Probabilistic Graphical Models **MVA Fall 2019**

Homework 2 — December 31

Instructor: Pierre Latouche *Student: Antoine Moulin, Marie Heurtevent*

1 Classification: K-means, and the EM algorithm

Consider a mixture model, with K components, where datapoints X_i, $i = 1, ..., n$ have a probability p_k to be in component k: $P(Z_i = k) = p_k$, and, conditional on $Z_i = k$, $X_i \sim \mathcal{N}(\mu_k, D_k)$, a multivariate Gaussian distribution with mean μ_k, and diagonal (not full) covariance matrix D_k.

1. Derive the expressions of the parameter $\theta_k = (p_k, \mu_k, D_k)$ at each iteration of the corresponding EM algorithm.

   Solution.
   First, let's compute the likelihood of the problem at the t-th iteration.

$$\begin{aligned}
\ell_t &= \log p_\theta(X, Z) \\
&= \sum_{i=1}^{n} \log p_\theta(X_i, Z_i) \\
&= \sum_{i=1}^{n} \log p_\theta(Z_i) + \sum_{i=1}^{n} \log p_\theta(X_i|Z_i) \\
\ell_t &= \sum_{i=1}^{n}\sum_{j=1}^{K} z_i^j \log p_j + \sum_{i=1}^{n}\sum_{j=1}^{K} z_i^j \log \mathcal{N}(X_i|\mu_{j,t}, D_{j,t})
\end{aligned}$$

with $z_i^j = \begin{cases} 1 \text{ if } z_i = j \\ 0 \text{ otherwise} \end{cases}$ and $\mathcal{N}(X_i|\mu, D) = \frac{1}{(2\pi)^{\frac{d}{2}}|D|^{\frac{1}{2}}} \exp\{-\frac{1}{2}(x_i - \mu)^T D^{-1}(x_i - \mu)\}$.

During the E-step, we will write the expectation of the complete likelihood at step t with respect to the conditional distribution of $Z|X$.
Since, all terms of the sum except for z_i^j are constant with regards to $Z|X$, we need to compute $\mathbb{E}_{(Z|X)}(z_i^j)$.

$$\begin{aligned}
\mathbb{E}_{(Z|X)}(z_i^j) =& p_{\theta_t}(z_i = j|x_i) \\
=& \frac{p_{\theta_t}(z_i = j, x_i)}{p_{\theta_t}(x_i)} \\
=& \frac{p_{\theta_t}(z_i = j)p_{\theta_t}(x_i|z_i = j)}{p_{\theta_t}(x_i)} \\
=& \frac{p_j \mathcal{N}(x_i|\mu_j, D_j)}{\sum_{j'} p_{j'} \mathcal{N}(x_i|\mu_{j'}, D_{j'})} \\
:=& \tau_i^j(\theta_t)
\end{aligned}$$

Since the value of θ_t will be fixed during the M-step, we can drop the dependence on θ_t and write τ_i^j. Hence,

$$\mathbb{E}_{(Z|X)}(l_t) = \sum_{i=1}^{n}\sum_{j=1}^{K} \tau_i^j \log p_j + \sum_{i=1}^{n}\sum_{j=1}^{K} \tau_i^j \log \mathcal{N}(X_i|\mu_{j,t}, D_{j,t})$$

$$\mathbb{E}_{(Z|X)}(l_t) = \sum_{i=1}^{n}\sum_{j=1}^{K} \tau_i^j \log p_j + \sum_{i=1}^{n}\sum_{j=1}^{K} \tau_i^j \left[\log\left(\frac{1}{(2\pi)^{\frac{d}{2}}}\right) + \log\left(\frac{1}{|D_{j,t}|^{\frac{1}{2}}}\right) - \frac{1}{2}(x_i - \mu_{j,t})^T D_{j,t}^{-1}(x_i - \mu_{j,t}) \right]$$

During the M-step, we will maximise the previous expectation with respect to $\theta_t = (p_t, \mu_t, D_t)$. The expectation is split into two independent terms so we can start by first maximizing with regards to p_t:

$$\max_p \sum_{i=1}^{n}\sum_{j=1}^{K} \tau_i^j \log p_j$$

which gives us:

$$p_{j,t+1} = \frac{\sum_{i=1}^{n} \tau_i^j}{\sum_{i=1}^{n}\sum_{j'=1}^{K} \tau_i^{j'}}$$
$$p_{j,t+1} = \frac{1}{n}\sum_{i=1}^{n} \tau_i^j$$

because $\sum_{j'=1}^{K} \tau_i^{j'} = \sum_{j'=1}^{K} p_\theta(z_i = j'|x_i) = 1$.

