Graphical Models Assignment

Team 17

Student IDs
22197823
22192584
22095744

Contents

1 Inference and learning

1.1 Two earthquakes

Part A

Figure 1 is a belief network representation of two explosion events, where s_1 and s_2 are the locations of each explosion, and v_i is the observed signal at sensor i with noise from $\mathcal{N}(0, \sigma)$. This is using the assumption that the observed sensor values are independent (given the explosion locations). Therefore we have:

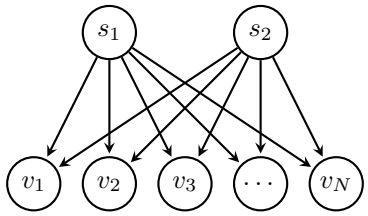
$$\begin{aligned} p(s_1, s_2, \mathbf{v}) &= p(s_1)p(s_2)\prod_{i=1}^{N} p(v_i|s_1, s_2) \\ &= p(s_1)p(s_2)\prod_{i=1}^{N} \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{1}{2\sigma^2}\left(v_i - \frac{0.5}{d_i^2(1)+0.1} - \frac{0.5}{d_i^2(2)+0.1}\right)^2} \end{aligned} \tag{1}$$

Where $d_i(j)$ is the L2 norm distance between sensor i and explosion site s_j. Similar to Example 1.12 we assume the explosion location priors are constant and uniform over the world space; $p(s_1) = p(s_2) = p_s$. Our aim is to plot the posterior of $p(s_1|\mathbf{v})$. Our first step is using the constant priors with Bayes formula, and marginalising over s_2 we get the following:

$$p(s_1|\mathbf{v}) = \frac{p_s^2}{p(\mathbf{v})} \sum_{s_2} \prod_{i=1}^{N} p(v_i|s_1, s_2)$$

This is a distribution conditioned on $p(\mathbf{v})$ and therefore it is not dependent on s_1, so both $p(\mathbf{v})$ and p_s are constants. So we have:

Figure 1 Belief Network diagram of 2 explosion events



$$
\begin{aligned}
p(s_1|\mathbf{v}) &\propto \sum_{s_2}\prod_{i=1}^{N} p(v_i|s_1,s_2) \\
&\propto \sum_{s_2}\prod_{i=1}^{N} \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{1}{2\sigma^2}\left(v_i-\frac{0.5}{d_i^2(1)+0.1}-\frac{0.5}{d_i^2(2)+0.1}\right)^2} \\
&\propto \sum_{s_2}\prod_{i=1}^{N} e^{-\frac{1}{2\sigma^2}\left(v_i-\frac{0.5}{d_i^2(1)+0.1}-\frac{0.5}{d_i^2(2)+0.1}\right)^2}
\end{aligned} \tag{2}
$$

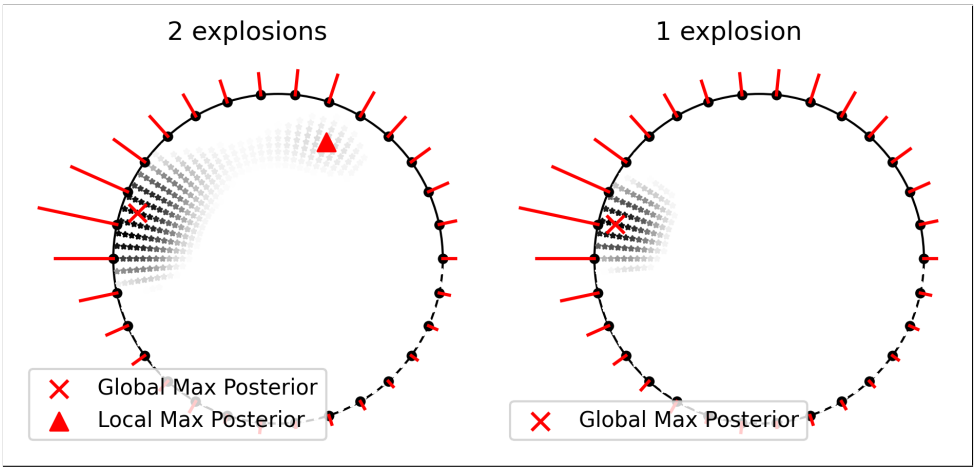


Figure 2 Posterior distribution for the location of explosions. Left figure assuming 2 explosions. Right figure assuming 1 explosion. Darker spots correspond to higher probability of explosion location. Observed signals at each sensor are represented by the red radial bars at each sensor location

Using *equation 2* we can calculate the posterior. To increase numerical stability we calculate the log of the posterior, otherwise the product of the exponentials can vanish to zero. A plot of the posterior can be seen in *figure 2*. A plot is also shown using the same data but assuming 1 explosion. In the plot on the left (assuming 2 explosions) the global maximum and a local maxima of the posterior are marked, indicating possible locations for both explosions. The plots were generated using code [*Appendix 1: Code: 1*].

