DD2434 MLADV - Assigment 2

Daniel Wass 940902-4750, dwass@kth.se

January 7, 2020

Questions

2.1 Knowing the rules

1. Yes.
2. **2.4:** Discussed problem formulations with my project group; Sonja Horn, Povel Forsare Kallman and Martin Koling.
   2.5: Discussed problem formulations with TA Hazal Koptagel.
   2.6: Discussed problem formulations with TA Negar Safinianaini.
3. No, I have not discussed solutions with anybody.

2.2 Dependencies in a Directed Graphical Model

4. Yes.
5. No.
6. Yes.
7. No.
8. No.
9. No.

2.4 Simple VI

12. Implemented in Python. See Appendix 1 for code.

13. The exact posterior is (Murphy, 2007, p. 8):

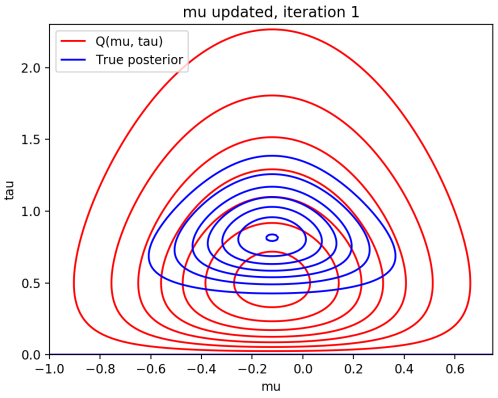
$$
\begin{gathered}
p(\mu, \tau|D) = p(D|\mu, \tau)p(\mu|\tau)p(\tau) = \\
\mathcal{NG}(\mu, \tau|\mu_N, \lambda_N, a_N, b_N), \\
\text{where} \\
\mu_N = \frac{\lambda_0\mu_0 + N\bar{x}}{\lambda_0 + N} \\
\lambda_N = \lambda_0 + N \\
a_N = a_0 + N/2 \\
b_N = b_0 + 0.5\sum_{i=1}^{N}(x_i - \bar{x})^2 + \frac{\lambda_0 N(\bar{x} - \mu_0)^2}{2(\lambda_0 + N)}
\end{gathered}
\tag{1}
$$

14. In Fig. 1 a) to d), the iterative process of the IV algorithm, implemented on 20 datapoints drawn from an iid zero mean, one variance, is visualised. We can see that the prior of τ is Gamma distributed whereas μ is normal distributed - over the μ-axis the distribution is symmetric, but over the τ-axis the distribution is more Gamma.

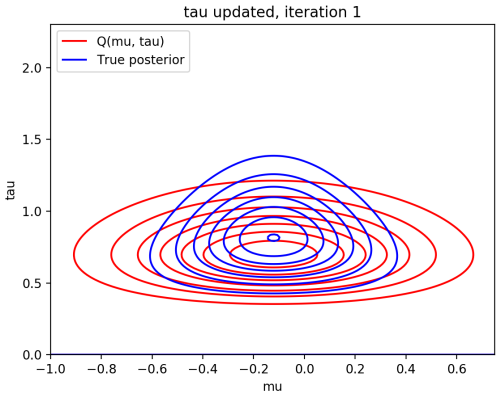
    The plot in e) shows a case of identical parameters except with more datapoints, i.e. ten times more. We see that with more datapoints, the inferred distribution is closer to the true posterior and that both these distributions are more certain of the values of τ and μ - i.e. less variance for both τ and μ. This is expected behaviour when increasing the number of datapoints since more data gives a bigger basis for inference. Furthermore, we see that τ becomes more Gaussian distributed, which is sensible as the prior - being Gamma - gets less impact on the posterior when the dataset increase.

    The plot in f) shows the case of the same parameters as in d) but with generating distribution variance set to four. This leads to a less certainty regarding μ, but a higher certainty regarding τ. The first is pretty straight-forward - it is reasonable that our uncertainty about the mean of our data increases as the variance of the datapoints increases. The latter, however more complex, is reasonable as well. The variance of τ is $\frac{a}{b^2}$, and as variance of the generated data $x_1, ..., x_N$ increases, b_N increases, and thus variance of τ increases. In the same way, decreasing variance of the generated data, leads to more certainty regarding μ and less certainty of τ.

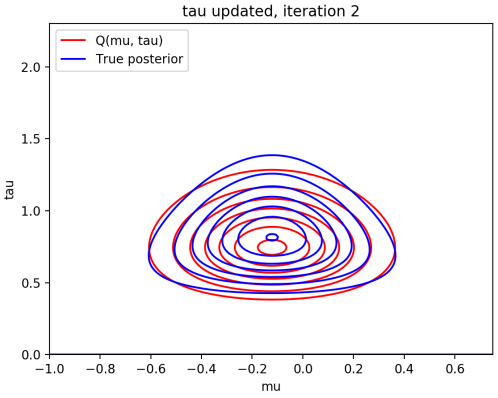
$$
b_N = b_0 + 0.5\,\mathbb{E}_\mu\left[\sum_{i}^{N}(x_i - \mu)^2 + \lambda_0(\mu - \mu_0)^2\right] \tag{2}
$$



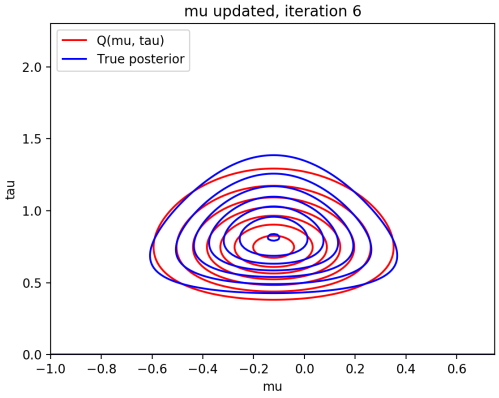
(a) After updating μ the first iteration.



