DD2434 Machine Learning, Advanced Course

Fernando García Sanz - fegs@kth.se

January 4, 2020

Contents

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 $\mu_k \perp \tau_k | X, ..., X^N$?*

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 $\mu \perp \beta' | X, ..., X^N$?*

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

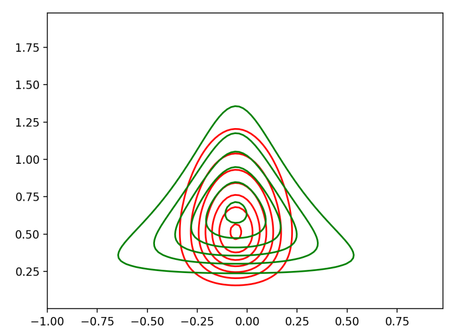
The exact posterior is defined by:

$$p(\mu|\tau) = \mathcal{N}(\mu|\mu_0, (\lambda_0\tau)^{-1})$$
$$p(\tau) = Gam(\tau|a_0, b_0)$$

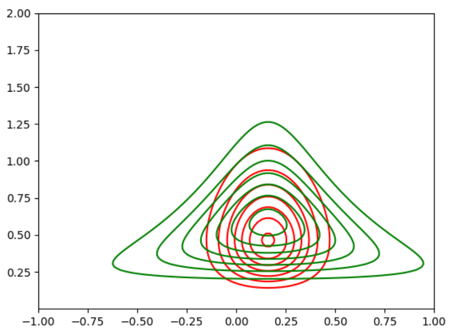
The joint of these distributions provides the exact posterior.

Question 2.4.14

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

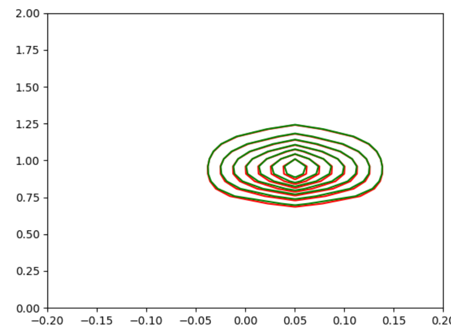


(a) Estimation 10 samples - 10 iterations

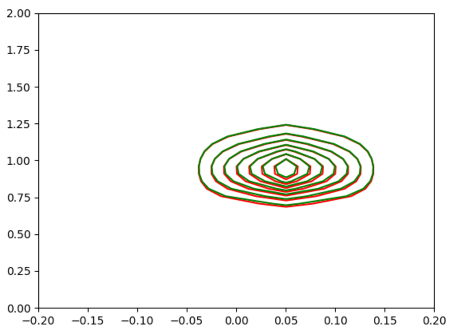


(b) Estimation 10 samples - 100 iterations

Figure 1: Approximations with 10 samples



(a) Estimation 100 samples - 10 iterations



(b) Estimation 100 samples - 100 iterations

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

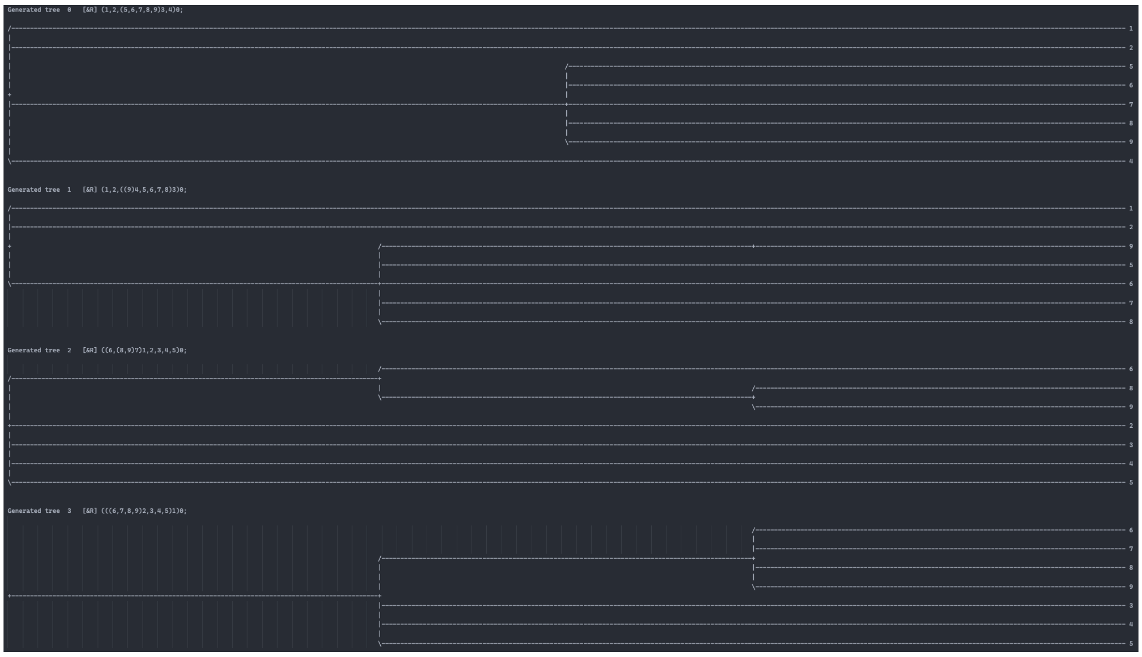
Question 2.5.15

See appendix B.

Question 2.5.16

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

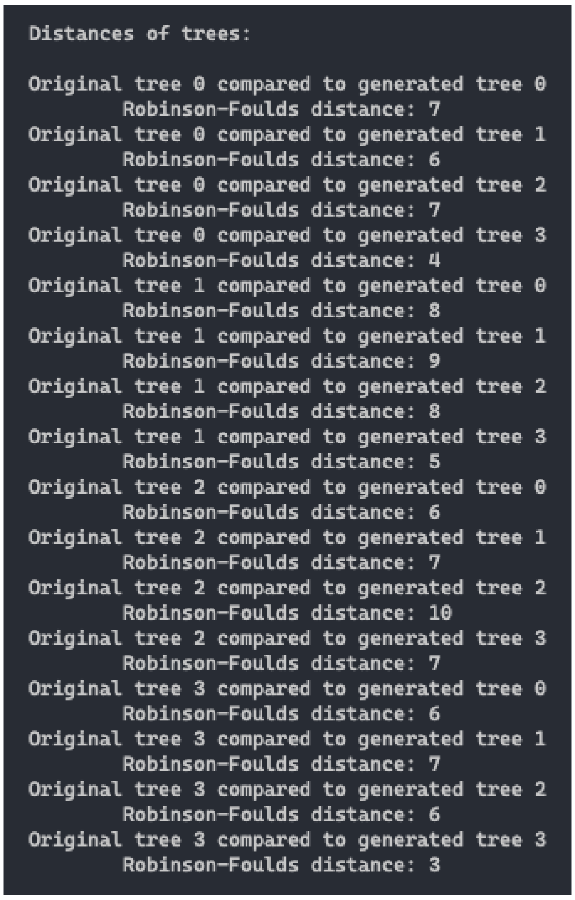
- The obtained trees for a configuration with 10 nodes, 20 observations and 4 clusters have been the followings:



```
Generated tree  0   [&R] (1,2,(5,6,7,8,9)3,4)0;

Generated tree  1   [&R] (1,2,((9)4,5,6,7,8)3)0;

Generated tree  2   [&R] ((6,(8,9)7)1,2,3,4,5)0;

Generated tree  3   [&R] (((6,7,8,9)2,3,4,5)1)0;
```

