Nimara Doumitrou-Daniil
Student, First Year
M.Sc. in Machine Learning
nimara@kth.se

Assignment 2

Machine Learning, advanced : December 2019 :
Due Tuesday, January 7, 2020

Contents

Q2.1

Q2.1.1

I confirm that I have read the instruction text.

Q2.1.2

I have discussed problem formulations with:

1. Adrian Campoy
2. Carles Balsells
3. Fernando Garcia
4. Gustavo Teodoro Beck
5. Livia Qian
6. Lucas Gongora

Q2.1.3

I have not discussed solutions with anybody.

Q2.2

In order to derive the dependencies, we follow the second method of evaluating dependencies, namely using the red dot notation on the graph whenever a d-separation rule occurs. Based on that, we deduced the following:

Q2.2.4

Yes, $\mu_k \perp \tau_k$.

Q2.2.5

No, $\mu_k \not\perp \tau_k | \boldsymbol{X}^1, ..., \boldsymbol{X}^n$.

Q2.2.6

Yes, $\mu \perp \beta'$.

Q2.2.7

No, $\mu \not\perp \beta' | \boldsymbol{X}^1, ...\boldsymbol{X}^n$.

Q2.2.8

No, $\boldsymbol{X}^n \not\perp S^n$.

Q2.2.9

No, $\boldsymbol{X}^n \not\perp S^n | \mu_k, \tau_k$.

2.4

2.4.12

Please refer to the appendix for the implementation, which can be summarized as follows:

1. Generate N iid data $x \sim N(\mu_x, \sigma_x^2)$ (parameters of your choosing).
2. Initialize prior hyperparameters $a_0, b_0, \mu_0, \lambda_0$.
3. Run the VI algorithm and compute $q(\tau)$, $q(\mu)$ (so that $q(\tau,\mu) = q(\tau)q(\mu)$).
4. **Update equations**:
   - a_N: $a_N = a_0 + \frac{N+1}{2}$, N : sample size
   - μ_N: $\mu_N = \frac{\lambda_0\mu_0 + \sum_n x_n}{\lambda_0 + N}$
   - b_N: $b_N = b_0 + \frac{\lambda_0}{2}(E_\mu[\mu^2]) + \mu_0^2 - 2E_\mu[\mu]\mu_0) + \frac{1}{2}\sum_n (x_n^2 + E_\mu[\mu^2] - 2E_\mu[\mu]x_n)$, where $E_\mu[\mu] = \mu_n$ and $E_\mu[\mu^2] = \mu_n^2 + \frac{1}{\lambda_N}$
   - λ_N: $\lambda_N = (\lambda_0 + N)\frac{a_N}{b_N}$

Note that only parameters b_n and λ_n need to be computed iteratively, while a_n, μ_n are computed directly from the data. Finally, we have $\tau \sim G(a_N, b_N)$ and $\mu \sim N(\mu_N, \lambda_N^{-1})$. We can thus compute $q(\tau)\cdot q(\mu) = q(\tau,\mu)$ (approximated posterior).

2.4.13

For this simple problem, we know the exact posterior of the parameters τ, μ. If $\mu \sim N(\mu_0, (\lambda_0\tau)^{-1})$, $\tau \sim G(a_0, b_0)$, then the joint prior follows a Normal-Gamma distribution (Bishop p. 470[1]):

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

The direct posterior can be derived utilizing the Bayes rule:

$$p(\tau,\mu|\boldsymbol{X}) \propto L(\boldsymbol{X}|\tau,\mu)\cdot\pi(\tau,\mu)$$

It can be shown [3] that:

$$p(\tau,\mu|\boldsymbol{X}) = NG(\frac{\lambda_0\mu_0 + N\bar{x}}{\lambda_0 + N}, \lambda_0 + N, a_0 + \frac{N}{2}, b_0 + \frac{1}{2}(N\cdot s + \frac{\lambda_0 N(\bar{x}-\mu_0)^2}{\lambda_0 + N}))$$

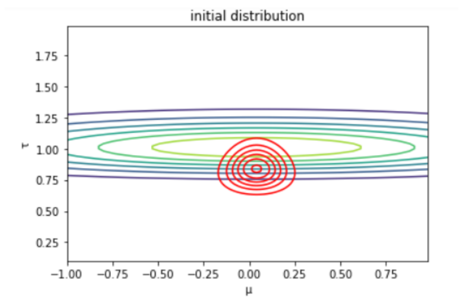
$$\bar{x} = \frac{\sum_n x_n}{N}, \quad s = \frac{\sum_n (x_n - \bar{x})^2}{N}$$

2.4.14

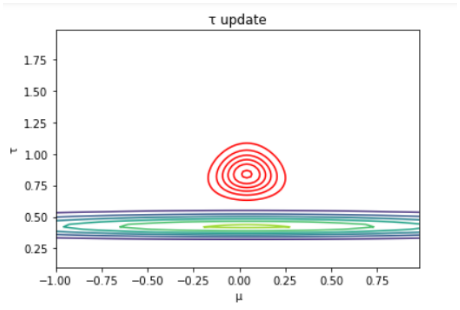
We ran the algorithm on three different cases (each for $N = 100$ samples):

1. $(a_0, b_0, \lambda_0, \mu_0) = (1, 50, 1, 1)$ with $x \sim N(0, 0.5)$
2. $(a_0, b_0, \lambda_0, \mu_0) = (1, 50, 1, 1)$ with $x \sim N(0.5, 0.5)$
3. $(a_0, b_0, \lambda_0, \mu_0) = (0, 0, 0, 0)$ with $x \sim N(0, 0.5)$

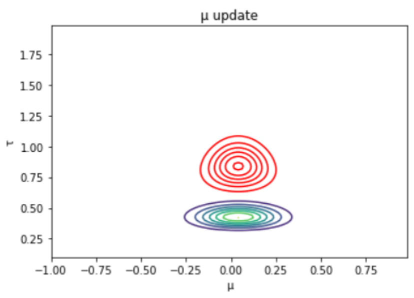
By comparing 1 and 2 we will notice the effect that our mean has on our posterior. Comparing 1 and 3 will show how even with "nonsensical" initial conditions, our distribution will still converge to the corresponding true posterior. We will also see how they differ (although they are both centered at $\mu = 0$).



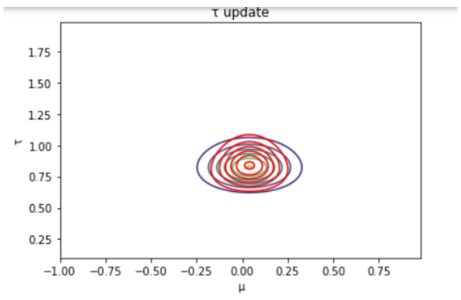
(a) Initial



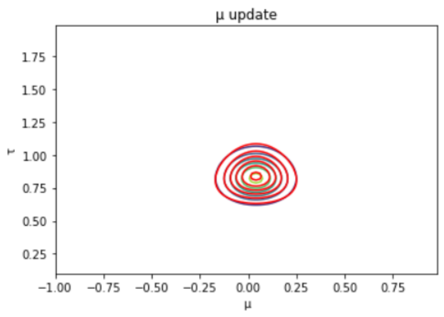
(b) Updating q_τ



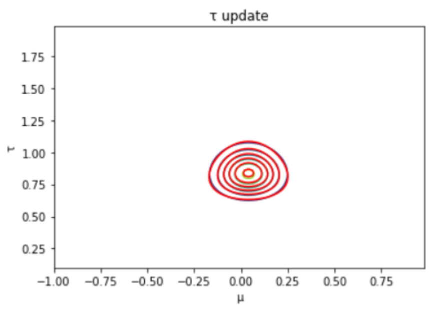
(c) Updating q_μ



(d) Updating q_τ

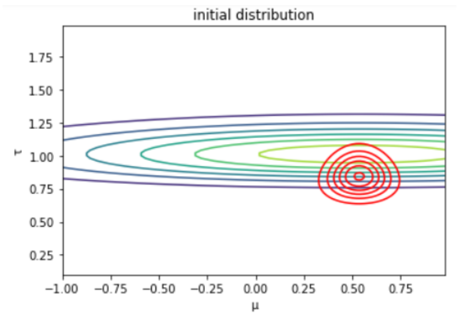


(e) Updating q_μ

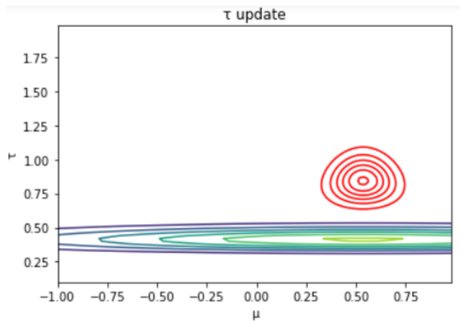


(f) Converged

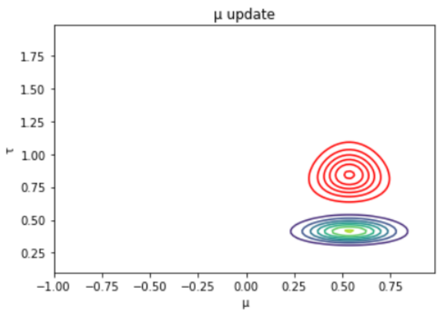
Figure 1: Visualization of VI inference when applied to this simple problem. The red contour corresponds to the real exact posterior, while the other contour to the approximated one. We showcase the initial distribution (after no update) and the distribution after each update (altering b_N and then λ_N and so on). We achieve convergence relatively quickly.



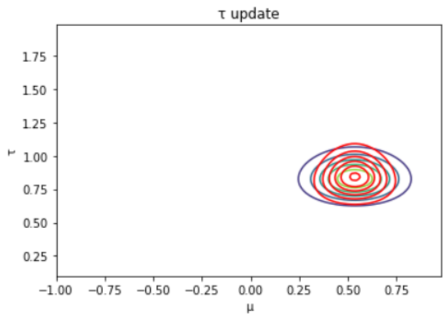
(a) Initial



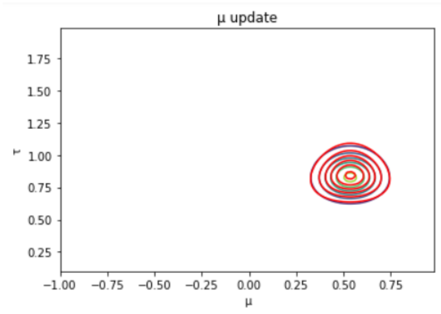
(b) Updating q_τ



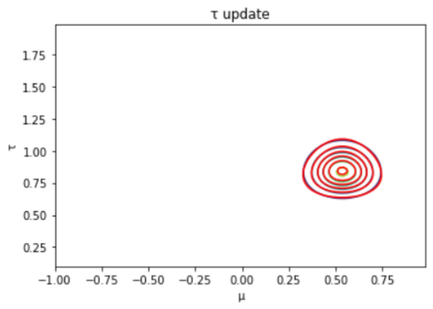
(c) Updating q_μ



(d) Updating q_τ

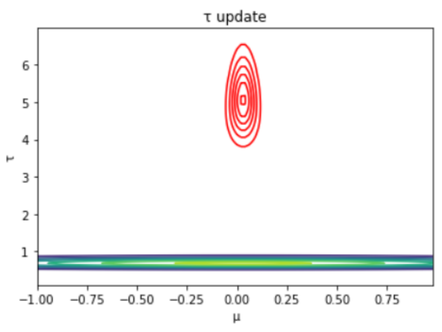


(e) Updating q_τ

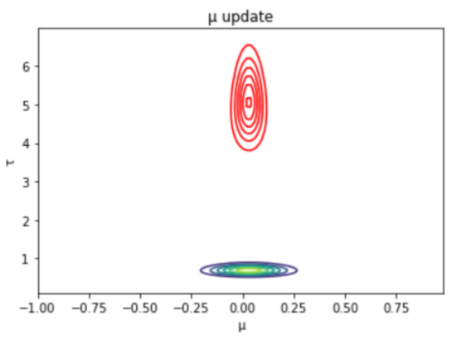


(f) Converged

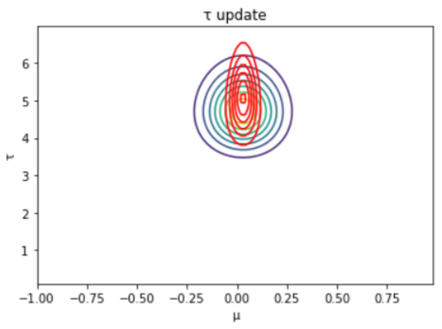
Figure 2: Visualization of VI inference when applied to this simple problem. The red contour corresponds to the real exact posterior, while the other contour to the approximated one. We showcase the initial distribution (after no update) and the distribution after each update (altering b_N and then λ_N and so on). We achieve convergence relatively quickly. Altering the mean of our data shifted the center of our posterior accordingly.



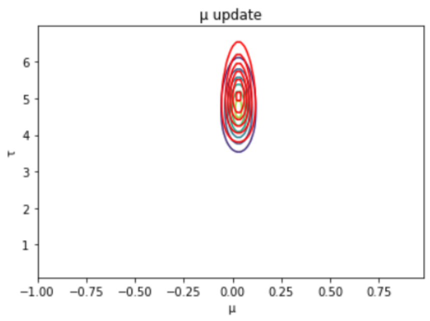
(a) Initial



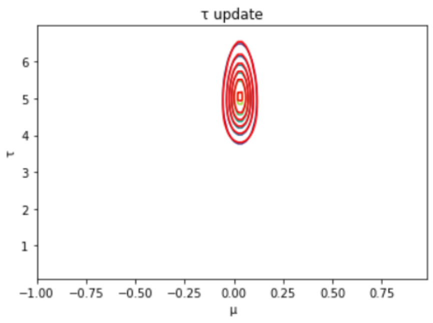
(b) Updating q_τ



(c) Updating q_μ



(d) Updating q_τ



(e) Converged

Figure 3: Visualization of VI inference when applied to this simple problem. The red contour corresponds to the real exact posterior, while the other contour to the approximated one. We do not showcase the initial distribution, for none can be defined for our initial values $(a_0, b_0, \lambda_0, \mu_0) = (0, 0, 0, 0)$. Even with a nonsensical initialization we nevertheless achieve convergence relatively quickly.

Based on these three visualizations as well as other experiments we conducted separately, we can deduce clearly that our posterior is directly affected by our prior hyperparameters as well as the data (as one would expect). Nevertheless, we are still able to converge to the underlying exact distribution after only a few iterations (due to the simplicity of the problem).

The effect of each parameter is captured in the exact form of our posterior:

$$NG(\frac{\lambda_0\mu_0 + N\bar{x}}{\lambda_0 + N}, \lambda_0 + N, a_0 + \frac{N}{2}, b_0 + \frac{1}{2}(N \cdot s + \frac{\lambda_0 N(\bar{x} - \mu_0)^2}{\lambda_0 + N})) = NG(\mu_{real}, \lambda_{real}, a_{real}, b_{real})$$

It can be shown that, under this distribution, $E[\mu] = \mu_{real}$, $E[\tau] = \frac{a_{real}}{b_{real}}$, $V[\mu] = \frac{b_{real}}{\lambda_{real}(a_{real}-1)}$, $V[\tau] = \frac{a_{real}}{b_{real}^2}$. The effect of our priors and data becomes clear through these quantities:

- This distribution is centered (in the μ axis) around $\frac{\lambda_0\mu_0 + N\bar{x}}{\lambda_0 + N}$ which is influenced by our prior hyperparemeters μ_0, λ_0 and our data points $\bar{x}$. Altering those values would reflect in a shift along the μ axis.

- The center along the τ axis is captured by $E[\tau] = \frac{a_{real}}{b_{real}}$, which in turn takes into account all of our hyperparameters, as well as our data's mean $\bar{x}$ and variance s. For instance, increasing a_0 leads to an increase of a_{real} and thus a vertical upward shift along the τ axis. In fact, this is what we see in the third experiment, where we essentially decreased b_{real} more than a_{real} (a_0 decreased by 1, while b_0 by 50), essentially increasing $E[\tau]$, shifting the mean upwards along the τ axis. We also notice an increase in variance $V[\tau]$ (spread along the τ axis) for the same reason.

- The spread of our posterior is captured in the $V[\mu], V[\tau]$, which in turn are affected by all of our hyperparameters as well as the mean and variance of our data points $\boldsymbol{x}$.

- As we increase N, the effect of the prior hyperparameters becomes mitigated by the effect of our data points. Note how, for large N $\mu_{real} \sim \bar{x}$, $\lambda_{real} \sim N$, $a_{real} \sim \frac{N}{2}$, $b_{real} \sim \frac{1}{2}(N \cdot s + \frac{\lambda_0 N(\bar{x}-\mu_0)^2}{\lambda_0 + N}) \sim \frac{1}{2}Ns$. This is why, even though μ_0 is non zero in the first experiment, the mean of the exact posterior is dominated by the mean of our iid x and is centered close to zero. On the other hand, shifting the mean of our iid affected the exact posterior substantially.

Finally, note that, even though we do not showcase this case here, N influences (as expected) the extent to which we approximate our posterior. Lowering N, makes our approximation less accurate (and more similar to the one seen in Bishop, where the distributions, though similar, did not approximate to the degree shown in our graphs).

2.5

2.5.15

Please refer to the appendix for the code implementation of said algorithm. The main premise of it can be summarized as follows:

1. Initialize your initial trees (of fixed length), thetas and pi randomly (tree mixture parameters) and create a corresponding tree mixture
2. Do so for 100 different random seed values ($0 - 99$) and run an EM algorithm for 10 iterations
3. Select the 10 best seeds (highest log likelihood) and run the EM algorithm proposed in the assignment for 100 iterations
4. Select the mixture with the highest log likelihood and plot it (as well as the likelihood)

The EM algorithm is exactly as in the assignment:

- **E-step**: Compute the responsibility $r_{n,k}$
- **M-step**: Update your parameters Θ_k, T_k

When computing the responsibility, we utilized a small number $\epsilon = 2.22044604925 * 10^{-16}$ in order to ensure that our responsibilities $r_{n,k}$ never become 0. In this way, we computed:

$$r_{n,k} = \frac{\pi_k p(x^n|T_k, \Theta_k) + \epsilon}{p(x^n) + K * \epsilon}$$

This subsequently ensured $\pi_k \neq 0$, avoiding degenerate cases.

In the m-step, when updating our tree T_k, we ran the Kruskal-v1 python file, and had to process the result in order to obtain the corresponding topology array. Even though there were multiple isomorphic tree structures, we opted into maintaining the same root and order of our vertices. That is, when presenting a topology array at iteration t of the form $[nan, parent(1), ..., parent(V)]$, then the first node is always x_1^n, the second x_2^n and so on. We also slightly altered the Maximum Spanning Tree algorithm: instead of printing the resulting array, we instead return it.