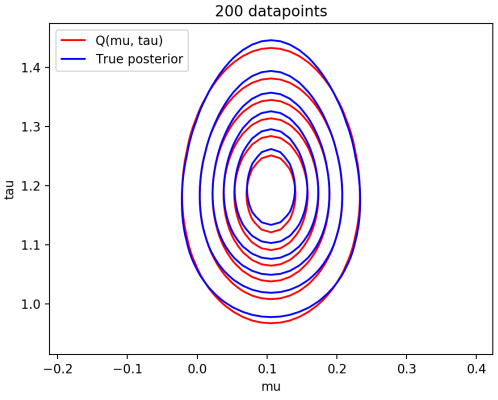
(b) After one full iteration.



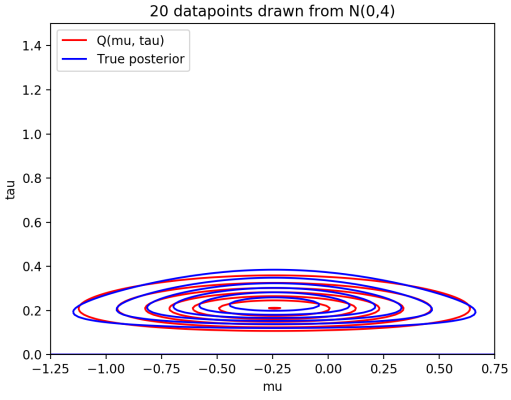
(c) After two full iterations.



(d) After convergence.



(e) 200 datapoints.



(f) Higher variance when generating data set.

Figure 1: The inferred distribution compared to the true posterior. The data is sampled from a Gaussian with variance one and zero mean. A-D shows one single case, whereas E and F shows one case each.

2.5 Mixture of trees with observable variables

15. Implemented in Python. See Appendix 2 for code.

16. The comparisons between the real and inferred models for the different datasets are presented in Table 1 and Table 2. In addition to the given datasets, results are presented for five new datasets and mixtures. Plots of how the complete log-likelihood and complete likelihood change over the iterations until convergence for the given datasets are presented in Figure 2. Inferred trees are called $i0, ..., iN-1$ and real trees are called $t0, ..., tN-1$, where N is the number of clusters.

    The RF-distance is not very helpful for finding structural similarities - it is simply the sum of the number of clades unique for each tree respectively. The maximum RF-distance ($2(n-3)$, where n = number of nodes) can be given to trees that actually are similar. However, an analysis of the results in terms of RF-distance and log likelihood are presented in the next section (2.5.17).

17. Five new datasets were generated. Their results and settings can all be seen in Table 1.

    Let us compare the trees that gives the lowest total RF-distance, e.g. the bold values in Table 2 for the 10/20/4-dataset (i0-t1, i1-t3, i2-t2 and i3-t0). This is one of the ways giving the shortest total RF-distance between the mixture models, e.g. $\min RF_{10/20/4} = 29$. In this way we have the total minimum mixture RF-distance, seen in Table 1. The "Weighted Mean RF" column is simply the minimum RF-distance divided by number of clusters and max RF-distance, and thus it is the distance in relation to max distance, averaged over the number of trees in the mixture. Looking at the unweighted distance we see a clear correlation between number of nodes and RF-distance, caused by the max-distance depending on the number of nodes.

    For 10 nodes and 4 clusters, the best result is given by 20 samples, implying that increased number of samples does **not** increase the similarity between real and inferred trees, which was not expected (note that the 10/200/4-dataset was drawn from another mixture than the other 10-nodes, 4 clusters datasets). However, for the two datasets with only one cluster, increasing the number of samples seems to decrease the RF-distance. Otherwise the weighted RF-distance seems to be relatively stable for all tests. The 100-nodes dataset gives a surprisingly good weighted RF - despite high dimensions and low amount of data. The same goes for the given 20/20/4-dataset, having a 1:1 dimensions/samples ratio, implying that the tree similarity is not sensitive to the dimensions/samples ratio. When looking at the test with more clusters, i.e. the 10/20/7-dataset, the model shows a slightly lower performance, regarding weighted min RF, compared to the 10/20/4-dataset. This implies that the more clusters, the more data should be needed, which is reasonable as else there will be less samples representing each cluster. However, the 10/20/1-dataset implies the contradictory, having a lower weighted min RF than the 10/20/4-dataset. When increasing the number of samples for the 1-cluster 10-nodes mixture, the RF-distance decreases. Table 1 shows for a 10 times increase, but when running additional tests for even more data, it shows the same. This implies that the RF-distance slightly decrease when increasing the number of samples, in contradiction to what is implied by the 10-nodes, 4-cluster datasets.

    When comparing the log likelihoods, it is important to remember that the mixture log likelihood scales with the number of samples and the number of clusters. Thus, the values are

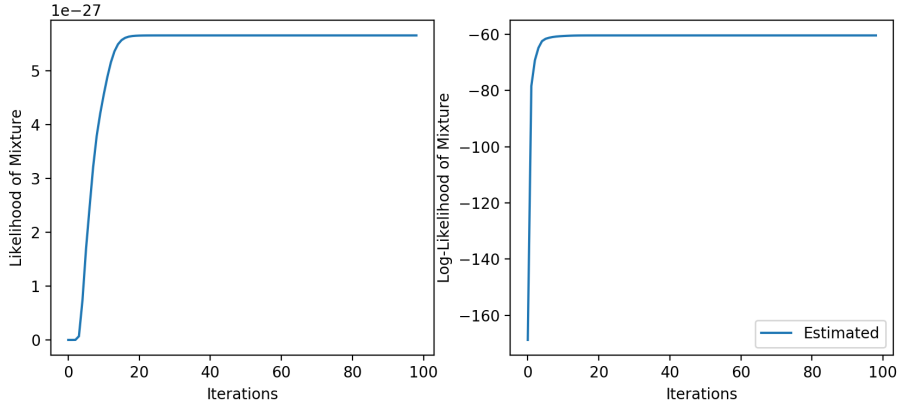
evaluated in relation to that when discussed below. However, we see tendencies of the real mixture's and the inferred one's becoming more similar to each other when the number of samples increase. This is expected behaviour as few samples allows the noise to have bigger impact, and the inferred trees thus becomes somewhat overfitted to the data. If we use more samples, the data will be a better representation of the model, and the learned mixture will have a likelihood closer to the real mixture's. Regarding the experiment of using huge trees (i.e. 100 nodes), the inferred log likelihood is very large compared to the true log likelihood. As the trees are huge, there are many sources of noise (the categorical distribution of each node) in the data, and the real mixture does not match the data very well (at least not for this few samples) whereas the inferred mixture overfits, matching the data very well. This, together with the 20/20/4-dataset implies that the number of samples should be higher than the number of nodes, which is fully sensible and corresponds to samples/dimension-relation norms for machine learning in general. This is however not shown in the weighted RF-distance, being similar to the tests with fewer nodes. To be noted is that the time complexity increases tremendously when using huge trees.