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: 7
Original tree 0 compared to generated tree 1
        Robinson-Foulds distance: 6
Original tree 0 compared to generated tree 2
        Robinson-Foulds distance: 7
Original tree 0 compared to generated tree 3
        Robinson-Foulds distance: 4
Original tree 1 compared to generated tree 0
        Robinson-Foulds distance: 8
Original tree 1 compared to generated tree 1
        Robinson-Foulds distance: 9
Original tree 1 compared to generated tree 2
        Robinson-Foulds distance: 8
Original tree 1 compared to generated tree 3
        Robinson-Foulds distance: 5
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: 10
Original tree 2 compared to generated tree 3
        Robinson-Foulds distance: 7
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: 6
Original tree 3 compared to generated tree 3
        Robinson-Foulds distance: 3
```

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:

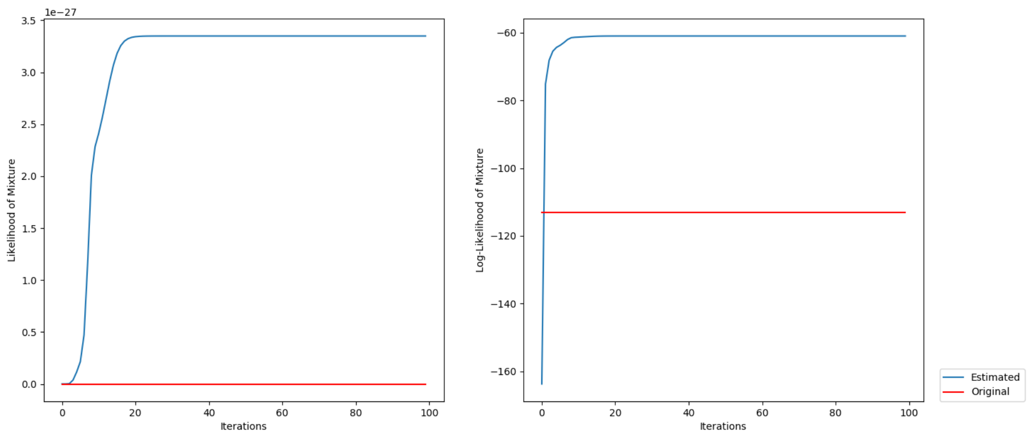
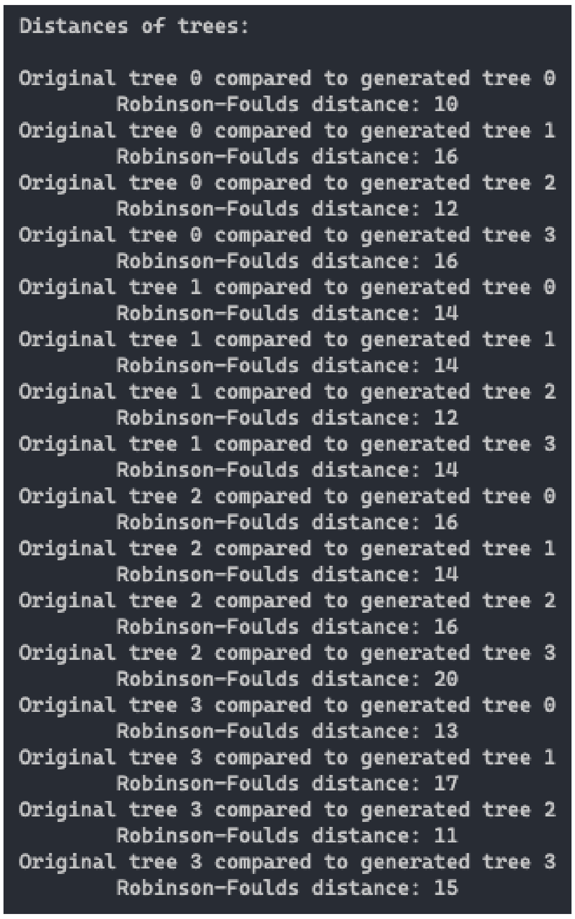


Figure 5: Likelihood and log-likelihood 10 - 20 - 4

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: 10
Original tree 0 compared to generated tree 1
        Robinson-Foulds distance: 16
Original tree 0 compared to generated tree 2
        Robinson-Foulds distance: 12
Original tree 0 compared to generated tree 3
        Robinson-Foulds distance: 16
Original tree 1 compared to generated tree 0
        Robinson-Foulds distance: 14
Original tree 1 compared to generated tree 1
        Robinson-Foulds distance: 14
Original tree 1 compared to generated tree 2
        Robinson-Foulds distance: 12
Original tree 1 compared to generated tree 3
        Robinson-Foulds distance: 14
Original tree 2 compared to generated tree 0
        Robinson-Foulds distance: 16
Original tree 2 compared to generated tree 1
        Robinson-Foulds distance: 14
Original tree 2 compared to generated tree 2
        Robinson-Foulds distance: 16
Original tree 2 compared to generated tree 3
        Robinson-Foulds distance: 20
Original tree 3 compared to generated tree 0
        Robinson-Foulds distance: 13
Original tree 3 compared to generated tree 1
        Robinson-Foulds distance: 17
Original tree 3 compared to generated tree 2
        Robinson-Foulds distance: 11
Original tree 3 compared to generated tree 3
        Robinson-Foulds distance: 15
```

Figure 6: Distances 20 - 20 - 4

  - Likelihood and log-likelihood comparison:

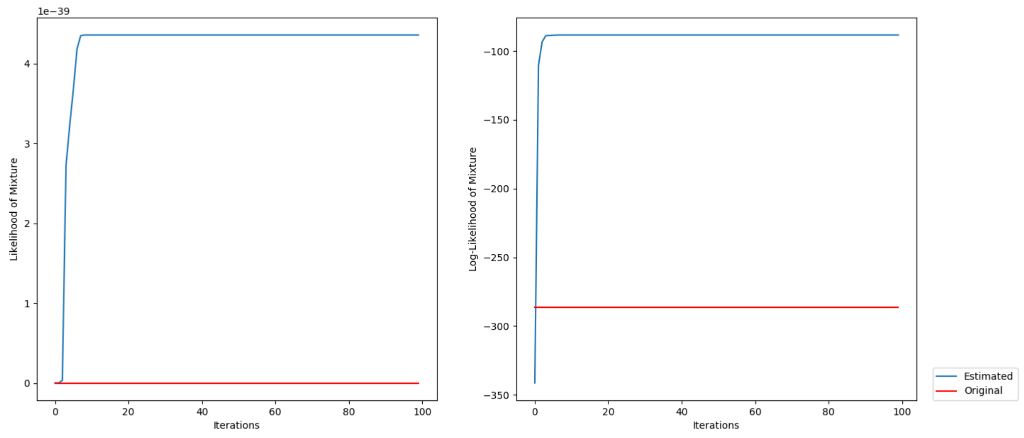
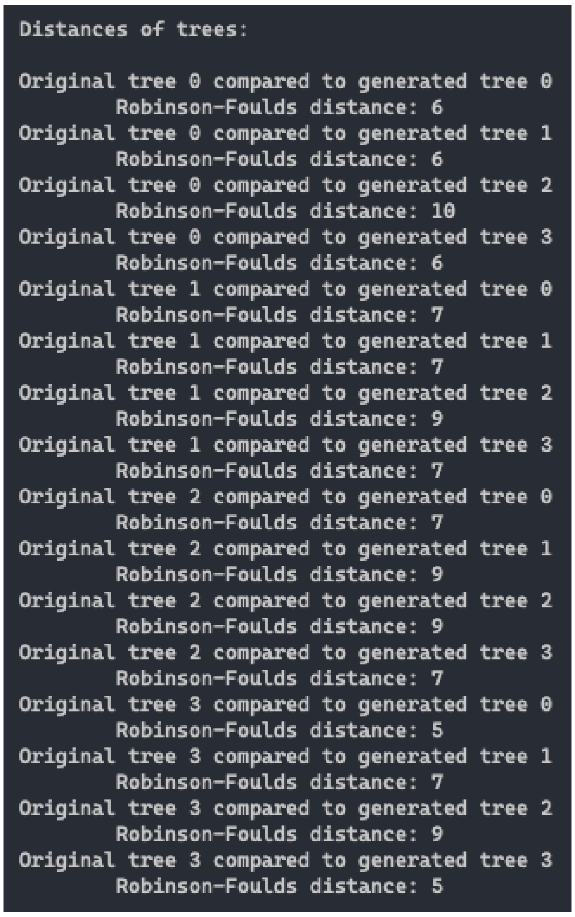


