

DD2434 Machine Learning, Advanced Course

Fernando García Sanz - fegs@kth.se

December 31, 2019



Contents

1	Knowing the rules	3
2	Dependencies in a Directed Graphical Model	3
3	Likelihood of a tree GM	4
4	Simple VI	4
5	Mixture of trees with observable variables	6
6	Super epicentra – EM	15
7	Super epicentra – VI	20
8	Sampling from a tree GM	25
9	Failing components VI	25

1 Knowing the rules

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

Yes, I have read it.

Question 2.1.2: *List all your collaborations concerning the problem formulations in this assignment.*

1. Adrian Campoy
2. Doumitru Nimara
3. Gustavo Teodoro Beck
4. Lucas Gongora
5. Flavia García

Question 2.1.3: *Have you discussed solutions with anybody?*

No, I have not discussed my solutions.

2 Dependencies in a Directed Graphical Model

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

Yes.

Question 2.2.5: *In the graphical model of Figure 1, is $\mu_k \perp \tau_k | X, \dots, X^N$?*

No.

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

Yes.

Question 2.2.7: *In the graphical model of Figure 2, is $\mu \perp \beta' | X, \dots, X^N$?*

No.

Question 2.2.8: *In the graphical model of Figure 2, is $X^n \perp S^n$ (not conditioned by anything)?*

No.

Question 2.2.9 *In the graphical model of Figure 2, is $X^n \perp S^n | \mu_k, \tau_k$*

No.

3 Likelihood of a tree GM

Question 2.3.10

Question 2.3.11

4 Simple VI

Question 2.4.12

The code can be found in appendix A.

Question 2.4.13

The exact posterior is defined by:

$$p(\mu|\tau) = \mathcal{N}(\mu|\mu_0, (\lambda_0\tau)^{-1})$$

$$p(\tau) = \text{Gam}(\tau|a_0, b_0)$$

The joint of these distributions provides the exact posterior.

Question 2.4.14

Once the code has been implemented, several tests have been performed, observing the following results:

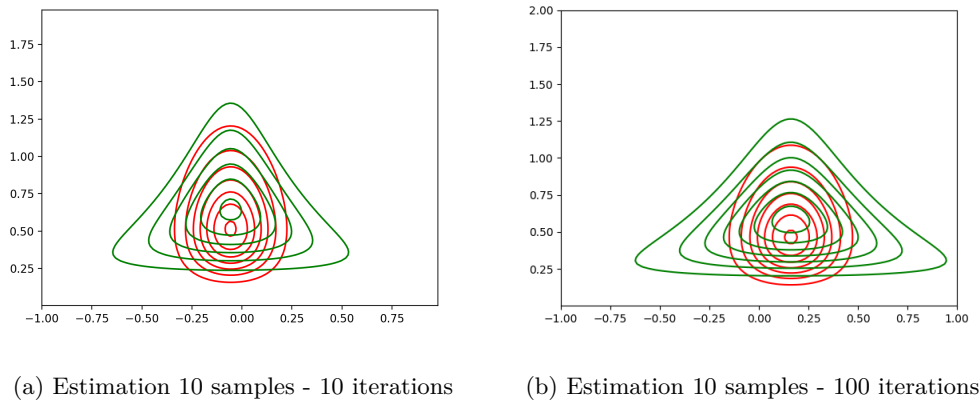


Figure 1: Approximations with 10 samples

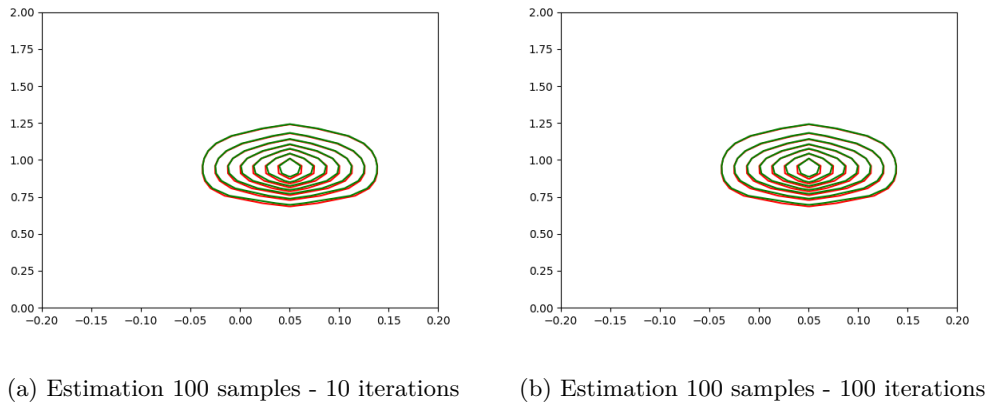


Figure 2: Approximations with 100 samples

The figures above express the different approximations performed by means of the *variational inference* algorithm for different combinations of the number of used samples and the number of iterations of the algorithm. The green shape is the original one and the red one is the one obtained from the approximation, and the horizontal axis represents the values of μ , while the vertical one represents the values of τ .

As it can be seen, in the first two plots the forms obtained are quite similar, even though the number of iterations is 10 times higher in the second one. In the last two, it has been necessary to enlarge the part of the image in which the shapes are drawn. These two show a much smaller area, providing a greater specification of the possible values μ and τ can take.

As a conclusion, it can be said that the number of used samples has a much greater relevance in the procedure, and that the algorithm converges really fast, since the results obtained when only varying the number of iterations are extremely similar.

5 Mixture of trees with observable variables

Question 2.5.15

See appendix B.

Question 2.5.16

Applying the EM algorithm to the provided data, the following results have been obtained:

- The obtained trees for a configuration with 10 nodes, 20 observations and 4 clusters have been the followings:

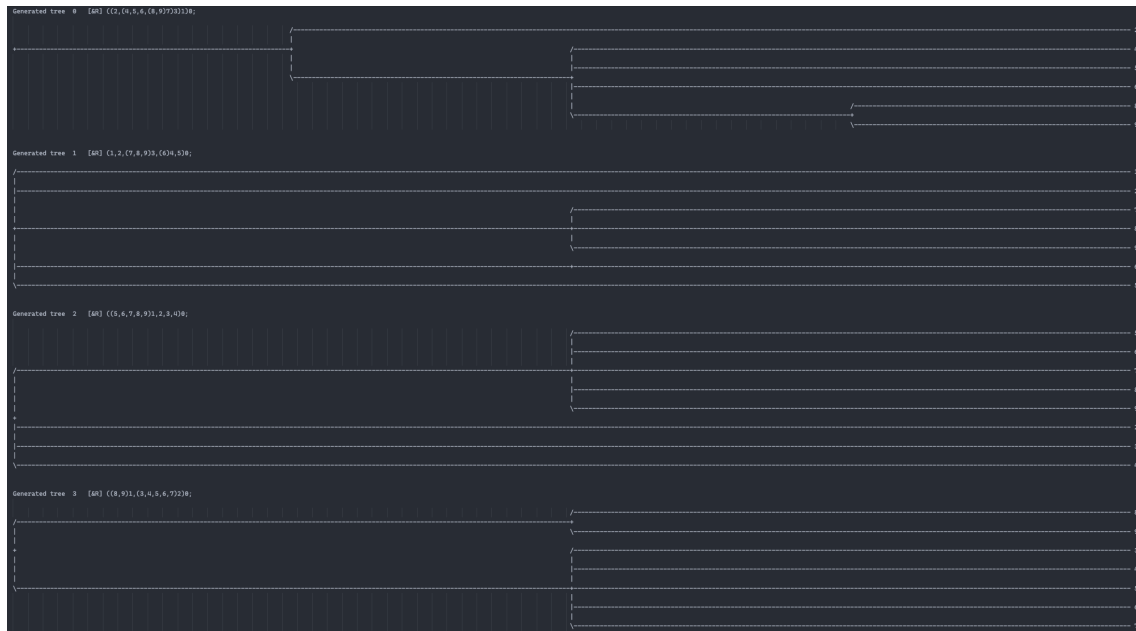


Figure 3: Inferred trees 10 - 20 - 4

- Applying Robinson-Foulds metric, it has been possible to compare the generated trees to the original ones, obtaining the following result:

```

Distances of trees:

Original tree 0 compared to generated tree 0
Robinson-Foulds distance: 8
Original tree 0 compared to generated tree 1
Robinson-Foulds distance: 6
Original tree 0 compared to generated tree 2
Robinson-Foulds distance: 5
Original tree 0 compared to generated tree 3
Robinson-Foulds distance: 7
Original tree 1 compared to generated tree 0
Robinson-Foulds distance: 9
Original tree 1 compared to generated tree 1
Robinson-Foulds distance: 7
Original tree 1 compared to generated tree 2
Robinson-Foulds distance: 6
Original tree 1 compared to generated tree 3
Robinson-Foulds distance: 6
Original tree 2 compared to generated tree 0
Robinson-Foulds distance: 9
Original tree 2 compared to generated tree 1
Robinson-Foulds distance: 7
Original tree 2 compared to generated tree 2
Robinson-Foulds distance: 8
Original tree 2 compared to generated tree 3
Robinson-Foulds distance: 8
Original tree 3 compared to generated tree 0
Robinson-Foulds distance: 7
Original tree 3 compared to generated tree 1
Robinson-Foulds distance: 7
Original tree 3 compared to generated tree 2
Robinson-Foulds distance: 2
Original tree 3 compared to generated tree 3
Robinson-Foulds distance: 6

```

Figure 4: Distances 10 - 20 - 4

- Finally, the last step taken was comparing likelihood and log-likelihood of the original mixture and the generated one:

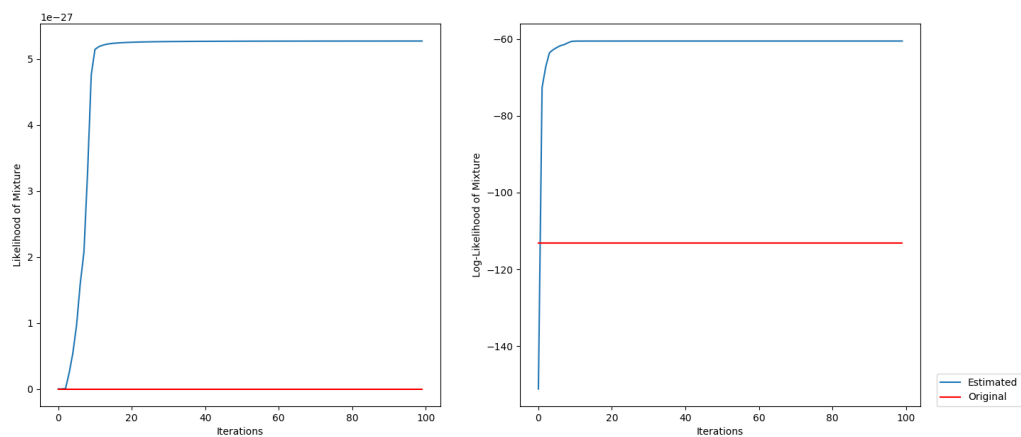


Figure 5: Likelihood and log-likelihood 10 - 20 - 4

The same procedure has been used with different configurations, obtaining the following results:

- For the case of 20 nodes, 20 observations and 4 clusters:
 - Robinson-Foulds metric:

```
Distances of trees:

Original tree 0 compared to generated tree 0
Robinson-Foulds distance: 15
Original tree 0 compared to generated tree 1
Robinson-Foulds distance: 14
Original tree 0 compared to generated tree 2
Robinson-Foulds distance: 12
Original tree 0 compared to generated tree 3
Robinson-Foulds distance: 13
Original tree 1 compared to generated tree 0
Robinson-Foulds distance: 13
Original tree 1 compared to generated tree 1
Robinson-Foulds distance: 16
Original tree 1 compared to generated tree 2
Robinson-Foulds distance: 14
Original tree 1 compared to generated tree 3
Robinson-Foulds distance: 15
Original tree 2 compared to generated tree 0
Robinson-Foulds distance: 17
Original tree 2 compared to generated tree 1
Robinson-Foulds distance: 16
Original tree 2 compared to generated tree 2
Robinson-Foulds distance: 16
Original tree 2 compared to generated tree 3
Robinson-Foulds distance: 19
Original tree 3 compared to generated tree 0
Robinson-Foulds distance: 18
Original tree 3 compared to generated tree 1
Robinson-Foulds distance: 13
Original tree 3 compared to generated tree 2
Robinson-Foulds distance: 13
Original tree 3 compared to generated tree 3
Robinson-Foulds distance: 16
```

Figure 6: Distances 20 - 20 - 4

- Likelihood and log-likelihood comparison:

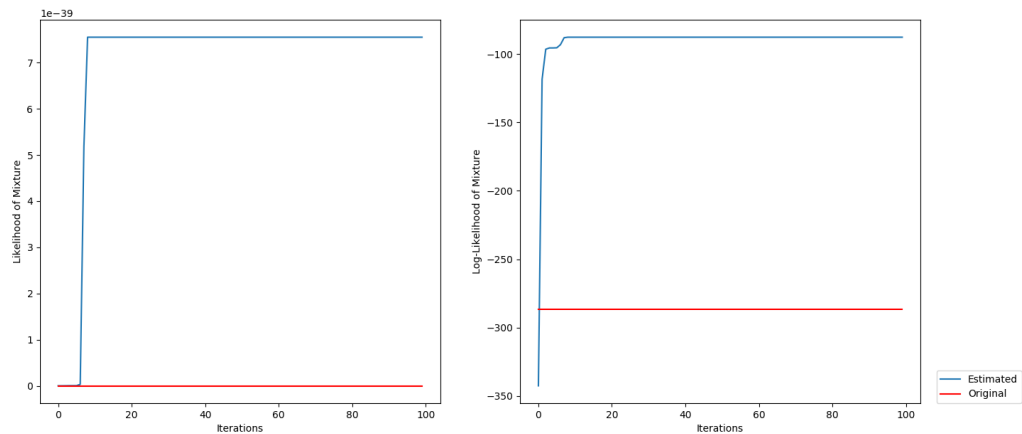


Figure 7: Likelihood and log-likelihood 20 - 20 - 4

- For the case of 10 nodes, 50 observations and 4 clusters:
 - Robinson-Foulds metric:

```

Distances of trees:

Original tree 0 compared to generated tree 0
  Robinson-Foulds distance: 5
Original tree 0 compared to generated tree 1
  Robinson-Foulds distance: 6
Original tree 0 compared to generated tree 2
  Robinson-Foulds distance: 6
Original tree 0 compared to generated tree 3
  Robinson-Foulds distance: 5
Original tree 1 compared to generated tree 0
  Robinson-Foulds distance: 6
Original tree 1 compared to generated tree 1
  Robinson-Foulds distance: 9
Original tree 1 compared to generated tree 2
  Robinson-Foulds distance: 5
Original tree 1 compared to generated tree 3
  Robinson-Foulds distance: 6
Original tree 2 compared to generated tree 0
  Robinson-Foulds distance: 10
Original tree 2 compared to generated tree 1
  Robinson-Foulds distance: 9
Original tree 2 compared to generated tree 2
  Robinson-Foulds distance: 7
Original tree 2 compared to generated tree 3
  Robinson-Foulds distance: 8
Original tree 3 compared to generated tree 0
  Robinson-Foulds distance: 6
Original tree 3 compared to generated tree 1
  Robinson-Foulds distance: 7
Original tree 3 compared to generated tree 2
  Robinson-Foulds distance: 7
Original tree 3 compared to generated tree 3
  Robinson-Foulds distance: 6

```

Figure 8: Distances 10 - 50 - 4

- Likelihood and log-likelihood comparison:

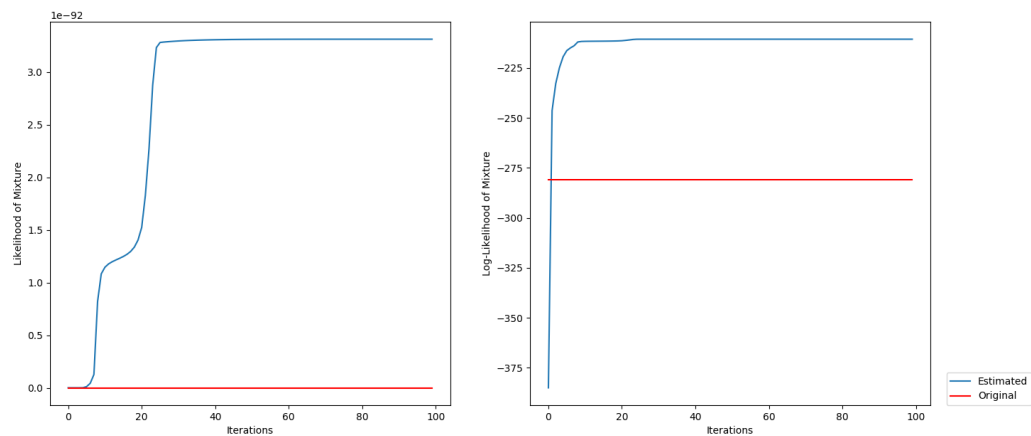


Figure 9: Likelihood and log-likelihood 10 - 50 - 4

- For the case of 10 nodes, 20 observations and 4 clusters, applying mixtures of 3 trees:
 - Robinson-Foulds metric:

```
Distances of trees:

Original tree 0 compared to generated tree 0
Robinson-Foulds distance: 5
Original tree 0 compared to generated tree 1
Robinson-Foulds distance: 8
Original tree 0 compared to generated tree 2
Robinson-Foulds distance: 9
Original tree 1 compared to generated tree 0
Robinson-Foulds distance: 6
Original tree 1 compared to generated tree 1
Robinson-Foulds distance: 5
Original tree 1 compared to generated tree 2
Robinson-Foulds distance: 6
Original tree 2 compared to generated tree 0
Robinson-Foulds distance: 8
Original tree 2 compared to generated tree 1
Robinson-Foulds distance: 9
Original tree 2 compared to generated tree 2
Robinson-Foulds distance: 8
```

Figure 10: Distances 10 - 20 - 4 - mixtures of 3 trees

- Likelihood and log-likelihood comparison:

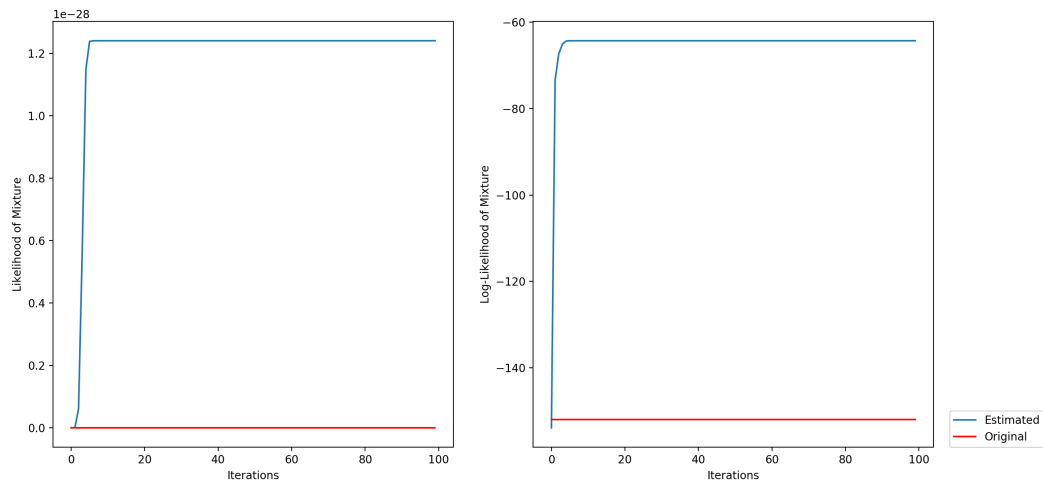


Figure 11: Likelihood and log-likelihood 10 - 20 - 4 - mixtures of 3 trees

As can be seen, the modification of the configuration produces quite different results. When the number of nodes was increased from 10 to 20, the distances between generated trees and the original ones practically doubled those of the previous case. A similar case happens with likelihood and log-likelihood after a proper number of iterations. Given that the number of nodes is much bigger, mixtures are less probable.

The case of increasing the number of observations from 20 to 50 has also affected the final result. Distances between trees do not vary that much; nonetheless, likelihood and log-likelihood are even smaller than in the previous case. Since the sequence of observations is much bigger, it is less probable.

Finally, the configuration was modified by changing the number of trees which compose the mixture from 4 to 3. This has triggered almost no changes in the probabilities shown by the generated trees unlike in the original ones. As the configuration was made for 4 clusters, applying mixtures of 3 has worsened the likelihood of the original trees, being even lower than the one sampled from the original mixture when 4 clusters were used.

Question 2.5.17

Now, some new mixtures have been generated by means of the code which can be seen in C.