We can now maximize the right term of the sum with regards to μ_t and D_t, which leaves us with:

$$\mu_{j,t+1} = \frac{\sum_{i=1}^{n} \tau_i^j x_i}{\sum_{i=1}^{n} \tau_i^j}$$
$$D_{j,t+1} = \frac{\sum_{i=1}^{n} \tau_i^j \left(x_i - \mu_{j,t+1}\right)\left(x_i - \mu_{j,t+1}\right)^T}{\sum_{i=1}^{n} \tau_i^j}$$

We can name $w_j^t := \sum_{i=1}^{n} \tau_i^j(\theta_t)$.
Hence, the updated expressions of the parameters at each iteration are

$$\boxed{\begin{aligned} p_{j,t+1} &= \frac{1}{n} w_j^t \\ \mu_{j,t+1} &= \frac{1}{w_j^t}\sum_{i=1}^{n} \tau_i^j x_i \\ D_{j,t+1} &= \frac{1}{w_j^t} Diag(x_i - \mu_{t+1})^2 \end{aligned}}$$

2. What may be the advantage of such a model, compared to the more standard Gaussian mixture model, where covariance matrices are full?

   Solution.
   In general, when the covariance matrices are full, there are $O\left(K\frac{d(d+1)}{2}\right)$ parameters. In this case, there are much fewer parameters : $O\left(Kd\right)$.
   For datasets that feature relatively independant features conditionally on the latent class, this approximation is quite valid with very similar performances, all the while retaining much fewer parameters.

3. Implement the algorithm, and compare the results with:

   - K-means
   - EM for a standard Gaussian mixture (full covariance)

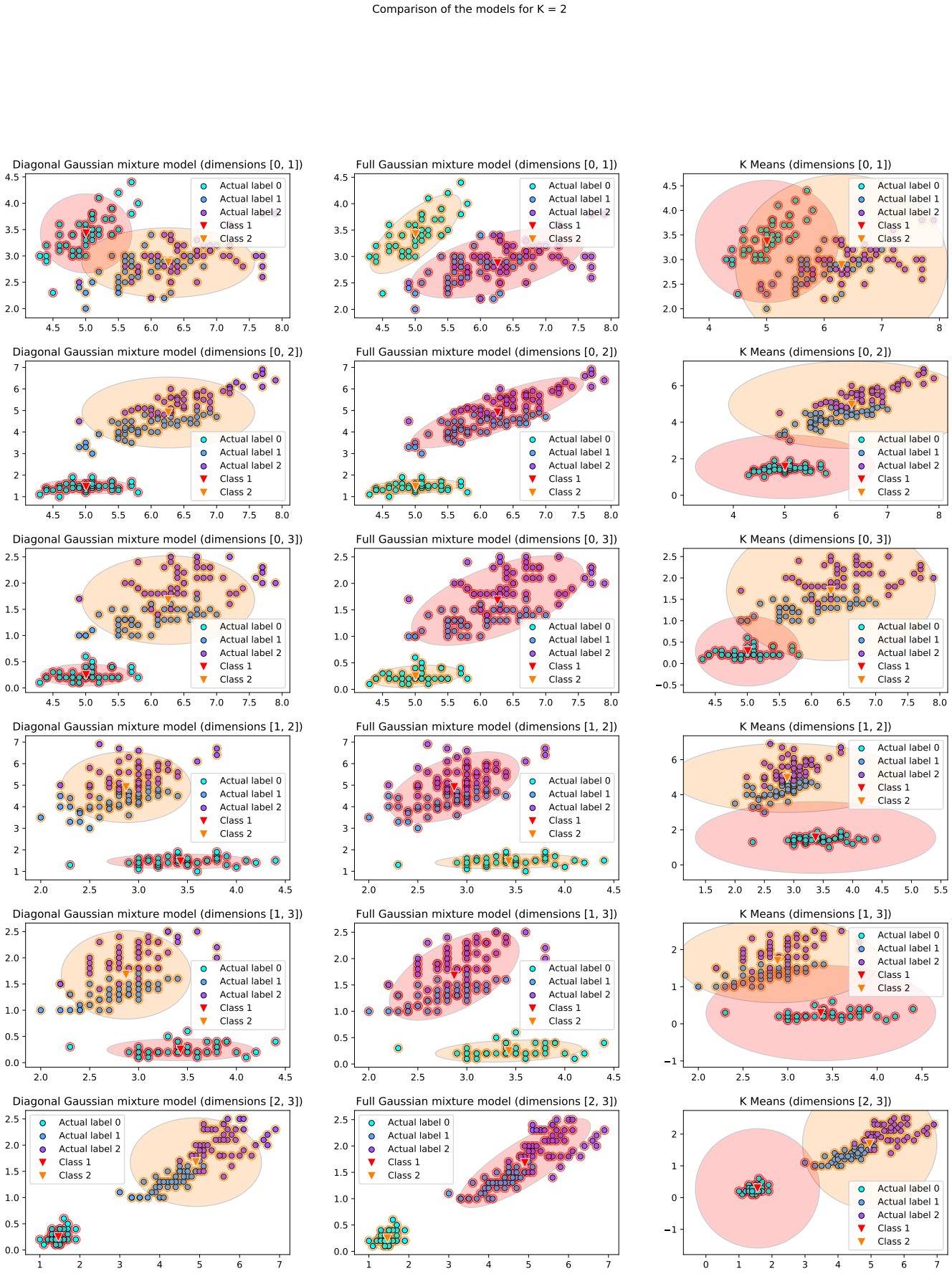
Comparison of the models for K = 2
Diagonal Gaussian mixture model (dimensions [0, 1])
Full Gaussian mixture model (dimensions [0, 1])
K Means (dimensions [0, 1])
Diagonal Gaussian mixture model (dimensions [0, 2])
Full Gaussian mixture model (dimensions [0, 2])
K Means (dimensions [0, 2])
Diagonal Gaussian mixture model (dimensions [0, 3])
Full Gaussian mixture model (dimensions [0, 3])
K Means (dimensions [0, 3])
Diagonal Gaussian mixture model (dimensions [1, 2])
Full Gaussian mixture model (dimensions [1, 2])
K Means (dimensions [1, 2])
Diagonal Gaussian mixture model (dimensions [1, 3])
Full Gaussian mixture model (dimensions [1, 3])
K Means (dimensions [1, 3])
Diagonal Gaussian mixture model (dimensions [2, 3])
Full Gaussian mixture model (dimensions [2, 3])
K Means (dimensions [2, 3])
Actual label 0
Actual label 1
Actual label 2
Class 1
Class 2