Figure 7: Likelihood and log-likelihood 20 - 20 - 4

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

Figure 8: Distances 10 - 50 - 4

  - Likelihood and log-likelihood comparison:

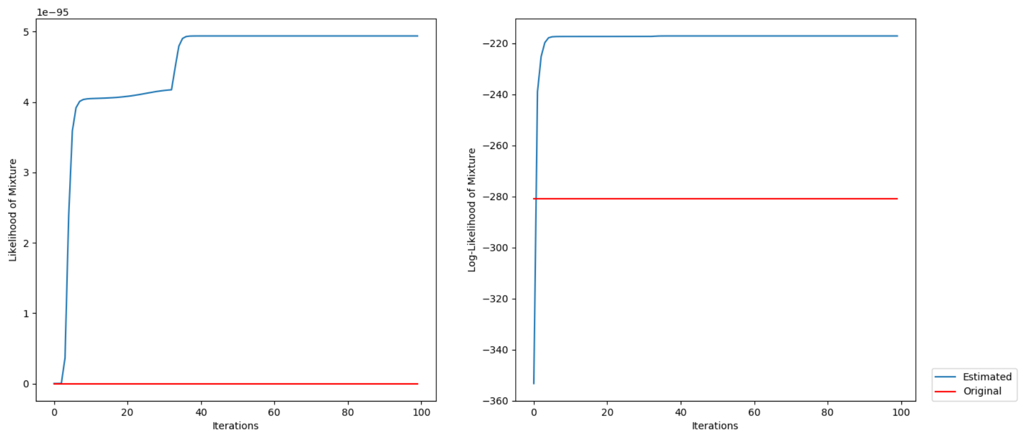
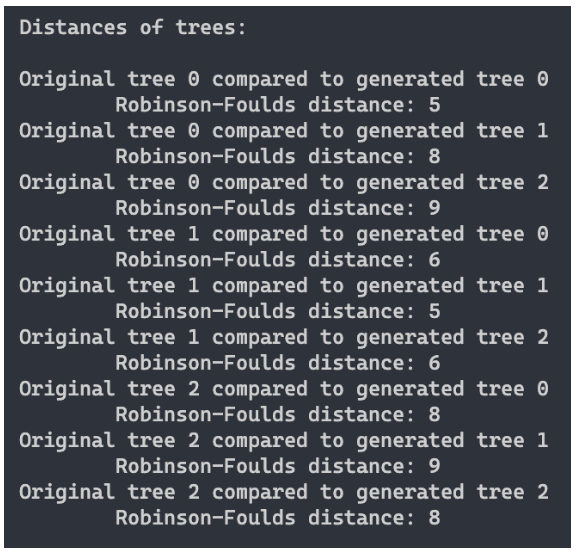


Figure 9: Likelihood and log-likelihood 10 - 50 - 4

- 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:

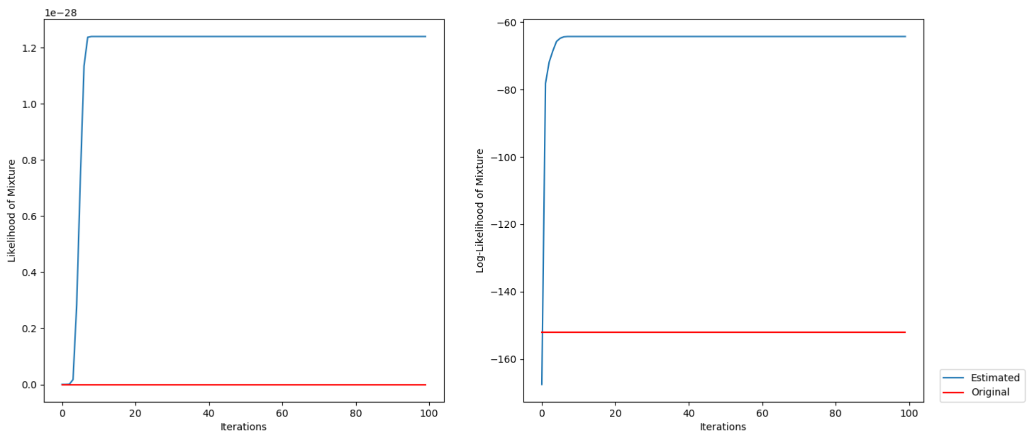


Figure 11: Likelihood and log-likelihood 10 - 20 - 4 - mixtures of 3 trees

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.

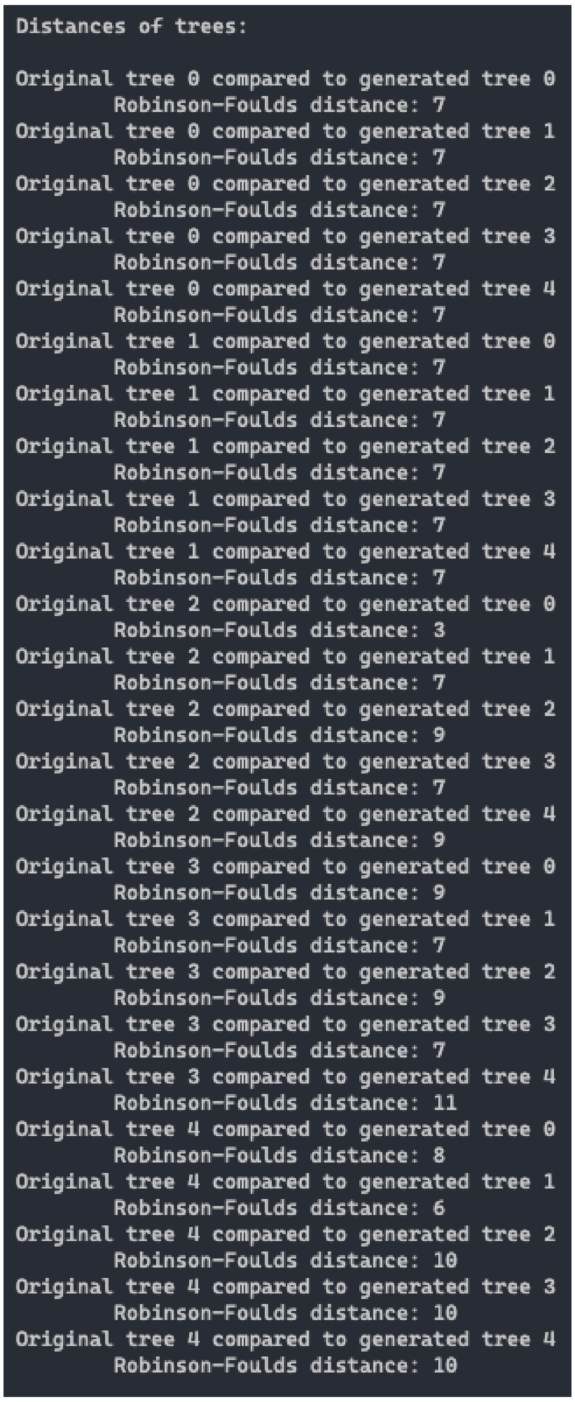
The case of increasing the number of observations from 20 to 50 has also affected the final result. Distances between trees do not vary that much; nonetheless, likelihood and log-likelihood are even smaller than in the previous case. Since the sequence of observations is much bigger, it is less probable.

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 B.