Part B

We want to calculate $\log(p(\mathbf{v}|H_2)) - \log(p(\mathbf{v}|H_1))$ where H_1 and H_2 are the hypothesis that there is 1 and 2 explosions respectively. In *Part A* we were assuming there were 2 explosions, therefore all formulas were already conditioned on H_2. Therefore *equation 1* is equal to $p(\mathbf{v}, s_1, s_2|H_2)$. Using this we have:

$$p(\mathbf{v}, s_1, s_2|H_2) = p(s_1)p(s_2)\prod_{i=1}^{N} p(v_i, s_1, s_2|H_2)$$

$$p(\mathbf{v}|H_2) = p_s^2 \sum_{s_1,s_2}\prod_{i=1}^{N} p(v_i, s_1, s_2|H_2)$$

$$\log(p(\mathbf{v}|H_2)) = 2\log(p_s) + \log\left(\sum_{s_1,s_2}\prod_{i=1}^{N}\frac{1}{\sqrt{2\pi\sigma^2}}e^{-\frac{1}{2\sigma^2}\left(v_i - \frac{0.5}{d_i^2(1)+0.1} - \frac{0.5}{d_i^2(2)+0.1}\right)^2}\right) \quad (3)$$

Similarly, conditioned under H_1 we have:

$$\log(p(\mathbf{v}|H_1)) = \log(p_s) + \log\left(\sum_{s_1}\prod_{i=1}^{N}\frac{1}{\sqrt{2\pi\sigma^2}}e^{-\frac{1}{2\sigma^2}\left(v_i - \frac{1}{d_i^2(1)+0.1}\right)^2}\right) \quad (4)$$

But calculating *equations 3* and *4* directly is numerically unstable due to the product of exponentials being small and vanishing to zero. Taking the log doesn't help unless a calculation method of taking the maximum exponential power out of the log sum, this method is shown in detailed in *Barber, 2012, p90* [1]. An example of this method is shown below with the simple case where we want to calculate A

$$A = \log\left(\sum_x e^{f(x)}\right) \quad (5)$$

let $\tilde{f} = \max_x f(x)$. Note that $\tilde{f}$ does not depend on x and therefore can be removed from the sum:

$$= \log\left(\sum_x e^{f(x)-\tilde{f}+\tilde{f}}\right)$$

$$A = \tilde{f} + \log\left(\sum_x e^{f(x)-\tilde{f}}\right)$$

By construction $f(x) - \tilde{f} \leq 1, \forall x$, with at least one term being equal to 1, which allows the dominant terms to be computed accurately.

Equations 3 and *4* are in the same form as *Equation 5*. Using this method we have the following result (generated from [*Appendix 1: Code: 1*]):
$\log(p(\mathbf{v}|H_2)) = -772.69106$
$\log(p(\mathbf{v}|H_1)) = -767.58319$
$\log(p(\mathbf{v}|H_2)) - \log(p(\mathbf{v}|H_1)) =$ **-5.10787**

Part C

To help answer this question, let E be a variable representing how many explosions there are. Therefore we have $p(\mathbf{v}|H_2) = p(\mathbf{v}|E = 2)$, and $p(\mathbf{v}|H_1) = p(\mathbf{v}|E = 1)$. Therefore we have:

$$\begin{aligned} \log(p(\mathbf{v}|H_2)) - \log(p(\mathbf{v}|H_1)) &= \log\left(\frac{p(\mathbf{v}|E=2)}{p(\mathbf{v}|E=1)}\right) \\ &= \log\left(\frac{p(E=1)p(\mathbf{v},E=2)}{p(E=2)p(\mathbf{v},E=1)}\right) \end{aligned} \tag{6}$$

Assuming we have no prior preference, then $p(H_1) = p(H_2) \implies p(E = 1) = p(E = 2)$. Using this and applying Bayes' formula again we get:

$$\begin{aligned} \log(p(\mathbf{v}|H_2)) - \log(p(\mathbf{v}|H_1)) &= \log\left(\frac{p(\mathbf{v},E=2)}{p(\mathbf{v},E=1)}\right) \\ &= \log\left(\frac{p(\mathbf{v})p(E=2|\mathbf{v})}{p(\mathbf{v})p(E=1|\mathbf{v})}\right) \\ &= \log\left(\frac{p(E=2|\mathbf{v})}{p(E=1|\mathbf{v})}\right) \end{aligned}$$

We've just shown *Equation 6* is the log of the ratio of probabilities having 2 explosions compared to 1 explosion. Relating this back to our solution in *Part B*, if we have no prior preference then log-odds of -5.10787 would imply with the observed earthquake signal data it is more likely there was only 1 explosion instead of 2.