2.5.16

We ran our algorithm on three distinct plotting the likelihoods of our mixture models (as they evolved after each iteration):

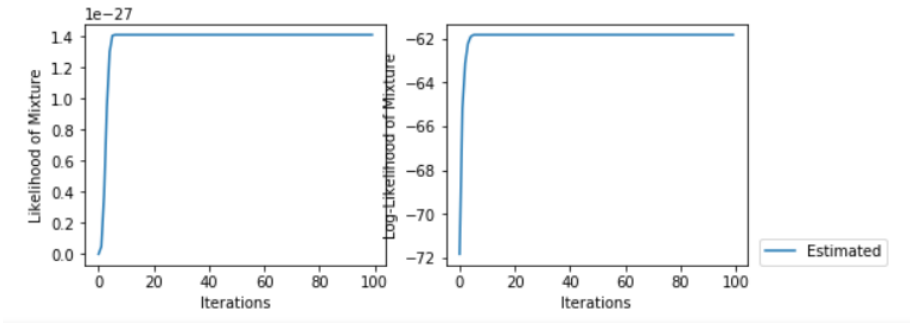


Figure 4: Likelihood for trees of size 10 and sample of size 20

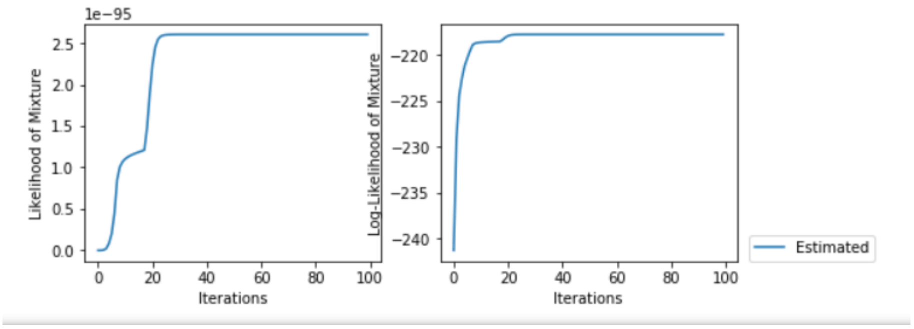


Figure 5: Likelihood for trees of size 10 and sample of size 50

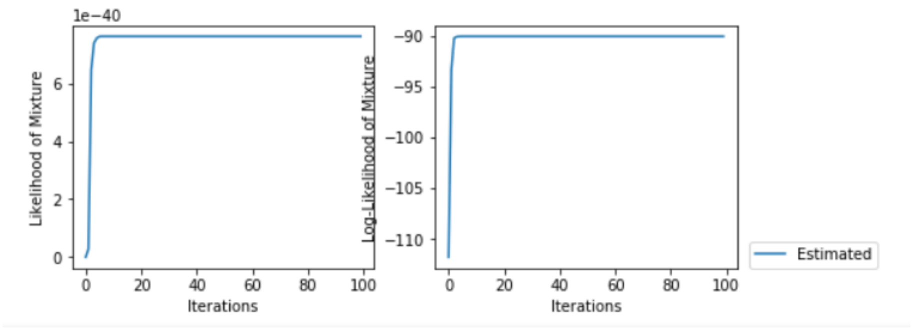


Figure 6: Likelihood for trees of size 20 and sample of size 20

We also compared the inferred models with the real ones, in regards to their tree similarity and their likelihood (rounded to two decimals). Recall that the mixture model likelihood is given by:

$$\ln p(\boldsymbol{X}|\Theta, T, \pi) = \sum_n \ln \big(\sum_k \pi_k p(x^n|\theta_k, T_k)\big)$$

Example	Real Model (log likelihood)	EM Model (log likelihood)
10-size 20-sample	-113.14	-61.82
10-size 50-sample	-280.85	-217.79
20-size 20-sample	-286.37	-90.07

We see that our trees have, in a sense, "overfit" the data, as showcased by the substantially better log likelihood we obtain on the data. Such a result is expected when one takes into account the disproportionate amount of model parameters relative to the small amount of data. In each of the provided test cases, the data never exceeds 50 observations, while our trees have a size of at least 10. Each such tree has many parameters: for example Θ_k has at least $\sim 10 \cdot 2 \cdot 2 = 40$ parameters. This, in combination with the large amount of possible trees that can equivalently produce such observations, is the main reason we observe a lot of 0 and 1 in our resulting Θ arrays and expect to find different trees than the real ones. If we increase the data we have for training (while keeping the other parameters fixed), we expect that the discrepancy between the real and the derived likelihood and trees to reduce. In fact, we can already see this effect in our log likelihoods:

- Keeping the size fixed to 10, and **increasing the sample size** from 20 to 50, **reduced the relative log likelihood improvement** $\frac{|\text{real-inferred}|}{|\text{real}|}$ from $\frac{113.14-61.82}{113.14} \sim 0.45$ to $\frac{280.85-217.79}{280.85} \sim 0.22$.
- Similarly, keeping the sample sized fixed to 20, and increasing the size of the trees to 20 (thus, **increasing the dimensionality of our parameters**) **increased the relative log likelihood improvement** from $\frac{113.14-61.82}{113.14} \sim 0.45$ to $\frac{286.37-90.07}{286.37} \sim 0.69$

Comparing them based on the Robinson-Foulds distance we obtain:

inferred vs real (10-20)	t_0	t_1	t_2	t_3
rt_0	8	8	8	8
rt_1	11	7	9	7
rt_2	11	9	9	11
rt_3	9	9	9	9

Table 1: Robinson-Foulds distance between real and inferred trees. We compute the distance between each pair of trees, since during the EM training, the order of the trees can be shuffled around.

inferred vs real (10-50)	t_0	t_1	t_2	t_3
rt_0	8	7	9	9
rt_1	11	8	8	10
rt_2	11	8	8	8
rt_3	9	8	8	8

Table 2: Robinson-Foulds distance between real and inferred trees. We compute the distance between each pair of trees, since during the EM training, the order of the trees can be shuffled around.

inferred vs real (20-20)	t_0	t_1	t_2	t_3
rt_0	21	21	22	19
rt_1	19	19	20	17
rt_2	23	21	16	21
rt_3	20	22	25	20

Table 3: Robinson-Foulds distance between real and inferred trees. We compute the distance between each pair of trees, since during the EM training, the order of the trees can be shuffled around.

As expected, due to to complexity of the model and the lacking (in size) data, we are unable to obtain an exact copy of our trees (zero distance). However, we observe what we explained earlier: **increasing** the sample size, **decreases** the average R-F distance (compare the 10-20 case with the 10-50).

2.5.17

We wish to further analyze this problem, by addressing the previous discrepancies (in likelihood and tree similarity) in cases where the proportion $\frac{\text{data}}{\text{complexity}}$ is more reasonable. We expect to smoothen the discrepancy as said ratio becomes increasingly reasonable.

We ran the following experiments:

- Mixture model of three trees of size 5 utilizing sample size of 20
- Mixture model of three trees of size 5 utilizing sample size of 200
- Mixture model of five trees of size 5 utilizing sample size of 200

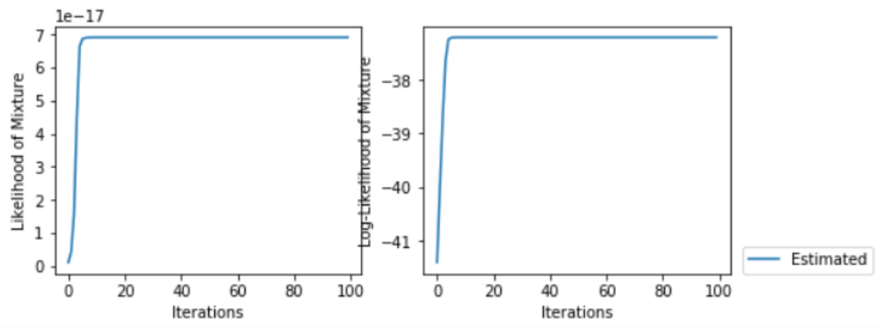


Figure 7: Likelihood for three trees of size 5 and sample of size 20

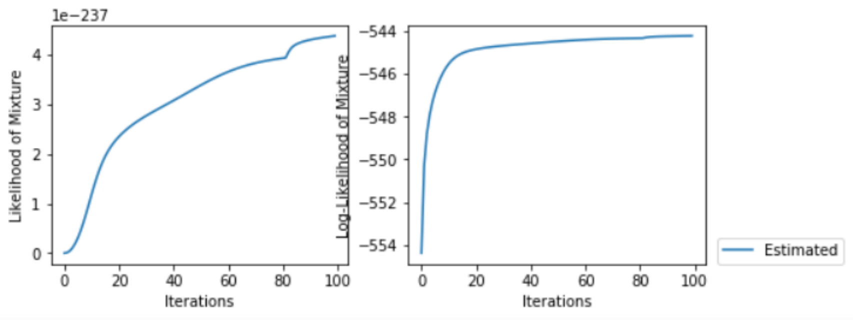


Figure 8: Likelihood for three trees of size 5 and sample of size 200

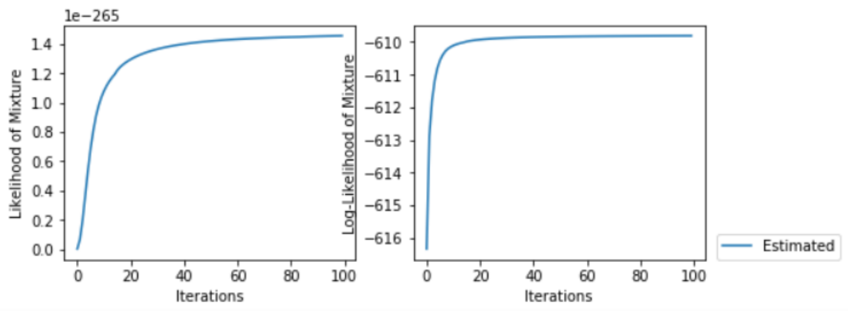


Figure 9: Likelihood for five trees of size 5 and sample of size 200

Example	Real Model (log likelihood)	EM Model (log likelihood)
5-size 20-sample	-55.02	-37.21
5-size 200-sample	-557.06	-544.23
5-size 200-sample 5-cluster	-625.40	-609.81

inferred vs real (2-20)	t_0	t_1	t_2
rt_0	4	4	4
rt_1	5	3	5
rt_2	4	0	4

Table 4: Robinson-Foulds distance between real and inferred trees. We compute the distance between each pair of trees, since during the EM training, the order of the trees can be shuffled around.

inferred vs real (2-200)	t_0	t_1	t_2
rt_0	4	4	4
rt_1	3	3	3
rt_2	0	2	4

Table 5: Robinson-Foulds distance between real and inferred trees. We compute the distance between each pair of trees, since during the EM training, the order of the trees can be shuffled around.

inferred vs real (2-200)	t_0	t_1	t_2	t_3	t_4
rt_0	4	4	4	4	3
rt_1	5	3	3	5	0
rt_2	6	4	4	4	3
rt_3	6	4	4	6	1
rt_4	3	5	5	3	4

Table 6: Robinson-Foulds distance between real and inferred trees. We compute the distance between each pair of trees, since during the EM training, the order of the trees can be shuffled around.

As expected, due to the relative simplicity (size 5) of the underlying trees, 20 samples generate a model more similar to the real one than in the previous experiments. In fact, we even manage to capture one of the trees (see rt_2 in the 2-20 and 2-200 experiment). We also see:

- The relative log likelihood increase is smaller than the previous experiments: $\frac{55.02-37.21}{55.02} \sim 0.32$.
- Augmenting the sample size decreases it even further to $\frac{557.06-544.23}{557.06} \sim 0.023$
- Increasing the complexity of our mixture model, by adding more components-clusters (5 instead of 3 trees), increases the relative increase from 0.023 to $\frac{625.4-609.81}{625.4} \sim 0.025$. Note how the increase is not as substantial.
- We also see that we manage to capture one of the trees in each case. In the second case, we note that increasing the sample size produced trees that are on average closer to the real ones (see rt_1). Lastly, increasing the number of clusters made the problem more complex which manifests itself in the RF metric which exhibits a higher (on average) distance.

In summary, this exercise explores the robustness of the EM algorithm when applied to complex mixture models and highlights the pivotal role of the sample size in approximating the real underlying distribution.

Q2.6

We aim to derive an EM algorithm for the parameters $\boldsymbol{\theta} = (\boldsymbol{\pi}, \boldsymbol{\mu_1}, ..., \boldsymbol{\mu_K}, \boldsymbol{\tau_1}, ..., \boldsymbol{\tau_K}, \lambda_1, ..., \lambda_k)$, where $\boldsymbol{\mu_k}$ and $\boldsymbol{\tau_k}$ are two dimensional, λ_k as described in the assignment. All of these parameters describe the distribution of our random variables Z_n, X_n (two dimensional), S_n.

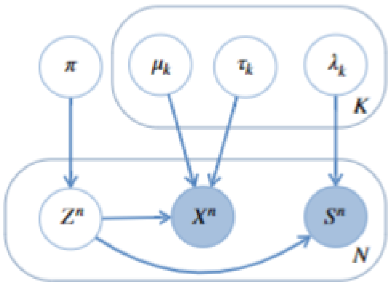


Figure 10: Graphical model of this problem.

The derivation for the EM algorithm is standard. In fact, in this simple case, one could work per coordinate of μ_k, τ_k, since X^n variable is independent (it follows a two dimensional Gaussian with a diagonal precision matrix with its diagonal given by the $\boldsymbol{\tau_k} = (\tau_{k1}, \tau_{k2})$). Then, for instance, the M step for the τ_{k1} will be a function of x_1^n (first coordinate) only and so on. However, even though this would make our calculations easier, we choose to derive the EM algorithm in a more general approach, looking at the whole joint two dimensional distribution.

First, we compute the expected log likelihood:

$$\sum_n E_{p(z^n|x^n,s^n,\theta^t)}[\log p[x^n,s^n,z^n|\theta] = \sum_n E_{p(z^n|x^n,s^n,\theta^t)}[\log p[x^n|z^n,\theta]p[s^n|z^n,\theta]p[z^n|\theta]]$$

$$= \sum_n E_{p(z^n|x^n,s^n,\theta^t)}[\log \prod_k (p[x^n|z^n=k,\theta_k]p[s^n|z^n=k,\theta_k]p[z^n=k|\theta_k])^{I(z^n=k)}]$$

$$= \sum_{n,k} E_{p(z^n|x^n,s^n,\theta^t)}[I(z^n=k)(\log p[x^n|z^n,\theta_k] + \log p[s^n|z^n,\theta_k] + \log p[z^n|\theta_k]])$$

$$= \sum_{n,k} E_{p(z^n|x^n,s^n,\theta^t)}[I(z^n=k)(\log N(\boldsymbol{x}^n|\boldsymbol{\mu_k},\boldsymbol{\tau_k}^{-1}I) + \log Poisson(s^n|\lambda_k) + \log \pi_k])$$

$$= \sum_{n,k} \underbrace{E_{p(z^n|x^n,s^n,\theta^t)}[I(z^n=k)]}_{r_{n,k}}(\log N(\boldsymbol{x}^n|\boldsymbol{\mu_k},\boldsymbol{\tau_k}^{-1}I) + \log Poisson(s^n|\lambda_k) + \log \pi_k)$$

$$= \underbrace{\sum_k [\sum_n r_{n,k}] \log \pi_k}_{A} + \sum_k \underbrace{\sum_n r_{n,k}(\log N(\boldsymbol{x}^n|\boldsymbol{\mu_k},\boldsymbol{\tau_k}^{-1}I) + \log Poisson(s^n|\lambda_k))}_{B_k}$$

We minimize this expression by first maximizing A and then B_k. When we examine A, we can notice that this expression is the same as the one we found when we tried to maximize a categorical distribution with a likelihood of:

$$\prod_k \pi_k^{\sum_n r_{n,k}}$$

In that case, we found in class (using Lagrange multipliers) that the optimal π_k were given by:

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

For our term B_k:

$$\log N(\boldsymbol{x}^n|\boldsymbol{\mu_k},\boldsymbol{\tau_k}^{-1}I) = \underbrace{\log N(x_1^n|\mu_{k1},\tau_{k1}^{-1}) + \log N(x_2^n|\mu_{k2},\tau_{k2}^{-1})}_{\text{independence}}$$

$$= \frac{\log \tau_{k1}}{2} - \frac{\tau_{k1}}{2}(x_1^n - \mu_{k1})^2 + \frac{\log \tau_{k2}}{2} - \frac{\tau_{k2}}{2}(x_2^n - \mu_{k2})^2 + C$$

Similarly, since:

$$Poisson(s^n|\lambda_k) = \frac{\lambda_k^{s^n} e^{\lambda_k}}{s^n!}$$

$$\log Poisson(s^n|\lambda_k) = s^n \log \lambda_k + \lambda_k + C$$

Finally, our expression for B_k is:

$$f_k(\mu_{k1}, \mu_{k2}, \tau_{k1}, \tau_{k2}, \lambda_k) = \sum_n r_{n,k}[\frac{\log \tau_{k1}}{2} - \frac{\tau_{k1}}{2}(x_1^n - \mu_{k1})^2 + \frac{\log \tau_{k2}}{2} - \frac{\tau_{k2}}{2}(x_2^n - \mu_{k2})^2 + s^n \log \lambda_k + \lambda_k] + C$$

Using simple calculus (derivation):

$$\frac{\partial f}{\partial \mu_{k1}} = \sum_n r_{n,k}\tau_{k1}(x_1^n - \mu_{k1}) = 0 \iff \mu_{k1} = \frac{\sum_n r_{n,k} x_1^n}{r_k}, \; r_k = \sum_n r_{n,k}$$

Similarly:

$$\mu_{k2} = \frac{\sum_n r_{n,k} x_2^n}{r_k}$$

For the precision:

$$\frac{\partial f}{\partial \tau_{k1}} = \sum_n r_{n,k}[\frac{1}{2\tau_{k1}} - \frac{(x_1^n - \mu_{k1})^2}{2}] = 0 \iff \sigma_{k1}^2 = \frac{1}{\tau_{k1}} = \frac{\sum_n r_{n,k}(x_1^n - \mu_{k1})^2}{r_k}$$

Again:

$$\sigma_{k2}^2 = \frac{1}{\tau_{k2}} = \frac{\sum_n r_{n,k}(x_2^n - \mu_{k2})^2}{r_k}$$

Lastly:

$$\frac{\partial f}{\partial \lambda_k} = \sum_n r_{n,k}[\frac{s^n}{\lambda_k} - 1] = 0 \iff \lambda_k = \frac{\sum_n r_{n,k} s^n}{r_k}$$

We have yet to compute $r_{n,k}$ however:

$$r_{n,k} = E_{p(z^n|x^n,s^n,\theta^t)}[I(z^n = k)] = p(z^n = k|x^n, s^n, \theta^t) = \frac{p(x^n, s^n|z^n = k, \theta^t)p(z^n = k|\theta^t)}{p(x^n, s^n|\theta^t)}$$

$$= \frac{\pi_k N(\boldsymbol{x^n}|\boldsymbol{\mu_k}, \boldsymbol{\tau_k}^{-1}I)Poisson(s^n|\lambda_k)}{\sum_j \pi_j N(\boldsymbol{x^n}|\boldsymbol{\mu_j}, \boldsymbol{\tau_j}^{-1}I)Poisson(s^n|\lambda_j)}$$

This concludes our EM algorithm. To summarize:

1. **E-Step**:

$$r_{n,k} = \frac{\pi_k N(\boldsymbol{x^n}|\boldsymbol{\mu_k}, \boldsymbol{\tau_k}^{-1}I)Poisson(s^n|\lambda_k)}{\sum_j \pi_j N(\boldsymbol{x^n}|\boldsymbol{\mu_j}, \boldsymbol{\tau_j}^{-1}I)Poisson(s^n|\lambda_j)}$$

2. **M-Step**:

- $\pi_k = \frac{\sum_n r_{n,k}}{\sum_{n,k} r_{n,k}} = \frac{\sum_n r_{n,k}}{N} = \frac{r_k}{N}$
- $\mu_{ki} = \frac{\sum_n r_{n,k} x_i^n}{r_k}, \; i \in \{1, 2\}$
- $\tau_{ki} = \frac{r_k}{\sum_n r_{n,k}(x_i^n - \mu_{ki})^2}, \; i \in \{1, 2\}$
- $\lambda_k = \frac{\sum_n r_{n,k} s^n}{r_k}$

As noted earlier, the parameters τ_{ki}, μ_{ki} could have been derived by working **per coordinate** (τ_{ki} does not depend on μ_{kj} or x_j^n, $i \neq j$ and vice versa). Furthermore, the parameter estimations are sensible:

1. Parameters $\boldsymbol{\mu_k}$ and λ_k determine the mean of the distributions of $\boldsymbol{x^n}$ and s^n respectively. Thus, taking a weighted (with weights the responsibilities $r_{n,k}$) average seems reasonable.

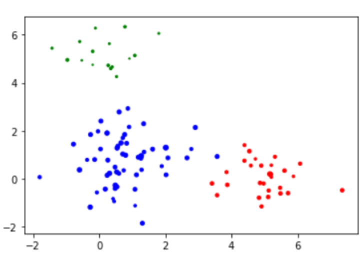
2. Parameter $\boldsymbol{\tau_k}$ refers to the precision of $\boldsymbol{x^n}$. Notice how our estimation corresponds to a "weighted" version of the inverse of the variance.