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: 7
Original tree 0 compared to generated tree 1
        Robinson-Foulds distance: 7
Original tree 0 compared to generated tree 2
        Robinson-Foulds distance: 7
Original tree 0 compared to generated tree 3
        Robinson-Foulds distance: 7
Original tree 0 compared to generated tree 4
        Robinson-Foulds distance: 7
Original tree 1 compared to generated tree 0
        Robinson-Foulds distance: 7
Original tree 1 compared to generated tree 1
        Robinson-Foulds distance: 7
Original tree 1 compared to generated tree 2
        Robinson-Foulds distance: 7
Original tree 1 compared to generated tree 3
        Robinson-Foulds distance: 7
Original tree 1 compared to generated tree 4
        Robinson-Foulds distance: 7
Original tree 2 compared to generated tree 0
        Robinson-Foulds distance: 3
Original tree 2 compared to generated tree 1
        Robinson-Foulds distance: 7
Original tree 2 compared to generated tree 2
        Robinson-Foulds distance: 9
Original tree 2 compared to generated tree 3
        Robinson-Foulds distance: 7
Original tree 2 compared to generated tree 4
        Robinson-Foulds distance: 9
Original tree 3 compared to generated tree 0
        Robinson-Foulds distance: 9
Original tree 3 compared to generated tree 1
        Robinson-Foulds distance: 7
Original tree 3 compared to generated tree 2
        Robinson-Foulds distance: 9
Original tree 3 compared to generated tree 3
        Robinson-Foulds distance: 7
Original tree 3 compared to generated tree 4
        Robinson-Foulds distance: 11
Original tree 4 compared to generated tree 0
        Robinson-Foulds distance: 8
Original tree 4 compared to generated tree 1
        Robinson-Foulds distance: 6
Original tree 4 compared to generated tree 2
        Robinson-Foulds distance: 10
Original tree 4 compared to generated tree 3
        Robinson-Foulds distance: 10
Original tree 4 compared to generated tree 4
        Robinson-Foulds distance: 10
```

Figure 12: Distances 10 - 2 - 5

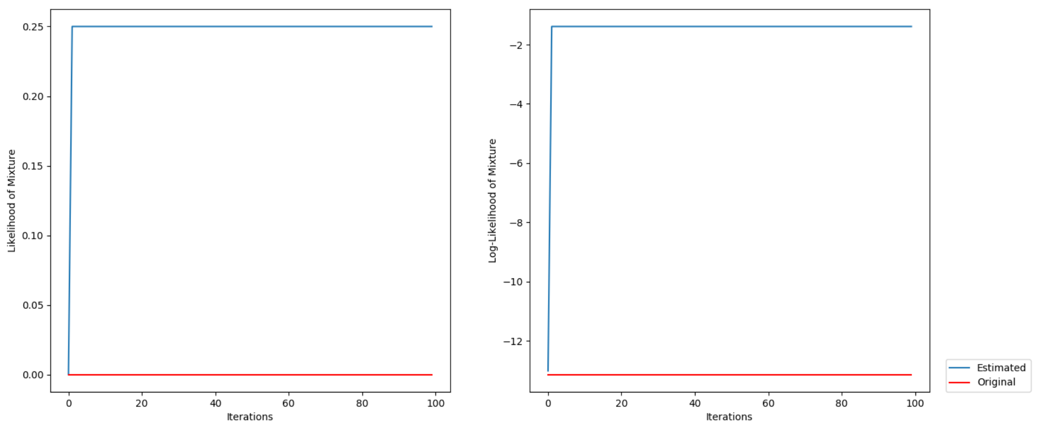
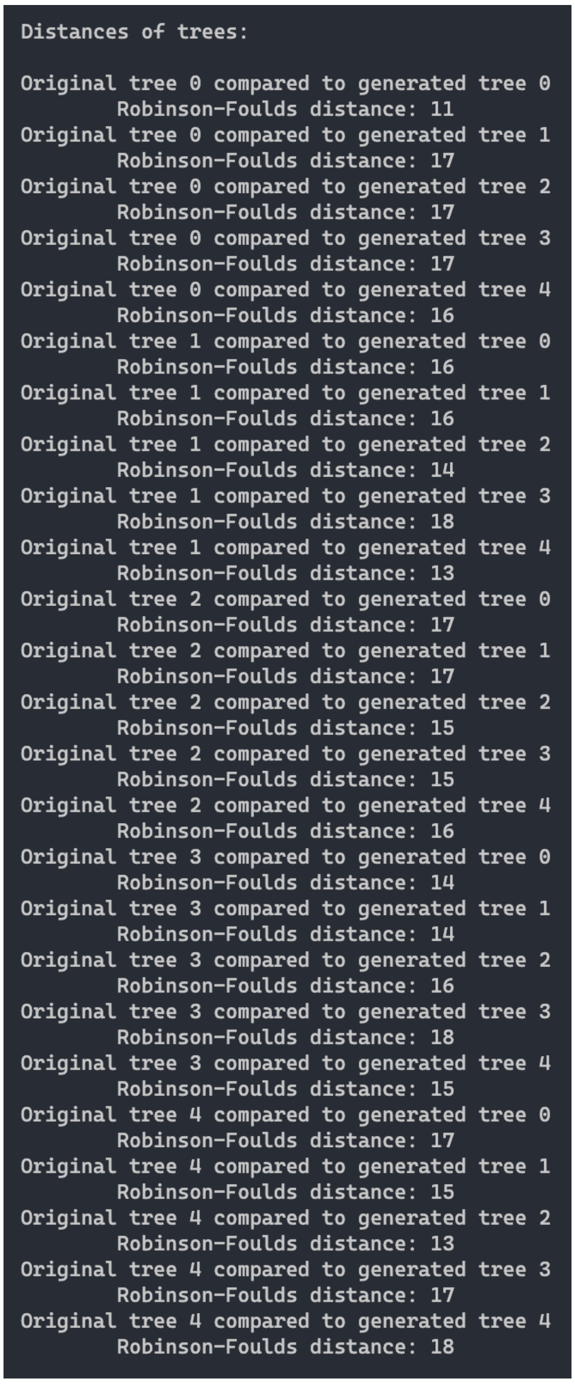


Figure 13: Likelihood and log-likelihood 10 - 2 - 5

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

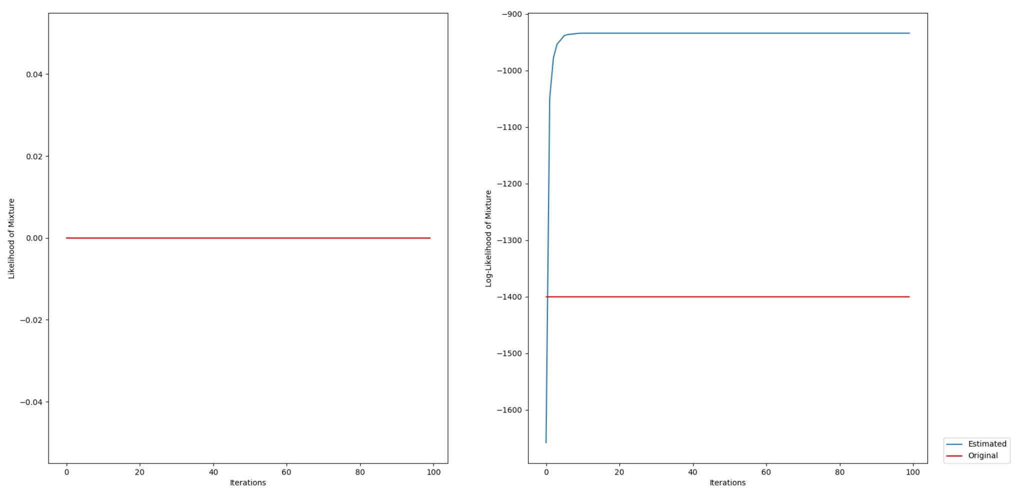


Figure 15: Likelihood and log-likelihood 20 - 100 - 5

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:

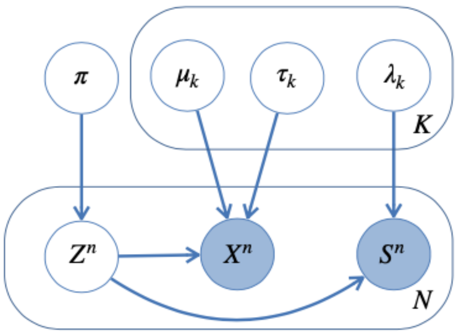


Figure 16: The K super epicentra model with priors

- 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.

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

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

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

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

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

Keeping expanding each term, they can be transformed into:

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

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

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

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

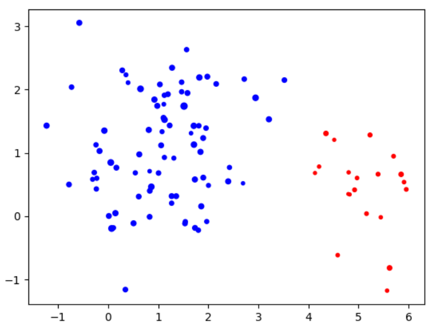
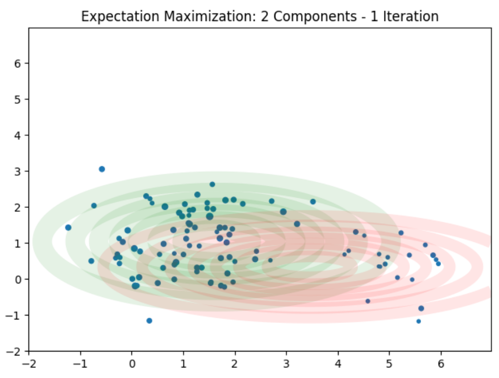
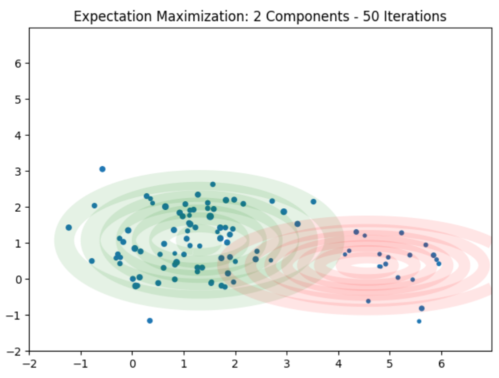


Figure 17: Provided data with 2 components



(a) EM for 2 components with 1 iteration



(b) EM for 2 components with 50 iterations