The case of 10 nodes, 2 samples and 5 clusters have been implemented. The results are the followings:

```
Distances of trees:

Original tree 0 compared to generated tree 0
Robinson-Foulds distance: 0
Original tree 0 compared to generated tree 1
Robinson-Foulds distance: 7
Original tree 0 compared to generated tree 2
Robinson-Foulds distance: 9
Original tree 0 compared to generated tree 3
Robinson-Foulds distance: 7
Original tree 0 compared to generated tree 4
Robinson-Foulds distance: 6
Original tree 1 compared to generated tree 0
Robinson-Foulds distance: 0
Original tree 1 compared to generated tree 1
Robinson-Foulds distance: 7
Original tree 1 compared to generated tree 2
Robinson-Foulds distance: 9
Original tree 1 compared to generated tree 3
Robinson-Foulds distance: 7
Original tree 1 compared to generated tree 4
Robinson-Foulds distance: 6
Original tree 2 compared to generated tree 0
Robinson-Foulds distance: 6
Original tree 2 compared to generated tree 1
Robinson-Foulds distance: 7
Original tree 2 compared to generated tree 2
Robinson-Foulds distance: 7
Original tree 2 compared to generated tree 3
Robinson-Foulds distance: 7
Original tree 2 compared to generated tree 4
Robinson-Foulds distance: 4
Original tree 3 compared to generated tree 0
Robinson-Foulds distance: 8
Original tree 3 compared to generated tree 1
Robinson-Foulds distance: 7
Original tree 3 compared to generated tree 2
Robinson-Foulds distance: 7
Original tree 3 compared to generated tree 3
Robinson-Foulds distance: 3
Original tree 3 compared to generated tree 4
Robinson-Foulds distance: 10
Original tree 4 compared to generated tree 0
Robinson-Foulds distance: 9
Original tree 4 compared to generated tree 1
Robinson-Foulds distance: 8
Original tree 4 compared to generated tree 2
Robinson-Foulds distance: 6
Original tree 4 compared to generated tree 3
Robinson-Foulds distance: 8
Original tree 4 compared to generated tree 4
Robinson-Foulds distance: 9
```

Figure 12: Distances 10 - 2 - 5

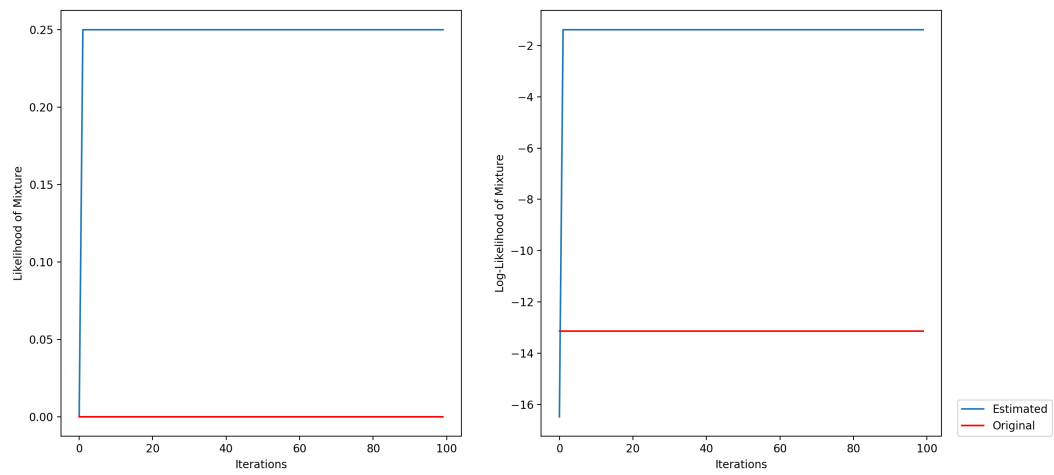


Figure 13: Likelihood and log-likelihood 10 - 2 - 5

The case of 20 nodes, 100 samples and 5 clusters have also been analyzed in this assignment, providing the following results:

```
Distances of trees:

Original tree 0 compared to generated tree 0
  Robinson-Foulds distance: 11
Original tree 0 compared to generated tree 1
  Robinson-Foulds distance: 17
Original tree 0 compared to generated tree 2
  Robinson-Foulds distance: 17
Original tree 0 compared to generated tree 3
  Robinson-Foulds distance: 17
Original tree 0 compared to generated tree 4
  Robinson-Foulds distance: 16
Original tree 1 compared to generated tree 0
  Robinson-Foulds distance: 16
Original tree 1 compared to generated tree 1
  Robinson-Foulds distance: 16
Original tree 1 compared to generated tree 2
  Robinson-Foulds distance: 14
Original tree 1 compared to generated tree 3
  Robinson-Foulds distance: 18
Original tree 1 compared to generated tree 4
  Robinson-Foulds distance: 13
Original tree 2 compared to generated tree 0
  Robinson-Foulds distance: 17
Original tree 2 compared to generated tree 1
  Robinson-Foulds distance: 17
Original tree 2 compared to generated tree 2
  Robinson-Foulds distance: 15
Original tree 2 compared to generated tree 3
  Robinson-Foulds distance: 15
Original tree 2 compared to generated tree 4
  Robinson-Foulds distance: 16
Original tree 3 compared to generated tree 0
  Robinson-Foulds distance: 14
Original tree 3 compared to generated tree 1
  Robinson-Foulds distance: 14
Original tree 3 compared to generated tree 2
  Robinson-Foulds distance: 16
Original tree 3 compared to generated tree 3
  Robinson-Foulds distance: 18
Original tree 3 compared to generated tree 4
  Robinson-Foulds distance: 15
Original tree 4 compared to generated tree 0
  Robinson-Foulds distance: 17
Original tree 4 compared to generated tree 1
  Robinson-Foulds distance: 15
Original tree 4 compared to generated tree 2
  Robinson-Foulds distance: 13
Original tree 4 compared to generated tree 3
  Robinson-Foulds distance: 17
Original tree 4 compared to generated tree 4
  Robinson-Foulds distance: 18
```

Figure 14: Distances 20 - 100 - 5

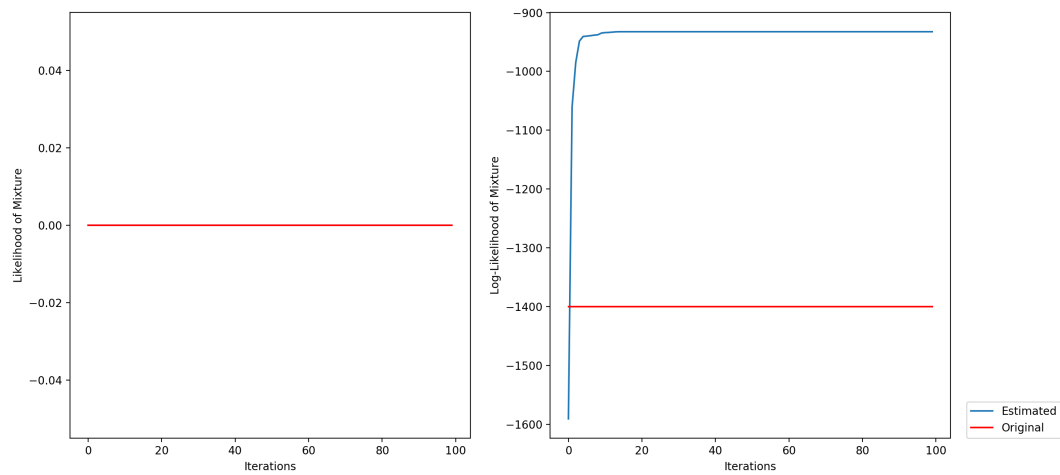


Figure 15: Likelihood and log-likelihood 20 - 100 - 5

As it can be seen, in the first case, distances do not vary a lot according to those which use 10 nodes in the previous section. Nevertheless, likelihood and log-likelihood are quite different. As only 2 observations are present in the configuration, the sequence is more probable to occur, returning high values in the probability of the inferred trees.

Totally the opposite happens in the second case. Distances, as related to the number of nodes, are similar to those of the previous section in which 20 nodes were employed. According to likelihood and log-likelihood, as a huge sequence of 100 observation was given, the probability of the sequence is much smaller, being practically zero in both cases, as it is shown in the plots.

6 Super epicentra – EM

Question 2.6.18

The following graphical model and data are given:

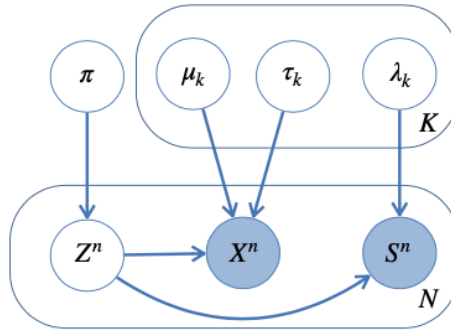


Figure 16: The K super epicentra model with priors

- Each super epicentra is modeled by a 2-dimensional Gaussian determining the location and a Poisson determining the strength.
- The entire model is a mixture of K components.
- The variable Z^n is a class variable that follows a categorical distribution π .
- X^n is modeled by a Gaussian distribution sampled from $\mu_k = (\mu_{k,1}, \mu_{k,2})$ and $\tau_k = (\tau_{k,1}, \tau_{k,2})$.
- S^n is modeled by a Poisson distribution sampled from λ_k .

As Z^n is not observed, the EM algorithm can be applied to find the maximum likelihood solution for the model. To do so, the following expression can be applied:

$$\log(p(X, S|\theta)) = \log \left\{ \sum_Z p(X, S, Z|\theta) \right\}$$

Summing over all values of Z , it is possible to get the probability of X and S happening. Nevertheless, Z is not an observed variable, so it is necessary to employ the expected value $\mathbb{E}_Z[\log(p(X, S, Z|\theta))]$ to perform this calculation. Employing the initial equation, the following is obtained:

$$\begin{aligned} p(X, S, Z|\theta) &= p(X|S, Z, \theta) p(S|Z, \theta) p(Z|\theta) = \\ &= p(X|S, Z, M, T) p(S|Z, \Lambda) p(Z|\Pi) \end{aligned}$$

According to the provided data, this expression can be rewritten using $p(Z = 1|\Pi)$, since Z

follows a categorical distribution, and when $Z = 0$, all multiplied by it will be equal to 0.

$$\begin{aligned} Z &\sim \text{Cat}(\pi_k) = \prod_{n=1}^N \prod_{k=1}^K \pi_k^{Z_{n,k}} \\ p(S|Z=1, \Lambda) &= \prod_{n=1}^N \prod_{k=1}^K \text{Poisson}(S_n|\lambda_k)^{Z_{n,k}} \\ p(X|Z=1, M, T) &= \prod_{n=1}^N \prod_{k=1}^K \mathcal{N}(X_n|\boldsymbol{\mu}_k, (\boldsymbol{\tau}_k)^{-1}I)^{Z_{n,k}} \end{aligned}$$

Using the obtained results, it is possible to get the following:

$$\begin{aligned} \log(p(X, S, Z|\theta)) &= \log(p(X|S, Z, M, T) p(S|Z, \Lambda) p(Z|\Pi)) \\ &= \log \left(\prod_{n=1}^N \prod_{k=1}^K (\mathcal{N}(X_n|\boldsymbol{\mu}_k, (\boldsymbol{\tau}_k)^{-1}I) \text{Poisson}(S_n|\lambda_k) \pi_k)^{Z_{n,k}} \right) = \\ &= \sum_{n=1}^N \sum_{k=1}^K Z_{n,k} \left(\log(\mathcal{N}(X_n|\boldsymbol{\mu}_k, (\boldsymbol{\tau}_k)^{-1}I)) + \log(\text{Poisson}(S_n|\lambda_k)) + \log(\pi_k) \right) \end{aligned}$$