Figure 1.1

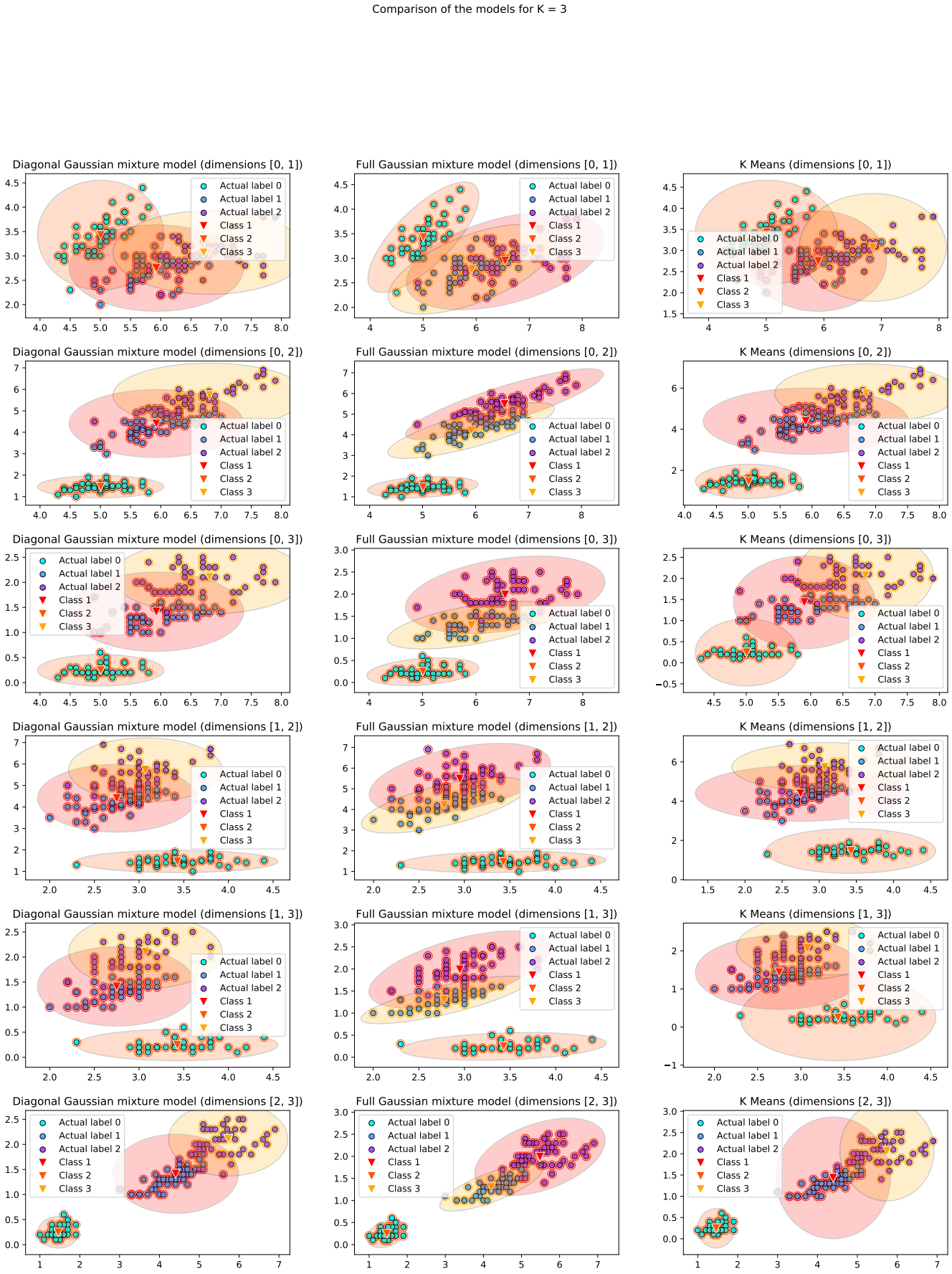
Comparison of the models for K = 3
Diagonal Gaussian mixture model (dimensions [0, 1])
Full Gaussian mixture model (dimensions [0, 1])
K Means (dimensions [0, 1])
Diagonal Gaussian mixture model (dimensions [0, 2])
Full Gaussian mixture model (dimensions [0, 2])
K Means (dimensions [0, 2])
Diagonal Gaussian mixture model (dimensions [0, 3])
Full Gaussian mixture model (dimensions [0, 3])
K Means (dimensions [0, 3])
Diagonal Gaussian mixture model (dimensions [1, 2])
Full Gaussian mixture model (dimensions [1, 2])
K Means (dimensions [1, 2])
Diagonal Gaussian mixture model (dimensions [1, 3])
Full Gaussian mixture model (dimensions [1, 3])
K Means (dimensions [1, 3])
Diagonal Gaussian mixture model (dimensions [2, 3])
Full Gaussian mixture model (dimensions [2, 3])
K Means (dimensions [2, 3])
Actual label 0
Actual label 1
Actual label 2
Class 1
Class 2
Class 3