Figure 18: EM algorithm for provided data with 2 components

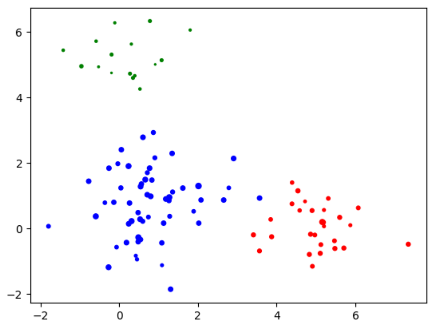
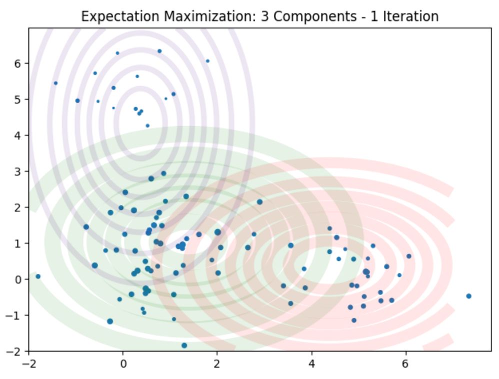
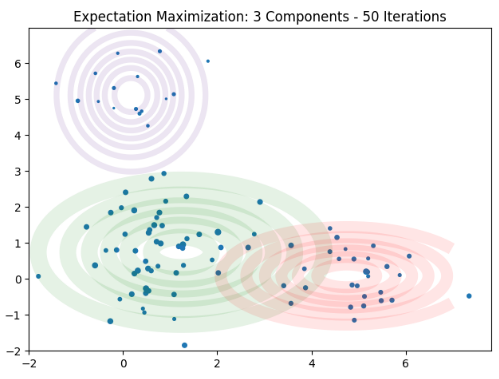


Figure 19: Provided data with 3 components



(a) EM for 3 components with 1 iteration



(b) EM for 3 components with 50 iterations

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:

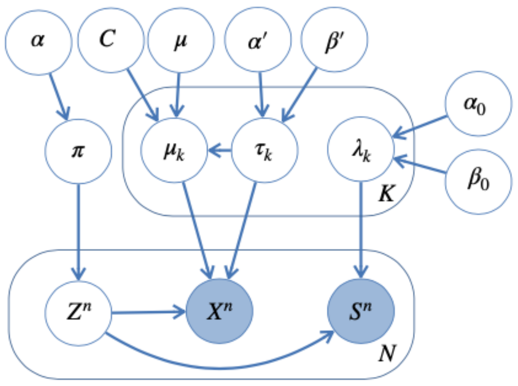


Figure 21: The K super epicentra model with priors

- π has a Dir(α) prior.
- $\tau_{k,i}$ has a Ga(α',β') prior.
- $\mu_{k,i}$ has a $\mathcal{N}(\mu, (C\tau_{k,i})^{-1})$ prior.
- λ_k has a Ga(α_0,β_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:

$$\pi \to \prod_{k=1}^{K} \pi_k^{\alpha_k - 1}$$

$$\tau_{k,i} \to \frac{(\beta')^{\alpha'}}{\Gamma(\alpha')} \, \tau_{k,i}^{[\alpha'-1]} \, e^{[-\beta' \tau_{k,i}]}$$

$$\mu_{k,i} \to \sqrt{\frac{C\tau_{k,i}}{2\pi}} \, e^{[-\frac{C\tau_{k,i}}{2}(\mu_{k,i}-\mu)^2]}$$

$$\lambda_k \to \frac{(\beta_0)^{\alpha_0}}{\Gamma(\alpha_0)} \, \lambda_k^{[\alpha_0-1]} \, e^{[-\beta_0 \lambda_k]}$$

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

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

$$\text{Being } \theta = \pi, \tau \underbrace{k, i}_{k}, \mu \underbrace{k, i}_{k}, \lambda_k$$

Now, considering variational inference:

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

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:

$$p(X, S, Z|M, T, \Lambda, \Pi) = \prod_{n=1}^{N} p(X_n|M, T)\, p(S_n|\Lambda)\ \underbrace{p(Z_n|\Pi)}_{\text{Categorical}} =$$
$$= \prod_{n=1}^{N}\prod_{k=1}^{K}(p(X_n|\boldsymbol{\mu}_k, \boldsymbol{\tau}_k)\, p(S_n|\lambda_k)\, p(Z_{n,k} = 1|\pi_k))^{Z_{n,k}}$$

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

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

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:

$$\mathbb{E}_\theta\left[\log\left(\prod_{n=1}^{N}\prod_{k=1}^{K} p(X_n|\boldsymbol{\mu}_k, \boldsymbol{\tau}_k)^{Z_{n,k}}\right) + \log\left(\prod_{n=1}^{N}\prod_{k=1}^{K} p(S_n|\lambda_k)^{Z_{n,k}}\right)\right.$$
$$\left. + \log\left(\prod_{n=1}^{N}\prod_{k=1}^{K} p(Z_{n,k} = 1|\pi_k\right)^{Z_{n,k}}\right] + const$$

Now, applying some properties:

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

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

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

Substituting in the expression, it takes the form of:

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

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

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

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:

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

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:

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

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:

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

Which is transformed into the following after removing constants:

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

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

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

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:

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

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