Part D

Assuming we have k explosions, we can extend *Equation 3* to the following (still assuming $p(s_j) = p_s$ is constant for all j):

$$\log(p(\mathbf{v}|H_k)) = k\log(p_s) + \log\left(\sum_{s_1,\ldots s_k}\prod_{i=1}^{N}\frac{1}{\sqrt{2\pi\sigma^2}}e^{-\frac{1}{2\sigma^2}\left(v_i-\frac{1}{k}\sum_{j=1}^{k}\frac{1}{d_i^2(j)+0.1}\right)^2}\right) \tag{7}$$

As before we are assuming there can be multiple explosions at the same world point, ie. s_i can be equal to s_j. From the above we can see the iterative sums over $s_1, \ldots, s_k$ are the cause of computational complexity for each additional explosion. Because we are summing over all possible configurations of $(s_1, s_2, \ldots, s_k)$, and each s_j can take S different values (where S is the number of discrete world points in our model), naively we have S^k different states of $(s_1, \ldots, s_k)$, which would give us a computational complexity of $O(S^k)$. But the product in *Equation 7* is symmetric, ie. the product only cares how many explosions are at each location, not where each specific explosion is. For example the product will have the same value for state $(s_1, \ldots, s_k) = (l_1, l_2, l_2, \ldots, l_2)$ as it does for state $(s_1, \ldots, s_k) = (l_2, l_1, l_2, \ldots, l_2)$,

and there are k many states that have one explosion at location l_1 and $k-1$ explosions at location l_2. Therefore we can use this to our advantage to reduce the number of calculations and only calculate the product for unique unordered states and multiply by the number of times that state can happen. So there are $\binom{n+k-1}{k}$ unique ways to choose $(s_1, \ldots, s_k)$ (unordered sampling with replacement). So the complexity of *Equation 7* is $O(\binom{n+k-1}{k})$.

1.2 Meeting scheduling

Part (a)

From the information given in the question, the following probability distribution can be constructed, displayed here in a table:

Delay D_i	Probability
$D_i \leq 0$	0.7
$0 \leq D_i < 5$	0.1
$5 \leq D_i < 10$	0.1
$10 \leq D_i < 15$	0.07
$15 \leq D_i < 20$	0.02
$20 \leq D_i$	0.01

This then gives the cumulative density function (CDF):

Delay $D_i < t$	$P(D_i < t)$
$D_i \leq 0$	0.7
$D_i < 5$	0.8
$D_i < 10$	0.9
$D_i < 15$	0.97
$D_i < 20$	0.99
$D_i < \infty$	1

I am trying to find what time, T_0, I should tell my friends to meet so that there is at least a 0.9 probability we catch the train. As the train is at 21:05,

$$T_0(N) = (21:05) - t_0(N)$$

where $t_0(N)$ is the amount of time I should allow for my friends expected delays, and N is the number of friends going on the trip. Independence is assumed for each friend, therefore, the probability that all N friends are delayed by less than

some time t is given by

$$\begin{aligned} P(D_1, .., D_N < t) &= P(D_1 < t)...P(D_N < t) \\ &= \prod_{j=1}^{N} P(D_j < t) \\ &= [P(D_i < t)]^N \end{aligned}$$

As each friend has the same CDF, we create another table for the CDF of the joint distribution $P(D_i < t)^N$ for $N = 3$, $N = 5$ and $N = 10$

t	$P(D_i < t)^3$	$P(D_i < t)^5$	$P(D_i < t)^{10}$
0	0.343	0.168	0.028
5	0.512	0.33	0.11
10	0.729	0.59	0.35
15	0.913	0.859	0.737
20	0.97	0.95	0.904

Assuming I will ask my friends to meet in increments of 5 minutes, the joint CDF table can be used to find $t_0(N)$ which is the smallest value of t such that $P(D_i < t)^N \geq 0.9$ for $N = 3$, $N = 5$ and $N = 10$.