Figure 1.2

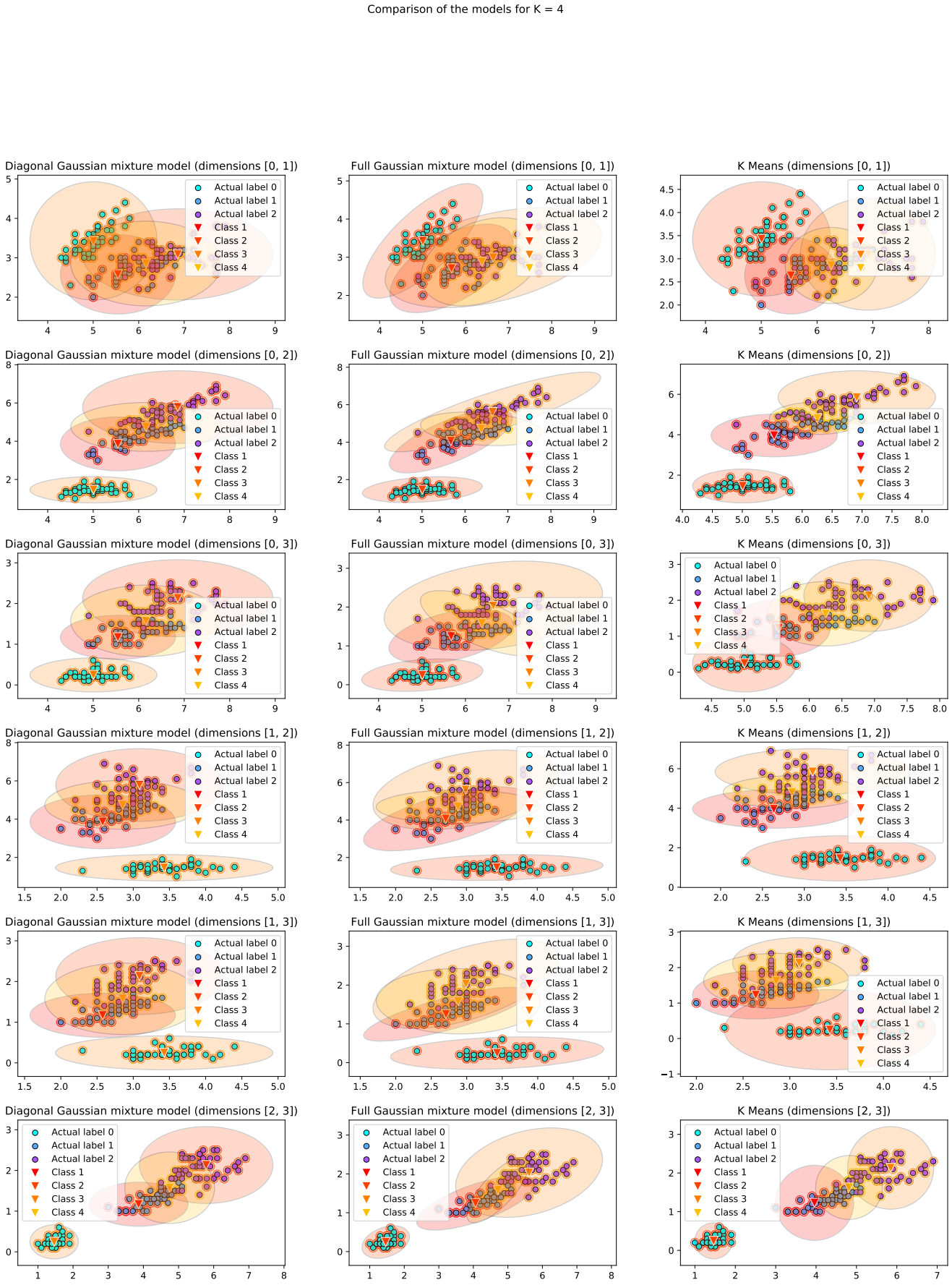
Comparison of the models for K = 4
Diagonal Gaussian mixture model (dimensions [0, 1])
Full Gaussian mixture model (dimensions [0, 1])
K Means (dimensions [0, 1])
Diagonal Gaussian mixture model (dimensions [0, 2])
Full Gaussian mixture model (dimensions [0, 2])
K Means (dimensions [0, 2])
Diagonal Gaussian mixture model (dimensions [0, 3])
Full Gaussian mixture model (dimensions [0, 3])
K Means (dimensions [0, 3])
Diagonal Gaussian mixture model (dimensions [1, 2])
Full Gaussian mixture model (dimensions [1, 2])
K Means (dimensions [1, 2])
Diagonal Gaussian mixture model (dimensions [1, 3])
Full Gaussian mixture model (dimensions [1, 3])
K Means (dimensions [1, 3])
Diagonal Gaussian mixture model (dimensions [2, 3])
Full Gaussian mixture model (dimensions [2, 3])
K Means (dimensions [2, 3])
Actual label 0
Actual label 1
Actual label 2
Class 1
Class 2
Class 3
Class 4