Table 1: Minimum total Robinson-Foulds distance for mixture and log likelihood for mixture ℓ.

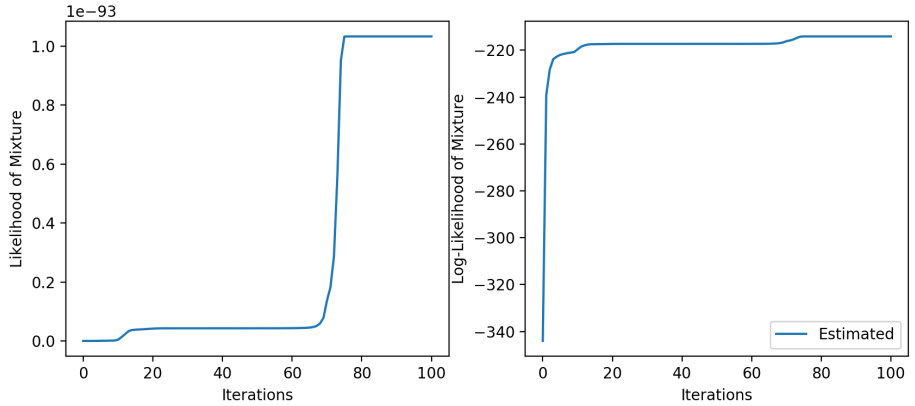
Nodes/Samples/Clusters	Min RF-distance	Weighted Min RF	Real ℓ	Inferred ℓ	$\Delta\ell$
10/20/4	29	0.518	-113.143	-60.438	52.705
10/50/4	34	0.607	-280.857	-214.108	66.74
20/20/4	73	0.537	-217.006	-85.758	131.248
10/200/4	33	0.589	-1188.428	-1111.679	76.749
100/20/4	406	0.523	-1046.738	-135.746	910.992
10/20/7	59	0.602	-124.746	-59.947	64.832
10/20/1	8	0.571	-97.585	-82.034	15.551
10/200/1	6	0.429	-1001.380	-982.909	18.472

Table 2: Robinson-Foulds comparison.

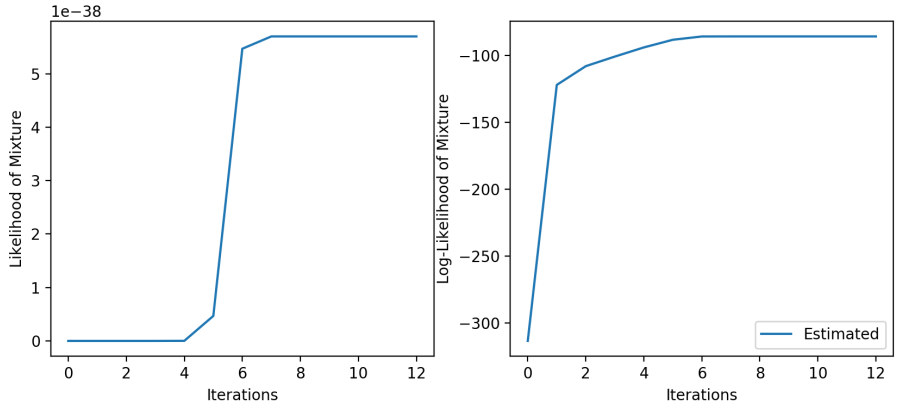
Nodes/Samples/Clusters	Real\\Inferred Tree	i0	i1	i2	i3	i4	i5	i6
10/20/4	t0	7	10	9	**6**			
	t1	**6**	7	8	7			
	t2	8	11	**8**	9			
	t3	8	**9**	10	7			
10/50/4	t0	8	10	8	9			
	t1	9	9	7	10			
	t2	9	11	11	10			
	t3	7	9	9	8			
20/20/4	t0	21	20	20	17			
	t1	17	20	20	15			
	t2	21	24	18	17			
	t3	22	23	21	18			
10/200/4	t0	7	9	10	8			
	t1	6	10	9	7			
	t2	9	11	10	8			
	t3	9	11	12	8			
100/20/4	t0	98	98	96	114			
	t1	105	101	105	113			
	t2	103	107	107	113			
	t3	106	98	102	106			
10/20/7	t0	9	9	11	10	9	9	9
	t1	8	10	12	11	8	10	10
	t2	9	9	7	10	9	9	9
	t3	11	11	11	10	11	9	11
	t4	8	10	10	9	8	10	10
	t5	9	9	9	10	9	11	9
	t6	10	10	10	9	10	8	10



(a) 10 nodes 20 samples dataset.



(b) 10 nodes 50 samples dataset.



(c) 20 nodes 20 samples dataset.

Figure 2: The likelihood of the mixture given the three datasets. After sieving and convergence/-completing 100 iterations.