These values of $P(D_i < t)^N$ have been highlighted in the table for $t = t_0(N)$. For $N = 3$, $t_0(3) = (00 : 15)$ and therefore I need to ask my friends to meet 15 minutes before train leaves. For $N = 5$ and $N = 10$, $t_0(5) = t_0(10) = (00 : 20)$ so I will ask them to meet 20 minutes before. This ensures there is at least 0.9 probability that we all make the train. Hence,

$$\begin{aligned} T_0(3) &= (21:05) - (00:15) = 20:50 \\ T_0(5) &= (21:05) - (00:20) = 20:45 \\ T_0(10) &= (21:05) - (00:20) = 20:45 \end{aligned}$$

Part (b)

Updating the model with the unobserved binary variables Z_i, for punctual ($Z_i = 1$) and not punctual ($Z_i = 0$), gives the following conditional distribution:

Delay D_i	$P(D_i \mid Z_i = 1)$	$P(D_i \mid Z_i = 0)$
$D_i \leq 0$	0.7	0.5
$0 \leq D_i < 5$	0.1	0.2
$5 \leq D_i < 10$	0.1	0.1
$10 \leq D_i < 15$	0.07	0.1
$15 \leq D_i < 20$	0.02	0.05
$20 \leq D_i$	0.01	0.05

This then gives the conditional CDF:

Delay $D_i < t$	$P(D_i < t \mid Z_i = 1)$	$P(D_i < t \mid Z_i = 0)$
$D_i \leq 0$	0.7	0.5
$D_i < 5$	0.8	0.7
$D_i < 10$	0.9	0.8
$D_i < 15$	0.97	0.9
$D_i < 20$	0.99	0.95
$D_i < \infty$	1	1

To calculate the marginal probabilities, I must use

$$P(D_i < t) = P(D_i < t \mid Z_i = 1)P(Z_i = 1) + P(D_i < t \mid Z_i = 0)P(Z_i = 0)$$

where $P(Z_i = 1) = \frac{2}{3}$ and $P(Z_i = 0) = \frac{1}{3}$.

I want to know the probability of missing the train, given the times $T_0(N)$ stated in part (a). The probability of missing the train is 1 minus the probability of catching the train: $P(M \mid t_0) = 1 - P(M' \mid t_0)$, where M is the event of missing the train and M' is therefore the event of catching it. Catching the train means every friend being delayed less than the allowed amount of time,

$$P(M' \mid t_0) = P(D_i < t_0 \ \forall i) = P(D_i < t_0)^N$$

Hence, to calculate this for $N = 3$, the probability of one friend catching the train is:

$$\begin{aligned} P(D_i < 15) &= P(D_i < 15 \mid Z_i = 1)(\frac{2}{3}) + P(D_i < 15 \mid Z_i = 0)(\frac{1}{3}) \\ &= 0.97(\frac{2}{3}) + 0.9(\frac{1}{3}) \\ &= 0.947 \end{aligned}$$

and, therefore, the probability that all three friends catch the train is

$$\begin{aligned} P(M' \mid t_0 = 15) &= P(D_i < 15)^3 \\ &= 0.947^3 \\ &= 0.848 \end{aligned}$$

Thus, the probability of missing the train is

$$\begin{aligned} P(M \mid t_0 = 15) &= 1 - P(M' \mid t_0 = 15) \\ &= 1 - 0.848 \\ &= 0.152 \end{aligned}$$

Using a similar logic, I can find this probability for $N = 5$ and $N = 10$. The

probability of one friend catching the train is

$$\begin{aligned} P(D_i < 20) &= P(D_i < 20 \mid Z_i = 1)(\frac{2}{3}) + P(D_i < 20 \mid Z_i = 0)(\frac{1}{3}) \\ &= 0.99(\frac{2}{3}) + 0.95(\frac{1}{3}) \\ &= 0.977 \end{aligned}$$

and therefore the probability the all friends catch the train is $0.977^5 = 0.889$ for five friends and $0.977^{10} = 0.79$ for 10 friends. Thus, the probability of missing the train is $1 - 0.889 = 0.111$ for 5 friends and $1 - 0.79 = 0.21$ for 10 friends.

Bonus

I asked 5 friends to meet 20 minutes early for the train, and we missed it. To find how many of my friends are not punctual, I need to find the posterior distribution

$$\begin{aligned} P(B = c \mid M) &= \frac{P(B = c, M)}{P(M)} \\ &= \frac{P(M \mid B = c)P(B = c)}{P(M)} \end{aligned}$$

where B is the number of friends who are not punctual, and M is the event of missing the train. I know from part (b) that $P(M) = 0.111$, therefore I need to calculate the condition probability $P(M \mid B = c)$ and the marginal probability $P(B = c)$.