Figure 1.3

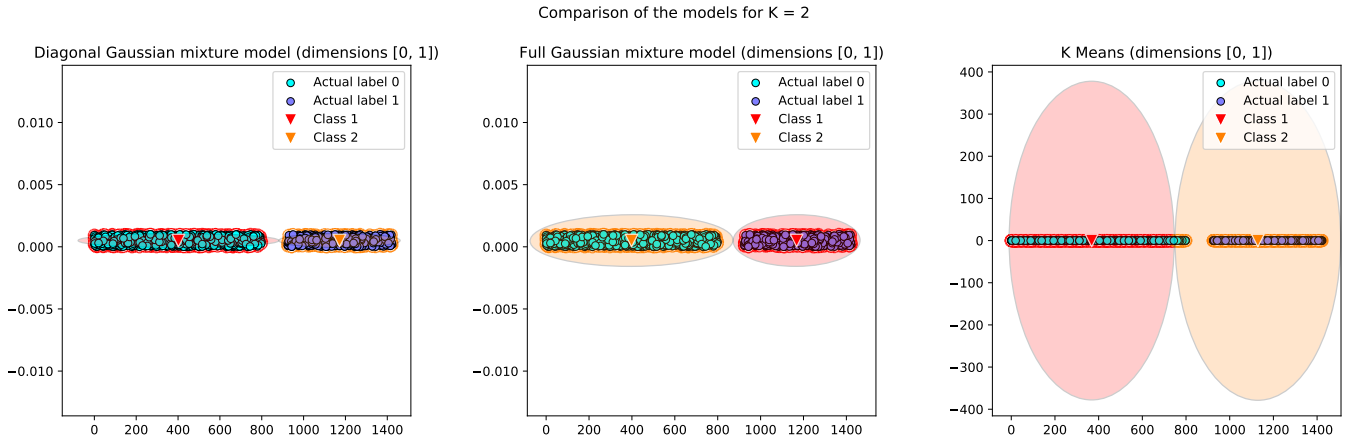

Figure 1.4

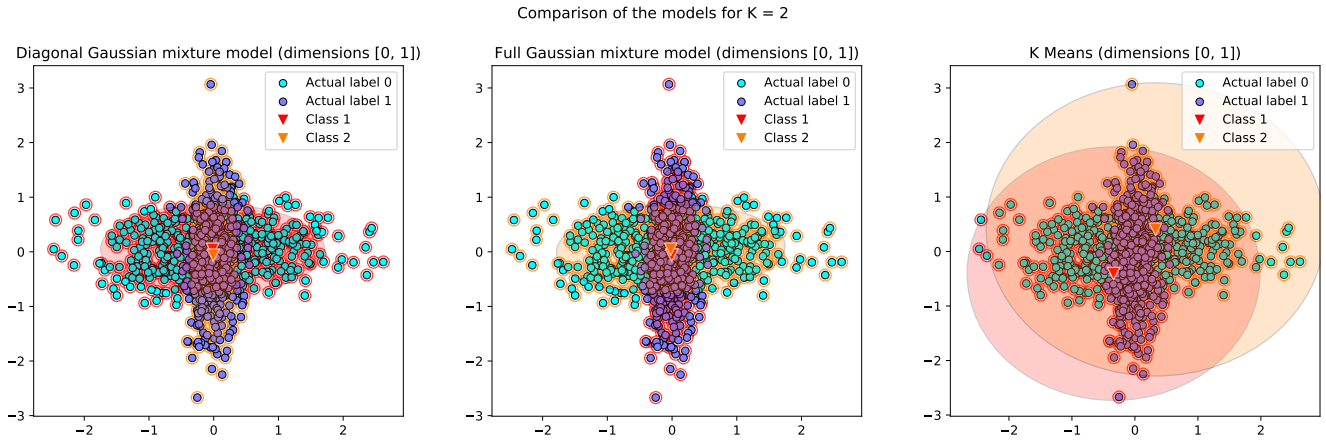

Figure 1.5

for the IRIS data set, and $K = 2, 3, 4$.

Solution.