Q2.6.19

The implementation of the code was fairly straight forward, given our previous calculations. We simply had to iteratively compute the responsibility $r_{n,j}$ and update our parameters $\pi_k, \mu_{ki}, \tau_{ki}, \lambda_k$ until our expected log likelihood converged to a certain value (see appendix).

Q2.6.20

We ran the algorithm on the proposed data sets, one consisting of three components ($k = 3$), while the other of two ($k = 2$):


(a) Three component data set

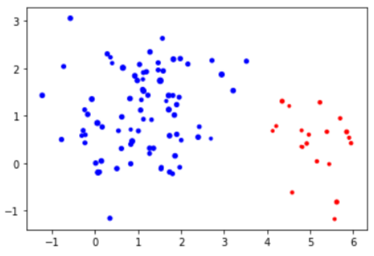
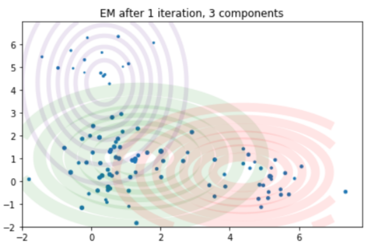
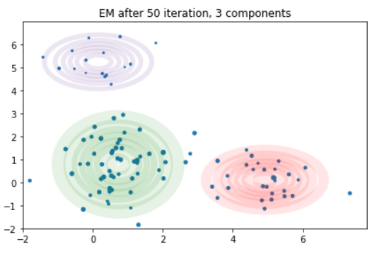

(b) One Component data set

Figure 11: Note we are only showcasing X.

We then ran the algorithm, visualizing the resulting Gaussian Mixture (μ_k, τ_k):

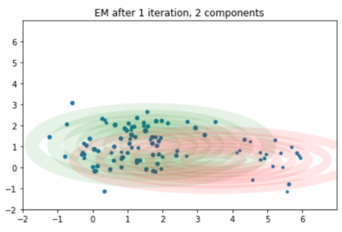


(a) After one iteration

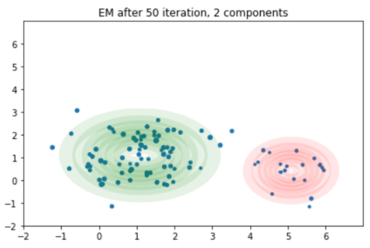


(b) After fifty iterations

Figure 12: Note that we reached convergence before 50 iterations (we only needed 16).



(a) After one iteration



(b) After fifty iterations

Figure 13: Note that we reached convergence before 50 iterations (we only needed 6).

The algorithms seem to have converged relatively quickly and the mixture distributions seem adequate. Since we generated the data ourselves, we can also compare our found parameters with those that generated the data. Note that it is not necessary to find the mixture models **in the same order**. For the first case, rounded to two decimal places and after 16 iterations (we also reordered the result accordingly so that the mixtures appear in the same order):

Parameter	Real	EM
π	$\begin{bmatrix}0.25\\0.5\\0.25\end{bmatrix}$	$\begin{bmatrix}0.29\\0.55\\0.16\end{bmatrix}$
μ	$\begin{bmatrix}5 & 0\\1 & 1\\0 & 5\end{bmatrix}$	$\begin{bmatrix}4.92 & 0.11\\0.73 & 0.78\\0.15 & 5.24\end{bmatrix}$
Σ	$\begin{bmatrix}0.5 & 0\\0 & 0.5\end{bmatrix}, \begin{bmatrix}0.92 & 0\\0 & 0.91\end{bmatrix}, \begin{bmatrix}0.5 & 0\\0 & 0.5\end{bmatrix}$	$\begin{bmatrix}0.72 & 0\\0 & 0.43\end{bmatrix}, \begin{bmatrix}0.74 & 0\\0 & 1\end{bmatrix}, \begin{bmatrix}0.62 & 0\\0 & 0.36\end{bmatrix}$
λ	$\begin{bmatrix}11.11\\14.22\\5.96\end{bmatrix}$	$\begin{bmatrix}11.53\\15.07\\5.19\end{bmatrix}$

Even though we have not converged to the real parameters (more data would almost surely reduce the differences as we saw in 2.5), we can say that, based on the parameters and the visualization, our algorithm has successfully approximated the real mixture model.

Q2.7

One could again work per coordinate (in order to make the calculations simpler), however we opt against it once more.

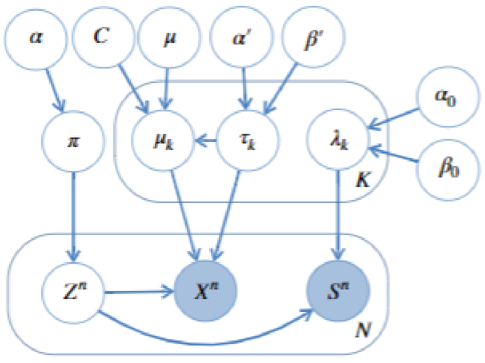


Figure 14: Graphical model of problem 2.7

In variational inference (VI), we aim to compute the posterior distribution of our latent variables $p(Z, \Pi, \Lambda, M, T|X^n, S^n)$ (recall that in the Bayesian approach, we would marginalize them out using this posterior). However, computing this posterior is often intractable. We instead aim to approximate it by a "simpler" (satisfies some assumptions) distribution $q(Z, \Pi, \Lambda, M, T)$. Our assumption is that

$$q(Z, \Pi, \Lambda, M, T) = q(Z)q(\Pi, \Lambda, M, T)$$

First, we examine the joint distribution (factorizing as the graphical model dictates):

$$p(S, X, Z, \Pi, M, T, \Lambda) = \underbrace{p(X|Z, M, T)p(Z|\Pi)p(S|Z, \Lambda)}_{p(X,Z,S|M,\Lambda,T,\Pi)} p(\Pi)p(M|T)p(T)p(\Lambda)$$

We next compute the complete likelihood:

$$p(X, Z, S|M, \Lambda, T, \Pi) = \prod_n [p(x^n|z^n, M, T)p(z^n|\Pi), p(s^n|z^n, \Lambda)] = \prod_{n,k} [p(x^n|\boldsymbol{\mu_k}, \boldsymbol{\tau_k}) \underbrace{p(z_{n,k} = 1)}_{\pi_k} p(s^n|\lambda_k)]^{z_{n,k}}$$

where $z^n = k \iff z_{n,k} = 1 \wedge z_{n,k'} = 0 \; \forall \; k' \neq k$

Then (only the complete likelihood depends on z):

$$\ln q^*(z) = E_{\Pi,M,T,\Lambda}[\ln p(S, X, \Pi, M, T, \Lambda)] \overset{+}{=} E_{\Pi,M,T,\Lambda}[\ln p(X, Z, S|M, \Lambda, T, \Pi)]$$

$$= \sum_{n,k} z_{n,k}[E_{M,T}[\ln N(x^n, |\boldsymbol{\mu_k}, \boldsymbol{\tau_k}^{-1} I) + E_\Pi[\ln \pi_k] + E_\Lambda[\ln Poisson(\lambda_k)]]$$

$$= \sum_{n,k} z_{n,k}[\frac{E_{\tau_{k1}}[\ln \tau_{k1}]}{2} - \frac{E_{\tau_{k1}}[\tau_{k1}]}{2} E_{\mu_{k1}}[(x_1^n - \mu_{k1})^2] + \frac{E_{\tau_{k2}}[\ln \tau_{k2}]}{2} - \frac{E_{\tau_{k2}}[\tau_{k2}]}{2} E_{\mu_{k2}}[(x_2^n - \mu_{k2})^2] -$$

$$- \ln(2\pi) + s^n E_{\lambda_k}[\log \lambda_k] + E_{\lambda_k}[\lambda_k] - \ln(s^n!) + E_\Pi[\ln \Pi_k]] = \sum_{n,k} z_{n,k} \ln \rho_{n,k}$$

Thus $q^*(z) \propto \prod_{n,k} \rho_{n,k}^{z_{n,k}}$, implying that $q^*(z) = \prod_{n,k} r_{n,k}^{z_{n,k}}$, $r_{n,k} = \frac{\rho_{n,k}}{\sum_k \rho_{n,k}}$. We will come back to the computation of $\rho_{n,k}$. We have thus computed $q(z)$, and now we must calculate $q^*(M, T, \Lambda, \Pi)$:

$$\ln q^*(M, T, \Pi, \Lambda) = E_z[\ln p(X, Z, S|M, \Lambda, T, \Pi) + \underbrace{\ln p(\Pi) + \ln p(M, T) + \ln P(\Lambda)}_{\text{independent of z}}] =$$

$$= E_z[\ln p(X, Z, S|M, \Lambda, T, \Pi)] + \ln p(\Pi) + \ln p(M, T) + \ln P(\Lambda)$$

$$\stackrel{\pm}{=} \underbrace{\sum_{n,k} E_z[z_{n,k}] \ln N(x^n|\boldsymbol{\mu_k}, \boldsymbol{t_k}^{-1}I) + \sum_{n,k} E_z[z_{n,k}] \ln \pi_k + \sum_{n,k} E_z[z_{n,k}] \ln Poisson(s^n|\lambda_k) +}_{E_z[\ln p(X,Z,S|M,\Lambda,T,\Pi)]}$$

$$+ \ln p(\Pi) + \ln p(M,T) + \ln P(\Lambda)$$

Notice how the terms of Π do not interact with those of (M,T) which in turn do not interact with Λ (as expected, since they are independent, according to the graphical model). This implies that we can process these three groups separately, and write $q(M,T,\Lambda,\Pi) = q(\Pi)\prod_k p(\lambda_k)p(\boldsymbol{\mu_k}, \boldsymbol{\tau_k})$. Lastly (as we saw in class), $E_z[z_{n,k}] = r_{n,k}$.

Before we continue, let us refresh our knowledge on the likelihoods of Gamma(a,b), $N(\mu,\tau)$, Dir(a) distributions:

Y	log-likelihood
Gamma(a,b)	$(a-1)\ln Y - bY$
$N(\mu,\tau^{-1})$	$-\frac{\tau}{2}Y^2 + \mu\tau Y$
Dir(a)	$(a-1)\ln Y$

Let us first examine Π, and more specifically, $q(\Pi)$. Focusing in the above expression in places where Π appears, we have:

$$\sum_{n,k} r_{n,k} \ln \pi_k + \underbrace{\sum_k (a_k - 1)\ln \pi_k}_{\ln p(\Pi)} = \sum_k (a_k - 1 + \sum_n r_{n,k}) \ln \pi_k$$

Therefore, $q^*(\Pi) = Dir(a^*)$:

$$a_k^* = a_k + \sum_n r_{n,k}$$

Let us now examine λ_k:

$$\sum_{n,k} r_{n,k} \underbrace{(s^n \ln \lambda_k - \lambda_k)}_{\ln Poisson(s^n|\lambda_k)} + \underbrace{\sum_k (a_0 - 1)\ln \lambda_k - \beta_0 \lambda_k}_{\ln P(\Lambda)}$$

Focusing on each λ_k:

$$\sum_n r_{n,k}(s^n \ln \lambda_k - \lambda_k) + (a_0 - 1)\ln \lambda_k - \beta_0\lambda_k = (a_0 + \sum_n r_{n,k}s^n)\ln \lambda_k - (\beta_0 + \sum_n r_{n,k})\lambda_k$$

Thus, $q^*(\lambda_k) = Ga(\lambda_k|a_0^*, \beta_0^*)$:

$$a_0^* = a_0 + \sum_n r_{n,k}s^n$$

$$\beta_0^* = \beta_0 + \sum_n r_{n,k}$$

Finally, let us examine (τ_{ki}, μ_{ki}):

$$\stackrel{\pm}{=} \underbrace{\sum_n r_{n,k}[\frac{\ln \tau_{k1}}{2} - \frac{\tau_{k1}}{2}(x_1^n - \mu_{k1})^2 + \frac{\ln \tau_{k2}}{2} - \frac{\tau_{k2}}{2}(x_2^n - \mu_{k2})^2] +}_{\sum_n E_z[z_{n,k}] \ln N(x^n|\boldsymbol{\mu_k}, \boldsymbol{t_k}^{-1}I)}$$

$$+ \underbrace{(a' - 1)(\ln \tau_{k1} + \ln \tau_{k2}) - \beta'(\tau_{k1} + \tau_{k2})}_{\ln p(\boldsymbol{\tau_k}) = \ln p(\boldsymbol{\tau_{k1}}) + \ln p(\boldsymbol{\tau_{k2}})} +$$

$$+ \underbrace{\frac{\ln \tau_{k1}}{2} - \frac{C\tau_{k1}}{2}(x_1^n - \mu_{k1})^2 + \frac{\ln \tau_{k2}}{2} - \frac{C\tau_{k2}}{2}(x_2^n - \mu_{k2})^2}_{\ln p(\boldsymbol{\mu_k}|\boldsymbol{\tau_k}) = \ln p(\boldsymbol{\mu_{k1}}|\boldsymbol{\tau_{k1}}) + \ln p(\boldsymbol{\mu_{k2}}|\boldsymbol{\tau_{k2}})}$$

By grouping up the terms for the Gaussian μ_{ki} (linear and quadratic) we can deduce that $q^*(\mu_{ki}) = N(\mu_i^*, \tau_i^*)$:

$$\tau_i^* = \tau_{ki}(C + \sum_n r_{k,n})$$

$$\mu_i^* = \frac{\tau_{ki} \sum_n r_{k,n} x_i^n + C\mu\tau_{ki}}{\tau_i^*}$$

while $q^*(\tau_{ki}) = Ga(a_i'^*, b_i'^*) = q^*(\tau_{ki}, \mu_{ki}) - q^*(\mu_{ki}|\tau_{ki})$:

$$a_i'^* = a' + \frac{1}{2}\sum_n r_{k,n}$$

$$b_i'^* = b' + \frac{C\mu^2}{2} + \frac{1}{2}\sum_n r_{k,n} x_i^n$$

Finally, $\rho_{n,k}$ is such that:

$$\ln \rho_{n,k} = \frac{E_{\tau_{k1}}[\ln \tau_{k1}]}{2} - \frac{E_{\tau_{k1}}[\tau_{k1}]}{2} E_{\mu_{k1}}[(x_1^n - \mu_{k1})^2] + \frac{E_{\tau_{k2}}[\ln \tau_{k2}]}{2} - \frac{E_{\tau_{k2}}[\tau_{k2}]}{2} E_{\mu_{k2}}[(x_2^n - \mu_{k2})^2] -$$

$$- \ln(2\pi) + s^n E_{\lambda_k}[\log \lambda_k] + E_{\lambda_k}[\lambda_k] - \ln(s^n!) + E_\Pi[\ln \Pi_k]]$$

We know the distributions of these variables, and thus:

- $E_{\mu_{ki}}[(x_i^n - \mu_{ki})^2]$ can be computed since $\mu_{ki} \sim N(\mu_i^*, \tau_i^*)$.
- Similarly, $E_{\tau_{ki}}[\tau_{ki}] = \frac{a_i'^*}{b_i'^*}$, since $\tau_{ki} \sim Ga(a_i'^*, b_i'^*)$. Equivalently, $E_{\lambda_k}[\lambda_k] = \frac{a_0^*}{\beta_0^*}$.
- $E_{\tau_{ki}}[\ln \tau_{ki}] = -\ln b_i'^* + \psi(a_i'^*)$, $E_{\lambda_k}[\ln \lambda_k] = -\ln \beta_0'^* + \psi(a_0'^*)$ ([6]).
- $E_{\pi_k}[\ln \pi_k] = \psi(a_k^*) - \psi(\sum_j a_j^*)$ ([7]), $\psi()$ the digamma function.

2.8

2.8.22

We first wish to derive an algorithm that computes:

$$s(u,i) = P[\sum X_{\downarrow u \cap O} = \text{odd} | X_u = i]$$

Where O: is the set of observations (leafs), and thus $\sum X_{\downarrow u \cap O} = \text{odd}$ describes the sum of the observations (leafs) located **below** u. Imagine u has two children, v and w. Then, since we conditioned on X_u, the observations located in the v subtree are independent of those located in the w subtree:

$$s(u,i) = P[\sum X_{\downarrow u \cap O} = \text{odd} | X_u = i] =$$

$$P[(\sum X_{\downarrow v \cap O} = \text{odd} \cap \sum X_{\downarrow w \cap O} = \text{even}) \cup (\sum X_{\downarrow v \cap O} = \text{even} \cap \sum X_{\downarrow w \cap O} = \text{odd}) | X_u = i]$$

$$= P[\sum X_{\downarrow v \cap O} = \text{odd} | X_u = i] \cdot P[\sum X_{\downarrow w \cap O} = \text{even} | X_u = i] +$$

$$P[\sum X_{\downarrow v \cap O} = \text{even} | X_u = i] \cdot P[\sum X_{\downarrow w \cap O} = \text{odd} | X_u = i]$$

Let us compute $P[\sum X_{\downarrow v \cap O} = \text{odd} | X_u = i]$ (the other expression is very similar):

$$P[\sum X_{\downarrow v \cap O} = \text{odd} | X_u = i] = \sum_{j \in [K]} P[\sum X_{\downarrow v \cap O} = \text{odd}, X_v = j | X_u = i]$$

$$= \sum_{j \in [K]} P[\sum X_{\downarrow v \cap O} = \text{odd} | X_v = j] \cdot P[X_v = j | X_u = i] = \sum_{j \in [K]} s(v,j) \cdot \theta_v(j,i)$$

Thus:

$$s(u,i) = (\sum_{j\in[K]} s(v,j)\cdot\theta_v(j,i))\cdot(\sum_{j\in[K]} \underbrace{(1-s(w,j))}_{p(even)=1-p(odd)} \cdot\theta_w(j,i))+(\sum_{j\in[K]} (1-s(v,j))\cdot\theta_v(j,i))\cdot(\sum_{j\in[K]} s(w,j)\cdot\theta_w(j,i))$$

We can compute this quantity recursively (utilizing dynamic programming), but we still need to define the base of this recursion. If $u \in O$ (u is a leaf), then trivially:

$$s(u,i) = P[\sum X_{\downarrow u\cap O} = \text{odd}|X_u = i] = P[X_u = odd|X_u = i] = \begin{cases} 1 & i \text{ is odd} \\ 0 & \text{otherwise} \end{cases}$$

This concludes the bottom-up Dynamic programming algorithm which computes $s(u,i) = P[\sum X_{u\cap O} = \text{odd}|X_u = i]$.

We can use this quantity to sample from our distribution in a root-to-leaves (similar to ancestral sampling) fashion:

1. Sample from the root r. Since we know that our leaves have an odd sum, the root's distribution is

$$P[X_r = i|\sum X_o = \text{odd}] \propto P[\sum X_o = \text{odd}|X_r = i]\cdot P[X_r = i] = s(r,i)\cdot\theta_r(i)$$

   That is,

$$P[X_r = i|\sum X_o = \text{odd}] = C\cdot s(r,i)\cdot\theta_r(i),\ C^{-1} = P[\sum X_o = \text{odd}] = \sum_{i\in[K]} s(r,i)\cdot\theta_r(i)$$

2. Let u be a child of $r = pa(u)$. Compute:

$$P[\sum X_{\downarrow u\cap O} = \text{odd }|X_{pa(u)} = j] = \sum_i P[\sum X_{\downarrow u\cap O} = \text{odd },X_u = i|X_{pa(u)} = j] = \sum_i s(u,i)\cdot\theta_u(i,j)$$

   Flip a coin (Bernoulli) based on this distribution to decide whether $\sum X_{\downarrow u\cap O} = \text{odd}$ or $\sum X_{\downarrow u\cap O} = \text{even}$. Because we know (recursively) the value of $\sum X_{\downarrow pa(v)\cap O} mod(2)$, $\sum X_{\downarrow u\cap O} mod(2)$ directly affects $(\sum X_{\downarrow v\cap O})mod(2)$, v the other child of $pa(u)$, since $(\sum X_{\downarrow pa(u)\cap O})mod(2) = (\sum X_{\downarrow u\cap O} + \sum X_{\downarrow v\cap O})mod(2)$. For instance, when $pa(u) = \text{root}$, since $\sum X_o = \text{odd}$, if $\sum X_{\downarrow u\cap O} = \text{odd}$, then $\sum X_{\downarrow v\cap O} = \text{even}$.