2.6 Super epicentra - EM

18. The locations of the earthquakes, $X \in \mathcal{R}^2$, follow a mixture of Gaussians and the strength, $S \in \mathcal{R}$, follow a mixture of Poissons. As $X \perp\!\!\!\perp S|Z$, the log likelihood for data X and S is:

$$\begin{aligned}
\ell(\theta; X, S, Z) &= \log \prod_n p(z_n, x_n, s_n|\theta_k) \\
&= \sum_n \log \left[\prod_k (\pi_k p(x_n|\theta_k) p(s_n|\theta_k))^{I(z_n=k)} \right] \\
&= \sum_n \sum_k I(z_n = k) \log \pi_k + \sum_n \sum_k I(z_n = k) \log p(x_n|\theta_k) + \sum_n \sum_k I(z_n = k) \log p(s_n|\theta_k).
\end{aligned} \tag{3}$$

This gives the expected complete data log likelihood Q(Z):

$$\begin{aligned}
\mathcal{Q}(Z) &= \mathbb{E}_{p(z_n|x_n,s_n,\theta)} \left[\ell(\theta; x_n, s_n, z_n) \right] \\
&= \sum_n \mathbb{E} \left[\sum_k I(z_n = k) \log \pi_k + \sum_k I(z_n = k) \log p(x_n|\theta_k) + \sum_k I(z_n = k) \log p(s_n|\theta_k) \right] \\
&= \sum_n \sum_k \mathbb{E}[I(z_n = k)] \log \pi_k + \sum_n \sum_k \mathbb{E}[I(z_n = k)] \log p(x_n|\theta_k) + \sum_n \sum_k \mathbb{E}[I(z_n = k)] \log p(s_n|\theta_k) \\
&= \sum_k \sum_n \gamma_{nk} \log \pi_k + \sum_k \sum_n \gamma_{nk} \log p(x_n|\mu_k, \Sigma_k) + \sum_k \sum_n \gamma_{nk} \log p(s_n|\lambda_k),
\end{aligned} \tag{4}$$

where the responsibilities γ_{nk} are:

$$\gamma_n k = \frac{\pi_k \mathcal{N}(x_n|\mu_k, \Sigma_k) \text{Poisson}(s_n; \lambda_k)}{\sum_{l=1}^{K} \pi_l \mathcal{N}(x_n|\mu_l, \Sigma_l) \text{Poisson}(s_n; \lambda_l)}. \tag{5}$$

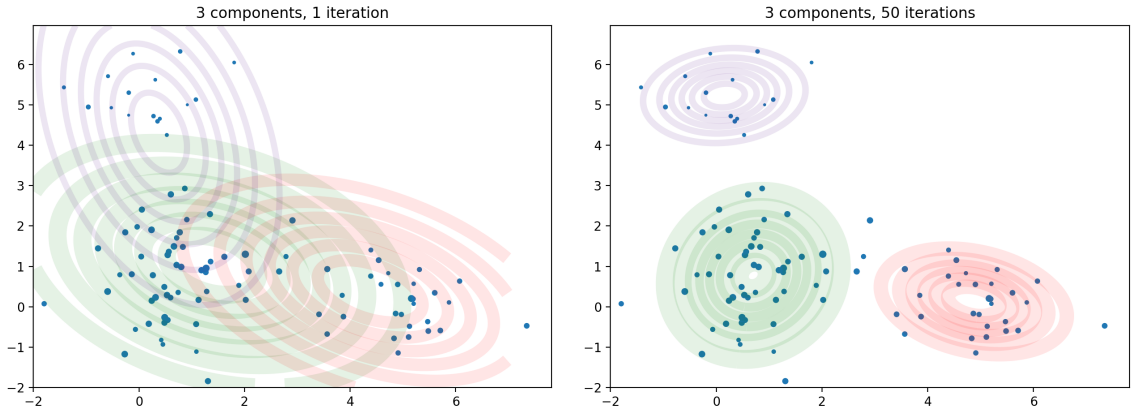
For each k, we search $\theta_k^* = \arg\max \ell(\theta_k, X, S, Z)$. We differentiate Q with respect to each parameter, and then set each partial derivative to zero to find the parameters that maximise the likelihood.

$$\begin{aligned}
\frac{\partial \mathcal{Q}}{\partial \pi_k} = 0 &\implies \pi_k' = \frac{\sum_n \gamma_{nk}}{N} \\
\frac{\partial \mathcal{Q}}{\partial \mu_k} = 0 &\implies \mu_k' = \frac{\sum_n \gamma_{nk} x_n}{\sum_n \gamma_{nk}} \\
\frac{\partial \mathcal{Q}}{\partial \Sigma_k} = 0 &\implies \Sigma_k' = \frac{\sum_n \gamma_{nk} (x_n - \mu_k')(x_n - \mu_k')^T}{\sum_n \gamma_{nk}} \\
\frac{\partial \mathcal{Q}}{\partial \lambda_k} = 0 &\implies \lambda_k' = \frac{\sum_n \gamma_{nk} s_n}{\sum_n \gamma_{nk}}
\end{aligned} \tag{6}$$

An EM algorithm for the model consists of computing γ, eq. 5 (E-step), followed by a maximising update of the model parameters (M-step) while keeping γ fixed. The E-step is then repeated with the new model parameters, followed by the M-step, and so on. The two steps are repeated until convergence of the log likelihood, eq. 3, is achieved. We have now optimised the model parameters.