Firstly, the conditional probability $P(M \mid B = c)$ can once again be rewritten in terms of the probability of catching the train, that is

$$P(M \mid B = c) = 1 - P(M' \mid B = c)$$

where

$$P(M' \mid B = c) = P(M' \mid Z_i = 0)^c P(M' \mid Z_i = 1)^{5-c}$$

is the product of the probability of each friend catching the train given whether they are punctual or not punctual. These values can be read from the previous table as $P(M' \mid Z_i = 0) = 0.95$ and $P(M' \mid Z_i = 1) = 0.99$ for $D_i < 20$, and therefore we have

$$P(M' \mid B = c) = 0.95^c 0.99^{5-c}$$

Secondly, the marginal probability of the number of friends that are not punctual B follows the Binomial distribution

$$B \sim Bin(n = 5, p = \frac{1}{3})$$

hence

$$P(B=c) = \binom{5}{c}\left(\frac{1}{3}\right)^{c}\left(\frac{2}{3}\right)^{5-c}$$

Thus, putting this all together, we get the posterior distribution:

$$P(B=c\,|\,M) = \frac{(1-0.95^{c}0.99^{5-c})\binom{5}{c}(\frac{1}{3})^{c}(\frac{2}{3})^{5-c}}{0.11}$$

which gives the following distribution

c	$P(B=c\,\|\,M)$
0	0.059
1	0.261
2	0.372
3	0.239
4	0.072
5	0.008

I can see from this table that the most likely scenario is that 2 of my friends are not punctual.

1.3 Three weather stations

Part A

Here we have a set of $N = 500$ sequences $\mathcal{V} = \{v^{n}_{1:T}, n = 1, \ldots, N\}$ where all sequences are of the same length $T = 100$.

This problem can be thought of as a mixture of first order Markov chains whereby the discrete hidden variable h with $\text{dom}(h) = \{1, 2, 3\}$ indexes the Markov chain $\prod_t p(v_t|v_{t-1}, h)$ with $\text{dom}(v) = \{0, 1, 2\}$. The graphical model for this problem can be found in figure 3.

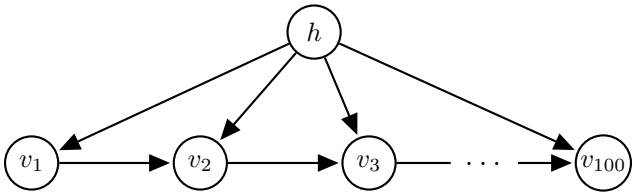


Figure 3 Mixture of first order Markov chains

Assuming the data are i.i.d so that $p(\mathcal{V}) = \prod_n p(v^{n}_{1:T})$, we define a mixture model for a single sequence $v_{1:T}$ such that

$$p(v_{1:T}) = \sum_{h=1}^{H} p(h)p(v_{1:T}|h) = \sum_{h=1}^{H} p(h)p(v_1|h)\prod_{t=2}^{T} p(v_t|v_{t-1}, h) \tag{8}$$

The parameters of this model are therefore $p(h)$ and $p(v_t|v_{t-1}, h)$ as well as an initial term $p(v_1|h)$.

The Expectation Maximisation (EM) algorithm is an iterative algorithm used for maximum likelihood estimation of the parameters of a model with hidden variables or missing data. In this scenario it will be used to estimate the parameters listed above. Once the parameters have been learned, the sequences can then be assigned to a particular station according to $p(h|v^n_{1:T})$. The steps of the EM-algorithm are as follows, using terminology found in [1]:

1. Initialise the parameters.

2. E-step. The posterior distribution is set according to

$$p^{old}(h|v^n_{1:T}) \propto p(h)p(v^n_{1:T}|h) = p(h)\prod_{t=1}^{T} p(v^n_t|v^n_{t-1}, h). \tag{9}$$

3. M-step. The goal here is to update the model parameters in order to maximise the energy term E given by

$$E = \sum_{n=1}^{N} \langle \log p(v^n_{1:T}, h) \rangle_{p^{old}(h|v^n_{1:T})} \tag{10}$$

$$= \sum_{n=1}^{N} \left\{ \langle \log p(h) \rangle_{p^{old}(h|v^n_{1:T})} + \sum_{t=1}^{T} \langle \log p(v_t|v_{t-1}, h) \rangle_{p^{old}(h|v^n_{1:T})} \right\} \tag{11}$$

   By considering the contribution to E from each parameter we get the following update rules

$$p^{new}(h) \propto \sum_{n=1}^{N} p^{old}(h|v^n_{1:T}) \tag{12}$$

$$p^{new}(v_t = i|v_{t-1} = j, h = k) \propto \sum_{n=1}^{N} p^{old}(h = k|v^n_{1:T}) \sum_{t=2}^{T} \mathbb{I}[v^n_t = i]\mathbb{I}[v^n_{t-1} = j] \tag{13}$$

$$p^{new}(v_1 = i|h = k) \propto \sum_{n=1}^{N} p^{old}(h = k|v^n_{1:T})\mathbb{I}[v^n_1 = i] \tag{14}$$

The E-step and M-step are then iteratively performed until convergence or a maximum number of iterations is reached.