3. Assume our previous coin flip deduced $\sum X_{\downarrow u\cap O} = \text{odd}$, then we compute:

$$P[X_u = i|\sum X_{\downarrow u\cap O} = \text{odd}, X_{pa(u)} = j] \propto P[X_u = i, \sum X_{\downarrow u\cap O} = \text{odd}|X_{pa(u)} = j] = s(u,i)\cdot\theta_u(i,j)$$

   and sample from it.

   If, on the other hand, the coin flip deduced $\sum X_{\downarrow u\cap O} = \text{even}$, then we instead compute:

$$P[X_u = i|\sum X_{\downarrow u\cap O} = \text{even}, X_{pa(u)} = j] \propto P[X_u = i, \sum X_{\downarrow u\cap O} = \text{even}|X_{pa(u)} = j] = (1-s(u,i))\cdot\theta_u(i,j)$$

   and sample from it

4. Recursively repeat steps 2 and 3, until we have sampled all of our leaves.

The proposed sampling algorithm is polynomial time, since at each node we perform $O(K)$ computations in order to compute the categorical distributions (more specifically steps 1 and 3 are $O(K)$, while step 2 is $O(1)$), and there are $O(V)$ such nodes to traverse, making it a $O(K * V)$ algorithm.

Intuitively, we are sampling from the desired q distribution, since we are respecting our tree CPD $\theta_u(i,j)$, while also ensuring that the sum of leaves-output remains odd. We are thus sampling from $p[X_O|\sum X_O = \text{odd}] = \frac{P[\sum X_O = \text{odd}|X_O]}{P[\sum X_O = \text{odd}]}\cdot p[X_O|\Theta,T] = C\cdot p[X_O|\Theta,T] = q(X_O)$. One could also reason inductively (utilizing a three node binary tree: root + two leafs, as base of induction, where it is clear to see how this is the same distribution and generalize using strong induction).

2.8.23

Please refer to the appendix for the implementation. The method that calculates the s matrix, alongside the sum is tree-DP(tree). The general idea was discussed in the previous question, where we derived the algorithm.

2.8.24

Please refer to the appendix for the implementation. We implemented two methods, alongside the sampling algorithm, as was suggested in the github code:

- test-sample-proportion(tree): Given a dictionary containing certain values for the leaf nodes nodes, and a leaf sample, calculates the proportion of sampled leaf sets that matches the values of the dictionary.
- test-node-sample-proportion(tree): Given a dictionary containing certain values for the leaf nodes, and a leaf sample, calculates same but for each leaf node separately instead of the whole set.

The implementation of the main sampling algorithm was fairly straightforward. This root-to-leafs algorithm utilized a stack, since we sought a BFS type tree traversal.

2.8.25

The final result we got by testing our dynamic programming algorithm for the test case (using the "load-tree()" function as was indicated in the read me) gave us:

$$P[\sum X_O = \text{odd}] = 0.5000002322336661$$

Interestingly, the sum is essentially 0.5. This is not unexpected, since our categorical distribution parameters θ were generated using a "uniform" Dirichlet prior (all a were equal to 1), spreading the θ evenly among even and odd outputs k. Furthermore, the tree was relatively deep. You can see this interesting effect in the $s(u, i)$ as well, where they become increasingly homogeneous as you move further away from the leaves. To break this symmetry, one simply needs to add a bias (in the a) towards either even or odd k. Then, our tree will be biased towards outputting only odd numbers or only even numbers and since even + even = even = odd + odd, the overall sum will be biased towards outputting an even number.

To prove this claim, we generated a binary tree with thetas based on alpha parameters: $[50, 1, 50, 1, 50]$, thus favoring (higher values) θ that correspond to even number outputs k $(0, 2, 4)$. Thus, our outputs are likely to be even, giving us an even sum. This gave us:

$$P[\sum X_O = \text{odd}] = 0.05004295104954254$$

Note that we did the same for $[1, 50, 1, 50, 1]$ and got a similarly small probability **because we had an even number of leaves** (high probability of even sum of odd numbers):

$$P[\sum X_O = \text{odd}] = 0.10114750100799448$$

In the second part of this question, we utilized our sampling algorithm to generate 1000 samples and computed the ratios discussed in section 2.8.24, giving us the following results:

$$\text{ratio} = 0.178$$

leaf-node	ratio
7	0.415
9	0.610
12	0.485
13	0.539
14	0.534
15	0.403
16	0.509
17	0.363
18	0.426
19	0.493
20	0.549

2.9

Our joint distribution factorizes like so:

$$p(Z, X, \Theta) = p(Z)p(\Theta)p(X|Z, \Theta)$$

Our Z probability is defined by its initial state probability (which is uniform for $C = 1$) and $p(Z_c|Z_{c-1})$ which can be succinctly represented by a transposition matrix $A_{s,j} = \frac{1}{3} \iff |s - j| \leq 1 mod(c)$, 0 otherwise. Let us examine $p(Z)$:

$$p(Z) = p(Z_1) \cdot \prod_{c=1}^{C-1} \prod_{j=1}^{R} \prod_{s=1}^{R} A_{j,s}^{z_{j,c} \cdot z_{s,c+1}} = \prod_{r=1}^{R} (\frac{1}{R})^{z_{r,1}} \cdot \prod_{c=1}^{C-1} \prod_{j=1}^{R} \prod_{s=1}^{R} A_{j,s}^{z_{j,c} \cdot z_{s,c+1}}$$

Furthermore,

$$p(\Theta) = p(\theta_f) \cdot \prod_{r=1}^{R} \prod_{c=1}^{C} p(\theta_{r,c})$$

and

$$p(X|Z, \Theta) = \prod_{r=1}^{R} \prod_{c=1}^{C} p(x_{r,c}|z_{r,c}, \Theta)$$

where

$$p(x_{r,c}|z_{r,c}, \Theta) = [\theta_f^{x_{r,c}}(1 - \theta_f)^{1-x_{r,c}}]^{z_{r,c}} \cdot [\theta_{r,c}^{x_{r,c}}(1 - \theta_{r,c})^{1-x_{r,c}}]^{1-z_{r,c}}$$

Finally:

$$p(\theta_{r,c}) = \frac{\theta_{r,c}^{a-1}(1 - \theta_{r,c})^{b-1}}{B(a, b)}, \; p(\theta_f) = \frac{\theta_f^{a_f-1}(1 - \theta_f)^{b_f-1}}{B(a_f, b_f)}$$

Let us now examine our approximate posterior:

$$q(Z, \Theta) = q(Z) \cdot q(\Theta)$$

For $q^*(Z)$ we have:

$$\ln q^*(Z) = E_\Theta[\ln p(Z)p(\Theta)p(X|Z, \Theta)] \overset{+}{=} E_\theta[\ln p(Z)] + E_\Theta[\ln p(X|Z, \Theta)] \overset{+}{=}$$

$$\underbrace{\sum_{r=1}^{R} z_{r,1} \ln \frac{1}{R} + \sum_{c=1}^{C-1} \sum_{j=1}^{R} \sum_{s=1}^{R} z_{j,c} \cdot z_{s,c+1} \ln A_{j,s}}_{\text{expectation disappears since } A_{j,s} \text{ is a fixed prior Hyperparameter}} \quad +$$

$$\sum_{r=1}^{R} \sum_{c=1}^{C} [z_{r,c}(x_{r,c}E_\Theta[\ln \theta_f] + (1 - x_{r,c})E_\Theta[\ln (1 - \theta_f)]) + (1 - z_{r,c})(x_{r,c}E_\Theta[\ln \theta_{r,c}] + (1 - x_{r,c})E_\Theta[\ln (1 - \theta_{r,c})])]$$

Note that if $\theta \sim Beta(a,b)$, then $E_\theta[\ln\theta] = \psi(a) - \psi(a+b)$, while $E_\theta[\ln(1-\theta)] = \psi(b) - \psi(a+b)$ ([4]). As such,

$$q^*(Z) \propto exp(\sum_{r=1}^{R} z_{r,1} \underbrace{\ln\frac{1}{R}}_{\ln\pi_r:\text{initial probability}} + \sum_{c=1}^{C-1}\sum_{j=1}^{R}\sum_{s=1}^{R} z_{j,c}\cdot z_{s,c+1} \underbrace{\ln A_{j,s}}_{\ln A_{j,s}:\text{transition matrix}} +$$

$$\sum_{r=1}^{R}\sum_{c=1}^{C}[z_{r,c}\underbrace{[(x_{r,c}(\psi(a_f^*) - \psi(a_f^*+b_f^*)) + (1-x_{r,c})(\psi(b_f^*) - \psi(a_f^*+b_f^*)))+}_{\text{Emission Matrix},\ln B_{c,r}}$$

$$\underbrace{-(x_{r,c}(\psi(a_{r,c}^*) - \psi(a_{r,c}^*+b_{r,c}^*)) - (1-x_{r,c})(\psi(b_{r,c}^*) - \psi(a_{r,c}^*+b_{r,c}^*)))]])}_{\ln B_{c,r}}$$

Note that the form of $q^*(Z)$ is proportional to that of an HMM (as we saw in class when we took the log-likelihood of an HMM to derive it's EM estimation) with R hidden states and $C=T$ timesteps and thus $q(Z_c)$ (which is required in order to compute the overall $q^*(Z)$) can be computed using standard HMM procedures.

For $q^*(\Theta)$:

$$\ln q^*(\Theta) = E_Z[\ln p(Z)p(\Theta)p(X|Z,\Theta)] \overset{+}{=} E_Z[\ln p(\theta)] + E_Z[\ln p(X|Z,\Theta)] =$$

$$\sum_{r,c}[\underbrace{E[z_{r,c}]}_{\gamma(r,c)}\cdot(x_{r,c}\ln\theta_f + (1-x_{r,c})\ln(1-\theta_f)) + (1-\gamma(r,c))(x_{r,c}\ln\theta_{r,c} + (1-x_{r,c})\ln(1-\theta_{r,c}))]+$$

$$+(a_f-1)\ln\theta_f + (bf-1)\ln(1-\theta_f) + \sum_{r,c}[(a-1)\ln\theta_{r,c} + (b-1)\ln(1-\theta_{r,c})]$$

For θ_f we have:

$$(\sum_{r,c}\gamma(r,c)x_{r,c} + a_f - 1)\ln\theta_f + (\sum_{r,c}\gamma(r,c)(1-x_{r,c}) + b_f - 1)\ln(1-\theta_f)$$

which implies $\theta_f \sim Beta(a_f^*, b_f^*)$:

$$a_f^* = a_f + \sum_{r,c}\gamma(r,c)x_{r,c}$$

$$b_f^* = b_f + \sum_{r,c}\gamma(r,c)(1-x_{r,c})$$

Similarly, for each $\theta_{r,c}$:

$$((1-\gamma(r,c))x_{r,c} + a - 1)\ln\theta_{r,c} + ((1-\gamma(r,c))(1-x_{r,c}) + b - 1)\ln(1-\theta_{r,c})$$

Thus $\theta_{r,c} \sim Beta(a_{r,c}^*, b_{r,c}^*)$:

$$a_{r,c}^* = (1-\gamma(r,c))x_{r,c} + a$$

$$b_{r,c}^* = (1-\gamma(r,c))(1-x_{r,c}) + b$$

We still need to compute $\gamma(r,c)$:

$$\gamma(r,c) = E[z_{r,c}] = p[z_{r,c}=1|\boldsymbol{X}] = p[Z_c = r|\boldsymbol{X}]$$

Recall, however, that we know that Z is characterized by an HMM. We can thus compute this probability the same way it is computed in traditional HMMs, that is:

$$\gamma(r,c) = p[Z_c = r|\boldsymbol{X}] = \frac{f_c(r)\cdot b_c(r))}{\sum_{r'} f_c(r)\cdot b_c(r)}$$

where $f_c(r)$, $b_c(r)$ are the outputs of the forward and backward algorithm in Hidden Markov Models. That is:

$$f_c(r) = \sum_{r'} f_{c-1}(r')\cdot B_{c,r}A_{r,r'}$$

$$f_1(r) = \pi_r \cdot B_{1,r}$$

and

$$b_c(r) = \sum_{r'} A_{r,r'} b_{c+1}(r) B_{c+1,r'}$$

$$b_C(r) = 1$$

For more information on the forward and backward algorithm, as well as the derivation of these formulas, please refer to this excellent reading from Mark Stamp ([2]). Finally, recall that we discussed how $B_{c,r} = exp(E_\theta[\ln p(x_{r,c}|z_{r,c}, \theta)])$ which was calculated and thus:

$$\begin{gathered}
B_{c,r} = exp(x_{r,c}(\psi(a_f^*) - \psi(a_f^* + b_f^*)) + (1 - x_{r,c})(\psi(b_f^*) - \psi(a_f^* + b_f^*)) - \\
-x_{r,c}(\psi(a_{r,c}^*) - \psi(a_{r,c}^* + b_{r,c}^*)) - (1 - x_{r,c})(\psi(b_{r,c}^*) - \psi(a_{r,c}^* + b_{r,c}^*))) = \\
= \frac{e^{x_{r,c}(\psi(a_f^*) - \psi(a_f^* + b_f^*)) + (1 - x_{r,c})(\psi(b_f^*) - \psi(a_f^* + b_f^*))}}{e^{x_{r,c}(\psi(a_{r',c}^*) - \psi(a_{r,c}^* + b_{r,c}^*)) + (1 - x_{r,c})(\psi(b_{r,c}^*) - \psi(a_{r,c}^* + b_{r,c}^*))}}
\end{gathered}$$

We can thus compute $\gamma(r, c)$ and hence compute $q^*(\Theta)$.

References

[1] Christopher M. Bishop. *Pattern Recognition and Machine Learning.* (2006).

[2] Mark Stamp. *A Revealing Introduction to Hidden Markov Models.* (2018).

[3] Normal-Gamma Distribution: https://en.wikipedia.org/wiki/Normal-gamma_distribution

[4] Beta Distribution: https://en.wikipedia.org/wiki/Beta_distribution

[5] Digamma Function: https://en.wikipedia.org/wiki/Digamma_function

[6] Gamma Distribution: https://en.wikipedia.org/wiki/Gamma_distribution

[7] Dirichlet Distribution: https://en.wikipedia.org/wiki/Dirichlet_distribution

Appendix

In this appendix I showcase my code implementation, beginning from exercise 2.4, up until the last programming assignment 2.8. In 2.5 I did not show the results of the cell running the sieving algorithm, since it was rather verbose and would add unnecessary pages to an already crowded report. The graphs are already shown in the report. There was some debate of whether or not to include the results of the kernels (since they might hinder readability). In the end, I opted into showing the results to bolster repeatability and ensure transparency. I believe that the tradeoff with readibility is worth it, as the overall structure is fairly clear, with each exercise code beginning with the algorithms, and ending with their application on certain data (and thus, it is easily skipable if someone only wishes to focus on the algorithms).

2_4

December 28, 2019

[1]:
```
#import needed libraries
import numpy as np
from scipy.special import gamma as gam
from scipy.stats import norm,gamma
import matplotlib.pyplot as plt
```

[2]:
```
# generate data
# Define hyperparameters
m_0 = 5
a_0=1
b_0=50
N = 100
lambda_0=1
#generate iid gaussians from N(m_x,  ~2_x)
seed_val = 1
np.random.seed(seed_val)
x = np.random.normal(0,0.5,N)
```

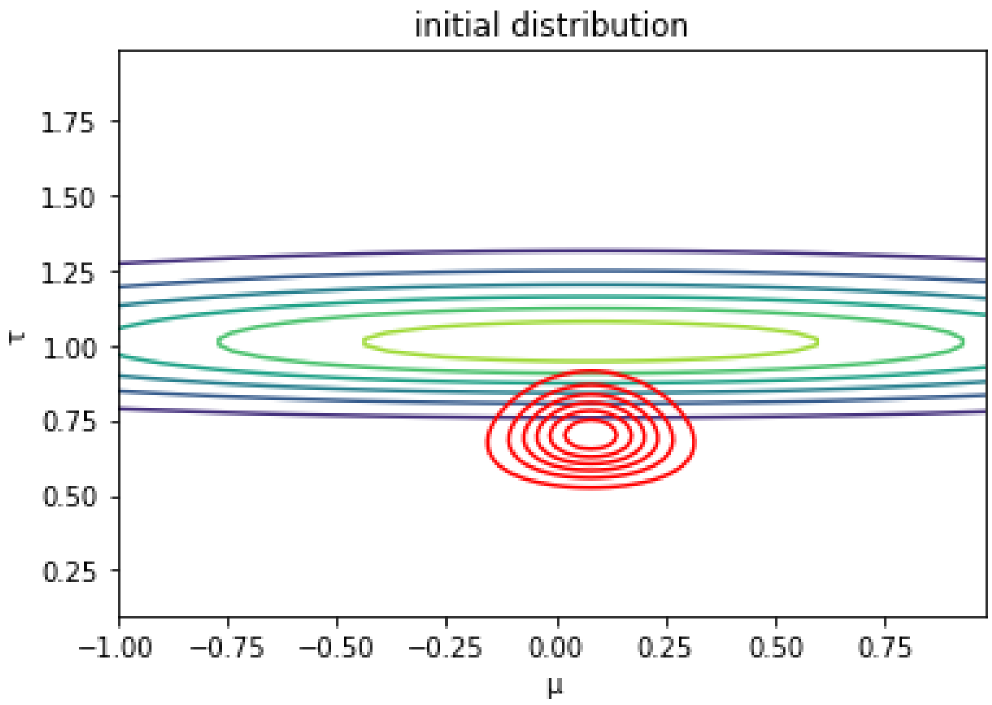
[3]:
```
#update equations

def updateEquations(m_0,lambda_0,a_0,b_0,tol,xn,it):
    N = len(xn)
    dif = 1 # dif measures the change of our parameters. We default it to 1 for␣
↪the first step
    iterat=0
    m_prev,lambda_prev,a_prev,b_prev = m_0,lambda_0,a_0,b_0 # store the initial␣
↪values
    m_n = (lambda_0*m_0 + np.sum(xn))/(lambda_0 + N) # Compute this once and␣
↪for all
    a_n = a_0 + (N+1)/2. # Similarly
    lambda_n=np.random.rand() #initialize lambda_n randomly to use it for b_n
    lambda_history=[lambda_n] #remember it
    b_history=[b_0] #remember b_0
    while(dif > tol and iterat<it): #while parameters change > tol and we have␣
↪not reached max iterations
        b_n = t_update(m_0,m_n,lambda_0,lambda_n,b_0,xn) #compute b_n
        lambda_n = m_update(lambda_0,a_n,b_n,N) #compute lambda_n
        dif = np.amax(a=np.array([lambda_n-lambda_prev,b_n-b_prev])) # dif =␣
↪max{|lambda_n-lambda_prev|,|b_n-b_prev|}
        m_prev,lambda_prev,a_prev,b_prev = m_n,lambda_n,a_n,b_n # remember the␣
↪last ones
        iterat=iterat+1
        lambda_history.append(lambda_n) #store last lambda_n
        b_history.append(b_n) #store last b_n

    return m_n,lambda_n,a_n,b_n,lambda_history,b_history

def m_update(lambda_0,a_n,b_n,N): #lambda_n update
    lambda_n = (lambda_0+N)*a_n/b_n
    return lambda_n

def t_update(m_0,m_n,lambda_0,lambda_n,b_0,xn): #b_n update
    #compute E[ ], E[ ^2]
    mean = m_n
    mean_sq = m_n**2 + 1./lambda_n
    #compute b_n
    b_n = b_0 + lambda_0/2. * (mean_sq + m_0**2 - 2* mean* m_0) + 1/2.* np.
↪sum(np.square(xn) + mean_sq - 2*mean*xn)
    return b_n
```