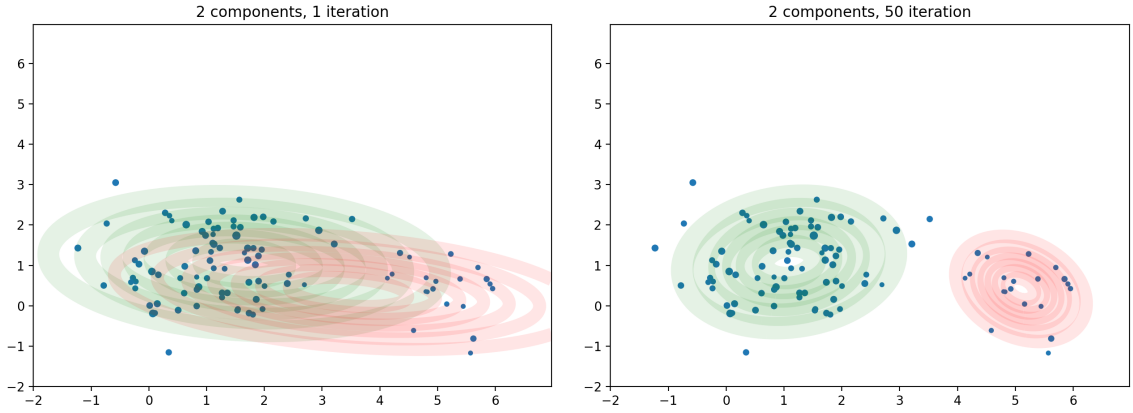
19. The algorithm was implemented in Python. See Appendix 3 for code.

20. The plots of the inferred distributions and the data are shown in Figure 3. First we make an initial guess for the parameters for the distribution of X and S. Then we then compute the responsibility of each sample and each cluster, i.e. the probability of the cluster given the samples x_n and s_n. We then optimise the parameters according to the γ we just computed with respect to all samples - i.e. letting each sample's impact on the new parameters be weighted by its responsibility for this current cluster. So, now we have new parameters, better explaining the clusters than the initial guess. This means that we can compute more accurate responsibilities, which we do. And when we have more accurate responsibilities we can compute new, more accurate parameters, and so on. This is the simple explanation of how the EM-algorithm is successful. The thickness of the points in Figure 3 represents the true strength of the earthquake sample and their grid represents its true location. The inferred clusters' line thickness represents the inferred rate λ_k of the cluster.



(a) Dataset 1 after one iteration.

(b) Dataset 1 after 50 iterations/convergence.



(c) Dataset 2 after one iteration.

(d) Dataset 2 after 50 iterattions/convergence.

Figure 3: The plots of the two different datasets after 1 iteration and 50 iterations. The linewidth of the distributions represents the inferred strength of the earthquake, whereas the linewidth of the datapoints represents the true strength.

References

Murphy, K. P. 2007. Conjugate bayesian analysis of the gaussian distribution. *def*, 1(2σ2):16.

Appendix

1. Code for 2.4 - Simple VI

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

# computes the true parameters. takes initial guesses and data as input and returns tru
def compute_true_parameters(mu0, lambda0, a0, b0, N, X):
    mu_data = np.mean(X)
    mu_true = ( lambda0 * mu0 + N * mu_data ) / (lambda0 + N)
    lambda_true = lambda0 + N
    a_true = a0 + N/2
    temp = np.square(X - mu_data)
    b_true = b0 + 1/2 * (np.sum(temp)) + (lambda0 * N * (mu_data - mu0)**2) / (2*(lambd

    return mu_true, lambda_true, a_true, b_true

# plot true distribution and inferred distribution
def plot_PDFs(muN, lambdaN, aN, bN,  mu_true, lambda_true, a_true, b_true, title):

    # axes for mu and tau
    mu = np.linspace(-1.25, 0.75, 150)
    tau = np.linspace(0, 1.5, 150)
    MU, TAU = np.meshgrid(mu, tau, indexing='ij')

    # q_mu(axis, mu, sigma)
    Q_mu = lambda x, mu, sigma: stats.norm.pdf(x, mu, 1/sigma)
    # q_tau(axis, a, b)
    Q_tau = lambda x, a, b: stats.gamma.pdf(x, a, loc=0, scale=1/b)

    # create matrix Q for plotting, Q = q_mu * q_tau
    Q = np.zeros_like(MU)
```

```
    for i in range(Q.shape[0]):
        for j in range(Q.shape[1]):
            Q[i][j] = Q_mu(mu[i], muN, sqrt(lambdaN)) * Q_tau(tau[j], aN, bN)

    # plot inferred distribution
    plt.plot()
    plt.title(title)
    CS = plt.contour(MU, TAU, Q, colors = "r")
    CS.collections[0].set_label("Q(mu, tau)")
    plt.legend(loc='upper left')

    plt.xlabel("mu")
    plt.ylabel("tau")

    # NG-distribution(mu, tau)
    normal_gamma = lambda mu, tau: (((b_true ** a_true) * sqrt(lambda_true))/(gamma(a_t

    # create matrix for true posterior
    true_posterior = np.zeros_like(MU)
    for i in range(true_posterior.shape[0]):
        for j in range(true_posterior.shape[1]):
            true_posterior[i][j] = normal_gamma(mu[i], tau[j])

    # plot true distribution
    CS2 = plt.contour(MU, TAU, true_posterior, colors = "blue")
    CS2.collections[0].set_label("True posterior")
    plt.legend(loc='upper left')

    plt.show()

# stepwise compute q(mu) and q(tau), plot twice, after q(mu) is updated and after aN an
# and returns updated parameters. (Also takes previously inferred aN, bN and iteration
def q_mu_tau(mu0, lambda0, a0, b0, E_tau, N, X, aN, bN, i):

    muN = (lambda0 * mu0 + np.sum(X)) / (lambda0 + N)
    lambdaN = (lambda0 + N) * E_tau

    mu_true, lambda_true, a_true, b_true = compute_true_parameters(mu0, lambda0, a0, b0
    plot_PDFs(muN, lambdaN, aN, bN, mu_true, lambda_true, a_true, b_true, str("mu_updat

    E_mu = muN
    E_mu2 = 1./lambdaN + E_mu ** 2

    # update aN and bN
    aN = a0 + N/2
    bN = b0 + 1/2 * (np.sum(np.square(X)) - 2 * E_mu * np.sum(X) + N * E_mu2 + lambda0
```

```
    # plot with tau updated
    plot_PDFs(muN, lambdaN, aN, bN, mu_true, lambda_true, a_true, b_true, "full iteratio

    return muN, lambdaN, aN, bN

if __name__ == "__main__":

    np.random.seed(11)

