):
  num_samples, num_nodes = samples.shape
  num_clusters = true_tree.num_clusters
  posterior = np.ones((num_samples, num_clusters))
  prior = np.ones(num_samples)
  for n, x in enumerate(samples):
      for k, tree in enumerate(true_tree.clusters):
          visited_nodes = [tree.root]
          num_visited_nodes = 1
          while num_visited_nodes <= num_nodes:
              curr_node = visited_nodes.pop(0)
```

```
            visited_nodes = visited_nodes + curr_node.descendants
            if num_visited_nodes == 1:
              curr_cat = x[int(curr_node.name)]
              posterior[n,k] *= curr_node.cat[curr_cat]
            else:
              parent_cat = x[int(curr_node.ancestor.name)]
              curr_cat = x[int(curr_node.name)]
              posterior[n,k] *= curr_node.cat[parent_cat][curr_cat]
            num_visited_nodes += 1

    prior[n] *= np.sum(posterior[n] * true_tree.pi)

return np.sum(np.log(prior))
```