Figures 1.1-1.2-1.3 , each for a specific $K \in \{2, 3, 4\}$, are made up of six lines for each possible pair of dimensions and and three columns comparing the confidence ellipses obtained via the diagonal Gaussian Mixture model and the full Gaussian Mixture model and the spherical clusters obtained by a K Means algorithm.

4. In which situations K-means is going to be significantly outperformed by the two EM algorithms discussed above?

Solution.

K-means performs well when the width of the different clusters are similar, since it works by minimizing the radius of each cluster. However, K-means has a lower performance than Gaussian mixtures when the widths are quite different, as seen on figure 1.4.

As well, K-means creates spheres, as opposed to the ellipses obtained with Gaussian mixtures (whether with diagonal or full covariance matrices). Therefore, Gaussian mixture models work much better than a K Means algorithm on elliptical data, as seen on figure 1.5.

2 Graphs, algorithms and Ising

To avoid over or underflow, it is often recommended to store and compute small quantities (such as probabilities, densities, etc.) on the ***log scale***. To multiply two quantities stored on the log-scale, we just add their logs. To add two such quantities, we use the log-sum trick: if $a \geq b$, compute $\log\left(e^a + e^b\right)$ as: $a + \log\left(1 + e^{b-a}\right)$; otherwise swap a and b.

1. Implement the sum-product algorithm for an undirected chain, using this trick, in order to compute all the forward and backward messages. (Explain how you represent the input of the algorithm, *i.e.*, the functions ψ_i and $\psi_{i,i+1}$).

   Solution. Let's consider $X_1, \ldots, X_n$ n random variables represented by an undirected chain and with joint distribution $p(x)$:

   $$p(x) = \frac{1}{Z} \prod_{i=1}^{n} \psi_i(x_i) \prod_{i=2}^{n} \psi_{i-1,i}(x_{i-1}, x_i)$$

   Recall that the forward and backward messages can be computed with:

   $$\mu_{j \to j+1}(x_{j+1}) = \sum_{x_j} \psi_j(x_j) \psi_{j,j+1}(x_j, x_{j+1}) \, \mu_{j-1 \to j}(x_j)$$
   $$\mu_{j \to j-1}(x_{j-1}) = \sum_{x_j} \psi_j(x_j) \psi_{j-1,j}(x_{j-1}, x_j) \, \mu_{j+1 \to j}(x_j)$$

   Then, for $j = 1, \ldots, n$, the messages can be used to compute the marginal distribution of the node j:

   $$p(x_j) = \frac{1}{Z} \mu_{j-1 \to j}(x_j) \, \psi_j(x_j) \, \mu_{j+1 \to j}(x_j)$$

   as well as the partition function:

   $$Z = \sum_{x_j} \mu_{j-1 \to j}(x_j) \, \psi_j(x_j) \, \mu_{j+1 \to j}(x_j)$$

   The functions $\psi_i, \psi_{i-1,i} \geq 0$ are the ***potentials***. In practice, they can be stored in arrays, e.g. if X_1 can take K_1 different values and X_2 can take K_2 different values, $\psi_1, \psi_2, \psi_{1,2}$ will be stored in three arrays with shape $(K_1,), (K_2,), (K_1, K_2)$, respectively. Note that it is the same whether the random variables are discrete or continuous because in the latter case, a discretized version is used. Besides, as indicated, the log-sum trick is used for the computations. Note that like multiplication operation in linear-scale becomes simple addition in log-scale; an addition operation in linear-scale becomes the log-sum-exp in the log-domain. Besides, the normalization changes. Consider a discrete probability distribution $(p_i)_i$, represented in log-scale as $(a_i)_i$ (*i.e.* $a_i = \log p_i$):

   $$\sum_i p_i = 1 \text{ iff } \sum_i e^{a_i} = 1$$
   $$\text{iff } \log \sum_i e^{a_i} = 0$$

   Hence, summing to one in the probability space is equivalent to have a log-sum-exp equal to 0 in the log-probability space. To make such a normalization, one can modify a_i as $a_i = a_i - LSE(a)$, where $LSE(a) = \log \sum_i e^{a_i}$.

   The code is available in the file `undirected_chain.py` where an example is shown.

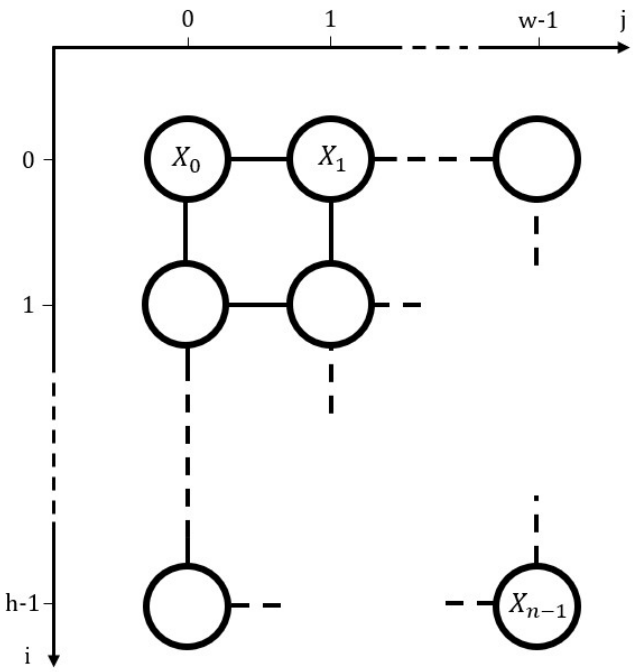

Figure 2.1: Graphical model associated to the Ising model. The grid has a size $h \times w$ and contains $n = h \times w$ vertices.