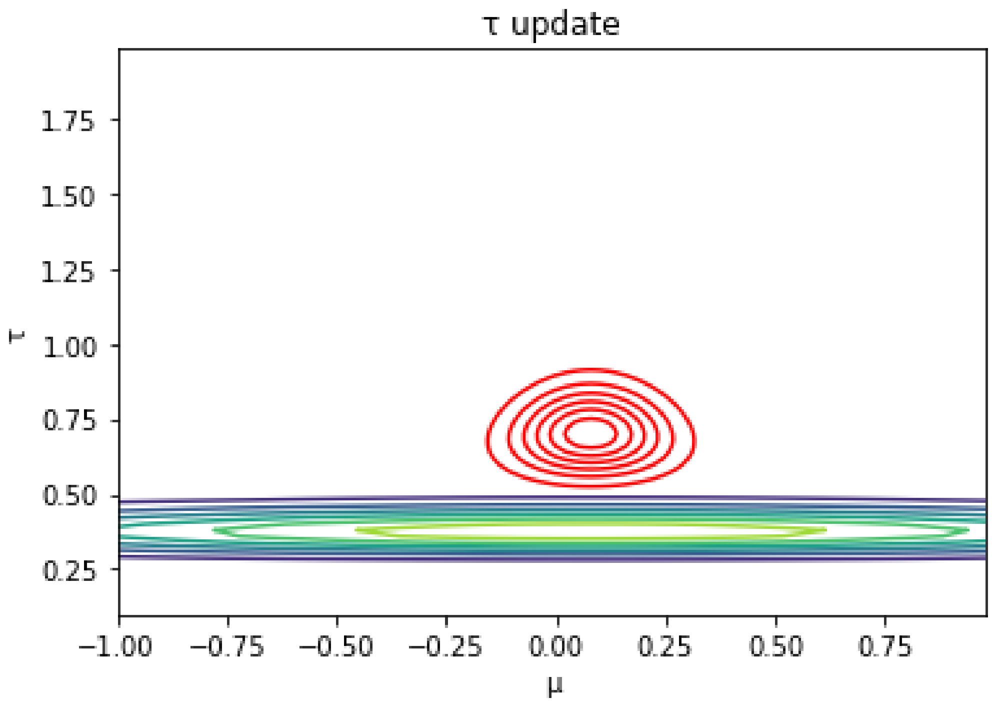
[4]:
```
#compute the parameters
m_n,lambda_n,a_n,b_n,lambda_h,b_h = updateEquations(m_0,lambda_0,a_0,b_0,0.01,␣
↪x, 100)
#print them
print('m_n=',m_n)
print('lambda_n=',lambda_n)
print('a_n=',a_n)
print('b_n=',b_n)
print('b history:',b_h)
print('lambda history:',lambda_h)
```

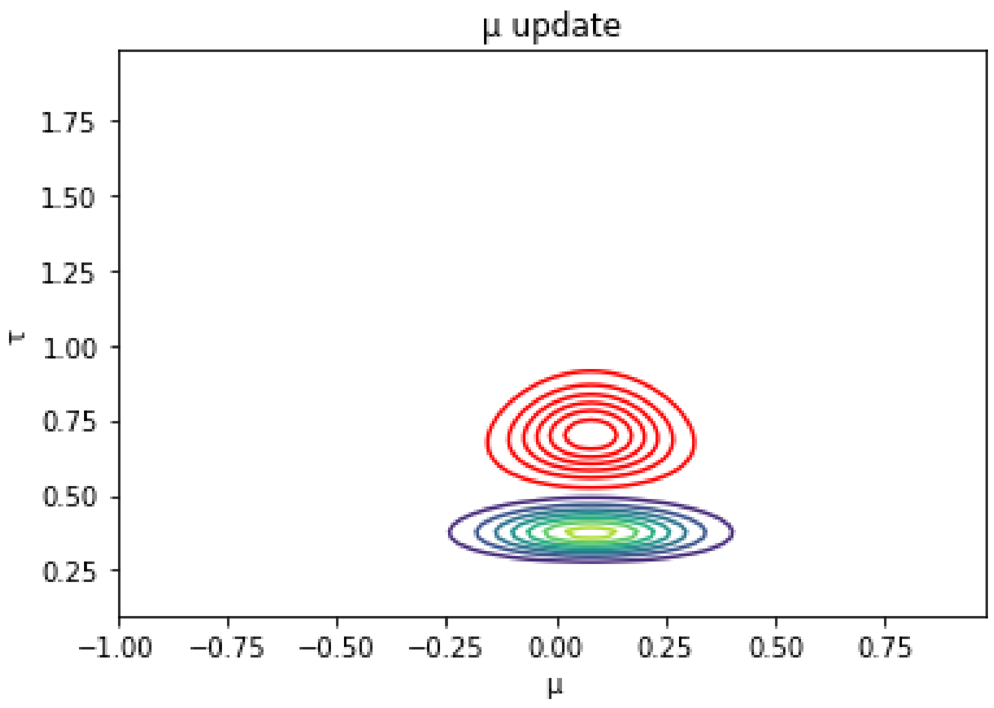
```
m_n= 0.07949646142361322
lambda_n= 71.52124396225533
a_n= 51.5
b_n= 72.72664332775089
b history: [50, 134.58979453101577, 73.32720038227713, 72.73241791481364,
72.72664332775089]
lambda history: [0.8071051956187791, 38.64706100581297, 70.93547787018991,
71.51556553629429, 71.52124396225533]
```

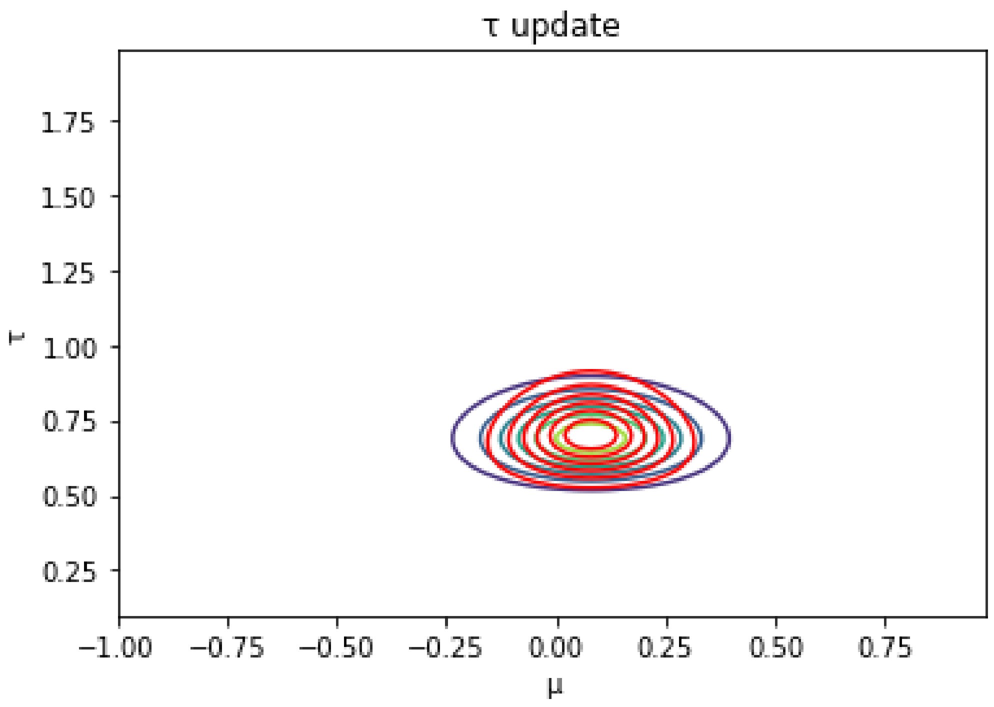
[5]:
```
def posterior(m,t,m_n,lambda_n,a_n,b_n): #approximated posterior q( , ) = q( )q( )
    q_m = np.sqrt(lambda_n) / np.sqrt(2*np.pi) * np.exp(-lambda_n/2 * (m -␣
↪m_n)**2)
    q_t = b_n**a_n /gam(a_n) * t**(a_n-1) *np.exp(-b_n*t)
    return q_m*q_t

def real_posterior(m,t,m_0,lambda_0,a_0,b_0,xn): #exact posterior
    N = len(xn)
    x_bar = np.mean(xn)
    s = np.var(xn)
    m_n = (lambda_0*m_0 + N*x_bar)/(lambda_0 + N)
    lambda_n = lambda_0 + N
    a_n = a_0 + N/2.0
    b_n = b_0 + 1/2.0 *(N*s + lambda_0*N*(x_bar - m_0)**2/lambda_n)
    result =  b_n**a_n * np.sqrt(lambda_n)/(gam(a_n)*np.sqrt(2*np.pi)) *␣
 ↪t**(a_n-1/2.) *np.exp(-b_n*t) * np.exp(-lambda_n/2. * t * (m - m_n)**2)
    return result
```

[6]:
```
#create grid
m_grid = np.arange(-1,1,0.02)
t_grid = np.arange(0.1,2,0.02)

M,T= np.meshgrid(m_grid,t_grid)
#begin printing
for i in range(0, len(lambda_h)-1):
    Z1 = posterior(M,T,m_n,lambda_h[i],a_n,b_h[i]) #distribution at the start␣
 ↪or after an  update
    Z2 = real_posterior(M,T,m_0,lambda_0,a_0,b_0,x)
    CS = plt.contour(M,T,Z1)
    CS = plt.contour(M,T,Z2, colors='red')
    if i==0:
        plt.title('initial distribution')
    else:
        plt.title('  update')
    plt.xlabel(' ')
    plt.ylabel(' ')
    plt.figure()
    Z1 = posterior(M,T,m_n,lambda_h[i],a_n,b_h[i+1]) #distribution after a ␣
 ↪update
    CS = plt.contour(M,T,Z1)
    CS = plt.contour(M,T,Z2, colors='red')
    plt.title('  update')
    plt.xlabel(' ')
    plt.ylabel(' ')
    plt.figure()
plt.show()
```

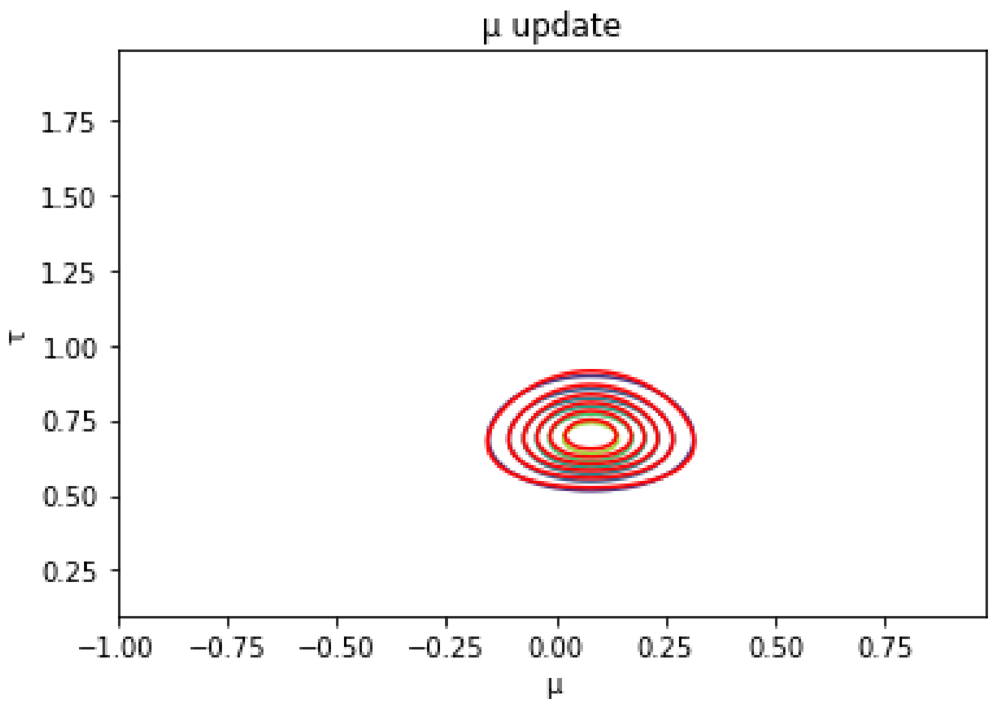

initial distribution
τ
μ
1.75
1.50
1.25
1.00
0.75
0.50
0.25
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75

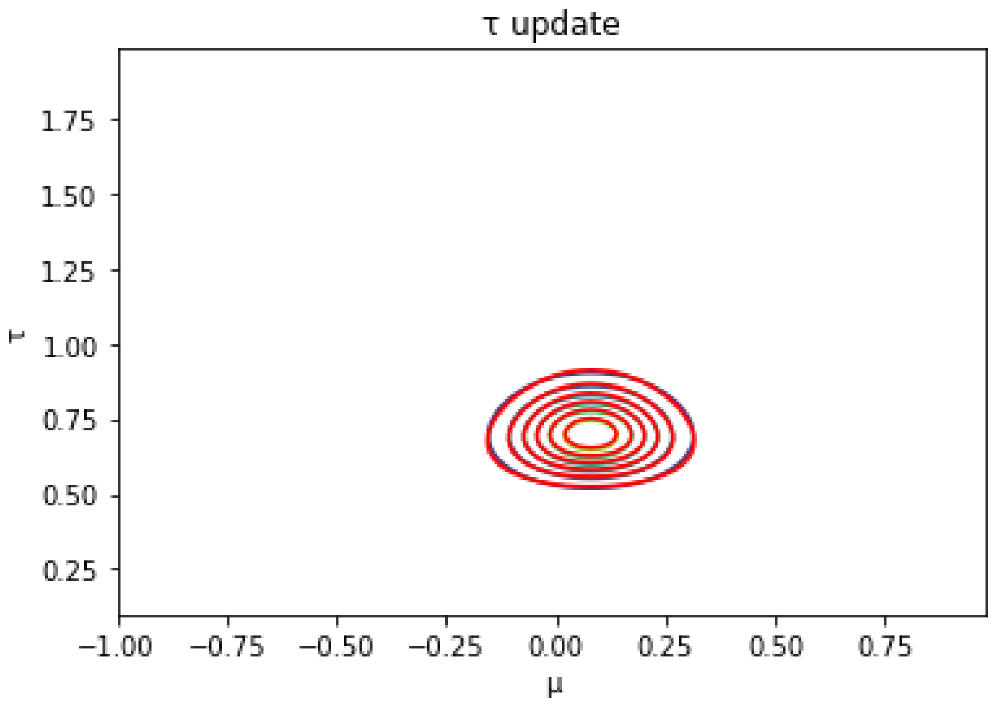

τ update
τ
μ
1.75
1.50
1.25
1.00
0.75
0.50
0.25
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75

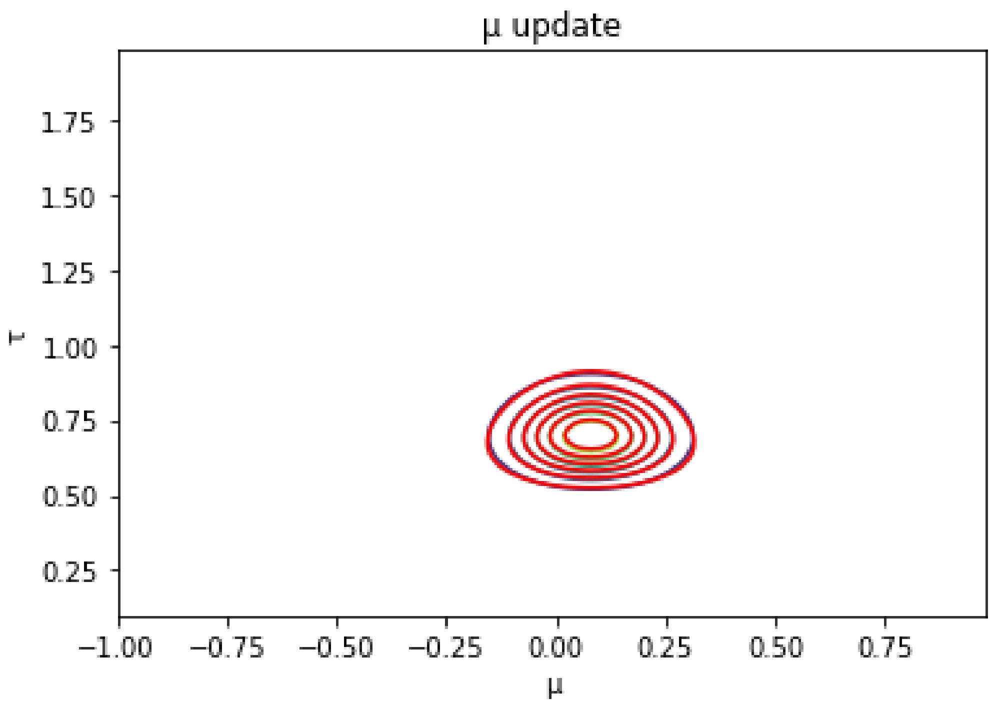

μ update
τ
μ
1.75
1.50
1.25
1.00
0.75
0.50
0.25
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75

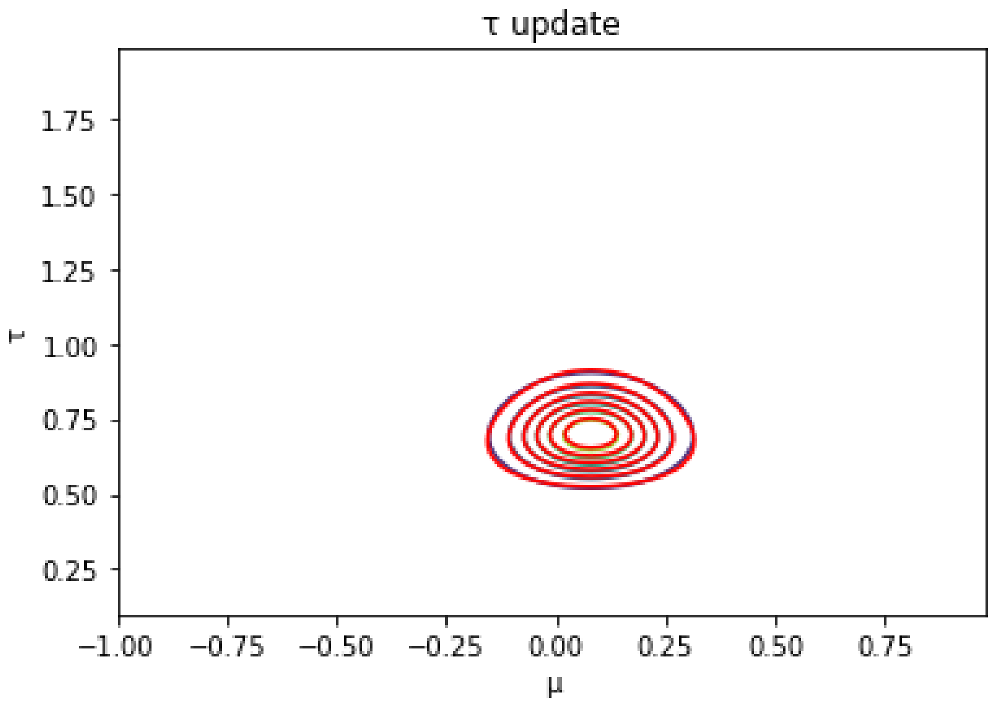

τ update
τ
μ
1.75
1.50
1.25
1.00
0.75
0.50
0.25
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75

5


μ update
τ
μ
1.75
1.50
1.25
1.00
0.75
0.50
0.25
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75


τ update
τ
μ
1.75
1.50
1.25
1.00
0.75
0.50
0.25
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75


μ update
τ
μ
1.75
1.50
1.25
1.00
0.75
0.50
0.25
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75


τ update
τ
μ
1.75
1.50
1.25
1.00
0.75
0.50
0.25
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75

```
<Figure size 432x288 with 0 Axes>
```

8

2_5

December 28, 2019

```
[15]: #import libraries
from Tree import TreeMixture
import argparse
import numpy as np
import matplotlib.pyplot as plt


def save_results(loglikelihood, topology_array, theta_array, filename):
    #This method can be used to store your results in file (I did not use this␣
 ↪however).
    """ 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:
 ↪ ", topology_array_filename,
          ", 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.
```