    N = 200

    # generate data from iid gaussian
    X = np.random.normal(0, 2, N)

    mu0 = 0
    lambda0 = 2
    a0 = 2
    b0 = 2

    aN = a0
    bN = b0

    # as E_tau = a/b, we make an initial guess accordingly
    E_tau = a0/b0

    for i in range(3):
        muN, lambdaN, aN, bN = q_mu_tau(mu0, lambda0, a0, b0, E_tau, N, X, aN, bN, i)
        E_tau = aN / bN
        print("    mu: %f \nlambda: %f \n      a: %f \n      b: %f \n      tau: %f" % (muN
        print("\n")
    print(compute_true_parameters(mu0, lambda0, a0, b0, N, X))
```

2. Code for 2.5 - Mixture of trees with observable variables

Note that the *full* algorithm according to the instructions, i.e. including sieving, is run by calling the function sieving().

```
import argparse
import numpy as np
import matplotlib.pyplot as plt
import scipy.stats as stats
import math
import itertools
import Kruskal_v2
from queue import Queue
import sys
import random
from Tree import TreeMixture, Tree, Node
import Kruskal_v2
import heapq
import pickle as pkl
import dendropy

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: ", 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)

# Initiates 100 mixtures with random parameters and runs them 10 iterations through the
# Returns lists of loglikelihood, topology and the categorical distributions theta for
# the 10 sieved mixtures aafter convergence. The loglikelihood-list has one element fro
# Input:    seed_val is the master seed for generating 100 random seeds.
#           samples is the array of samples
#           mixtures is the number of mixtures to sieve
#           num_clusters is the number of clusters generating the data
#           max_num_iter is the iteration limit if no convergence
def sieving(seed_val, samples, mixtures, num_clusters, max_num_iter=100):

    np.random.seed(seed_val)
```

```
    random_100_seeds = np.array(random.sample(range(10000), mixtures))

    log_likelihoods = np.zeros(mixtures)

    # compute likelihood for the 100 random seeds and save the loglikelihood of the 10t
    print("--- SIEVING ---")
    for s, seed in enumerate(random_100_seeds):
        log_likelihood, _, _ = em_algorithm(seed, samples, num_clusters, 10)
        log_likelihoods[s] = log_likelihood.pop()

    # --- Find top 10 Mixtures ---
    top10_indices = heapq.nlargest(10, range(len(log_likelihoods)), log_likelihoods.tak
    top10_seeds = random_100_seeds[top10_indices]

    print("--- Run to convergence ---")
    # --- Run to convergence ---
    bar = -99999
    best_log_likelihood = []
    best_theta = []
    best_topology = []
    best_seed = -1
    for seed in top10_seeds:
        log_likelihood, topology, theta = em_algorithm(seed, samples, num_clusters)
        print("...running...")
        if log_likelihood[-1] > bar:
            bar = log_likelihood[-1]
            best_log_likelihood = log_likelihood
            best_theta = theta
            best_topology = topology
            best_seed = seed

    print("best seed:", best_seed)

    return best_log_likelihood, best_topology, best_theta

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, nun
    :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 a
                Dimensions: (num_iterations, ) Note: num_iterations does not have to be
    :return: topology_list: A list of tree topologies. Type: numpy array. Dimensions: (
    :return: theta_list: A list of tree CPDs. Type: numpy array. Dimensions: (num_clust
```

```
    You can change the function signature and add new parameters. Add them as parameter
    i.e.
    Function template: def em_algorithm(seed_val, samples, k, max_num_iter=10):
    You can change it to: def em_algorithm(seed_val, samples, k, max_num_iter=10, new_p
    """
    # Set the seed
    np.random.seed(seed_val)

    N = samples.shape[0]
    num_nodes = samples.shape[1]


    # Create one mixture
    M = TreeMixture(num_clusters=num_clusters, num_nodes=num_nodes)
    M.pi = np.ones(num_clusters)
    M.pi = M.pi/num_clusters
    M.simulate_trees(seed_val=seed_val)

    ''' Try with generating parameters '''
    # for k, tree in enumerate(M.clusters):
    #     theta = np.load(("data/q_2_5_tm_10node_50sample_4clusters.pkl_tree_%i_theta.n
    #     topology = np.load(("data/q_2_5_tm_10node_50sample_4clusters.pkl_tree_%i_topo
    #     tree = tree.load_tree_from_direct_arrays(topology, theta_array=theta)


    llh_list = []

    log_likelihood = compute_log_likelihood(M, samples, N, num_clusters, num_nodes)
    likelihood = np.exp(log_likelihood)
    likelihood_sum1 = np.sum(likelihood, axis=1)
    llh_list.append(np.sum(np.log(likelihood_sum1), axis=0))

    for iteration in range(max_num_iter):

        # Compute responsibilities
        responsibility_matrix = compute_responsibility(M, samples, N, num_clusters, num

        M.pi = np.sum(responsibility_matrix, axis = 0)
        M.pi = M.pi / N

        # Update tree T in M to be MST
        # Update theta according to new T
        for k, tree in enumerate(M.clusters):
            new_topology = update_Topology(k, responsibility_matrix, samples, N, num_no
            theta = tree.get_theta_array()
```

```
            new_theta = update_Theta(k, new_topology, samples, responsibility_matrix, t
            tree.load_tree_from_direct_arrays(new_topology, new_theta)

        # Compute new log_likelihood for each datapoint
        log_likelihood = compute_log_likelihood(M, samples, N, num_clusters, num_nodes)

        # Compute log likelihood for mixture and append to list
        likelihood = np.exp(log_likelihood)
        likelihood_sum1 = np.sum(likelihood, axis=1)
        llh_list.append(np.sum(np.log(likelihood_sum1), axis=0))

        if iteration > 10 and abs(llh_list[-2] - llh_list[-1]) < 1e-16:
            print("---- convergence ----\nat iteration: ", iteration)
            break