```python
import numpy as np
import matplotlib.pyplot as plt
import scipy.stats as stats
from tqdm import tqdm

# This function computes the value of tau_N
def compute_tauN(tau_0, mu_0, vector, a_N, b_N):

    result = (tau_0 + len(vector)) * (a_N / b_N)
    return result

# This function computes the values of b_N
def compute_bN(a_0, b_0, vector, mu_0, tau_N):

    term1 = (1 / tau_N) + pow(np.mean(vector),2) + pow(mu_0,2) - 2 * np.mean(vector
    ) * mu_0
    term2 = 0
    for x in vector:
        term2 += pow(x,2) + (1 / tau_N) + pow(np.mean(vector),2) - 2 * np.mean(
    vector) * x
    result = b_0 + 0.5 * tau_0 * term1 + 0.5 * term2

    return result

#This function computes the approximated distribution
def getQ(mu_axis, tau_axis, mu_N, tau_N, a_N, b_N):

    q_mu_tau = np.zeros((len(mu_axis), len(tau_axis)))

    for i in tqdm(range(len(mu_axis))):
        for j in range(len(tau_axis)):
            q_mu = stats.norm(mu_N, 1/(tau_N)).pdf(mu_axis[i])
            q_tau = stats.gamma.pdf(tau_axis[j], a_N, loc=0, scale=(1/b_N))
            q_mu_tau[j][i] = q_mu * q_tau

    return q_mu_tau

#This function computes the original distribution
def getP(mu_axis, tau_axis, mu_N, tau_N, a_N, b_N):

    p_mu_tau = np.zeros((len(mu_axis), len(tau_axis)))

    for i in tqdm(range(len(mu_axis))):
        for j in range(len(tau_axis)):
            if(tau_axis[j] == 0):
                tau_axis[j] = 0.00001 # Correction for 0 denominator
            p_mu = stats.norm(mu_N, 1/(tau_N * tau_axis[j])).pdf(mu_axis[i])
            p_tau = stats.gamma.pdf(tau_axis[j], a_N, loc=0, scale=(1/b_N))
            p_mu_tau[j][i] = p_mu * p_tau

    return p_mu_tau

if __name__ == "__main__":

    # VARIABLES
    iterations = 10                 # Number of iterations
    n = 10                          # Number of samples
    mu_0 = 0                        # Initial value mu_0
    mu_N = 1                        # Initial value mu_N
    tau_0 = 1                       # Initial value tau_0
    tau_N = 1                       # Initial value mu_N
    a_0 = 0                         # Initial value a_0
    a_N = 1                         # Initial value a_N
    b_0 = 1                         # Initial value b_0
    b_N = 1                         # Initial value b_N
    mu_axis = np.linspace(-1,1,100)  # X axis grid
    tau_axis = np.linspace(0,2,100)  # Y axis grid

    # SAMPLES GENERATION
    vector = np.random.normal(0,1,n)

    mu_N = (tau_0 * mu_0 + np.sum(vector)) / (tau_0 + len(vector))  # mu_N value computation (it won't update)
    a_N = a_0 + (len(vector)) / 2                                   # a_N value computation (it won't update)

    q_mu_tau = np.zeros((len(mu_axis), len(tau_axis)))
    p_mu_tau = np.zeros((len(mu_axis), len(tau_axis)))

    for i in range(iterations):
        tau_N = compute_tauN(tau_0, mu_0, vector, a_N, b_N)
        b_N = compute_bN(a_0, b_0, vector, mu_N, tau_N)

    p_mu_tau = getP(mu_axis, tau_axis, mu_N, tau_N, a_N, b_N)
    q_mu_tau = getQ(mu_axis, tau_axis, mu_N, tau_N, a_N, b_N)
    #plt.axis([-0.2,0.2, 0, 2])                                     # Uncomment this line to modify the grid area
    plt.contour(mu_axis, tau_axis, q_mu_tau, colors='red')
    plt.contour(mu_axis, tau_axis, p_mu_tau, colors='green')
    plt.show()
```

B Mixture of trees with observable variables

Exercise 2.5

```
import argparse
import numpy as np
import matplotlib.pyplot as plt
import sys
import networkx as nx
import dendropy
import Kruskal_v1 as kruskal
from Tree import Tree, TreeMixture
from tqdm import tqdm

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

    likelihood_filename = filename + "_em_loglikelihood.npy"
    topology_array_filename = filename + "_em_topology.npy"
    theta_array_filename = filename + "_em_theta.npy"
    print("Saving log-likelihood to ", likelihood_filename, ", topology_array to: "
    , 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 computeLikelihood(samples, topology, theta):
    result = theta[0][samples[0]]
    for i in range(1, len(topology)):
        result *= theta[i][samples[int(topology[i])]][samples[i]]

    return (result + sys.float_info.epsilon)

def computeLogLikelihood(pi, likelihood):
    result = 0
    for i in range(likelihood.shape[0]):
        aux = 0
        for k in range(likelihood.shape[1]):
            aux += pi[k] * likelihood[i,k]
        result += np.log(aux)

    return result

def computeCondQ(responsibility, samples, node1, node2, value1, value2):
    num = 0
    denom = 0
    for i in range(samples.shape[0]):
        if (samples[i,node1] == value1):
            denom += responsibility[i]
            if (samples[i,node2] == value2):
                num += responsibility[i]

    return (num / (denom + sys.float_info.epsilon))

def computeQ(responsibility, samples, node, val):
    num = 0
    for i in range(samples.shape[0]):
        if (samples[i,node] == val):
            num += responsibility[i]
    denom = np.sum(responsibility) + sys.float_info.epsilon

    return (num / denom)

def computeQJoint(responsibility, samples, node1, node2, val1, val2):
    num = 0
    for i in range(samples.shape[0]):
        if (samples[i,node1] == val1) and (samples[i,node2] == val2):
            num += responsibility[i]
    denom = np.sum(responsibility) + sys.float_info.epsilon

    return (num / denom)

def computeResponsibility(num_clusters, samples, pi, likelihood):
    result = np.zeros((samples.shape[0], num_clusters))
    for i in range(samples.shape[0]):
        for k in range(num_clusters):
            result[i,k] = pi[k] * likelihood[i,k]
        result[i] = (result[i] + sys.float_info.epsilon) / (np.sum(result[i]) +
    num_clusters * sys.float_info.epsilon)

    return result

def calculateI(responsibility, samples, node1, node2):
    result = 0
    for i in range(2):
        for j in range(2):
            q_node1 = computeQ(responsibility, samples, node1, i)
            q_node2 = computeQ(responsibility, samples, node2, j)
            q_joint = computeQJoint(responsibility, samples, node1, node2, i, j)
            if (q_joint != 0):
                if (q_node1 == 0):
                    q_node1 = sys.float_info.epsilon
                if (q_node2 == 0):
                    q_node2 = sys.float_info.epsilon
                if ((q_node1 * q_node2) == 0):
                    result += q_joint * np.log(q_joint / (sys.float_info.min))
                else:
                    result += q_joint * np.log(q_joint / (q_node1 * q_node2))

    return result

def computeTheta(theta_list, responsibility, samples, topology, num_nodes, num_clusters):
    result = theta_list
    for k in range(num_clusters):
        for i in range(num_nodes):
            if (i == 0):
                result[k][0,0] = computeQ(responsibility[:,k], samples, 0, 0)
                result[k][0,1] = computeQ(responsibility[:,k], samples, 0, 1)
            else:
                result[k][i,0][0]= computeCondQ(responsibility[:,k], samples, int(topology[k][i]), i, 0, 0)
                result[k][i,0][1]= computeCondQ(responsibility[:,k], samples, int(topology[k][i]), i, 0, 1)
                result[k][i,1][0]= computeCondQ(responsibility[:,k], samples, int(topology[k][i]), i, 1, 0)
                result[k][i,1][1]= computeCondQ(responsibility[:,k], samples, int(topology[k][i]), i, 1, 1)

    return result

def computationsEM(iterations, samples, num_clusters, tm, topology_list, theta_list):
    pi = tm.pi
    loglikelihood = np.zeros(iterations)

    for it in range(iterations):
        num_samples = samples.shape[0]
        num_nodes = samples.shape[1]
        likelihood = np.zeros((num_samples, num_clusters)) # Probability of having this sample per tree

        # Compute likelihood per sample
        for i in range(num_samples):
            for k in range(num_clusters):
                likelihood[i,k] = computeLikelihood(samples[i,:], topology_list[k], theta_list[k])

        # Computation of responsibilities
        responsibility = computeResponsibility(num_clusters, samples, pi,
    likelihood)

        # Computation of pi'
        res_sum = np.sum(responsibility, axis=0)
        total_sum = np.sum(res_sum)
        pi = np.zeros(len(res_sum))
        for i in range(len(res_sum)):
            pi[i] = res_sum[i] / total_sum

        # Get the IQ for using as weights
        IQ = np.zeros((num_nodes, num_nodes, num_clusters))
        for k in range(num_clusters):
            for i in range(len(topology_list[k])):
                for j in range(len(topology_list[k])):
                    if (i != j):
                        IQ[i,j,k] = calculateI(responsibility[:,k], samples, i, j)