1

```
    :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)

    You can change the function signature and add new parameters. Add them as␣
↪parameters with some default values.
    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_param_1=[], new_param_2=123):
    """

    # 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.
    print("Running EM algorithm...")
    #from Tree import TreeMixture
    #randomly initialize our model
    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)
    loglikelihood = []

    topology_list = []
    theta_list = []
    #initialize our mixture model
    for i in range(num_clusters):
        topology_list.append(tm.clusters[i].get_topology_array())
        theta_list.append(tm.clusters[i].get_theta_array())
    topology_list = np.array(topology_list)
    theta_list = np.array(theta_list)
    pi = tm.pi
    #run the EM max_num_iter times
    for iter_ in range(max_num_iter):
        #e-step
        r = responsibility(topology_list,theta_list,pi,samples,num_clusters)
        #m-step
        pi = pi_update(samples,num_clusters,r)
        trees = tree_update(samples,num_clusters,r)
        for k in range(0,num_clusters):
            thet = update_theta(samples,r,trees[k],k)
            topology_list[k]=trees[k]
            theta_list[k]=thet
        #s= model_log_likelihood(topology_list,theta_list,r,pi,samples)
        #if (len(loglikelihood)>0 and s < loglikelihood[-1]):
        loglikelihood.
↪append(model_log_likelihood(topology_list,theta_list,r,pi,samples))

    #convert lists to arrays
    loglikelihood = np.array(loglikelihood)
    topology_list = np.array(topology_list)
    theta_list = np.array(theta_list)

    return loglikelihood, topology_list, theta_list

def sieving(tries,samples,num_clusters): #sieving algorithm
    log_list = np.zeros(tries)
    for i in range(0,tries): #first, run EM for 100 different seeds (0-99) for␣
↪10 iterations
        print('iteration number:', i)
        loglikelihood, topology_array, theta_array = em_algorithm(i, samples,␣
↪num_clusters=num_clusters,max_num_iter=10)
        log_list[i] = loglikelihood[-1] #keep the final log_likelihood
    print('log_list',log_list)
    ind = np.argpartition(log_list, -10)[-10:] #get the 10 best likelihoods
    updated_log_list = log_list[ind]
    print('updated_list',updated_log_list) #show it
    for i in range(0,updated_log_list.shape[0]): #for these 10 options, train up␣
↪to 100 iterations
        loglikelihood, topology_array, theta_array = em_algorithm(i, samples,␣
↪num_clusters=num_clusters,max_num_iter=100)
        updated_log_list[i] = loglikelihood[-1]
    best = np.argmax(updated_log_list)
    #print(best)
    #print(ind[best])
    return em_algorithm(ind[best], samples,␣
↪num_clusters=num_clusters,max_num_iter=100)
```

[16]:
```
#compute likelihood
def computeLikelihood(tree_ar, theta_ar,sample):
    result = theta_ar[0][sample[0]] #root has no parents
    for i in range(1,tree_ar.shape[0]):
        result = result * theta_ar[i][sample[int(tree_ar[i])]][sample[i]]␣
↪#result[node][value of parent][value of node]
    return result
```

[17]:
```
#compute responsibility
def responsibility(topology_list,theta_list,pi,samples,clusters_num):
    r= np.ones((samples.shape[0],clusters_num)) #initialize
    for n in range(0,samples.shape[0]):
        summ = 0 # this is our denominator
        for k in range(0,clusters_num): #r[n][k] = pi_k *␣
 ↪p(samples[n]|theta_k,topology_k) / denominator
            r[n][k] = pi[k] *␣
 ↪computeLikelihood(topology_list[k],theta_list[k],samples[n]) #this is the␣
 ↪numerator
            summ = summ + r[n][k] # denominator-normalizer
        eps = np.finfo(np.float64).eps #2.22*e-16
        r[n] = (r[n]+eps)/(summ+clusters_num*eps) #normalize using this epsilon
    return r
```

[18]:
```
#pi-update
def pi_update(samples,clusters_num,r):
    N = samples.shape[0]
    pi = np.ones(clusters_num)
    pi = np.sum(r,axis=0)/N #pi_k = r_k / N
    return pi
#tree-update
def tree_update(samples,clusters_num,r):
    i_table = create_i_table(samples,r)
    tree_arr=np.ones((clusters_num,samples.shape[1]))
    for k in range(0,clusters_num):
        gr = create_Graph(samples,i_table[k]) #create a graph using mutual␣
 ↪information i_table[k]
        mst = gr.maximum_spanning_tree() #get the Maximum Spanning Tree
        tree_arr[k]=tree_repr(mst) #transform said tree into its correspodning␣
 ↪topology array



    return tree_arr
```

[19]:
```
def q_joint(samples,r,s,a,t,b,k): #calculating q_k(X_s=a,X_t=b)
    N = samples.shape[0]
    q = 0
    denominator = np.sum(r,axis=0)[k]
    for i in range(0,N):
        if(samples[i][s]==a and samples[i][t]==b):
            q+=r[i][k]
    q=q/(denominator+np.finfo(np.float64).eps)
    return q
def q_single(samples,r,s,a,k): #calculating q_k(X_s=a)
    N=samples.shape[0]
    q = 0
    denominator = np.sum(r,axis=0)[k]
    for i in range(0,N):
        if (samples[i][s]==a):
            q+=r[i][k]
    q=q/(denominator+np.finfo(np.float64).eps)
    return q
def q_cond(samples,r,s,a,t,b,k): #calculating q_k(X_s=a|X_t=b)
    N = samples.shape[0]
    q = 0
    denominator=0
    for i in range(0,N):
        if(samples[i][t]==b):
            denominator += r[i][k]
        if(samples[i][s]==a and samples[i][t]==b):
            q+=r[i][k]
    q=q/(denominator+np.finfo(np.float64).eps)
    return q
```

[20]:
```
def i_k(samples,r,s,t,k): #Mutual information I_k (X_s,X_t)
    result = 0
    eps = np.finfo(np.float64).eps
    for a in range(0,2):
        for b in range(0,2):

            if (q_joint(samples,r,s,a,t,b,k)>0): #if the joint is zero, the␣
 ↪corresponding term is zero
                result+= q_joint(samples,r,s,a,t,b,k) * np.
 ↪log(q_joint(samples,r,s,a,t,b,k)/
 ↪(q_single(samples,r,s,a,k)*q_single(samples,r,t,b,k)))
    return result

def create_i_table(samples,r): #Mutual information matrix I
    result = np.zeros((r.shape[1],samples.shape[1],samples.shape[1]))
    for k in range(0,r.shape[1]):
        for s in range(0,samples.shape[1]):
            for t in range(0,samples.shape[1]):
                if s!=t: #at diagonal, assume 0
                    result[k][s][t] = i_k(samples,r,s,t,k)
    return result
```

[21]:
```
from Kruskal_v1 import Graph
#mst
def create_Graph(samples,i_table): #given mutual information matrix I, create a␣
 ↪graph
    g = Graph(samples.shape[1])
    for i in range(0,samples.shape[1]):
        for j in range(i+1,samples.shape[1]): #only unite them once. No need to␣
↪consider both (u,v) and (v,u)
            g.addEdge(i,j,i_table[i][j])
    return g
```

```
[22]: def tree_repr(mst): # reshape our maximum spanning tree result into the␣
↪corrseponding topology array
    result = np.zeros(len(mst)+1)
    result[0]=np.nan #we maintain our root
    found = [0]
    i=1
    #mst is a list containing elements [u,v,w], w=edge(u,v)
    while(True):
        for dic in mst:
            if(i==0): #ignore the root, it's already inside (and has 'nan')
                break
            #the two following cases makes sure that we always parent->child,␣
↪where parent is a node
            #already discovered. This ensures that the topology list is in fact␣
↪a single tree.
            #(there is always a path from the root to one of the nodes)
            if dic[0] not in found and dic[0]==i and dic[1] in found: #first␣
↪case: u was previously found
                found.append(dic[0])
                result[dic[0]]=dic[1]
            if dic[0] in found and dic[1] not in found and dic[1]==i: #second␣
↪case:v was previously found
                found.append(dic[1])
                result[dic[1]]=dic[0]
            if (len(found)>=result.shape[0]): #we found all the nodes
                break
        if(len(found)>=result.shape[0]):
            break
        i=(i+1)%result.shape[0] # i = (i+1)mod(tree.size). We loop through all␣
↪the nodes until we find them all
    return result
```

```
[23]: def update_theta(samples,r,tree_ar,k):
    result = []
    result.append([q_single(samples,r,0,0,k),q_single(samples,r,0,1,k)]) #root␣
↪has no parents
    for i in range(1,samples.shape[1]):
        t = int(tree_ar[i])#store the parent
        #compute q_k(X_s=a,X_t=b), a,b in {0,1}
        result.append([np.array([q_cond(samples,r,i,a,t,0,k) for a in␣
↪range(0,2)]),np.array([q_cond(samples,r,i,a,t,1,k) for a in range(0,2)])])
```

```
    return np.array(result)
```

[24]:
```
#compute the model_log_likelihood
def model_log_likelihood(tree_ar,theta_ar,r,pi,samples): #ECLL
    res = 0
    k = pi.shape[0]
    N = samples.shape[0]
    for i in range(0,N):
        summ = 0
        for j in range(0,k):
            tree_arr = tree_ar[j]
            theta_arr= theta_ar[j]
            likely = computeLikelihood(tree_arr,theta_arr,samples[i]) #compute␣
 ↪likelihood

            if(likely>=np.finfo(np.float64).eps): #To compensate for the non␣
 ↪zero r_nk
                summ += pi[j]*likely
        res += np.log(summ) # likelihood = sum_n ln(sum_k pi_k␣
 ↪p(x_n|T_k,Theta_k))
    return res
```

[ ]:
```
#choose number of clusters
num_clusters=3
#pick sample file
samples = np.loadtxt('data/q_2_5_tm_5node_20sample_3clusters.pkl_samples.txt',␣
 ↪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")

loglikelihood, topology_array, theta_array = sieving(100,samples,num_clusters)
#sieving(100,samples,num_clusters)#em_algorithm(1, samples,␣
 ↪num_clusters=num_clusters)

print("\n3. Save, print and plot the results.\n")

save_results(loglikelihood, topology_array, theta_array, 'outputfile')

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

```
plt.figure(figsize=(8, 3))
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(loc=(1.04, 0))
plt.show()
```

[27]:
```
#get the real underlying tree mixture models. The saved files must respect the␣
 ↪name same name format as the given files
def get_real_parameters(tree_size,sample_size,cluster_num):
    theta_real=[]
    topology_real=[]
    pi_real = np.load('data/
 ↪q_2_5_tm_'+str(tree_size)+'node_'+str(sample_size)+'sample_'+str(cluster_num)+'clusters.
 ↪pkl_pi.npy', allow_pickle=True)
    for i in range(0,cluster_num):
        theta_real.append(np.load('data/
 ↪q_2_5_tm_'+str(tree_size)+'node_'+str(sample_size)+'sample_'+str(cluster_num)+'clusters.
 ↪pkl_tree_'+str(i)+'_theta.npy', allow_pickle=True))
        topology_real.append(np.load('data/
 ↪q_2_5_tm_'+str(tree_size)+'node_'+str(sample_size)+'sample_'+str(cluster_num)+'clusters.
 ↪pkl_tree_'+str(i)+'_topology.npy', allow_pickle=True))

    return np.array(theta_real), np.array(topology_real),np.array(pi_real)
theta_real,topology_real,pi_real = get_real_parameters(5,20,3)
```

[28]:
```
r = responsibility(topology_real,theta_real,pi_real,samples,3)
print(model_log_likelihood(topology_real,theta_real,r,pi_real,samples))
```

```
-55.023875734478096
```

[29]:
```
print(loglikelihood[-1])
```

```
-37.210687549397115
```

[32]:
```
#Compare tree structures
import dendropy
from Tree import Tree
tns = dendropy.TaxonNamespace()
t0=Tree()
t0.load_tree_from_direct_arrays(topology_real[0])
t0 = dendropy.Tree.get(data=t0.newick, schema="newick", taxon_namespace=tns)
```

```
t1=Tree()
t1.load_tree_from_direct_arrays(topology_real[1])
t1 = dendropy.Tree.get(data=t1.newick, schema="newick", taxon_namespace=tns)

t2=Tree()
t2.load_tree_from_direct_arrays(topology_real[2])
t2 = dendropy.Tree.get(data=t2.newick, schema="newick", taxon_namespace=tns)

#remove comments if you want to test case for 5 clusters
'''
t3=Tree()
t3.load_tree_from_direct_arrays(topology_real[3])
t3 = dendropy.Tree.get(data=t3.newick, schema="newick", taxon_namespace=tns)

t4=Tree()
t4.load_tree_from_direct_arrays(topology_real[4])
t4 = dendropy.Tree.get(data=t4.newick, schema="newick", taxon_namespace=tns)
'''
rt0 = Tree()
rt0.load_tree_from_direct_arrays(topology_array[0])
rt0 = dendropy.Tree.get(data=rt0.newick, schema="newick", taxon_namespace=tns)
print("\tInferred Tree 0: ", rt0.as_string("newick"))
rt0.print_plot()

rt1 = Tree()
rt1.load_tree_from_direct_arrays(topology_array[1])
rt1 = dendropy.Tree.get(data=rt1.newick, schema="newick", taxon_namespace=tns)
print("\tInferred Tree 1: ", rt1.as_string("newick"))
rt1.print_plot()

rt2 = Tree()
rt2.load_tree_from_direct_arrays(topology_array[2])
rt2 = dendropy.Tree.get(data=rt2.newick, schema="newick", taxon_namespace=tns)
print("\tInferred Tree 2: ", rt2.as_string("newick"))
rt2.print_plot()

#remove comments if you want to test case for 5 clusters
'''rt3 = Tree()
rt3.load_tree_from_direct_arrays(topology_array[3])
rt3 = dendropy.Tree.get(data=rt3.newick, schema="newick", taxon_namespace=tns)
print("\tInferred Tree 3: ", rt3.as_string("newick"))
rt3.print_plot()

rt4 = Tree()
rt4.load_tree_from_direct_arrays(topology_array[4])
rt4 = dendropy.Tree.get(data=rt4.newick, schema="newick", taxon_namespace=tns)
```

```
print("\tInferred Tree 3: ", rt4.as_string("newick"))
rt4.print_plot()'''

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

print("\tt0 vs inferred trees")
print("\tRF distance: \t", dendropy.calculate.treecompare.
 ↪symmetric_difference(t0, rt0))
print("\tRF distance: \t", dendropy.calculate.treecompare.
 ↪symmetric_difference(t0, rt1))
print("\tRF distance: \t", dendropy.calculate.treecompare.
 ↪symmetric_difference(t0, rt2))
'''print("\tRF distance: \t", dendropy.calculate.treecompare.
 ↪symmetric_difference(t0, rt3))
print("\tRF distance: \t", dendropy.calculate.treecompare.
 ↪symmetric_difference(t0, rt4))'''

print("\tt1 vs inferred trees")
print("\tRF distance: \t", dendropy.calculate.treecompare.
 ↪symmetric_difference(t1, rt0))
print("\tRF distance: \t", dendropy.calculate.treecompare.
 ↪symmetric_difference(t1, rt1))
print("\tRF distance: \t", dendropy.calculate.treecompare.
 ↪symmetric_difference(t1, rt2))
'''print("\tRF distance: \t", dendropy.calculate.treecompare.
 ↪symmetric_difference(t1, rt3))
print("\tRF distance: \t", dendropy.calculate.treecompare.
 ↪symmetric_difference(t1, rt4))
'''
print("\tt2 vs inferred trees")
print("\tRF distance: \t", dendropy.calculate.treecompare.
 ↪symmetric_difference(t2, rt0))
print("\tRF distance: \t", dendropy.calculate.treecompare.
 ↪symmetric_difference(t2, rt1))
print("\tRF distance: \t", dendropy.calculate.treecompare.
 ↪symmetric_difference(t2, rt2))
'''print("\tRF distance: \t", dendropy.calculate.treecompare.
 ↪symmetric_difference(t2, rt3))
print("\tRF distance: \t", dendropy.calculate.treecompare.
 ↪symmetric_difference(t2, rt4))

print("\tt3 vs inferred trees")
print("\tRF distance: \t", dendropy.calculate.treecompare.
 ↪symmetric_difference(t3, rt0))
print("\tRF distance: \t", dendropy.calculate.treecompare.
 ↪symmetric_difference(t3, rt1))
```

```
print("\tRF distance: \t", dendropy.calculate.treecompare.
 ↪symmetric_difference(t3, rt2))
print("\tRF distance: \t", dendropy.calculate.treecompare.
 ↪symmetric_difference(t3, rt3))
print("\tRF distance: \t", dendropy.calculate.treecompare.
 ↪symmetric_difference(t3, rt4))

print("\tt4 vs inferred trees")
print("\tRF distance: \t", dendropy.calculate.treecompare.
 ↪symmetric_difference(t4, rt0))
print("\tRF distance: \t", dendropy.calculate.treecompare.
 ↪symmetric_difference(t4, rt1))
print("\tRF distance: \t", dendropy.calculate.treecompare.
 ↪symmetric_difference(t4, rt2))
print("\tRF distance: \t", dendropy.calculate.treecompare.
 ↪symmetric_difference(t4, rt3))
print("\tRF distance: \t", dendropy.calculate.treecompare.
 ↪symmetric_difference(t4, rt4))'''
```

```
        Inferred Tree 0:  [&R] ((((1)4)2)3)0;

+-------------------+-------------------+-------------------+------------------- 1


        Inferred Tree 1:  [&R] (((3,4)1)2)0;

                                                  /------------------------- 3
+-------------------------+-------------------------+
                                                  \------------------------- 4


        Inferred Tree 2:  [&R] ((((1)3)2)4)0;

+-------------------+-------------------+-------------------+------------------- 1



4.2 Compare trees and print Robinson-Foulds (RF) distance:

        t0 vs inferred trees
        RF distance:     4
        RF distance:     4
        RF distance:     4
        t1 vs inferred trees
        RF distance:     5
        RF distance:     3
        RF distance:     5
```

11

```
        t2 vs inferred trees
        RF distance:     4
        RF distance:     0
        RF distance:     4
```

[32]:
```
'print("\tRF distance: \t",
dendropy.calculate.treecompare.symmetric_difference(t2, rt3))\nprint("\tRF
distance: \t", dendropy.calculate.treecompare.symmetric_difference(t2,
rt4))\n\nprint("\tt3 vs inferred trees")\nprint("\tRF distance: \t",
dendropy.calculate.treecompare.symmetric_difference(t3, rt0))\nprint("\tRF
distance: \t", dendropy.calculate.treecompare.symmetric_difference(t3,
rt1))\nprint("\tRF distance: \t",
dendropy.calculate.treecompare.symmetric_difference(t3, rt2))\nprint("\tRF
distance: \t", dendropy.calculate.treecompare.symmetric_difference(t3,
rt3))\nprint("\tRF distance: \t",
dendropy.calculate.treecompare.symmetric_difference(t3, rt4))\n\nprint("\tt4 vs
inferred trees")\nprint("\tRF distance: \t",
dendropy.calculate.treecompare.symmetric_difference(t4, rt0))\nprint("\tRF
distance: \t", dendropy.calculate.treecompare.symmetric_difference(t4,
rt1))\nprint("\tRF distance: \t",
dendropy.calculate.treecompare.symmetric_difference(t4, rt2))\nprint("\tRF
distance: \t", dendropy.calculate.treecompare.symmetric_difference(t4,
rt3))\nprint("\tRF distance: \t",
dendropy.calculate.treecompare.symmetric_difference(t4, rt4))'
```

[62]:
```python
#Create your own tree sample
#Here I created the 5 size 5 tree mixture model
seed_val = 1
tm = TreeMixture(num_clusters=5, num_nodes=5)
tm.simulate_pi(seed_val=seed_val)
tm.simulate_trees(seed_val=seed_val)
tm.sample_mixtures(num_samples=200, seed_val=seed_val)
tm.save_mixture('data/q_2_5_tm_5node_200sample_5clusters.pkl',True)
```

```
Creating random tree with fixed number of nodes...
Creating random tree with fixed number of nodes...
Creating random tree with fixed number of nodes...
Creating random tree with fixed number of nodes...
Creating random tree with fixed number of nodes...
Sampling tree nodes...
Sampling tree nodes...
Sampling tree nodes...
Sampling tree nodes...
Sampling tree nodes...
Saving tree mixture to  data/q_2_5_tm_5node_200sample_5clusters.pkl , samples
to:  data/q_2_5_tm_5node_200sample_5clusters.pkl_samples.txt ...
Saving pi to  data/q_2_5_tm_5node_200sample_5clusters.pkl_pi.npy , samples to:
data/q_2_5_tm_5node_200sample_5clusters.pkl_samples.npy , sample assignments to
```