Applying this to the expected value, the following is gotten:

$$\begin{aligned} \mathbb{E}_Z[\log(p(X, S, Z|\theta))] &= \mathbb{E}_Z \left[\sum_{n=1}^N \sum_{k=1}^K Z_{n,k} \left(\log(\mathcal{N}(X_n|\boldsymbol{\mu}_k, (\boldsymbol{\tau}_k)^{-1}I)) + \log(\text{Poisson}(S_n|\lambda_k)) + \log(\pi_k) \right) \right] = \\ &= \sum_{n=1}^N \sum_{k=1}^K \mathbb{E}_Z[Z_{n,k}] \left(\log(\mathcal{N}(X_n|\boldsymbol{\mu}_k, (\boldsymbol{\tau}_k)^{-1}I)) + \log(\text{Poisson}(S_n|\lambda_k)) + \log(\pi_k) \right) \end{aligned}$$

Keeping expanding each term, they can be transformed into:

$$\begin{aligned} \log(\mathcal{N}(X_n|\boldsymbol{\mu}_k, (\boldsymbol{\tau}_k)^{-1}I)) &= \log \left(\frac{1}{\sqrt{(2\pi)^K |\boldsymbol{\tau}_k^{-1}|}} \exp \left[\frac{1}{2} (X_n - \boldsymbol{\mu}_k)^T (\boldsymbol{\tau}_k^{-1}I)^{-1} (X_n - \boldsymbol{\mu}_k) \right] \right) = \\ &= -\frac{1}{2} (K(\log(2\pi)) + \log(|\boldsymbol{\tau}_k^{-1}|)) + \left[\frac{1}{2} (X_n - \boldsymbol{\mu}_k)^T (\boldsymbol{\tau}_k^{-1}I)^{-1} (X_n - \boldsymbol{\mu}_k) \right] \\ \log(\text{Poisson}(S_n|\lambda_k)) &= \log \left(\frac{\lambda_k^{S_n} e^{-\lambda_k}}{S_n!} \right) = S_n \log(\lambda_k) - \lambda_k - \log(S_n!) \end{aligned}$$

From here, it is possible to obtain (Expectation step):

$$\gamma(Z_{n,k}) = \frac{\mathcal{N}(X_n|\boldsymbol{\mu}_k, (\boldsymbol{\tau}_k)^{-1}I) \text{Poisson}(S_n|\lambda_k) \pi_k}{\sum_{k=1}^K \mathcal{N}(X_n|\boldsymbol{\mu}_k, (\boldsymbol{\tau}_k)^{-1}I) \text{Poisson}(S_n|\lambda_k) \pi_k}$$

And applying this function $\gamma(Z_{n,k})$, the new parameters can be estimated (Maximization step):

$$\begin{aligned}\boldsymbol{\mu}_k^{new} &= \frac{1}{N_k} \sum_{n=1}^N \gamma(Z_{n,k}) X_n \\ \boldsymbol{\tau}_k^{new} &= \frac{1}{N_k} \sum_{n=1}^N \gamma(Z_{n,k}) (X_n - \boldsymbol{\mu}_k)^T (X_n - \boldsymbol{\mu}_k) \\ \pi_k^{new} &= \frac{N_k}{N} \\ \lambda_k^{new} &= \frac{1}{N_k} \sum_{n=1}^N \gamma(Z_{n,k}) S_n\end{aligned}$$

Knowing that:

$$N_k = \sum_{n=1}^N \gamma(Z_{n,k})$$

This procedure is repeated iteratively until either the estimated parameters or the log-likelihood converges.

Question 2.6.19

See appendix C.

Question 2.6.20

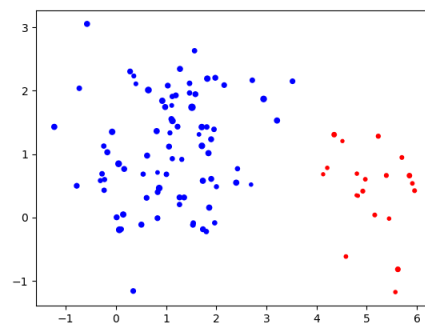


Figure 17: Provided data with 2 components

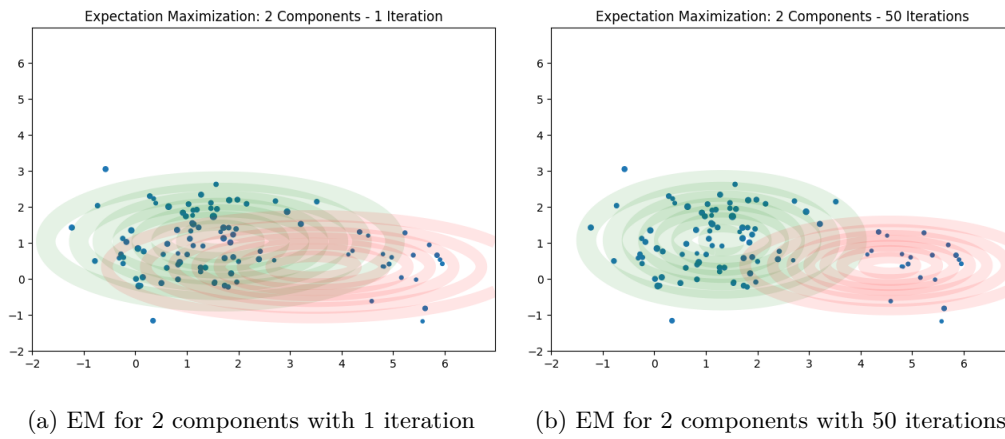


Figure 18: EM algorithm for provided data with 2 components

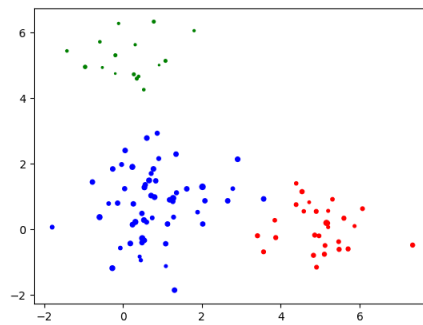


Figure 19: Provided data with 3 components

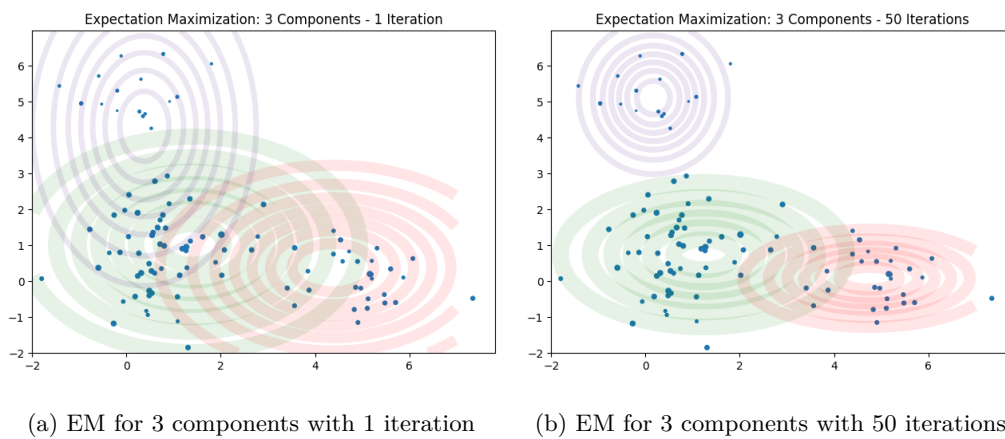


Figure 20: EM algorithm for provided data with 3 components

7 Super epicentra – VI

Question 2.7.21

The following graphical model and data are given:

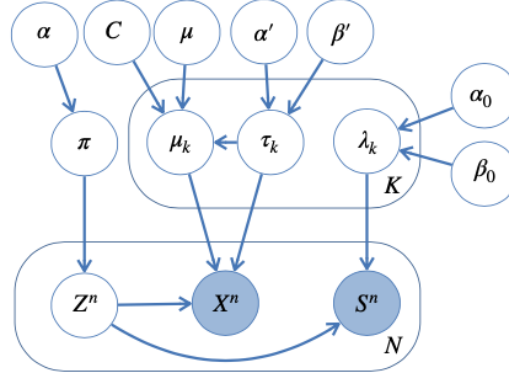


Figure 21: The K super epicentra model with priors

- π has a $\text{Dir}(\alpha)$ prior.
- $\tau_{k,i}$ has a $\text{Ga}(\alpha', \beta')$ prior.
- $\mu_{k,i}$ has a $\mathcal{N}(\mu, (C\tau_{k,i})^{-1})$ prior.
- λ_k has a $\text{Ga}(\alpha_0, \beta_0)$ prior.
- Z^n is modeled by a categorical distribution.
- X^n is modeled by a bi-dimensional normal distribution.
- S^n is modeled by a Poisson distribution.

From here, it can be said that:

$$\begin{aligned} \pi &\rightarrow \prod_{k=1}^K \pi_k^{\alpha_k - 1} \\ \tau_{k,i} &\rightarrow \frac{(\beta')^{\alpha'}}{\Gamma(\alpha')} \tau_{k,i}^{[\alpha' - 1]} e^{[-\beta' \tau_{k,i}]} \\ \mu_{k,i} &\rightarrow \sqrt{\frac{C\tau_{k,i}}{2\pi}} e^{[-\frac{C\tau_{k,i}}{2} (\mu_{k,i} - \mu)^2]} \\ \lambda_k &\rightarrow \frac{(\beta_0)^{\alpha_0}}{\Gamma(\alpha_0)} \lambda_k^{[\alpha_0 - 1]} e^{[-\beta_0 \lambda_k]} \end{aligned}$$

The posterior distribution for this graphical model can be obtained by:

$$p(Z, \theta | X, S) = p(Z, \Pi, T, M, \Lambda | X, S)$$

$$\text{Being } \theta = \pi, \underbrace{\tau_{k,i}}_k, \underbrace{\mu_{k,i}}_k, \lambda_k$$

Now, considering variational inference:

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

$$\log(q^*(Z)) = \mathbb{E}_{\Pi, T, M, \Lambda} [\log(\underbrace{p(\Pi, T, M, \Lambda, X, S, Z)}_{\text{Joint distribution}})] + \text{const}$$

And now, checking independence by means of the provided graphical model:

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

Focusing on the first term:

$$\begin{aligned} p(X, S, Z|M, T, \Lambda, \Pi) &= \prod_{n=1}^N p(X_n|M, T) p(S_n|\Lambda) \underbrace{p(Z_n|\Pi)}_{\text{Categorical}} \\ &= \prod_{n=1}^N \prod_{k=1}^K (p(X_n|\mu_k, \tau_k) p(S_n|\lambda_k) p(Z_{n,k} = 1|\pi_k))^{Z_{n,k}} \end{aligned}$$

From here, it can be substituted in the expected value expression derived before:

$$\log(q^*(Z)) = \mathbb{E}_{\theta} \left[\log \left(\prod_{n=1}^N \prod_{k=1}^K (p(X_n|\mu_k, \tau_k) p(S_n|\lambda_k) p(Z_{n,k} = 1|\pi_k))^{Z_{n,k}} p(M, T, \Lambda, \Pi) \right) \right] + \text{const}$$