        # Create the graphs
        graphs = list()
        for k in range(num_clusters):
            graphs.append(kruskal.Graph(num_nodes))
            for i in range(num_nodes):
                for j in range(i+1, num_nodes):
                    graphs[-1].addEdge(i, j, IQ[i,j,k])

        # Get the Maximum Spanning Tree from each graph
        tree = np.zeros((num_nodes-1, 3, num_clusters))
        for k in range(num_clusters):
            result = graphs[k].maximum_spanning_tree()
            cnt = 0
            for u_aux, v_aux, weight_aux in result:
                tree[cnt,0,k] = u_aux
                tree[cnt,1,k] = v_aux
                tree[cnt,2,k] = weight_aux
                cnt += 1

        # Creation of the tree
        topology_list = list()
        for k in range(num_clusters):
            topology_list.append(np.zeros(num_nodes))
            topology_list[-1][0] = np.nan
            max_tree = nx.Graph()
            for i in range(tree.shape[0]):
                max_tree.add_edge(tree[i,0,k], tree[i,1,k])
            finaltree = list(nx.bfs_edges(max_tree, 0))
            for i in range(num_nodes - 1):
```

```python
                topology_list[-1][int(finaltree[i][1])] = finaltree[i][0]

        # Computation of theta'
        theta_list = computeTheta(theta_list, responsibility, samples, topology_list, num_nodes, num_clusters)

        loglikelihood[it] = computeLogLikelihood(pi, likelihood)

    return loglikelihood


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

    # Initialize the needed variables
    sieving = 100
    max_log = float("-inf")
    best_seed = 0

    # Get the best seed for likelihood
    for siev in tqdm(range(sieving)):
        # Set the seed
        aux_seed = seed_val + siev # Try with all seeds from @param:seed_val to @param:seed_val + sieving
        np.random.seed(aux_seed)

        # Generate tree mixture
        tm = TreeMixture(num_clusters=num_clusters, num_nodes=samples.shape[1])
        tm.simulate_pi(seed_val=aux_seed)
        tm.simulate_trees(seed_val=aux_seed)
        topology_list = []
        theta_list = []

        for i in range(num_clusters):
            topology_list.append(tm.clusters[i].get_topology_array())
            theta_list.append(tm.clusters[i].get_theta_array())

        # Run 10 iterations according to this mixture
        loglikelihood = computationsEM(10, samples, num_clusters, tm, topology_list, theta_list)

        aux = loglikelihood[-1]
        if (aux > max_log):
            max_log = aux
            best_seed = aux_seed

    # -------------------- End of sieving -------------------- #

    # Variable initialization
    np.random.seed(best_seed)
    topology_list = [] # Dimensions: (num_clusters, num_nodes)
    theta_list = [] # Dimensions: (num_clusters, num_nodes, 2)
    tm = TreeMixture(num_clusters = num_clusters, num_nodes = samples.shape[1])
    tm.simulate_pi(seed_val = best_seed)
    tm.simulate_trees(seed_val = best_seed)

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

    # Beginning of iterations
    pi = tm.pi
    loglikelihood = computationsEM(max_num_iter, samples, num_clusters, tm,
    topology_list, theta_list)

    return loglikelihood, topology_list, theta_list


def main():
    # Code to process command line arguments
    parser = argparse.ArgumentParser(description='EM algorithm for likelihood of a
    tree GM.')
    parser.add_argument('sample_filename', type=str,
                        help='Specify the name of the sample file (i.e data/
    example_samples.txt)')
    parser.add_argument('output_filename', type=str,
                        help='Specify the name of the output file (i.e data/
    example_results.txt)')
    parser.add_argument('num_clusters', type=int, help='Specify the number of
    clusters (i.e 3)')
    parser.add_argument('--seed_val', type=int, default=42, help='Specify the seed
    value for reproducibility (i.e 42)')
    parser.add_argument('--real_values_filename', type=str, default="",
                        help='Specify the name of the real values file (i.e data/
    example_tree_mixture.pkl)')
    # You can add more default parameters if you want.

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

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

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

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

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

    print("\n2. Run EM Algorithm.\n")
    max_iterations = 100 # Maximum number of iterations for the EM algorithm
    loglikelihood, topology_array, theta_array = em_algorithm(args.seed_val,
    samples, args.num_clusters, max_iterations)

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

    save_results(loglikelihood, topology_array, theta_array, args.output_filename)

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

    print("\n4. Retrieve real results and compare.\n")
    if args.real_values_filename != "":
        print("\tComparing the results with real values...")

        print("\t4.1. Make the Robinson-Foulds distance analysis.\n")

        tns = dendropy.TaxonNamespace()
        original_tree = list()
        original_topology = list()

        for k in range(args.num_clusters):
            filename = args.real_values_filename + "_tree_" + str(k) + "_topology.
    npy"
            original_topology.append(np.load(filename))
            original_tree.append(Tree())
            original_tree[-1].load_tree_from_direct_arrays(original_topology[-1])
            original_tree[-1] = dendropy.Tree.get(data = original_tree[-1].newick,
    schema = "newick", taxon_namespace = tns)

        generated_tree = list()

        for k in range(args.num_clusters):
            generated_tree.append(Tree())
            generated_tree[-1].load_tree_from_direct_arrays(topology_array[k])
            generated_tree[-1] = dendropy.Tree.get(data = generated_tree[-1].newick
    , schema = "newick", taxon_namespace = tns)
            print("Generated tree ", k, " ",generated_tree[-1].as_string("newick"))
```

```python
            generated_tree[-1].print_plot()

        print("\tDistances of trees:\n")
        for k in range(args.num_clusters):
            for i in range(args.num_clusters):
                print("\tOriginal tree",k,"compared to generated tree",i)
                print("\t\tRobinson-Foulds distance:", dendropy.calculate.treecompare.symmetric_difference(original_tree[k], generated_tree[i]))


        print("\n\t4.2. Make the likelihood comparison.\n")

        original_theta = list()
        for k in range(args.num_clusters):
            filename = args.real_values_filename + "_tree_" + str(k) + "_theta.npy"
            original_theta.append(np.load(filename, allow_pickle = True))

        filename = args.real_values_filename + "_pi.npy"
        original_pi = np.load(filename)
        original_likelihood = np.zeros((num_samples, args.num_clusters))
        for i in range(num_samples):
            for k in range(args.num_clusters):
                original_likelihood[i,k] = computeLikelihood(samples[i,:], original_topology[k], original_theta[k])

        original_log_likelihood = computeLogLikelihood(original_pi, original_likelihood)
        original_log_likelihood_array = [original_log_likelihood for i in range(max_iterations)]

        plt.figure(figsize=(16, 7))
        plt.subplot(121)
        plt.plot(np.exp(loglikelihood), label='Estimated')
        plt.plot(np.exp(original_log_likelihood_array), label='Real',color = 'r')
        plt.ylabel("Likelihood of Mixture")
        plt.xlabel("Iterations")
        plt.subplot(122)
        plt.plot(loglikelihood, label='Estimated')
        plt.plot(original_log_likelihood_array, label='Original',color = 'r')
        plt.ylabel("Log-Likelihood of Mixture")
        plt.xlabel("Iterations")
        plt.legend(loc=(1.04, 0))
        plt.show()

        print("End of execution.\n")

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

Tree Generator

```
from Tree import TreeMixture
import Exercise2_5
import argparse

# This file has the pupose of creating new samples for testing exercise 5

parser = argparse.ArgumentParser()
parser.add_argument("seed", help="Introduce the seed to generate trees", type=int)
parser.add_argument("samples", help="Introduce the number of samples", type=int)
parser.add_argument("nodes", help="Introduce the number of nodes", type=int)
parser.add_argument("clusters", help="Introduce the number of clusters", type=int)
args = parser.parse_args()
print("Generating tree with seed:", args.seed, "\tsamples:", args.samples,
        "\tnodes:", args.nodes, "\tclusters:", args.clusters)
tm = TreeMixture(num_clusters=args.clusters, num_nodes=args.nodes)
tm.simulate_pi(seed_val=args.seed)
tm.simulate_trees(seed_val=args.seed)
tm.sample_mixtures(num_samples=args.samples, seed_val=args.seed)
path = 'data/q_2_5_tm_'+str(args.nodes)+'node_'+str(args.samples)+'sample_'+str(
    args.clusters)+'clusters.pkl'
tm.save_mixture(path, True)
```