The Ising model assumes n binary variables $X_1, \ldots, X_n$, which are jointly distributed as follows:

$$p(x_1, \ldots, x_n) = \frac{1}{Z(\alpha, \beta)} \exp\left(\alpha \sum_i x_i + \beta \sum_{i \sim j} \mathbb{I}_{x_i = x_j}\right)$$

where the relation $i \sim j$ means that i and j are "neighbours". Specifically, each variable is associated to a point in a $2D$ grid, of size $h \times w$ (height times width), and two variables are neighbours if they are at distance one on that grid (*i.e.* immediately to the left, right, up or down).

2. For $w = 10, h = 100, \alpha = 0$, use your implementation from point 1 to compute ***exactly*** $Z(\alpha, \beta)$ as a function of β, and plot it. (Hint: recall the idea behind the junction tree algorithm).

   Solution. The graph $G = (V, E)$ corresponding to the Ising model is represented in figure 2.1. For simplicity and to fit with the code, the indices are translated from $[\![1, n]\!]$ to $[\![0, n-1]\!]$. Let $k \in [\![0, n-1]\!]$. There exists a pair $(i, j) \in [\![0, h-1]\!] \times [\![0, w-1]\!]$ such that $k = i \times w + j$. The node associated to X_k can be denoted by $v_{i,j}$ and corresponds to the node at the i-th row and the j-column. The idea here is to turn every row into a super-node to retrieve the case of the question 1 with an undirected chain that contains h nodes who can take 2^w different states. This choice comes from the fact that $w = 10 < 100 = h$ and this is more efficient to have 2^{10} possible states rather than 2^{100}. Let's rewrite the joint distribution:

$$\begin{aligned} p(x_0, \ldots, x_{n-1}) &= \frac{1}{Z(\alpha, \beta)} \exp\left(\alpha \sum_{k=0}^{n-1} x_k + \beta \sum_{k \sim l} \mathbb{I}_{x_k = x_l}\right) \\ &= \frac{1}{Z(\alpha, \beta)} \left(\prod_{k=0}^{n-1} e^{\alpha x_k}\right) \left(\prod_{k \sim l} e^{\beta \mathbb{I}_{x_k = x_l}}\right) \end{aligned}$$

   Switching to the new representation:

$$
\begin{aligned}
p\left(x_0, \ldots, x_{n-1}\right) &= \frac{1}{Z\left(\alpha, \beta\right)} \left(\prod_{i=0}^{h-1} \prod_{j=0}^{w-1} e^{\alpha v_{i,j}} \right) \left(\underbrace{\prod_{i=0}^{h-1} \prod_{j=0}^{w-2} e^{\beta \mathbb{I}_{v_{i,j}=v_{i,j+1}}}}_{\text{horizontal edges}} \underbrace{\prod_{i=0}^{h-2} \prod_{j=0}^{w-1} e^{\beta \mathbb{I}_{v_{i,j}=v_{i+1,j}}}}_{\text{vertical edges}} \right) \\
&= \frac{1}{Z\left(\alpha, \beta\right)} \prod_{i=0}^{h-1} \left(\prod_{j=0}^{w-1} e^{\alpha v_{i,j}} \prod_{j=0}^{w-2} e^{\beta \mathbb{I}_{v_{i,j}=v_{i,j+1}}} \right) \prod_{i=0}^{h-2} \left(\prod_{j=0}^{w-1} e^{\beta \mathbb{I}_{v_{i,j}=v_{i+1,j}}} \right) \\
&= \frac{1}{Z\left(\alpha, \beta\right)} \prod_{i=0}^{h-1} \phi_i\left(x_{R_i}\right) \prod_{i=0}^{h-2} \phi_{i,i+1}\left(x_{R_i}, x_{R_{i+1}}\right) \\
&\triangleq \tilde{p}\left(x_{R_0}, \ldots, x_{R_{h-1}}\right)
\end{aligned}
$$

where R_i is just the i-th, i.e. the set of nodes $\{v_{i,j}\}_{j \in [\![0, w-1]\!]}$. Hence, it is possible to use the undirected chain formed by the super-nodes $x_{R_0}, \ldots, x_{R_{h-1}}$ to compute $Z\left(\alpha, \beta\right)$. The result is shown in figure 2.2, with $\beta \in [-2, 2]$. This can be obtained by running the file `ising.py`, which contains both the class `Ising` and a small script to plot the log-partition function.