The term $p(M, T, \Lambda, \Pi)$ does not depend on Z , so it can be included into the constant. Now, splitting the equation by means of logarithms, it is obtained:

$$\begin{aligned} &\mathbb{E}_{\theta} \left[\log \left(\prod_{n=1}^N \prod_{k=1}^K p(X_n|\mu_k, \tau_k)^{Z_{n,k}} \right) + \log \left(\prod_{n=1}^N \prod_{k=1}^K p(S_n|\lambda_k)^{Z_{n,k}} \right) \right. \\ &\quad \left. + \log \left(\prod_{n=1}^N \prod_{k=1}^K p(Z_{n,k} = 1|\pi_k)^{Z_{n,k}} \right) \right] + \text{const} \end{aligned}$$

Now, applying some properties:

$$\mathbb{E}_{\theta} \left[\sum_{n=1}^N \sum_{k=1}^K Z_{n,k} (\log(p(X_n|\mu_k, \tau_k)) + \log(p(S_n|\lambda_k)) + \log(p(Z_{n,k} = 1|\pi_k))) \right] + \text{const}$$

Knowing that $X \sim \mathcal{N}(\mu_k | (\tau_k)^{-1})$, $S \sim \text{Poisson}(\lambda_k)$ and $Z \sim \text{Cat}(\pi_k)$, the expression terms can be transformed into:

$$\begin{aligned} \log(\mathcal{N}(X_n | \mu_k, (\tau_k)^{-1} I)) &= \log \left(\frac{1}{\sqrt{(2\pi)^K |\tau_k^{-1}|}} \exp \left[\frac{1}{2} (X_n - \mu_k)^T (\tau_k^{-1} I)^{-1} (X_n - \mu_k) \right] \right) = \\ &= -\frac{1}{2} (K(\log(2\pi)) + \log(|\tau_k^{-1}|)) + \frac{1}{2} (X_n - \mu_k)^T (\tau_k^{-1} I)^{-1} (X_n - \mu_k) \\ \log(p(S_n | \lambda_k)) &= \log \left(\frac{\lambda_k^{S_n} e^{-\lambda_k}}{S_n!} \right) = S_n \log(\lambda_k) - \lambda_k - \log(S_n!) \\ \log(p(Z_{n,k} = 1 | \pi_k)) &= \log(\pi_k) \end{aligned}$$

Substituting in the expression, it takes the form of:

$$\begin{aligned} \log(q^*(Z)) &= \mathbb{E}_\theta \left[\sum_{n=1}^N \sum_{k=1}^K Z_{n,k} \left(-\frac{1}{2} (K(\log(2\pi)) + \log(|\tau_k^{-1}|)) + \frac{1}{2} (X_n - \mu_k)^T (\tau_k^{-1} I)^{-1} (X_n - \mu_k) + \right. \right. \\ &\quad \left. \left. + S_n \log(\lambda_k) - \lambda_k - \log(S_n!) + \log(\pi_k) \right) \right] + \text{const} \\ \log(q^*(Z)) &= \sum_{n=1}^N \sum_{k=1}^K Z_{n,k} \left(-\frac{1}{2} (K(\log(2\pi)) + \mathbb{E}_\tau[\log(|\tau_k^{-1}|)]) + \mathbb{E}_{\mu,\tau} \left[\frac{1}{2} (X_n - \mu_k)^T (\tau_k^{-1} I)^{-1} (X_n - \mu_k) \right] + \right. \\ &\quad \left. + S_n \mathbb{E}_\lambda[\log(\lambda_k)] - \mathbb{E}_\lambda[\lambda_k] - \log(S_n!) + \mathbb{E}_\pi[\log(\pi_k)] \right) + \text{const} \end{aligned}$$

This equation can be written as a function of ρ , where $\log(\rho)$ will be equivalent to the terms inside the parenthesis:

$$\begin{aligned} \log(q^*(Z)) &= \sum_{n=1}^N \sum_{k=1}^K Z_{n,k} \log(\rho_{n,k}) + \text{const} = \sum_{n=1}^N \sum_{k=1}^K \log(\rho_{n,k}^{Z_{n,k}}) + \text{const} = \\ &= \log \left(\prod_{n=1}^N \prod_{k=1}^K \rho_{n,k}^{Z_{n,k}} \right) + \text{const} \\ q^*(Z) &= \prod_{n=1}^N \prod_{k=1}^K \rho_{n,k}^{Z_{n,k}} + \text{const} \propto \prod_{n=1}^N \prod_{k=1}^K \rho_{n,k}^{Z_{n,k}} \end{aligned}$$

Even so, it is necessary to normalize this distribution, since for each value of n , the quantities $Z_{n,k}$ sum to 1 for all values of k , obtaining:

$$q^*(Z) = \prod_{n=1}^N \prod_{k=1}^K r_{n,k}^{Z_{n,k}} \quad \text{where} \quad r_{n,k} = \frac{\rho_{n,k}}{\sum_{k=1}^K \rho_{n,k}}$$

Now, it is necessary to deal with $q^*(M, T, \Lambda, \Pi)$. It is possible to perform a similar derivation than before, just changing the expected value to be over Z instead of being over the parameters:

$$\log(q^*(M, T, \Lambda, \Pi)) = \mathbb{E}_Z[\log(p(\Pi, T, M, \Lambda, X, S, Z))] + \text{const}$$

From this expression, as before, it is obtained:

$$p(\Pi, T, M, \Lambda, X, S, Z) = p(X, S, Z|M, T, \Lambda, \Pi) p(M, T, \Lambda, \Pi)$$

Nevertheless, in this case, the second term will not be a constant and has to be derived as well:

$$\begin{aligned} \log(q^*(M, T, \Lambda, \Pi)) &= \mathbb{E}_Z \left[\sum_{n=1}^N \sum_{k=1}^K Z_{n,k} (\log(p(X_n|\boldsymbol{\mu}_k, \boldsymbol{\tau}_k)) + \log(p(S_n|\lambda_k)) + \log(p(Z_{n,k} = 1|\pi_k))) \right] + \\ &+ \mathbb{E}_Z [\log(p(M, T, \Lambda, \Pi))] + \text{const} = \\ &= \mathbb{E}_Z \left[\sum_{n=1}^N \sum_{k=1}^K Z_{n,k} \left(-\frac{1}{2} (K(\log(2\pi)) + \log(|\boldsymbol{\tau}_k^{-1}|)) + \frac{1}{2} (X_n - \boldsymbol{\mu}_k)^T (\boldsymbol{\tau}_k^{-1} I)^{-1} \right. \right. \\ &\quad \left. \left. (X_n - \boldsymbol{\mu}_k) + S_n \log(\lambda_k) - \lambda_k - \log(S_n!) + \log(\pi_k) \right) \right] + \mathbb{E}_Z [\log(p(M, T, \Lambda, \Pi))] + \text{const} \end{aligned}$$

Now, focusing on the second expected value term:

$$\mathbb{E}_Z [\log(p(M, T, \Lambda, \Pi))] = \mathbb{E}_Z [\log(p(M|T) p(T) p(\Lambda) p(\Pi))]$$

As none of these probabilities depends on Z , it is possible to substitute by their correspondent density functions:

$$\begin{aligned} \mathbb{E}_Z [\log(p(M, T, \Lambda, \Pi))] &= \log(p(M|T) p(T) p(\Lambda) p(\Pi)) = \\ &= \sum_{k=1}^K -\frac{1}{2} \log(2\pi(C\boldsymbol{\tau}_k)^{-1}) + \frac{(\boldsymbol{\mu}_k - \mu)^2}{2(C\boldsymbol{\tau}_k)^{-1}} + \alpha' \log(\beta') - \log(\Gamma(\alpha')) + \\ &+ (\alpha' - 1) \log(\boldsymbol{\tau}_k) - \beta' \boldsymbol{\tau}_k + (\alpha_k - 1) \log(\pi_k) + \alpha_0 \log(\beta_0) - \log(\Gamma(\alpha_0)) + \\ &+ (\alpha_0 - 1) \log(\lambda_k) - \beta_0 \lambda_k \end{aligned}$$

This is added to the previous equation, obtaining:

$$\begin{aligned} \log(q^*(M, T, \Lambda, \Pi)) &= \sum_{n=1}^N \sum_{k=1}^K \mathbb{E}_Z [Z_{n,k}] \left(-\frac{1}{2} (K(\log(2\pi)) + \log(|\boldsymbol{\tau}_k^{-1}|)) + \frac{1}{2} (X_n - \boldsymbol{\mu}_k)^T (\boldsymbol{\tau}_k^{-1} I)^{-1} \right. \\ &\quad \left. (X_n - \boldsymbol{\mu}_k) + S_n \log(\lambda_k) - \lambda_k - \log(S_n!) + \log(\pi_k) \right) + \sum_{k=1}^K -\frac{1}{2} \log(2\pi(C\boldsymbol{\tau}_k)^{-1}) + \\ &+ \frac{(\boldsymbol{\mu}_k - \mu)^2}{2(C\boldsymbol{\tau}_k)^{-1}} + \alpha' \log(\beta') - \log(\Gamma(\alpha')) + (\alpha' - 1) \log(\boldsymbol{\tau}_k) - \beta' \boldsymbol{\tau}_k + (\alpha_k - 1) \log(\pi_k) + \\ &+ \alpha_0 \log(\beta_0) - \log(\Gamma(\alpha_0)) + (\alpha_0 - 1) \log(\lambda_k) - \beta_0 \lambda_k + \text{const} \end{aligned}$$

Which is transformed into the following after removing constants:

$$\begin{aligned} \log(q^*(M, T, \Lambda, \Pi)) &= \sum_{n=1}^N \sum_{k=1}^K \mathbb{E}_Z[Z_{n,k}] \left(-\frac{1}{2} \log(|\boldsymbol{\tau}_k^{-1}|) + \frac{1}{2} (X_n - \boldsymbol{\mu}_k)^T (\boldsymbol{\tau}_k^{-1} I)^{-1} (X_n - \boldsymbol{\mu}_k) + \right. \\ &\quad \left. + S_n \log(\lambda_k) - \lambda_k - \log(S_n!) + \log(\pi_k) \right) + \sum_{k=1}^K -\frac{1}{2} \log((C\boldsymbol{\tau}_k)^{-1}) + \frac{(\boldsymbol{\mu}_k - \boldsymbol{\mu})^2}{2(C\boldsymbol{\tau}_k)^{-1}} + \\ &\quad + (\alpha' - 1) \log(\boldsymbol{\tau}_k) - \beta' \boldsymbol{\tau}_k + (\alpha_0 - 1) \log(\lambda_k) - \beta_0 \lambda_k + (\alpha_k - 1) \log(\pi_k) + \text{const} \end{aligned}$$