Part B

The EM-algorithm as described above and implemented in the code found in appendix 2 (adapted from the BRML toolbox provided with [1]) yields the

following results

h	$p(h)$
1	0.2997
2	0.192
3	0.5083

Table 1: Learned $p(h)$

v_1	$p(v_1\|h=1)$	$p(v_1\|h=2)$	$p(v_1\|h=3)$
0	0.1921	0.4271	0.4612
1	0.4193	0.5729	0.3312
2	0.3886	0	0.2076

Table 2: Learned $p(v_1|h)$

v_t	$p(v_t\|v_{t-1}=0,h=1)$	$p(v_t\|v_{t-1}=1,h=1)$	$p(v_t\|v_{t-1}=2,h=1)$
0	0.0597	0.1301	0.4176
1	0.1395	0.3238	0.4278
2	0.8008	0.5461	0.1546

Table 3: Learned $p(v_t|v_{t-1}, h = 1)$

v_t	$p(v_t\|v_{t-1}=0,h=2)$	$p(v_t\|v_{t-1}=1,h=2)$	$p(v_t\|v_{t-1}=2,h=2)$
0	0.3887	0.2406	0.5453
1	0.1485	0.5155	0.0176
2	0.4628	0.244	0.4371

Table 4: Learned $p(v_t|v_{t-1}, h = 2)$

v_t	$p(v_t\|v_{t-1}=0,h=3)$	$p(v_t\|v_{t-1}=1,h=3)$	$p(v_t\|v_{t-1}=2,h=3)$
0	0.0705	0.1368	0.5108
1	0.4947	0.2259	0.4372
2	0.4348	0.6374	0.052

Table 5: Learned $p(v_t|v_{t-1}, h = 3)$

The log likelihood for these parameters is -45254.5104.

Part C

The EM-algorithm will converge to local optima depending on the initial values set for the parameters. In order to be confident that a global maximum has been

n	$h=1$	$h=2$	$h=3$
1	0.000	1.000	0.000
2	0.9998	0.000	0.0002
3	0.000	0.000	1.000
4	0.000	0.000	1.000
5	0.000	0.000	1.000
6	1.000	0.000	0.000
7	0.000	1.000	0.000
8	0.000	0.000	1.000
9	0.000	0.000	1.000
10	0.000	0.000	1.000

Table 6: Posterior distribution $p(h|v_{1:T}^n)$

reached, it's recommended to run the algorithm several times, each with different initial values, and select the learned parameters with the highest likelihood.

The initialisation strategy we employed was to randomly initialise all parameter values from a uniform distribution on $[0, 1)$. The values were then normalised to ensure a valid distribution over hidden states or observed value transitions.

An alternative method of initialisation we tried was to generate an empirical distribution on $p(v_t|v_{t-1}, h)$ using the count of observed value transitions in the given data and use this as the initialised value, keeping it uniform across different values of h. This strategy converged to the same values as the random initialisation in roughly the same number of iterations, as shown in figure 4, however it has the drawback of using more computational power to calculate the prior informed by the data.

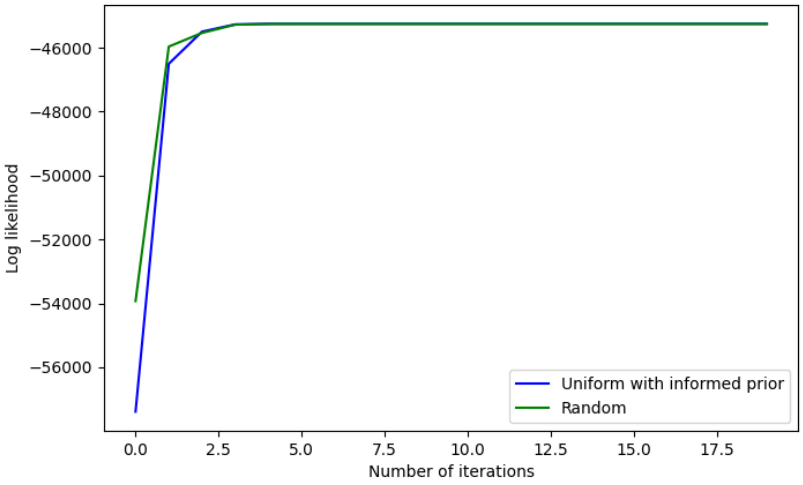


Figure 4 Log likelihood of learned parameters from EM-algorithm with different initialisation strategies.