    # load theta and topology
    topology_list=[]
    theta_list = []

    for k, tree in enumerate(M.clusters):
        topology_list.append(tree.get_topology_array())
        theta_list.append(tree.get_theta_array())


    return llh_list, topology_list, theta_list

# compute log likelihood. returns a NxK matrix of the log likelihood for each sample an
def compute_log_likelihood(M, samples, N, num_clusters, num_nodes):
    log_likelihood = np.zeros((N, num_clusters))
    pi_array = M.pi

    for n, sample in enumerate(samples):

        for k, tree in enumerate(M.clusters):
            topology = tree.get_topology_array()

            visit_list = [tree.root]

            while len(visit_list) != 0:

                cur_node = visit_list.pop(0)
                visit_list = visit_list + cur_node.descendants
                cur_index = int(cur_node.name)
                cur_cat = cur_node.cat
                if cur_node == tree.root:
                    log_likelihood[n,k] += np.log(cur_cat[sample[cur_index]])
```

```
                    continue
                parent_value = sample[int(topology[cur_index])]
                log_likelihood[n,k] += np.log(cur_cat[parent_value][sample[cur_index]])

            log_likelihood[n,k] += np.log(pi_array[k])

    return log_likelihood

# compute responsibility. returns a NxK matrix of the responsibilities for each sample
def compute_responsibility(M, samples, N, num_clusters, num_nodes, log_likelihood):
    responsibility_matrix = np.zeros((N, num_clusters))
    pi_array = M.pi

    likelihood = np.exp(log_likelihood)

    for n, sample in enumerate(samples):
        evidence = np.sum(likelihood[n,:])

        for k in range(num_clusters):
            responsibility_matrix[n,k] = (likelihood[n,k] / evidence) + sys.float_info.

        responsibility_matrix[n,:] = responsibility_matrix[n,:] / np.sum(responsibility_

    return responsibility_matrix

# returns the topology corresponding to the maximum spanning tree. uses the Kruskal alg
def update_Topology(k, responsibility_matrix, samples, N, num_nodes):
    undirected_topology = list(itertools.combinations(list(range(num_nodes)), 2))
    num_edges = len(undirected_topology)

    ab_combinations = [[0,0],[0,1],[1,0],[1,1]]

    graph = {'vertices': list(range(num_nodes)),
             'edges': set()}

    for e, edge in enumerate(undirected_topology):

        node1 = edge[0]
        node2 = edge[1]

        weight = 0

        for ab in ab_combinations:
            value1 = ab[0]
            value2 = ab[1]
```

```
            Q_joint = q_joint(value2, value1, node2, node1, samples, responsibility_mat

            if Q_joint == 0:
                continue

            Q_node1 = q_joint(0, value1, node2, node1, samples, responsibility_matrix[:
            Q_node2 = q_joint(value2, 0, node2, node1, samples, responsibility_matrix[:

            weight += Q_joint * (np.log(Q_joint) - (np.log(Q_node1) + np.log(Q_node2)))

        graph['edges'].add((node1, node2, weight))

    MST = Kruskal_v2.maximum_spanning_tree(graph)

    new_topology = np.full(num_nodes, np.nan)
    q = Queue()
    q.put(0)
    checked = []
    while not q.empty():
        parent = q.get()
        for e, edge in enumerate(MST):
            if edge not in checked:
                if parent == edge[0]:
                    new_topology[edge[1]] = parent
                    q.put(edge[1])
                    checked.append(edge)
                elif parent == edge[1]:
                    new_topology[edge[0]] = parent
                    q.put(edge[0])
                    checked.append(edge)

    return new_topology

# q(X_s = a, X_t=b), X_s = value1, X_t = value2, a = value1, b = value2
def q_joint(value2, value1, node2, node1, samples, responsibility_matrix_k):
    joint_indices = [] # np.where((samples[:, node1] == value1) & samples[:, node2] ==
    for n, sample in enumerate(samples):
        if sample[node1] == value1 and sample[node2] == value2:
            joint_indices.append(n)

    return np.sum(responsibility_matrix_k[joint_indices]) / np.sum(responsibility_matri

# q(X_s = a), X_s = node1, value = a
def q_node(value, node1, samples, responsibility_matrix_k):
    indices = []
```

```
    for n, sample in enumerate(samples):
        if sample[node1] == value:
            indices.append(n)

    return np.sum(responsibility_matrix_k[indices]) / np.sum(responsibility_matrix_k)

# returns the new categorical distributions theta.
def update_Theta(k, new_topology, samples, responsibility_matrix, theta):
    for child, parent in enumerate(new_topology):

        if not child:
            theta[child][0] = q_node(0, child, samples, responsibility_matrix[:,k])
            theta[child][1] = q_node(1, child, samples, responsibility_matrix[:,k])
            continue
        parent = int(parent)
        for parent_value in range(2):

            denom = q_node(parent_value, parent, samples, responsibility_matrix[:,k])

            for child_value in range(2):
                if denom == 0:
                    theta[child][parent_value][child_value] = 1
                    continue

                nom = q_joint(parent_value, child_value, parent, child, samples, respon

                if nom == 0:
                    theta[child][parent_value][child_value] = sys.float_info.epsilon
                    continue

                theta[child][parent_value][child_value] = nom / denom

            theta[child][parent_value][0] = theta[child][parent_value][0] / (theta[chil
            theta[child][parent_value][1] = theta[child][parent_value][1] / (theta[chil

    return theta

def main():
    # Code to process command line arguments
    parser = argparse.ArgumentParser(description='EM algorithm for likelihood of a tree
    parser.add_argument('sample_filename', type=str,
                        help='Specify the name of the sample file (i.e data/example_sam
    parser.add_argument('output_filename', type=str,
                        help='Specify the name of the output file (i.e data/example_res
    parser.add_argument('num_clusters', type=int, help='Specify the number of clusters
    parser.add_argument('--seed_val', type=int, default=42, help='Specify the seed valu
```