And now, applying:

$$q(M, T, \Lambda, \Pi) = q(\Pi) \prod_{k=1}^K q(M, T, \Lambda)$$

From the terms of $\log(q^*(M, T, \Lambda, \Pi))$ which depend on Π :

$$\begin{aligned} \log(q(\Pi)) &= \sum_{k=1}^K (\alpha_k - 1) \log(\pi_k) + \sum_{n=1}^N \sum_{k=1}^K r_{n,k} \log(\pi_k) \\ q(\Pi) &= \prod_{k=1}^K \pi_k^{(\alpha_k - 1)} + \prod_{n=1}^N \prod_{k=1}^K \pi_k^{r_{n,k}} \end{aligned}$$

The same procedure is also applied to $\prod_{k=1}^K q(M, T, \Lambda)$:

$$\begin{aligned} \prod_{k=1}^K q(T, M, \Lambda) &= \prod_{k=1}^K q(M|T) q(T) q(\Lambda) \\ q(M|T) q(T) &= \prod_{n=1}^N \prod_{k=1}^K \left(\frac{-1}{\sqrt{\tau_k}} + e^{-\frac{(X_n - \boldsymbol{\mu}_k)^2}{2\tau_k}} \right)^{\mathbb{E}_Z[Z_{n,k}]} + \prod_{k=1}^K \frac{-1}{\sqrt{(C\tau_k)^{-1}}} e^{\frac{-(\boldsymbol{\mu}_k - \boldsymbol{\mu})^2}{2(C\tau_k)^{-1}}} + \tau_k^{(\alpha' - 1)} e^{(-\beta' \tau_k)} \\ q(\Lambda) &= \prod_{n=1}^N \prod_{k=1}^K \left(\frac{\lambda_k^{S_n}}{\lambda_k} \right)^{\mathbb{E}_Z[Z_{n,k}]} + \lambda_k^{\alpha_0 - 1} e^{-\beta_0 \lambda_k} \end{aligned}$$

From this point, with the obtained expressions, it is possible to compute the approximation knowing that:

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

Therefore, the final expression is equal to:

$$\begin{aligned} q(Z, \Pi, T, M, \Lambda) &= \left(\prod_{n=1}^N \prod_{k=1}^K r_{n,k}^{Z_{n,k}} \right) \left(\prod_{k=1}^K \pi_k^{(\alpha_k - 1)} + \prod_{n=1}^N \prod_{k=1}^K \pi_k^{r_{n,k}} \right) \left(\prod_{n=1}^N \prod_{k=1}^K \left(\frac{-1}{\sqrt{\tau_k}} + e^{-\frac{(X_n - \boldsymbol{\mu}_k)^2}{2\tau_k}} \right)^{\mathbb{E}_Z[Z_{n,k}]} + \right. \\ &\quad \left. + \prod_{k=1}^K \frac{-1}{\sqrt{(C\tau_k)^{-1}}} e^{\frac{-(\boldsymbol{\mu}_k - \boldsymbol{\mu})^2}{2(C\tau_k)^{-1}}} + \tau_k^{(\alpha' - 1)} e^{(-\beta' \tau_k)} \right) \left(\prod_{n=1}^N \prod_{k=1}^K \left(\frac{\lambda_k^{S_n}}{\lambda_k} \right)^{\mathbb{E}_Z[Z_{n,k}]} + \lambda_k^{\alpha_0 - 1} e^{-\beta_0 \lambda_k} \right) \end{aligned}$$

8 Sampling from a tree GM

Question 2.8.22

Question 2.8.23

Question 2.8.24

Question 2.8.25

9 Failing components VI

A Simple VI

Exercise 2.4

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 import scipy.stats as stats
4 from tqdm import tqdm
5
6 # This function computes the value of tau_N
7 def compute_tauN(tau_0, mu_0, vector, a_N, b_N):
8
9     result = (tau_0 + len(vector)) * (a_N / b_N)
10    return result
11
12 # This function computes the values of b_N
13 def compute_bN(a_0, b_0, vector, mu_0, tau_N):
14
15     term1 = (1 / tau_N) + pow(np.mean(vector),2) + pow(mu_0,2) - 2 * np.mean(vector) * mu_0
16     term2 = 0
17     for x in vector:
18         term2 += pow(x,2) + (1 / tau_N) + pow(np.mean(vector),2) - 2 * np.mean(vector) * x
19     result = b_0 + 0.5 * tau_0 * term1 + 0.5 * term2
20
21     return result
22
23 #This function computes the approximated distribution
24 def getQ(mu_axis, tau_axis, mu_N, tau_N, a_N, b_N):
25
26     q_mu_tau = np.zeros((len(mu_axis), len(tau_axis)))
27
28     for i in tqdm(range(len(mu_axis))):
29         for j in range(len(tau_axis)):
30             q_mu = stats.norm(mu_N, 1/(tau_N)).pdf(mu_axis[i])
31             q_tau = stats.gamma.pdf(tau_axis[j], a_N, loc=0, scale=(1/b_N))
32             q_mu_tau[j][i] = q_mu * q_tau
33
34     return q_mu_tau
35
36 #This function computes the original distribution
37 def getP(mu_axis, tau_axis, mu_N, tau_N, a_N, b_N):
38
39     p_mu_tau = np.zeros((len(mu_axis), len(tau_axis)))
40
41     for i in tqdm(range(len(mu_axis))):

```

```

42     for j in range(len(tau_axis)):
43         if(tau_axis[j] == 0):
44             tau_axis[j] = 0.00001 # Correction for 0 denominator
45             p_mu = stats.norm(mu_N, 1/(tau_N * tau_axis[j])).pdf(mu_axis[i])
46             p_tau = stats.gamma.pdf(tau_axis[j], a_N, loc=0, scale=(1/b_N))
47             p_mu_tau[j][i] = p_mu * p_tau
48
49     return p_mu_tau
50
51 if __name__ == "__main__":
52
53     # VARIABLES
54     iterations = 10                # Number of iterations
55     n = 10                        # Number of samples
56     mu_0 = 0                      # Initial value mu_0
57     mu_N = 1                      # Initial value mu_N
58     tau_0 = 1                     # Initial value tau_0
59     tau_N = 1                     # Initial value mu_N
60     a_0 = 0                       # Initial value a_0
61     a_N = 1                       # Initial value a_N
62     b_0 = 1                       # Initial value b_0
63     b_N = 1                       # Initial value b_N
64     mu_axis = np.linspace(-1,1,100) # X axis grid
65     tau_axis = np.linspace(0,2,100) # Y axis grid
66
67     # SAMPLES GENERATION
68     vector = np.random.normal(0,1,n)
69
70     mu_N = (tau_0 * mu_0 + np.sum(vector)) / (tau_0 + len(vector)) # mu_N value
71     a_N = a_0 + (len(vector)) / 2                                # a_N value
72     computation (it won't update)
73
74     q_mu_tau = np.zeros((len(mu_axis), len(tau_axis)))
75     p_mu_tau = np.zeros((len(mu_axis), len(tau_axis)))
76
77     for i in range(iterations):
78         tau_N = compute_tauN(tau_0, mu_0, vector, a_N, b_N)
79         b_N = compute_bN(a_0, b_0, vector, mu_N, tau_N)
80
81     p_mu_tau = getP(mu_axis, tau_axis, mu_N, tau_N, a_N, b_N)
82     q_mu_tau = getQ(mu_axis, tau_axis, mu_N, tau_N, a_N, b_N)
83     #plt.axis([-0.2,0.2, 0, 2]) # Uncomment
84     this line to modify the grid area
85     plt.contour(mu_axis, tau_axis, q_mu_tau, colors='red')
86     plt.contour(mu_axis, tau_axis, p_mu_tau, colors='green')
87     plt.show()

```

B Mixture of trees with observable variables

Exercise 2.5

```

1 import argparse
2 import numpy as np
3 import matplotlib.pyplot as plt
4 import sys
5 import networkx as nx
6 import dendropy
7 import Kruskal_v1 as kruskal
8 from Tree import Tree, TreeMixture
9 from tqdm import tqdm
10
11 def save_results(loglikelihood, topology_array, theta_array, filename):
12     """ This function saves the log-likelihood vs iteration values,
13         the final tree structure and theta array to corresponding numpy arrays. """
14
15     likelihood_filename = filename + "_em_loglikelihood.npy"
16     topology_array_filename = filename + "_em_topology.npy"
17     theta_array_filename = filename + "_em_theta.npy"
18     print("Saving log-likelihood to ", likelihood_filename, ", topology_array to: ",
19           , topology_array_filename,
20           ", theta_array to: ", theta_array_filename, "...")
21     np.save(likelihood_filename, loglikelihood)
22     np.save(topology_array_filename, topology_array)
23     np.save(theta_array_filename, theta_array)
24
25 def computeLikelihood(samples, topology, theta):
26     result = theta[0][samples[0]]
27     for i in range(1, len(topology)):
28         result *= theta[i][samples[int(topology[i])]][samples[i]]
29
30     return (result + sys.float_info.epsilon)
31
32 def computeLogLikelihood(pi, likelihood):
33     result = 0
34     for i in range(likelihood.shape[0]):
35         aux = 0
36         for k in range(likelihood.shape[1]):
37             aux += pi[k] * likelihood[i,k]
38         result += np.log(aux)
39
40     return result
41
42 def computeCondQ(responsibility, samples, node1, node2, value1, value2):
43     num = 0

```

```

43     denom = 0
44     for i in range(samples.shape[0]):
45         if (samples[i,node1] == value1):
46             denom += responsibility[i]
47         if (samples[i,node2] == value2):
48             num += responsibility[i]
49
50     return (num / (denom + sys.float_info.epsilon))
51
52 def computeQ(responsibility, samples, node, val):
53     num = 0
54     for i in range(samples.shape[0]):
55         if (samples[i,node] == val):
56             num += responsibility[i]
57     denom = np.sum(responsibility) + sys.float_info.epsilon
58
59     return (num / denom)
60
61 def computeQJoint(responsibility, samples, node1, node2, val1, val2):
62     num = 0
63     for i in range(samples.shape[0]):
64         if (samples[i,node1] == val1) and (samples[i,node2] == val2):
65             num += responsibility[i]
66     denom = np.sum(responsibility) + sys.float_info.epsilon
67
68     return (num / denom)
69
70 def computeResponsibility(num_clusters, samples, pi, likelihood):
71     result = np.zeros((samples.shape[0], num_clusters))
72     for i in range(samples.shape[0]):
73         for k in range(num_clusters):
74             result[i,k] = pi[k] * likelihood[i,k]
75             result[i] = (result[i] + sys.float_info.epsilon) / (np.sum(result[i]) +
76 num_clusters * sys.float_info.epsilon)
77
78     return result
79
80 def calculateI(responsibility, samples, node1, node2):
81     result = 0
82     for i in range(2):
83         for j in range(2):
84             q_node1 = computeQ(responsibility, samples, node1, i)
85             q_node2 = computeQ(responsibility, samples, node2, j)
86             q_joint = computeQJoint(responsibility, samples, node1, node2, i, j)
87             if (q_joint != 0):
88                 if (q_node1 == 0):
89                     q_node1 = sys.float_info.epsilon

```