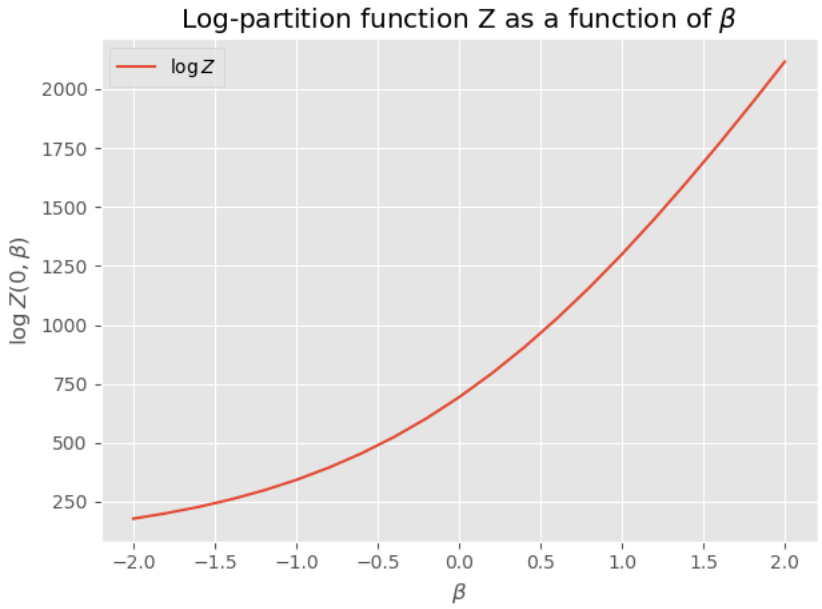

Figure 2.2: Log-partition function as a function of β and with $\alpha = 0$.

3. Implement loopy belief propagation to obtain a faster approximation of $Z\left(\alpha, \beta\right)$. (Explain.) For which values of β the approximation error gets larger?

   Solution. The implementation of the loopy belief propagation algorithm is available in the file `loopy_belief_propagation.py`. For each node, we send a message to its neighbors using the following formula and we iterate:

$$
\mu_{i \to j}\left(x_j\right) = \sum_{x_i} \psi_i\left(x_i\right) \psi_{i,j}\left(x_i, x_j\right) \prod_{k \in \mathcal{N}(i) \setminus \{j\}} \mu_{k \to i}\left(x_i\right)
$$

   The j-th marginal distribution becomes: $p\left(x_j\right) = \frac{1}{Z(\alpha,\beta)} \phi_i\left(x_i\right) \prod_{k \in \mathcal{N}(j)} \mu_{k \to j}\left(x_j\right)$.

   The result for ten passes is shown in figure 2.3. We see that it is very similar to the result obtained in the previous question, except that the values are different. Besides, the log-partition function obtained now depends on both the marginal used and the number of passes. For instance, with the top-left corner node and after 5 passes, the marginal is computed using the information of 15 nodes, where as with the node indexed by $(5, 5)$, 61 nodes are used. It corresponds to the 5-order neighborhood of the two nodes, respectively.

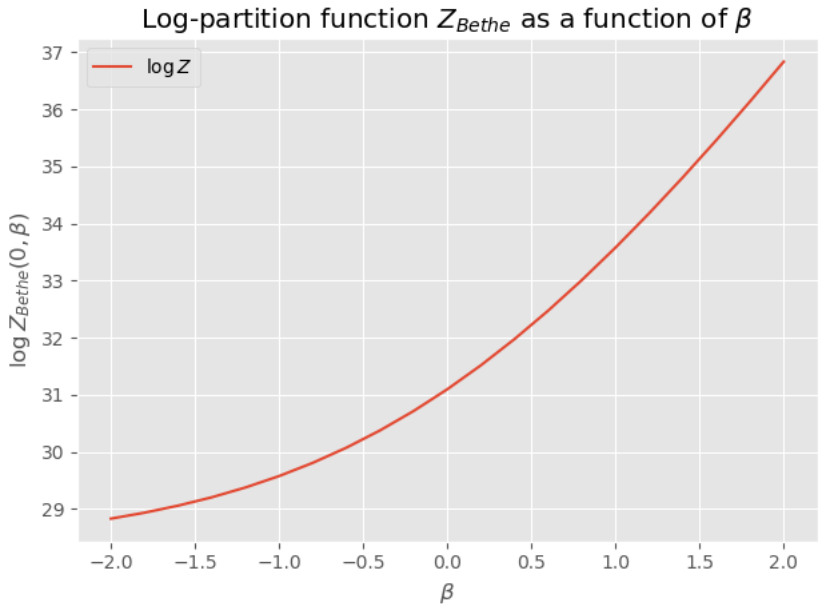

Figure 2.3: Approximation of the log-partition function as a function of β.

We see that when β is small, the difference between the two functions is small, and vice-versa. We do not really know what is expected here and it is not fully detailed in the lecture notes, but in the literature, it is possible to find some formulas computing the quantity $\frac{Z}{Z_{Bethe}}$, where Z_{Bethe} is the Bethe approximation obtained with the loopy belief algorithm. This quantity involves the so-called "beliefs" terms.