```
parser.add_argument('--real_values_filename', type=str, default="",
                    help='Specify the name of the real values file (i.e data/examp
# You can add more default parameters if you want.

print("Hello World!")
print("This file demonstrates the flow of function templates of question 2.5.")

print("\n0. Load the parameters from command line.\n")

args = parser.parse_args()
print("\tArguments are:", args)

print("\n1. Load samples from txt file.\n")

samples = np.loadtxt(args.sample_filename, delimiter="\t", dtype=np.int32)
num_samples, num_nodes = samples.shape
print("\tnum_samples:", num_samples, "\tnum_nodes:", num_nodes)
print("\tSamples: \n", samples)

print("\n2. Run EM Algorithm.\n")


""" LOAD TREES FROM RESULT-FILE """
# filename = args.output_filename

# theta_array = np.load(filename + "_em_theta.npy", allow_pickle=True)
# topology_array = np.load(filename + "_em_topology.npy", allow_pickle=True)
# loglikelihood = np.load(filename + "_em_loglikelihood.npy", allow_pickle=True)
"""                ---                """

""" COMPUTE TREES BY SIEVING """
loglikelihood, topology_array, theta_array = sieving(args.seed_val, samples, 100, n

# loglikelihood, topology_array, theta_array = em_algorithm(889, samples, num_clust

print("\n3. Save, print and plot the results.\n")
save_results(loglikelihood, topology_array, theta_array, args.output_filename)
"""                ---                """

for i in range(args.num_clusters):
    print("\n\tCluster:", i)
    print("\tTopology: \t", topology_array[i])
    print("\tTheta: \t", theta_array[i])

plt.figure(figsize=(10, 4))
plt.subplot(121)
```

```
plt.plot(np.exp(loglikelihood), label='Estimated')
plt.ylabel("Likelihood of Mixture")
plt.xlabel("Iterations")
plt.subplot(122)
plt.plot(loglikelihood, label='Estimated')
plt.ylabel("Log-Likelihood of Mixture")
plt.xlabel("Iterations")
plt.legend()
plt.show()

print("\n4. Retrieve real results and compare.\n")
if args.real_values_filename != "":
    print("\tComparing the results with real values...")
    print("\t4.1. Make the Robinson-Foulds distance analysis.\n")

    true_file = open(args.real_values_filename, 'rb')
    true_M = pkl.load(true_file)
    tns = dendropy.TaxonNamespace()

    true_trees = []
    for t in true_M.clusters:
        tree = dendropy.Tree.get(data=t.get_tree_newick(), schema="newick", taxon_n
        true_trees.append(tree)

    inferred_trees = []
    for k in range(args.num_clusters):
        t = Tree()
        t.load_tree_from_direct_arrays(topology_array[k], theta_array[k])
        tree = dendropy.Tree.get(data=t.get_tree_newick(), schema="newick", taxon_n
        inferred_trees.append(tree)

    print("\n4.2 Compare trees and print Robinson-Foulds (RF) distance:\n")

    for t, true_tree in enumerate(true_trees):
        print(("\tt%i vs inferred trees" % t))
        for i, inferred_tree in enumerate(inferred_trees):
            rf = dendropy.calculate.treecompare.symmetric_difference(inferred_tree,
            print(("\tRF distance between t%i and i%i: \t" % (t, i)), rf)


    print("\t4.2. Make the likelihood comparison.\n")
    true_likelihood = compute_log_likelihood(true_M, samples, samples.shape[0], arg
    temp_sum = np.sum(np.exp(true_likelihood), axis=1)
    true_log_likelihood = np.sum(np.log(temp_sum), axis=0)

    print(true_log_likelihood)
```

```
        print(loglikelihood[-1])
        print("difference (log): ", true_log_likelihood-loglikelihood[-1])

if __name__ == "__main__":
    main()
```

3. Code for 2.6 - Super epicentra EM

The code for the three functions requested are presented below.

```
def _do_estep(self, X, S):
        """
        E-step
        """

        Px = np.zeros((X.shape[0], self.n_components))
        Ps = np.zeros((S.shape[0], self.n_components))
        for k in range(self.n_components):
            normal_distr = multivariate_normal(
                mean=self.means[k],
                cov=self.covs[k])
            Px[:,k] = normal_distr.pdf(X)

            for n in range(self.n_row):
                Ps[n,k] = np.power(self.rates[k], S[n]) * np.exp(-self.rates[k]) / np.m


        numerator = Px * self.weights * Ps
        denominator = numerator.sum(axis=1)[:, np.newaxis]
        self.r = numerator / denominator

        self.weights = self.r.mean(axis=0)

        return self

    def _do_mstep(self, X, S):
        """M-step, update parameters"""

        for k in range(self.n_components):
            r_k = self.r[:,k]
            r_k_sum = np.sum(r_k)

            temp = [0,0]
            for n in range(self.n_row):
                temp += r_k[n] * X[n]

            self.means[k] = temp / r_k_sum

            self.covs[k] = np.cov(X.T, aweights=(r_k/r_k_sum).flatten(), bias=True)

            self.rates[k] = np.sum(r_k * S)
            self.rates[k] /= r_k_sum
```

```
        return self

    def _compute_log_likelihood(self, X, S):
        """compute the log likelihood of the current parameter"""
        log_likelihood = 0

        Px = np.zeros((X.shape[0], self.n_components))
        Ps = np.zeros((S.shape[0], self.n_components))

        for k in range(self.n_components):
            normal_distr = multivariate_normal(
                mean=self.means[k],
                cov=self.covs[k])
            Px[:,k] = normal_distr.pdf(X)

            for n in range(self.n_row):
                Ps[n,k] = np.power(self.rates[k], S[n]) * np.exp(-self.rates[k]) / np.m

        likelihood = Px * Ps * self.weights

        likelihood = np.sum(likelihood, axis=1)
        log_likelihood = np.sum(np.log(likelihood), axis=0)

        return log_likelihood
```