```

89         if (q_node2 == 0):
90             q_node2 = sys.float_info.epsilon
91         if ((q_node1 * q_node2) == 0):
92             result += q_joint * np.log(q_joint / (sys.float_info.min))
93         else:
94             result += q_joint * np.log(q_joint / (q_node1 * q_node2))
95
96     return result
97
98 def computeTheta(theta_list, responsibility, samples, topology, num_nodes,
99                  num_clusters):
100     result = theta_list
101     for k in range(num_clusters):
102         for i in range(num_nodes):
103             if (i == 0):
104                 result[k][0,0] = computeQ(responsibility[:,k], samples, 0, 0)
105                 result[k][0,1] = computeQ(responsibility[:,k], samples, 0, 1)
106             else:
107                 result[k][i,0][0] = computeCondQ(responsibility[:,k], samples, int(
108 topology[k][i]), i, 0, 0)
109                 result[k][i,0][1] = computeCondQ(responsibility[:,k], samples, int(
110 topology[k][i]), i, 0, 1)
111                 result[k][i,1][0] = computeCondQ(responsibility[:,k], samples, int(
112 topology[k][i]), i, 1, 0)
113                 result[k][i,1][1] = computeCondQ(responsibility[:,k], samples, int(
114 topology[k][i]), i, 1, 1)
115
116     return result
117
118 def computationsEM(iterations, samples, num_clusters, tm, topology_list, theta_list
119 ):
120     pi = tm.pi
121     loglikelihood = np.zeros(iterations)
122
123     for it in range(iterations):
124         num_samples = samples.shape[0]
125         num_nodes = samples.shape[1]
126         likelihood = np.zeros((num_samples, num_clusters)) # Probability of having
127 this sample per tree
128
129         # Compute likelihood per sample
130         for i in range(num_samples):
131             for k in range(num_clusters):
132                 likelihood[i,k] = computeLikelihood(samples[i,:], topology_list[k],
133 theta_list[k])
134
135         # Computation of responsibilities

```

```

128     responsibility = computeResponsibility(num_clusters, samples, pi,
129                                           likelihood)
130
131     # Computation of pi'
132     res_sum = np.sum(responsibility, axis=0)
133     total_sum = np.sum(res_sum)
134     pi = np.zeros(len(res_sum))
135     for i in range(len(res_sum)):
136         pi[i] = res_sum[i] / total_sum
137
138     # Get the IQ for using as weights
139     IQ = np.zeros((num_nodes, num_nodes, num_clusters))
140     for k in range(num_clusters):
141         for i in range(len(topology_list[k])):
142             for j in range(len(topology_list[k])):
143                 if (i != j):
144                     IQ[i,j,k] = calculateI(responsibility[:,k], samples, i, j)
145
146     # Create the graphs
147     graphs = list()
148     for k in range(num_clusters):
149         graphs.append(kruskal.Graph(num_nodes))
150         for i in range(num_nodes):
151             for j in range(i+1, num_nodes):
152                 graphs[-1].addEdge(i, j, IQ[i,j,k])
153
154     # Get the Maximum Spanning Tree from each graph
155     tree = np.zeros((num_nodes-1, 3, num_clusters))
156     for k in range(num_clusters):
157         result = graphs[k].maximum_spanning_tree()
158         cnt = 0
159         for u_aux, v_aux, weight_aux in result:
160             tree[cnt,0,k] = u_aux
161             tree[cnt,1,k] = v_aux
162             tree[cnt,2,k] = weight_aux
163             cnt += 1
164
165     # Creation of the tree
166     topology_list = list()
167     for k in range(num_clusters):
168         topology_list.append(np.zeros(num_nodes))
169         topology_list[-1][0] = np.nan
170         max_tree = nx.Graph()
171         for i in range(tree.shape[0]):
172             max_tree.add_edge(tree[i,0,k], tree[i,1,k])
173         finaltree = list(nx.bfs_edges(max_tree, 0))
174         for i in range(num_nodes - 1):

```

```

174         topology_list[-1][int(finaltree[i][1])] = finaltree[i][0]
175
176         # Computation of theta'
177         theta_list = computeTheta(theta_list, responsibility, samples,
178                                   topology_list, num_nodes, num_clusters)
179
180         loglikelihood[it] = computeLogLikelihood(pi, likelihood)
181
182     return loglikelihood
183
184 def em_algorithm(seed_val, samples, num_clusters, max_num_iter):
185
186     # Initialize the needed variables
187     sieving = 100
188     max_log = float("-inf")
189     best_seed = 0
190
191     # Get the best seed for likelihood
192     for siev in tqdm(range(sieving)):
193         # Set the seed
194         aux_seed = seed_val + siev # Try with all seeds from @param:seed_val to
195         @param:seed_val + sieving
196         np.random.seed(aux_seed)
197
198         # Generate tree mixture
199         tm = TreeMixture(num_clusters=num_clusters, num_nodes=samples.shape[1])
200         tm.simulate_pi(seed_val=aux_seed)
201         tm.simulate_trees(seed_val=aux_seed)
202         topology_list = []
203         theta_list = []
204
205         for i in range(num_clusters):
206             topology_list.append(tm.clusters[i].get_topology_array())
207             theta_list.append(tm.clusters[i].get_theta_array())
208
209         # Run 10 iterations according to this mixture
210         loglikelihood = computationsEM(10, samples, num_clusters, tm, topology_list
211                                     , theta_list)
212
213         aux = loglikelihood[-1]
214         if (aux > max_log):
215             max_log = aux
216             best_seed = aux_seed
217
218     # ----- End of sieving ----- #

```



```

218     # Variable initialization
219     np.random.seed(best_seed)
220     topology_list = [] # Dimensions: (num_clusters, num_nodes)
221     theta_list = [] # Dimensions: (num_clusters, num_nodes, 2)
222     tm = TreeMixture(num_clusters = num_clusters, num_nodes = samples.shape[1])
223     tm.simulate_pi(seed_val = best_seed)
224     tm.simulate_trees(seed_val = best_seed)
225
226     for k in range(num_clusters):
227         topology_list.append(tm.clusters[k].get_topology_array())
228         theta_list.append(tm.clusters[k].get_theta_array())
229
230     # Beginning of iterations
231     pi = tm.pi
232     loglikelihood = computationsEM(max_num_iter, samples, num_clusters, tm,
233                                   topology_list, theta_list)
234
235     return loglikelihood, topology_list, theta_list
236
237 def main():
238     # Code to process command line arguments
239     parser = argparse.ArgumentParser(description='EM algorithm for likelihood of a
240     tree GM.')
241     parser.add_argument('sample_filename', type=str,
242                         help='Specify the name of the sample file (i.e data/
243     example_samples.txt)')
244     parser.add_argument('output_filename', type=str,
245                         help='Specify the name of the output file (i.e data/
246     example_results.txt)')
247     parser.add_argument('num_clusters', type=int, help='Specify the number of
248     clusters (i.e 3)')
249     parser.add_argument('--seed_val', type=int, default=42, help='Specify the seed
250     value for reproducibility (i.e 42)')
251     parser.add_argument('--real_values_filename', type=str, default="",
252                         help='Specify the name of the real values file (i.e data/
253     example_tree_mixture.pkl)')
254     # You can add more default parameters if you want.
255
256     print("This file demonstrates the flow of function templates of question 2.5.")
257
258     print("\n0. Load the parameters from command line.\n")
259
260     args = parser.parse_args()
261     print("\tArguments are: ", args)
262
263     print("\n1. Load samples from txt file.\n")

```

```

258
259     samples = np.loadtxt(args.sample_filename, delimiter="\t", dtype=np.int32)
260     num_samples, num_nodes = samples.shape
261     print("\tnum_samples: ", num_samples, "\tnum_nodes: ", num_nodes)
262     print("\tSamples: \n", samples)
263
264     print("\n2. Run EM Algorithm.\n")
265     max_iterations = 100 # Maximum number of iterations for the EM algorithm
266     loglikelihood, topology_array, theta_array = em_algorithm(args.seed_val,
267                                                              samples, args.num_clusters, max_iterations)
268
269     print("\n3. Save, print and plot the results.\n")
270
271     save_results(loglikelihood, topology_array, theta_array, args.output_filename)
272
273     for i in range(args.num_clusters):
274         print("\n\tCluster: ", i)
275         print("\tTopology: \t", topology_array[i])
276         print("\tTheta: \t", theta_array[i])
277
278     print("\n4. Retrieve real results and compare.\n")
279     if args.real_values_filename != "":
280         print("\tComparing the results with real values...")
281
282         print("\t4.1. Make the Robinson-Foulds distance analysis.\n")
283
284         tns = dendropy.TaxonNamespace()
285         original_tree = list()
286         original_topology = list()
287
288         for k in range(args.num_clusters):
289             filename = args.real_values_filename + "_tree_" + str(k) + "_topology.
290             npy"
291             original_topology.append(np.load(filename))
292             original_tree.append(Tree())
293             original_tree[-1].load_tree_from_direct_arrays(original_topology[-1])
294             original_tree[-1] = dendropy.Tree.get(data = original_tree[-1].newick,
295             schema = "newick", taxon_namespace = tns)
296
297         generated_tree = list()
298
299         for k in range(args.num_clusters):
300             generated_tree.append(Tree())
301             generated_tree[-1].load_tree_from_direct_arrays(topology_array[k])
302             generated_tree[-1] = dendropy.Tree.get(data = generated_tree[-1].newick
303             , schema = "newick", taxon_namespace = tns)
304             print("Generated tree ", k, " ", generated_tree[-1].as_string("newick"))

```

```

301         generated_tree[-1].print_plot()
302
303     print("\tDistances of trees:\n")
304     for k in range(args.num_clusters):
305         for i in range(args.num_clusters):
306             print("\tOriginal tree",k,"compared to generated tree",i)
307             print("\t\tRobinson-Foulds distance:", dendropy.calculate.
treecompare.symmetric_difference(original_tree[k], generated_tree[i]))
308
309
310     print("\n\t4.2. Make the likelihood comparison.\n")
311
312     original_theta = list()
313     for k in range(args.num_clusters):
314         filename = args.real_values_filename + "_tree_" + str(k) + "_theta.npy"
315         original_theta.append(np.load(filename, allow_pickle = True))
316
317     filename = args.real_values_filename + "_pi.npy"
318     original_pi = np.load(filename)
319     original_likelihood = np.zeros((num_samples, args.num_clusters))
320     for i in range(num_samples):
321         for k in range(args.num_clusters):
322             original_likelihood[i,k] = computeLikelihood(samples[i,:],
original_topology[k], original_theta[k])
323
324     original_log_likelihood = computeLogLikelihood(original_pi,
original_likelihood)
325     original_log_likelihood_array = [original_log_likelihood for i in range(
max_iterations)]
326
327     plt.figure(figsize=(16, 7))
328     plt.subplot(121)
329     plt.plot(np.exp(loglikelihood), label='Estimated')
330     plt.plot(np.exp(original_log_likelihood_array), label='Real',color = 'r')
331     plt.ylabel("Likelihood of Mixture")
332     plt.xlabel("Iterations")
333     plt.subplot(122)
334     plt.plot(loglikelihood, label='Estimated')
335     plt.plot(original_log_likelihood_array, label='Original',color = 'r')
336     plt.ylabel("Log-Likelihood of Mixture")
337     plt.xlabel("Iterations")
338     plt.legend(loc=(1.04, 0))
339     plt.show()
340
341     print("End of execution.\n")
342
343 if __name__ == "__main__":

```