```
data/q_2_5_tm_5node_200sample_5clusters.pkl_sample_assignments.npy ...
Saving tree to  data/q_2_5_tm_5node_200sample_5clusters.pkl_tree_0 ...
Saving Newick string to
data/q_2_5_tm_5node_200sample_5clusters.pkl_tree_0_newick.txt ...
Saving topology to
data/q_2_5_tm_5node_200sample_5clusters.pkl_tree_0_topology.npy , theta to:
data/q_2_5_tm_5node_200sample_5clusters.pkl_tree_0_theta.npy ,  samples to
data/q_2_5_tm_5node_200sample_5clusters.pkl_tree_0_samples.npy  and
data/q_2_5_tm_5node_200sample_5clusters.pkl_tree_0_filtered_samples.npy ...
Saving topology to
data/q_2_5_tm_5node_200sample_5clusters.pkl_tree_0_topology.txt ,  samples to
data/q_2_5_tm_5node_200sample_5clusters.pkl_tree_0_samples.txt  and
data/q_2_5_tm_5node_200sample_5clusters.pkl_tree_0_filtered_samples.txt ...
Saving tree to  data/q_2_5_tm_5node_200sample_5clusters.pkl_tree_1 ...
Saving Newick string to
data/q_2_5_tm_5node_200sample_5clusters.pkl_tree_1_newick.txt ...
Saving topology to
data/q_2_5_tm_5node_200sample_5clusters.pkl_tree_1_topology.npy , theta to:
data/q_2_5_tm_5node_200sample_5clusters.pkl_tree_1_theta.npy ,  samples to
data/q_2_5_tm_5node_200sample_5clusters.pkl_tree_1_samples.npy  and
data/q_2_5_tm_5node_200sample_5clusters.pkl_tree_1_filtered_samples.npy ...
Saving topology to
data/q_2_5_tm_5node_200sample_5clusters.pkl_tree_1_topology.txt ,  samples to
data/q_2_5_tm_5node_200sample_5clusters.pkl_tree_1_samples.txt  and
data/q_2_5_tm_5node_200sample_5clusters.pkl_tree_1_filtered_samples.txt ...
Saving tree to  data/q_2_5_tm_5node_200sample_5clusters.pkl_tree_2 ...
Saving Newick string to
data/q_2_5_tm_5node_200sample_5clusters.pkl_tree_2_newick.txt ...
Saving topology to
data/q_2_5_tm_5node_200sample_5clusters.pkl_tree_2_topology.npy , theta to:
data/q_2_5_tm_5node_200sample_5clusters.pkl_tree_2_theta.npy ,  samples to
data/q_2_5_tm_5node_200sample_5clusters.pkl_tree_2_samples.npy  and
data/q_2_5_tm_5node_200sample_5clusters.pkl_tree_2_filtered_samples.npy ...
Saving topology to
data/q_2_5_tm_5node_200sample_5clusters.pkl_tree_2_topology.txt ,  samples to
data/q_2_5_tm_5node_200sample_5clusters.pkl_tree_2_samples.txt  and
data/q_2_5_tm_5node_200sample_5clusters.pkl_tree_2_filtered_samples.txt ...
Saving tree to  data/q_2_5_tm_5node_200sample_5clusters.pkl_tree_3 ...
Saving Newick string to
data/q_2_5_tm_5node_200sample_5clusters.pkl_tree_3_newick.txt ...
Saving topology to
data/q_2_5_tm_5node_200sample_5clusters.pkl_tree_3_topology.npy , theta to:
data/q_2_5_tm_5node_200sample_5clusters.pkl_tree_3_theta.npy ,  samples to
data/q_2_5_tm_5node_200sample_5clusters.pkl_tree_3_samples.npy  and
data/q_2_5_tm_5node_200sample_5clusters.pkl_tree_3_filtered_samples.npy ...
Saving topology to
data/q_2_5_tm_5node_200sample_5clusters.pkl_tree_3_topology.txt ,  samples to
data/q_2_5_tm_5node_200sample_5clusters.pkl_tree_3_samples.txt  and
```

```
data/q_2_5_tm_5node_200sample_5clusters.pkl_tree_3_filtered_samples.txt ...
Saving tree to  data/q_2_5_tm_5node_200sample_5clusters.pkl_tree_4 ...
Saving Newick string to
data/q_2_5_tm_5node_200sample_5clusters.pkl_tree_4_newick.txt ...
Saving topology to
data/q_2_5_tm_5node_200sample_5clusters.pkl_tree_4_topology.npy , theta to:
data/q_2_5_tm_5node_200sample_5clusters.pkl_tree_4_theta.npy ,  samples to
data/q_2_5_tm_5node_200sample_5clusters.pkl_tree_4_samples.npy  and
data/q_2_5_tm_5node_200sample_5clusters.pkl_tree_4_filtered_samples.npy ...
Saving topology to
data/q_2_5_tm_5node_200sample_5clusters.pkl_tree_4_topology.txt ,  samples to
data/q_2_5_tm_5node_200sample_5clusters.pkl_tree_4_samples.txt  and
data/q_2_5_tm_5node_200sample_5clusters.pkl_tree_4_filtered_samples.txt ...
```

[ ]:

14

2_6

January 2, 2020

[2]:
```
#import libraries
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.mlab as mlab
from scipy.stats import multivariate_normal, poisson
```

[3]:
```
#generate the data
def generate_data(n_data, means, covariances, weights, rates):
    n_clusters, n_features = means.shape
    data = np.zeros((n_data, n_features))
    poission_data = np.zeros(n_data)
    colors = np.zeros(n_data, dtype='str')
    for i in range(n_data):
        # pick a cluster id and create data from this cluster
        k = np.random.choice(n_clusters, size=1, p=weights)[0]
        x = np.random.multivariate_normal(means[k], covariances[k])
        data[i] = x
        poission_data[i] = np.random.poisson(rates[k])
        if k == 0:
            colors[i] = 'red'
        elif k == 1:
            colors[i] = 'blue'
        elif k == 2:
            colors[i] = 'green'

    return data, poission_data, colors


# means, covs: means and covariances of Gaussians
# rates: rates of Poissons
# title: title of the plot defining which EM iteration
def plot_contours(X, S, means, covs, title, rates):
    plt.figure()
    plt.scatter(X[:, 0], X[:, 1], s=S)

    delta = 0.025
    k = means.shape[0]
    x = np.arange(-2.0, 7.0, delta)
    y = np.arange(-2.0, 7.0, delta)
    X, Y = np.meshgrid(x, y)
    col = ['green', 'red', 'indigo']
    for i in range(k):
        mean = means[i]
        cov = covs[i]
        sigmax = np.sqrt(cov[0][0])
        sigmay = np.sqrt(cov[1][1])
        sigmaxy = cov[0][1] / (sigmax * sigmay)
        pos = np.dstack((X, Y))
        rv = multivariate_normal(mean,cov)
        plt.contour(X, Y, rv.pdf(pos), colors=col[i], linewidths=rates[i],␣
↪alpha=0.1)

    plt.title(title)
    plt.tight_layout()
```

```
[4]: class EM:

    def __init__(self, n_components, n_iter, tol, seed):
        self.n_components = n_components
        self.n_iter = n_iter
        self.tol = tol
        self.seed = seed
        self.it=0

    def fit(self, X, S):

        # data's dimensionality
        self.n_row, self.n_col = X.shape

        # initialize parameters
        np.random.seed(self.seed)
        chosen = np.random.choice(self.n_row, self.n_components, replace=False)
        self.means = X[chosen]
        self.weights = np.full(self.n_components, 1 / self.n_components)
        if self.n_components == 3:
            self.rates = (np.mean(S) * np.ones(self.n_components) / np.array([1,␣
↪2, 3])[np.newaxis]).flatten()
        elif self.n_components == 2:
            self.rates = (np.mean(S) * np.ones(self.n_components) / np.array([1,␣
↪2])[np.newaxis]).flatten()
        shape = self.n_components, self.n_col, self.n_col
        self.covs = np.full(shape, np.cov(X, rowvar=False))
        new_covs = []
        for c in self.covs:
```

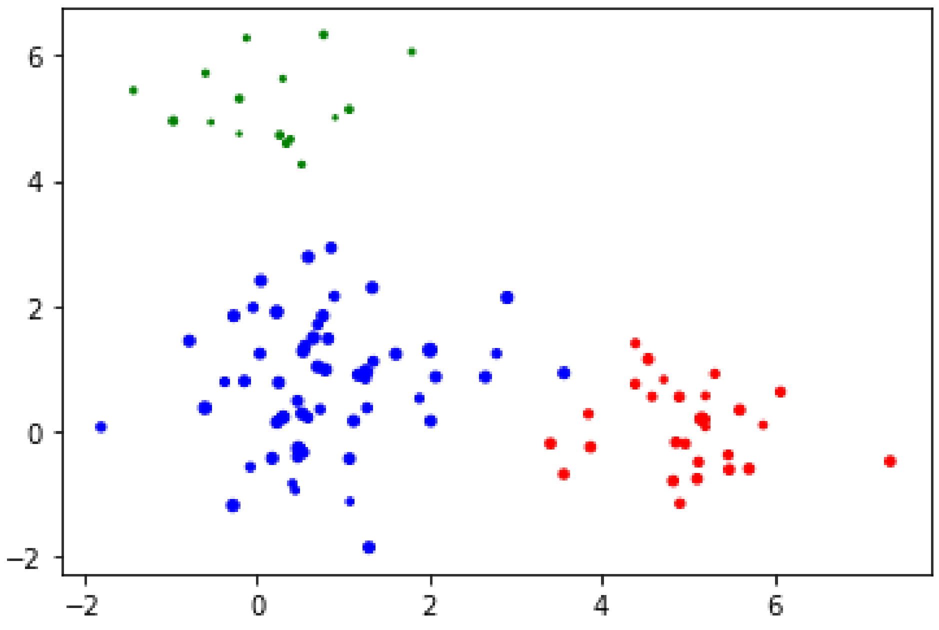
2

```
            new_covs = np.append(new_covs, np.diag(np.diag(c))) # making the␣
↪covariances diagonal (question assumption)
        self.covs = np.array(new_covs).reshape(self.n_components, 2, 2)

        log_likelihood = -np.inf
        self.converged = False

        for i in range(self.n_iter):
            self._do_estep(X, S) #run e-step
            self._do_mstep(X, S) #run m-step
            log_likelihood_new = self._compute_log_likelihood(X, S)

            if (log_likelihood_new - log_likelihood) <= self.tol:
                self.converged = True
                break

            log_likelihood = log_likelihood_new
        print('pi:',self.weights)
        print('mu:',self.means)
        print('sigma:',self.covs)
        print('lambda:',self.rates)
        return self

    def _do_estep(self, X, S):
        """
        E-step
        """
        #Compute responsibility

        N = X.shape[0]
        k = self.n_components

        self.r = np.ones(shape=(N,k))
        for i in range(0,N):
            normalizer = 0
            for j in range(0,k):
                self.r[i][j] = self.weights[j] * multivariate_normal(self.
↪means[j],self.covs[j]).pdf(X[i]) * poisson(self.rates[j]).pmf(S[i])
                normalizer+=self.r[i][j]
            self.r[i] = self.r[i]/normalizer
        return self

    def _do_mstep(self, X, S):
        """M-step, update parameters"""
        N = X.shape[0]
        k = self.n_components
        for j in range(0,k):
            r_k = np.sum(self.r,axis=0)[j]
            self.weights[j] = r_k/N
            self.rates[j]= np.dot(self.r[:,j],S)/r_k
            for i in range(0,X.shape[1]):
                self.means[j][i] = np.dot(self.r[:,j],X[:,i])/r_k
                self.covs[j][i][i] = np.dot(self.r[:,j],np.square(X[:,i] - self.
↪means[j][i]))/r_k
        return self

    def _compute_log_likelihood(self, X, S):
        """compute the log likelihood of the current parameter"""
        log_likelihood = 0
        self.it+=1
        N = X.shape[0]
        k = self.n_components
        A = 0
        for j in range(0,k):
            r_k = np.sum(self.r,axis=0)[j]
            A = A + r_k*np.log(weights[j])
        B = 0
        for i in range(0, N):
            for j in range(0,k):
                B = B + self.r[i][j]*(multivariate_normal(self.means[j],self.
↪covs[j]).pdf(X[i]) + poisson(self.rates[j]).pmf(S[i]))
        log_likelihood = A + B
        return log_likelihood
```

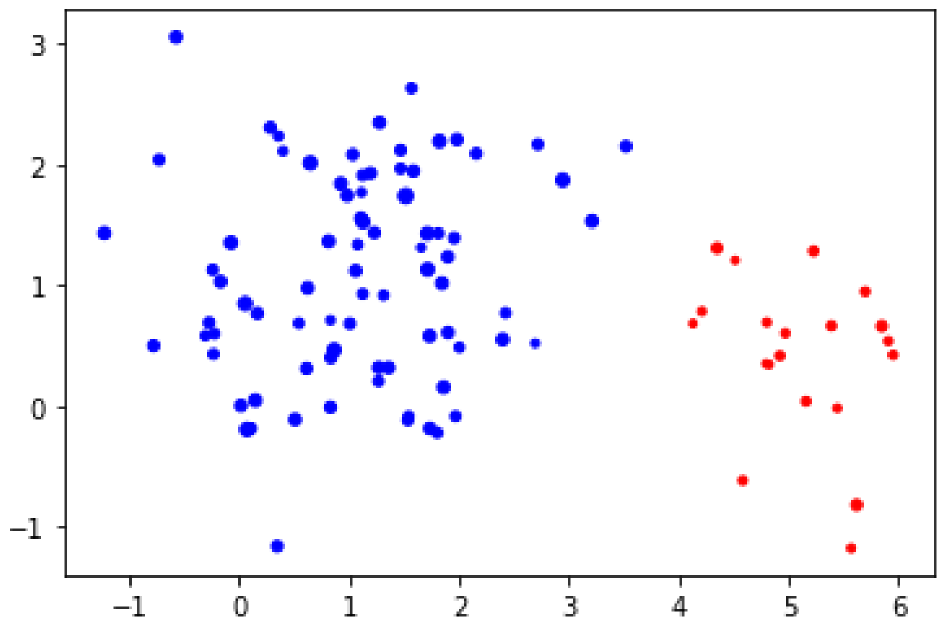
```
[5]: ##############################################################################################

# params for 3 clusters
means = np.array([
    [5, 0],
    [1, 1],
    [0, 5]
])

covariances = np.array([
    [[.5, 0.], [0, .5]],
    [[.92, 0], [0, .91]],
    [[.5, 0.], [0, .5]]
])

weights = [1 / 4, 1 / 2, 1 / 4]

# params for 2 clusters
means_2 = np.array([
    [5, 0],
    [1, 1]
])

covariances_2 = np.array([
    [[.5, 0.], [0, .5]],
    [[.92, 0], [0, .91]]
])

weights_2 = [1 / 4, 3 / 4]

np.random.seed(3)

rates = np.random.uniform(low=.2, high=20, size=3)
print("Poisson rates for 3 components:")
print(rates)

rates_2 = np.random.uniform(low=.2, high=20, size=2)
print("Poisson rates for 2 components:")
print(rates_2)

# generate data
X, S, colors = generate_data(100, means, covariances, weights, rates)
plt.scatter(X[:, 0], X[:, 1], s=S, c=colors) # the Poisson data is shown through␣
 ↪size of the points: s
plt.show()

X_2, S_2, colors_2 = generate_data(100, means_2, covariances_2, weights_2,␣
 ↪rates_2)
plt.scatter(X_2[:, 0], X_2[:, 1], s=S_2, c=colors_2) # the Poisson data is shown␣
 ↪through size of the points: s
plt.show()

##############################################################################################

em = EM(n_components=3, n_iter=1, tol=1e-4, seed=1)
em.fit(X, S)

# plot: call plot_contours and give it the params updated from EM with 3␣
 ↪components (after 1 iteration)
plot_contours(X, S, em.means, em.covs, 'EM after 1 iteration, 3 components', em.
 ↪rates)
plt.show()

em = EM(n_components=3, n_iter=50, tol=1e-4, seed=1)
em.fit(X, S)
```

```
# plot: call plot_contours and give it the params updated from EM with 3␣
 ↪components (after 50 iterations)
plot_contours(X, S, em.means, em.covs, 'EM after 50 iteration, 3 components', em.
 ↪rates)
plt.show()

em_2 = EM(n_components=2, n_iter=1, tol=1e-4, seed=1)
em_2.fit(X_2, S_2)

# plot: call plot_contours and give it the params updated from EM with 2␣
 ↪components (after 1 iteration)
plot_contours(X_2, S_2, em_2.means, em_2.covs, 'EM after 1 iteration, 2␣
 ↪components', em_2.rates)
plt.show()

em_2 = EM(n_components=2, n_iter=50, tol=1e-4, seed=1)
em_2.fit(X_2, S_2)

# plot: call plot_contours and give it the params updated from EM with 2␣
 ↪components (after 50 iterations)
plot_contours(X_2, S_2, em_2.means, em_2.covs, 'EM after 50 iteration, 2␣
 ↪components', em_2.rates)
plt.show()
```

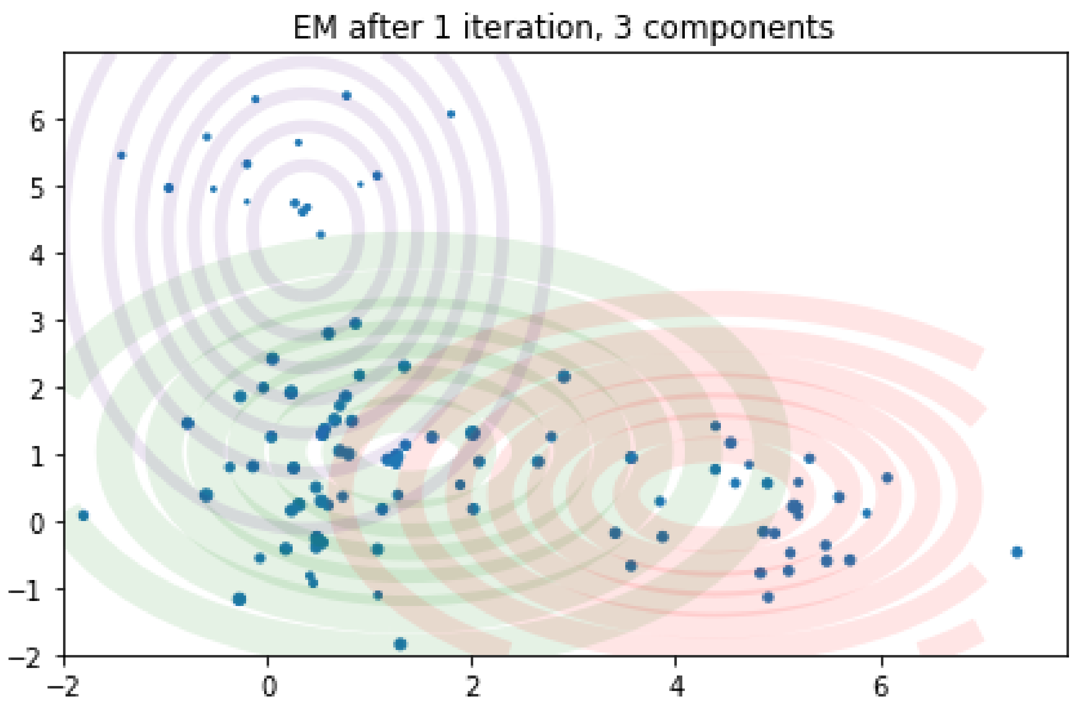
```
Poisson rates for 3 components:
[11.10579847 14.22132689  5.95991383]
Poisson rates for 2 components:
[10.31438658 17.8803497 ]
```



6

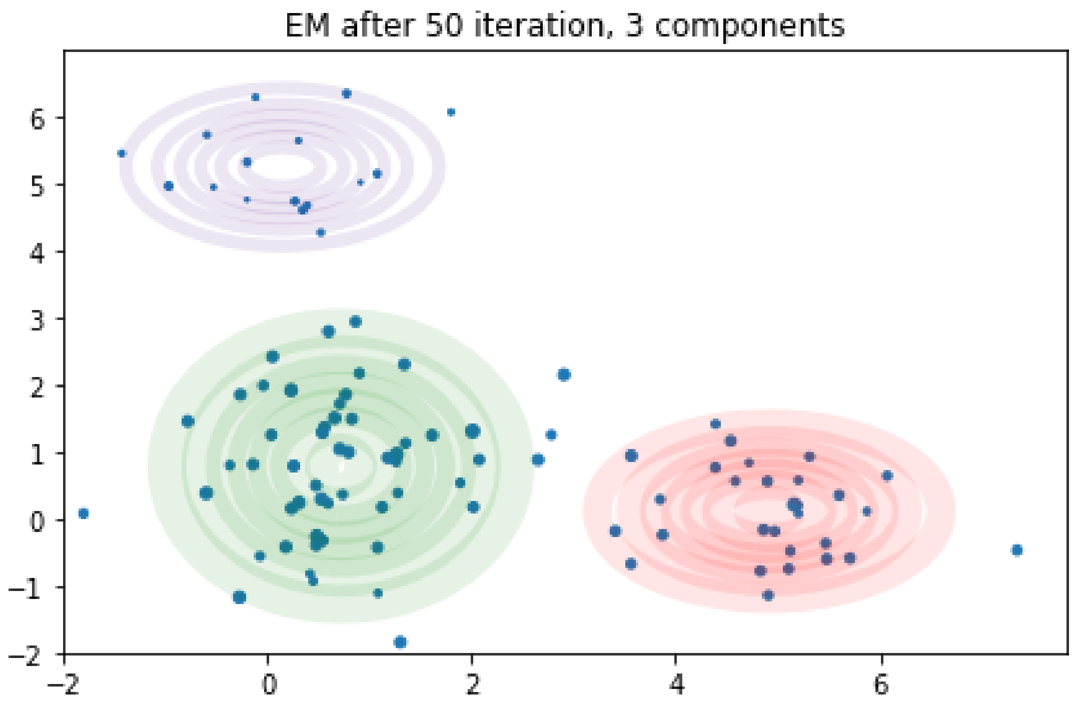