Kruskal Algorithm

```
# Code taken from https://www.geeksforgeeks.org/kruskals-minimum-spanning-tree-
    algorithm-greedy-algo-2/
# Python program for Kruskal's algorithm to find Minimum ST of a given connected,
    undirected and weighted graph
# # This code is contributed by Neelam Yadav


class Graph:
    # Class to represent a graph
    def __init__(self, vertices):
        self.V = vertices  # No. of vertices
        self.graph = []  # default dictionary to store graph

    # function to add an edge to graph
    def addEdge(self, u, v, w):
        self.graph.append([u, v, w])

    # A utility function to find set of an element i (uses path compression
    technique)
    def find(self, parent, i):
        if parent[i] == i:
            return i
        return self.find(parent, parent[i])

    # A function that does union of two sets of x and y (uses union by rank)
    def union(self, parent, rank, x, y):
        xroot = self.find(parent, x)
        yroot = self.find(parent, y)

        # Attach smaller rank tree under root of high rank tree (Union by Rank)
        if rank[xroot] < rank[yroot]:
            parent[xroot] = yroot
        elif rank[xroot] > rank[yroot]:
            parent[yroot] = xroot

        # If ranks are same, then make one as root and increment its rank by one
        else:
            parent[yroot] = xroot
            rank[xroot] += 1

    # The main function to construct MST using Kruskal's algorithm
    def KruskalMST(self):

        result = []  # This will store the resultant MST

        i = 0  # An index variable, used for sorted edges
        e = 0  # An index variable, used for result[]

        # Step 1:  Sort all the edges in non-decreasing order of their weight.
        # If we are not allowed to change the given graph, we can create a copy of
    graph
        self.graph = sorted(self.graph, key=lambda item: item[2])

        parent = []
        rank = []

        # Create V subsets with single elements
        for node in range(self.V):
            parent.append(node)
            rank.append(0)

        # Number of edges to be taken is equal to V-1
        while e < self.V - 1:

            # Step 2: Pick the smallest edge and increment the index for next
    iteration
            u, v, w = self.graph[i]
            i = i + 1
            x = self.find(parent, u)
            y = self.find(parent, v)

            # If including this edge does't cause cycle,
            # include it in result and increment the index of result for next edge
            if x != y:
                e = e + 1
                result.append([u, v, w])
                self.union(parent, rank, x, y)
            # Else discard the edge

        # print the contents of result[] to display the built MST
        print("Following are the edges in the constructed MST")
        for u, v, weight in result:
            print("%d -- %d == %d" % (u, v, weight))

    def maximum_spanning_tree(self):
        """ This function is the modified version of KruskalMST function.
            Given a graph with weighted edges, this function returns the maximum spanning tree. """

        #print("Running maximum spanning tree algorithm...")

        result = []  # This will store the resultant MST

        i = 0  # An index variable, used for sorted edges
        e = 0  # An index variable, used for result[]

        # Step 1:  Sort all the edges in non-INCREASING order of their weight.
        # If we are not allowed to change the given graph, we can create a copy of graph
        self.graph = sorted(self.graph, key=lambda item: item[2])[::-1]

        parent = []
        rank = []

        # Create V subsets with single elements
        for node in range(self.V):
            parent.append(node)
            rank.append(0)

        # Number of edges to be taken is equal to V-1
        while e < self.V - 1:

            # Step 2: Pick the LARGEST edge and increment the index for next iteration
            u, v, w = self.graph[i]
            i = i + 1
            x = self.find(parent, u)
            y = self.find(parent, v)

            # If including this edge does't cause cycle,
            # include it in result and increment the index of result for next edge
            if x != y:
                e = e + 1
                result.append([u, v, w])
                self.union(parent, rank, x, y)
            # Else discard the edge

        # print the contents of result[] to display the built MST
        #print("Following are the edges in the constructed MST")
        #for u, v, weight in result:
            #print("%d -- %d == %d" % (u, v, weight))

        return result


def main():
    print("Hello World!")
    print("This file demonstrates the usage of the functions.")
    print("The codes are taken from "
          "https://www.geeksforgeeks.org/kruskals-minimum-spanning-tree-algorithm-
    greedy-algo-2/.")

    print("\nCreate a graph and print it.")
    g = Graph(4)
    g.addEdge(0, 1, 10)
    g.addEdge(0, 2, 6)
    g.addEdge(0, 3, 5)
    g.addEdge(1, 3, 15)
    g.addEdge(2, 3, 4)
    print(g.graph)

    print("\nRun Kruskal's algorithm.")
    g.KruskalMST()

    print("\nRun maximum spanning tree algorithm.")
    g.maximum_spanning_tree()


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

The class `Tree.py` was also modified, but as the modification just consists of changing the return type when obtaining the *theta array* to `np.array`, it makes no sense to display the entire file here.

C Super epicentra – EM

Exercise 2.6

```python
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.mlab as mlab
from scipy.stats import multivariate_normal, poisson
import math


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]
        positions = np.dstack((X,Y))
        Z = multivariate_normal(mean, cov)
        plt.contour(X, Y, Z.pdf(positions), colors=col[i], linewidths=rates[i], alpha=0.1)

    plt.title(title)
    plt.tight_layout()


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

    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:
            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)
```

```python

        log_likelihood = 0
        self.converged = False

        for i in range(self.n_iter):
            self._do_estep(X, S)
            self._do_mstep(X, S)
            log_likelihood_new = self._compute_log_likelihood(X, S)

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

            log_likelihood = log_likelihood_new

        return self

    def _do_estep(self, X, S):
        num = np.zeros((self.n_row, self.n_components))
        denom = np.zeros(self.n_row)
        self.gamma = np.zeros((self.n_row, self.n_components))
        for i in range(self.n_row):
            for j in range(self.n_components):
                num[i,j] = self.weights[j] * multivariate_normal(self.means[j], self.covs[j]).pdf(X[i]) * poisson(self.rates[j]).pmf(S[i])
        self.gamma = (num.T / num.sum(1)).T

        return self

    def _do_mstep(self, X, S):
        n_k = self.gamma.sum(0)
        elems = self.n_row
        for i in range(self.n_components):
            for j in range(self.n_col):
                self.means[i,j] = sum(self.gamma[:,i] * X[:,j]) / n_k[i]
                diff = X[:,j] - self.means[i,j]
                self.covs[i,j,j] = sum(self.gamma[:,i] * diff * np.transpose(diff)) / n_k[i]
                self.rates[i] = sum(self.gamma[:,i] * S[:]) / n_k[i]
            self.weights[i] = n_k[i] / elems

        return self

    def _compute_log_likelihood(self, X, S):
        log_likelihood = 0
        for i in range(self.n_components):
            log_likelihood = sum(self.gamma[:,i] * (self.weights[i] *
    multivariate_normal(self.means[i], self.covs[i]).pdf(X[::]) * poisson(self.
    rates[i]).pmf(S[:])))
        log_likelihood = np.log(log_likelihood).sum()

        return log_likelihood

# 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()

# Plots of EM results #

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, means=em.means,covs=em.covs, title="Expectation Maximization: 3
     Components - 1 Iteration", rates=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, means=em.means,covs=em.covs, title="Expectation Maximization: 3
     Components - 50 Iterations", rates=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, means=em_2.means,covs=em_2.covs, title="Expectation
    Maximization: 2 Components - 1 Iteration", rates=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, means=em_2.means,covs=em_2.covs, title="Expectation
    Maximization: 2 Components - 50 Iterations", rates=em_2.rates)
plt.show()
```

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.probabilisticworld.com/conditional-dependence-independence/`, Accessed: 2019-12-11.

[5] *Mathematicalmonk*, `https://www.youtube.com/user/mathematicalmonk`, Accessed: 2019-12-30.