```
344 main()
```

C Super epicentra – EM

Exercise 2.6

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 import matplotlib.mlab as mlab
4 from scipy.stats import multivariate_normal, poisson
5 import math
6
7
8 def generate_data(n_data, means, covariances, weights, rates):
9     n_clusters, n_features = means.shape
10    data = np.zeros((n_data, n_features))
11    poission_data = np.zeros(n_data)
12    colors = np.zeros(n_data, dtype='str')
13    for i in range(n_data):
14        # pick a cluster id and create data from this cluster
15        k = np.random.choice(n_clusters, size=1, p=weights)[0]
16        x = np.random.multivariate_normal(means[k], covariances[k])
17        data[i] = x
18        poission_data[i] = np.random.poisson(rates[k])
19        if k == 0:
20            colors[i] = 'red'
21        elif k == 1:
22            colors[i] = 'blue'
23        elif k == 2:
24            colors[i] = 'green'
25
26    return data, poission_data, colors
27
28
29 # means, covs: means and covariances of Gaussians
30 # rates: rates of Poissons
31 # title: title of the plot defining which EM iteration
32 def plot_contours(X, S, means, covs, title, rates):
33     plt.figure()
34     plt.scatter(X[:, 0], X[:, 1], s=S)
35
36     delta = 0.025
37     k = means.shape[0]
38     x = np.arange(-2.0, 7.0, delta)
39     y = np.arange(-2.0, 7.0, delta)
```

```

40     X, Y = np.meshgrid(x, y)
41     col = ['green', 'red', 'indigo']
42     for i in range(k):
43         mean = means[i]
44         cov = covs[i]
45         positions = np.dstack((X,Y))
46         Z = multivariate_normal(mean, cov)
47         plt.contour(X, Y, Z.pdf(positions), colors=col[i], linewidths=rates[i],
48                    alpha=0.1)
49
50     plt.title(title)
51     plt.tight_layout()
52
53 class EM:
54
55     def __init__(self, n_components, n_iter, tol, seed):
56         self.n_components = n_components
57         self.n_iter = n_iter
58         self.tol = tol
59         self.seed = seed
60
61     def fit(self, X, S):
62
63         # data's dimensionality
64         self.n_row, self.n_col = X.shape
65
66         # initialize parameters
67         np.random.seed(self.seed)
68         chosen = np.random.choice(self.n_row, self.n_components, replace=False)
69         self.means = X[chosen]
70         self.weights = np.full(self.n_components, 1 / self.n_components)
71         if self.n_components == 3:
72             self.rates = (np.mean(S) * np.ones(self.n_components) / np.array([1, 2,
73 3])[np.newaxis]).flatten()
74         elif self.n_components == 2:
75             self.rates = (np.mean(S) * np.ones(self.n_components) / np.array([1,
76 2])[np.newaxis]).flatten()
77         shape = self.n_components, self.n_col, self.n_col
78         self.covs = np.full(shape, np.cov(X, rowvar=False))
79         new_covs = []
80         for c in self.covs:
81             new_covs = np.append(new_covs, np.diag(np.diag(c))) # making the
82             covariances diagonal (question assumption)
83         self.covs = np.array(new_covs).reshape(self.n_components, 2, 2)
84
85         log_likelihood = 0

```

```

83     self.converged = False
84
85     for i in range(self.n_iter):
86         self._do_estep(X, S)
87         self._do_mstep(X, S)
88         log_likelihood_new = self._compute_log_likelihood(X, S)
89
90         if (log_likelihood - log_likelihood_new) <= self.tol:
91             self.converged = True
92             break
93
94         log_likelihood = log_likelihood_new
95
96     return self
97
98     def _do_estep(self, X, S):
99         num = np.zeros((self.n_row, self.n_components))
100        denom = np.zeros(self.n_row)
101        self.gamma = np.zeros((self.n_row, self.n_components))
102        for i in range(self.n_row):
103            for j in range(self.n_components):
104                num[i,j] = self.weights[j] * multivariate_normal(self.means[j],
105                self.covs[j]).pdf(X[i]) * poisson(self.rates[j]).pmf(S[i])
106            self.gamma = (num.T / num.sum(1)).T
107
108        return self
109
110    def _do_mstep(self, X, S):
111        n_k = self.gamma.sum(0)
112        elems = self.n_row
113        for i in range(self.n_components):
114            for j in range(self.n_col):
115                self.means[i,j] = sum(self.gamma[:,i] * X[:,j]) / n_k[i]
116                diff = X[:,j] - self.means[i,j]
117                self.covs[i,j,j] = sum(self.gamma[:,i] * diff * np.transpose(diff))
118                / n_k[i]
119                self.rates[i] = sum(self.gamma[:,i] * S[:]) / n_k[i]
120                self.weights[i] = n_k[i] / elems
121
122        return self
123
124    def _compute_log_likelihood(self, X, S):
125        log_likelihood = 0
126        for i in range(self.n_components):
127            log_likelihood = sum(self.gamma[:,i] * (self.weights[i] *
128            multivariate_normal(self.means[i], self.covs[i]).pdf(X[:,i]) * poisson(self.
129            rates[i]).pmf(S[:,i])))

```

```
126     log_likelihood = np.log(log_likelihood).sum()
127
128     return log_likelihood
129
130 # params for 3 clusters
131 means = np.array([
132     [5, 0],
133     [1, 1],
134     [0, 5]
135 ])
136
137 covariances = np.array([
138     [[.5, 0.], [0, .5]],
139     [[.92, 0], [0, .91]],
140     [[.5, 0.], [0, .5]]
141 ])
142
143 weights = [1 / 4, 1 / 2, 1 / 4]
144
145 # params for 2 clusters
146 means_2 = np.array([
147     [5, 0],
148     [1, 1]
149 ])
150
151 covariances_2 = np.array([
152     [[.5, 0.], [0, .5]],
153     [[.92, 0], [0, .91]]
154 ])
155
156 weights_2 = [1 / 4, 3 / 4]
157
158 np.random.seed(3)
159
160 rates = np.random.uniform(low=.2, high=20, size=3)
161 print("Poisson rates for 3 components:")
162 print(rates)
163
164 rates_2 = np.random.uniform(low=.2, high=20, size=2)
165 print("Poisson rates for 2 components:")
166 print(rates_2)
167
168 # generate data
169 X, S, colors = generate_data(100, means, covariances, weights, rates)
170 plt.scatter(X[:, 0], X[:, 1], s=S, c=colors) # the Poisson data is shown through
171         size of the points: s
172 plt.show()
```

```

172
173 X_2, S_2, colors_2 = generate_data(100, means_2, covariances_2, weights_2, rates_2)
174 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
175 plt.show()
176
177 # Plots of EM results #
178
179 em = EM(n_components=3, n_iter=1, tol=1e-4, seed=1)
180 em.fit(X, S)
181 # plot: call plot_contours and give it the params updated from EM with 3 components
    (after 1 iteration)
182 plot_contours(X, S, means=em.means, covs=em.covs, title="Expectation Maximization: 3
    Components - 1 Iteration", rates=em.rates)
183 plt.show()
184
185 em = EM(n_components=3, n_iter=50, tol=1e-4, seed=1)
186 em.fit(X, S)
187 # plot: call plot_contours and give it the params updated from EM with 3 components
    (after 50 iterations)
188 plot_contours(X, S, means=em.means, covs=em.covs, title="Expectation Maximization: 3
    Components - 50 Iterations", rates=em.rates)
189 plt.show()
190
191 em_2 = EM(n_components=2, n_iter=1, tol=1e-4, seed=1)
192 em_2.fit(X_2, S_2)
193 # plot: call plot_contours and give it the params updated from EM with 2 components
    (after 1 iteration)
194 plot_contours(X_2, S_2, means=em_2.means, covs=em_2.covs, title="Expectation
    Maximization: 2 Components - 1 Iteration", rates=em_2.rates)
195 plt.show()
196
197 em_2 = EM(n_components=2, n_iter=50, tol=1e-4, seed=1)
198 em_2.fit(X_2, S_2)
199 # plot: call plot_contours and give it the params updated from EM with 2 components
    (after 50 iterations)
200 plot_contours(X_2, S_2, means=em_2.means, covs=em_2.covs, title="Expectation
    Maximization: 2 Components - 50 Iterations", rates=em_2.rates)
201 plt.show()

```

Tree Generator

```

1 from Tree import TreeMixture
2 import Exercise2_5
3 import argparse
4
5 # This file has the pupose of creating new samples for testing exercise 5

```



```
6
7 parser = argparse.ArgumentParser()
8 parser.add_argument("seed", help="Introduce the seed to generate trees", type=int)
9 parser.add_argument("samples", help="Introduce the number of samples", type=int)
10 parser.add_argument("nodes", help="Introduce the number of nodes", type=int)
11 parser.add_argument("clusters", help="Introduce the number of clusters", type=int)
12 args = parser.parse_args()
13 print("Generating tree with seed:", args.seed, "\tsamples:", args.samples,
14       "\tnodes:", args.nodes, "\tclusters:", args.clusters)
15 tm = TreeMixture(num_clusters=args.clusters, num_nodes=args.nodes)
16 tm.simulate_pi(seed_val=args.seed)
17 tm.simulate_trees(seed_val=args.seed)
18 tm.sample_mixtures(num_samples=args.samples, seed_val=args.seed)
19 path = 'data/q_2_5_tm_'+str(args.nodes)+'node_'+str(args.samples)+'sample_'+str(
20       args.clusters)+'clusters.pkl'
20 tm.save_mixture(path, True)
```

References

- [1] C. M. Bishop, *Pattern recognition and machine learning*. Springer Science+ Business Media, 2006.
- [2] *Variational bayes and the mean-field approximation*, <https://bjlkeng.github.io/posts/variational-bayes-and-the-mean-field-approximation/>, Accessed: 2019-12-30.
- [3] M. Meila and M. I. Jordan, “Learning with mixtures of trees”, *Journal of Machine Learning Research*, vol. 1, no. Oct, pp. 1–48, 2000.
- [4] *When dependence between events is conditional*, <https://www.probablisticworld.com/conditional-dependence-independence/>, Accessed: 2019-12-11.
- [5] *Mathematicalmonk*, <https://www.youtube.com/user/mathematicalmonk>, Accessed: 2019-12-30.