```
pi: [0.68097201 0.19777761 0.12125038]
mu: [[1.39090819 1.02556175]
 [4.37920861 0.38004614]
 [0.37925052 4.31561061]]
sigma: [[[3.13652994 0.        ]
  [0.         2.23115745]]

 [[3.255987   0.        ]
  [0.         1.97223875]]

 [[1.37627583 0.        ]
  [0.         4.8222996 ]]]
lambda: [14.53663478  9.92163139  4.93756871]
```



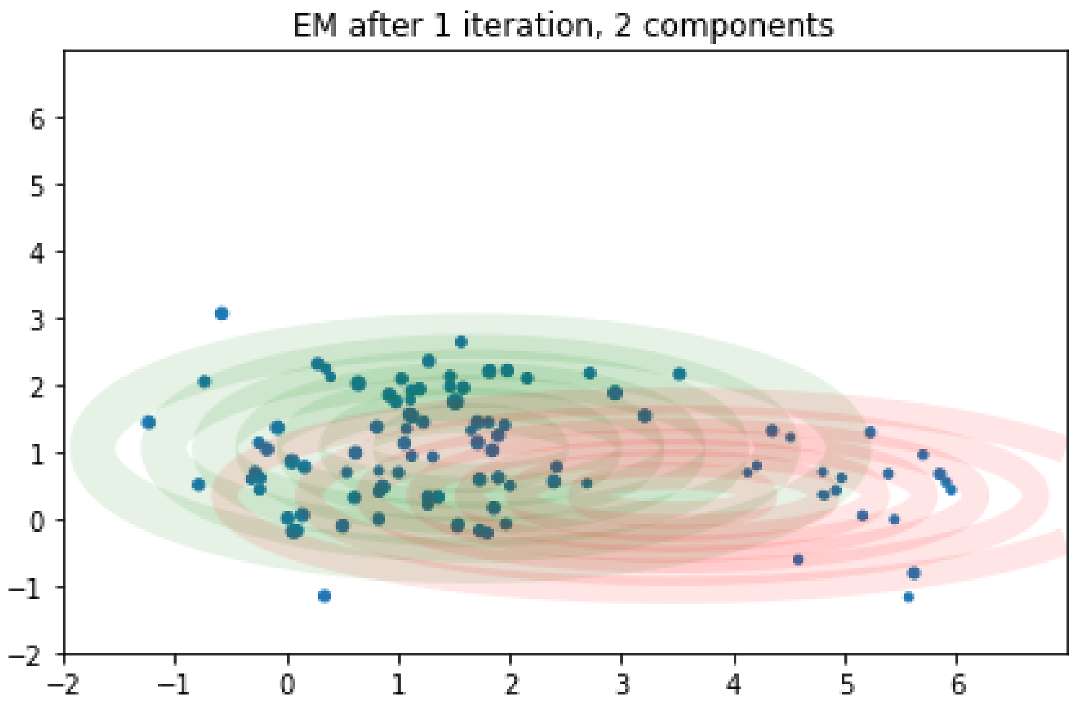
```
pi: [0.5475732  0.29243667 0.15999013]
mu: [[0.72511652 0.78045156]
 [4.91681763 0.10859925]
 [0.15220355 5.23594892]]
sigma: [[[0.73994704 0.        ]
  [0.         1.08016108]]

 [[0.71574933 0.        ]
  [0.         0.42573112]]

 [[0.61676029 0.        ]
  [0.         0.3628337 ]]]
lambda: [15.0777882  11.53710122  5.18741871]
```

8



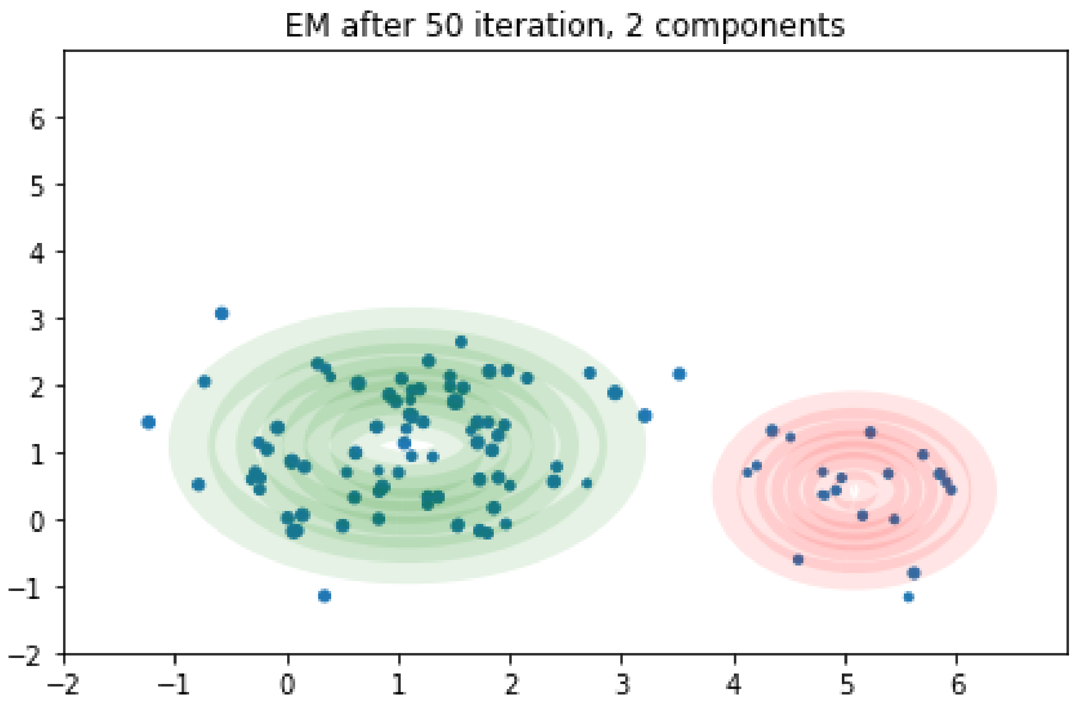
```
pi: [0.86842593 0.13157407]
mu: [[1.65010138 1.04401885]
 [3.45744028 0.34031114]]
sigma: [[[2.82765927 0.        ]
  [0.         0.69238652]]

 [[3.95525471 0.        ]
  [0.         0.48962477]]]
lambda: [17.51305948 10.42154465]
```

9



```
pi: [0.79954804 0.20045196]
mu: [[1.08597745 1.08596335]
 [5.08655122 0.41480906]]
sigma: [[[0.89610443 0.        ]
  [0.         0.70871026]]

 [[0.33777008 0.        ]
  [0.         0.41630876]]]
lambda: [18.03375427 10.78137203]
```

10


EM after 50 iteration, 2 components
6
5
4
3
2
1
0
−1
−2
−2
−1
0
1
2
3
4
5
6

```
[ ]:
```

2_8

January 3, 2020

[1]:
```
"""
File for task 2.8 in assignment 2 of DD2434.
See README for more instructions and information on generating/loading trees.
"""
#import needed libraries
import numpy as np
from Tree import Tree

def test_binary_trees(seed_val=0, k=2, num_nodes=5):
    """
    Test function for generating binary trees from Tree.py and printing topology.
    :param: seed_val: Numpy seed.
    :param: k: Number of possible values for a node.
    :param: num_nodes: Number of total nodes in the tree. Has to be odd, or it␣
 ↪will auto correct to +1.
    :return: binary_tree: A generated binary tree. Tree object from Tree.py.
    """
    binary_tree = Tree()
    #This is where I tinkered with the alphas to see the effect they have on the␣
 ↪P[sum=odd]
    binary_tree.create_random_binary_tree(seed_val=seed_val, k=k,␣
 ↪num_nodes=num_nodes,alpha=[1.0,1.0,1.0,1.0,1.0])

    # Print tree
    binary_tree.print()

    # Print topology
    binary_tree.print_topology()

    return binary_tree

def load_tree():
    """
    Function for loading the tree for the assignment.
    :return: Loaded tree
    """
    binary_tree = Tree()
```

1

```python
    binary_tree.load_tree('tree_task_2_8')

    # Print tree
    # binary_tree.print()

    # Print topology
    binary_tree.print_topology()

    return binary_tree

def load_interval(path='interval_task_2_8.npy', str=False):
    """
    Function for loading the interval dictionary for the assignment.
    :param: str: Load from text file or numpy file.
    :return: Dictionary of interval.
    """
    if not str:
        mode_dict = np.load(path, allow_pickle=True).item()
    else:
        from ast import literal_eval
        with open(path, "r") as file:
            data = file.readlines()
        mode_dict = literal_eval(data[0])
    return mode_dict

def test_node_sample_proportion(dict, tree, n=1000):
    """
    Function for sampling n leaf sets and computing, for each leaf node,
    the proportion of samples that are in the provided interval.
    :param: dict: Interval dictionary containing node keys and lists of possible␣
 ↪node values.
    :param: tree: Generated tree. Tree object from Tree.py.
    :param: n: Number of samples.
    :return: Dictionary with proportion of samples that are in the interval for␣
 ↪each node.
    """
    n_samples = {}
    for i in range(n):
        leaf_samples = odd_sum_sampling(tree)
        if i == 0:
            n_samples = {key: 0 for key in leaf_samples.keys()}
        for key, val in leaf_samples.items():
            if val in dict[key]:
                n_samples[key] += 1/n
    return n_samples

def test_sample_proportion(dict, tree, n=1000):
```

2

```
    """
    Function for sampling n leaf sets and computing the proportion of sets that␣
↪are in the interval dict.
    :param: dict: Interval dictionary containing node keys and lists of possible␣
↪node values.
    :param: tree: Generated tree. Tree object from Tree.py.
    :param: n: Number of samples.
    :return: Proportion of samples that are in interval dict.
    """
    n_samples = 0
    for i in range(n):
        leaf_samples = odd_sum_sampling(tree)
        for key, val in leaf_samples.items():
            if val not in dict[key]:
                break
        else:
            n_samples += 1
    return n_samples/n

def tree_DP(tree):
    # Implementation of Dynamic programming algorithm compute P[sum=odd]
    tree_arr = tree.get_topology_array()
    theta_arr=tree.get_theta_array()
    k = theta_arr[0].shape[0]
    s = np.zeros((tree_arr.shape[0],k))
    explored = find_leaves(tree_arr) #get the leaves
    explored_thus_far=explored[:] #this is what we explored thus far
    for v in explored_thus_far:
        for i in range(0,k):
            s[v,i] = i%2 #for the leaves, it is simply i mod(2)
        explored.remove(v) #remove them from explored
        if(int(tree_arr[v]) not in explored):
            explored.append(int(tree_arr[v])) #add its parent if it is not␣
↪already in it
    iterator = 0

    while(len(explored)>0): #while there is nothing left to explore
        v = int(explored[iterator]) #get father index
        if(has_a_child(v,tree_arr,explored_thus_far)==False): #explore it only␣
↪if we have seen all of its children
            children = find_children(v,tree_arr) #get its already explored␣
↪children
            for i in range(0,k):
                first_element = np.dot(s[children[0]],theta_arr[children[0]][i])␣
↪* np.dot(1-s[children[1]],theta_arr[children[1]][i])
```

```
                second_element = np.
↪dot(1-s[children[0]],theta_arr[children[0]][i]) * np.
↪dot(s[children[1]],theta_arr[children[1]][i])
                s[v,i] = first_element+second_element #calculate s(v,i)
            explored.remove(v) #We have explored it, so remove it
            explored_thus_far.append(v) #Store it in this list showcasing what␣
↪we have seen thus far (so that its father can use this info)
            explored.append(tree_arr[v])#Add his parent, as a future exploration
            if (v==0): #if we reach the root
                break
        iterator=(iterator+1)%len(explored) #'fancy' way of looping through the␣
↪list
    #print('final s(v,i):',s)
    #Uncomment if you want to see the full 's' matrix
    final_sum=0
    for i in range(0,k):
        final_sum += s[0,i]*theta_arr[0][i]
    #print('P[sum=odd]',final_sum)
    return s, final_sum

def find_leaves(tree_arr):
    result=[]
    for i in range(1,tree_arr.shape[0]): # 0 always root (never leaf)
        is_leaf=True
        for j in range(1,tree_arr.shape[0]):
            if (tree_arr[j]==i): #if you found a node pointing at it, it is not␣
↪a leaf
                is_leaf=False

        if(is_leaf):
            result.append(i)
    return result

def find_children(v,tree_arr):
    result=[]
    for i in range(0,tree_arr.shape[0]):
        if (tree_arr[i]==v): #if a node is pointing at it, then add it as its␣
↪child
            result.append(int(i))
    return result

def has_a_child(v,tree_arr,tree_list):
    #this method checks whether node v has a child THAT IS NOT in the tree_list,␣
↪based
    #on the tree_arr
    result = True
    children_explored=0
    children = find_children(v,tree_arr)
    for i in children:
        if i in tree_list:
            children_explored+=1
    if children_explored==2: #if both are located in the tree_list, it has no␣
 ↪childs left outside of it (binary tree)
        result = False
    return result

def odd_sum_sampling(tree):
    # Sampling algorithm
    tree_arr = tree.get_topology_array()
    theta_arr=tree.get_theta_array()
    k = theta_arr[0].shape[0]
    s = tree_DP(tree)[0]
    #sample from root
    picked = np.zeros(tree_arr.shape[0]) #picked[i] = value of node i
    p_r = np.zeros(k) #probability of posterior categorical P[x_r = i | sum=odd]
    normalizer = 0
    for i in range(0,k):
        p_r[i] = s[0,i]*theta_arr[0][i]
        normalizer += p_r[i]
    p_r = p_r/normalizer
    picked[0] = np.random.choice(k,1,p=p_r) #sample from categorical
    leaves_left = find_leaves(tree_arr) #leaves we have to sample from left
    parity = (-1)*np.ones(tree_arr.shape[0]) #parity[i] = sum[leafs below i] =␣
 ↪odd (1) or even (0)
    parity[0]=1 #sum of all leaves are odd
    stack =[0] #root
    while (len(stack)>0): #similar to BFS
        parent=stack[0]
        children = find_children(parent,tree_arr) #get its children
        u = int(children[0])
        v = int(children[1])
        if(len(find_children(u,tree_arr))>0): #add them in the list only if they␣
 ↪are not leaves
            stack.append(u)
        if(len(find_children(v,tree_arr))>0):
            stack.append(v)
        stack.remove(parent) #remove the parent (no need to examine him again)
        u_parity = np.zeros(2) #sum=even or odd
        #calculate bernoulli for sum[below u]
        for i in range(0,k):
            u_parity[1] += s[u][i]*theta_arr[u][int(picked[parent])][i]
        u_parity[0]=1-u_parity[1]
```

5

```
        parity[u] = np.random.choice(2,1,p=u_parity) #sample from it: 0->even,␣
↪1->odd
        if(parity[parent]==1): #sum[parrent] mod(2) = sum[u] + sum[v] mod(2)
            parity[v] = 1-parity[u]
        else:
            parity[v] = parity[u]
        picked[u] = sample_node(u,parity[u],s,int(picked[parent]),theta_arr)␣
↪#sample based on your parity
        picked[v] = sample_node(v,parity[v],s,int(picked[parent]),theta_arr)

    #print(picked)
    #print(parity)
    #print(picked)
    #print(picked[leaves_left])
    #create a dictionary containing the leaf sample
    result_dictionary = dict([(str(i),int(picked[i])) for i in leaves_left])
    #print(result_dictionary)
    return result_dictionary

def sample_node(u,parity,s,parent_value,theta_arr):
    prob = np.zeros(theta_arr.shape[1]) #initialize
    k = prob.shape[0]
    if(parity==1): #see assignment paper for formula
        normalizer = 0
        for i in range(0,k):
            prob[i]=s[u][i]*theta_arr[u][parent_value][i]
            normalizer +=prob[i]
        prob = prob/normalizer
    else:
        normalizer = 0
        for i in range(0,k):
            prob[i]=(1-s[u][i])*theta_arr[u][parent_value][i]
            normalizer +=prob[i]
        prob = prob/normalizer
    #np.random.seed(1)
    #print('probabilities',prob)
    return np.random.choice(k,1,p=prob) #pick
def main():
    # Test tree
    binary_tree = test_binary_trees(seed_val=0, k=2, num_nodes=5)

    # Load tree
    # binary_tree = load_tree()

    # Load interval dictionary
    # most_freq_dict = load_interval('interval_task_2_8.npy')
    # node_most_freq_dict = load_interval('node_interval_task_2_8.npy')
```

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

```
Creating random binary tree with fixed number of nodes...
Error! Alpha needs to contain k positive values!
```

[2]:
```
test_tree = load_tree()
```

```
Loading tree from  tree_task_2_8 ...
Printing tree topology...
        0
                1
                        3
                                7
                                8
                                        17
                                        18
                        4
                                19
                                20
                2
                        5
                                11
                                        15
                                        16
                                12
                        6
                                9
                                10
                                        13
                                        14
```

[3]:
```
prob_test=tree_DP(test_tree)[1]
print(prob_test)
```

```
0.5000002322336661
```

[4]:
```
samples = odd_sum_sampling(test_tree)
print('samples',samples)
```

```
samples {'7': 3, '9': 3, '12': 3, '13': 0, '14': 2, '15': 1, '16': 3, '17': 4,
'18': 1, '19': 1, '20': 0}
```

[7]:
```
#interval task 2_8
most_freq_dict = load_interval('interval_task_2_8.npy')
test_sample_proportion(most_freq_dict,test_tree, n=1000)
```

```
[7]: 0.178
```

```
[8]: #node interval 2_8
     most_freq_dict = load_interval('node_interval_task_2_8.npy')
     test_node_sample_proportion(most_freq_dict, test_tree, n=1000)
```

```
[8]: {'7': 0.4150000000000003,
      '9': 0.6100000000000004,
      '12': 0.4850000000000004,
      '13': 0.5390000000000004,
      '14': 0.5340000000000004,
      '15': 0.4030000000000003,
      '16': 0.5090000000000003,
      '17': 0.36300000000000027,
      '18': 0.4260000000000003,
      '19': 0.4930000000000004,
      '20': 0.5490000000000004}
```

```
[ ]:
```

8