When working with long sequences of data, as is the case in this example, it

can be common to encounter numerical underflow issues due to calculating the product of many probabilities. To mitigate this issue we instead used logs so we work with summations rather than products. This means that instead of equation 9 we use the following

$$\log p^{old}(h|v_{1:T}^n) = \log p(h) + \sum_{t=1}^{T} \log p(v_t^n|v_{t-1}^n, h) \tag{15}$$

Part D

If all parameter values are initialised uniformly, e.g. $p(h = 1) = p(h = 2) = p(h = 3) = 1/3$, the following parameters are learned.

h	$p(h)$
1	0.3333
2	0.3333
3	0.3333

Table 7: Learned $p(h)$

v_1	$p(v_1\|h = 1)$	$p(v_1\|h = 2)$	$p(v_1\|h = 3)$
0	0.374	0.374	0.374
1	0.404	0.404	0.404
2	0.222	0.222	0.222

Table 8: Learned $p(v_1|h)$

v_t	$p(v_t\|v_{t-1} = 0, h = 1)$	$p(v_t\|v_{t-1} = 1, h = 1)$	$p(v_t\|v_{t-1} = 2, h = 1)$
0	0.1598	0.1445	0.487
1	0.306	0.2835	0.3502
2	0.5343	0.572	0.1628

Table 9: Learned $p(v_t|v_{t-1}, h = 1)$

v_t	$p(v_t\|v_{t-1} = 0, h = 2)$	$p(v_t\|v_{t-1} = 1, h = 2)$	$p(v_t\|v_{t-1} = 2, h = 2)$
0	0.1598	0.1445	0.487
1	0.306	0.2835	0.3502
2	0.5343	0.572	0.1628

Table 10: Learned $p(v_t|v_{t-1}, h = 2)$

The log likelihood for these parameters is -49473.5473. Since this value is less than that achieved with random initialisation, we may conclude that uniform

v_t	$p(v_t\|v_{t-1}=0,h=3)$	$p(v_t\|v_{t-1}=1,h=3)$	$p(v_t\|v_{t-1}=2,h=3)$
0	0.1598	0.1445	0.487
1	0.306	0.2835	0.3502
2	0.5343	0.572	0.1628

Table 11: Learned $p(v_t|v_{t-1}, h = 3)$

initialisation is not a good idea. Inspecting the results above we can also see that all of the learned parameters are uniform across all values of h which also suggests that the algorithm has not converged to a globally optimum solution and has instead got stuck in a local maximum. This happens because at each step of the algorithm the update to $p(h)$ is generated from the initialised values of $p(h)$, $p(v_t|v_{t-1}, h)$ and $p(v_1|h)$. If these are all uniform across h, which is the case with uniform initialisation, then then updates to $p(h)$ will be uniform, i.e. $p^{new}(h) = 1/3$ for all values of h. If $p(h)$ remains uniform throughout the algorithm then the other parameter(s) will be be uniform across h. We lose our ability to cluster sequences in this case since the posterior $p(h|v^n_{1:T})$ will be the same for all values of h and n, suggesting that any sequence is equally likely to have come from each of the weather stations.

2 LDPC codes

2.1 Question 1

For a parity check matrix H given by

$$H = \begin{pmatrix} 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 & 0 \end{pmatrix} \tag{16}$$

we can recover the generator matrix G by first reducing H to Row Reduced Echelon Form and then permuting the columns of the RREF matrix to be in the form $\hat{H} := [PI_{N-K}]$, where N is the codeword length and K is the block length. G can then be retrieved from $\hat{H}$ using the relationship $G = [I_K P]^T$.

Running the above algorithm on H using the code listed in appendix 1, we get the following results:

$$\hat{H} = \begin{pmatrix} 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 1 \end{pmatrix} \tag{17}$$

$$G = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \\ 1 & 0 & 1 \end{pmatrix} \tag{18}$$

As expected, $\hat{H}Gt = 0$ for all $t \in \{0,1\}^K$ blocks with operations performed in $\mathbb{F}_2$.

2.2 Question 2

The factor graph for the matrix H is displayed in figure 5.

The nodes on the left are check nodes and correspond to the rows of a parity check matrix. The nodes on the right are message nodes, also known as bit nodes or variable nodes, and correspond to the columns of the parity check matrix.

Figure 5 corresponds to the following distribution

$$p(\mathbf{x}) = f_1(x_1, x_2, x_3, x_4) f_2(x_3, x_4, x_6) f_3(x_1, x_4, x_5) \tag{19}$$

For a bit node x_n, a check node f_m, and an observed bit y_n, the updates used for the messages are as follows:

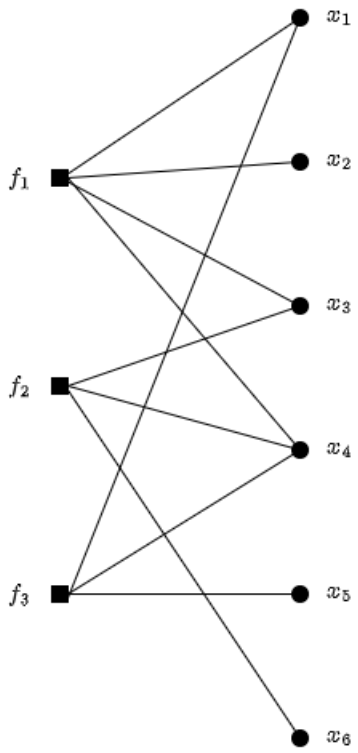


Figure 5 Factor graph representation of H

1. Initialisation: $\mu_{x_n \to f_m}(x_n) = p(y_n|x_n)$
2. Check-to-bit messages:

$$\mu_{f_m \to x_n}(x_n) = \sum_{x_{n'}: n' \in \mathcal{N}(m) \backslash n} \mathbb{I}[x_n + \textstyle\sum_{n'} x_{n'} = 0] \prod_{x_{n'}: n' \in \mathcal{N}(m) \backslash n} \mu_{x_{n'} \to f_m}(x_{n'})$$

   where $\mathcal{N}(m) \backslash n$ defines the set of bits n that participate in check m, excluding bit n

3. Bit-to-check messages:

$$\mu_{x_n \to f_m} \propto p(y_n|x_n) \prod_{m' \in \mathcal{N}(n) \backslash m} \mu_{f_{m'} \to x_n}(x_n)$$

   where $\mathcal{N}(n)$ defines the set of checks in which bit n participates.

2.3 Question 3

Given a transmitted word x, a received word y is decoded using Loopy Belief Propagation for Binary Symmetric Channels as described in [2] and [3]. It is an iterative algorithm whereby messages are passed from bit node to check node and then back again at each step. The messages represent probabilities about the values of the bit nodes conditioned on information from previous rounds of the algorithm and other messages passed along the edges of the factor graph. The aim

of the algorithm is to find a decoding $x_n^* = \arg\max p(x_n|y)$, ensuring that x^* is a valid codeword.

The code for the algorithm can be found in appendix 2 and is described by the following process for a parity check matrix H, a received word y, and noise ratio p. Note that we are working with logarithms to mitigate numerical underflow issues.

0. Initialise the messages from bit node to check node. Note that $H_{ij} = 1$ if there is a connection between check node i and bit node j, otherwise $H_{ij} = 0$. A connection corresponds to an edge in the factor graph representation of H and since messages are only passed along edges of the factor graph, we can keep track of the latest messages in a matrix P which is 0 for all i, j such that $H_{ij} = 0$. For the initial messages, we set

$$P_{ij} = \begin{cases} \log(1-p) - \log(p) & \text{if } y_j = 0 \\ \log(p) - \log(1-p) & \text{if } y_j = 1 \end{cases}$$

where $H_{ij} = 1$, and we set $P^0 := P$ as P^0 is needed for later steps in the algorithm.

1. Send messages from check node to bit node. A message from check node c to bit node v at the ℓth round of the algorithm is given by

$$m_{cv}^{(\ell)} = \log \frac{1 + \prod_{v' \in V_c \setminus \{v\}} \tanh\left(m_{v'c}^{(\ell)}/2\right)}{1 - \prod_{v' \in V_c \setminus \{v\}} \tanh\left(m_{v'c}^{(\ell)}/2\right)} \tag{20}$$

where V_c is the set of bit nodes incident to c and $m_{v'c}^{(\ell)}$ are the messages from bit node to check node in round ℓ of the algorithm. Each element $P_{ij} \neq 0$ is then set to $m_{ij}^{(\ell)}$ using equation 20.

2. Generate a tentative decoding. For each bit node x_j, we sum the latest messages over the check nodes in which x_j participates, i.e. the columns of P. If the sum is less than 0, we decode x_j to a 1, otherwise we decode it to a 0. With this decoded vector, denoted $\mathbf{d}$, we then test whether it is a valid codeword by performing $H\mathbf{d} = \mathbf{z}$ in $\mathbb{F}_2$. If $\mathbf{z} = \mathbf{0}$, then $\mathbf{d}$ is a valid codeword and we can halt the algorithm as we have converged to a maximum likelihood solution, namely $x^* = \mathbf{d}$. If not, then we proceed to step 3.

3. Send messages from bit node to check node. A message from bit node v to check node c at the ℓth round of the algorithm is given by

$$m_{vc}^{(\ell)} = m_v + \sum_{c' \in C_v \setminus \{c\}} m_{c'v}^{(\ell-1)} \tag{21}$$

where m_v are the initial messages described in step 0 and C_v is the set of

check nodes incident to bit node v. Each element $P_{ij} \neq 0$ is then set to $m_{ji}^{(\ell)}$ using equation 21. Repeat from step 1.

Running the algorithm on the given vector y_1 and parity check matrix H_1 converges to the following decoded 1000×1 vector in 8 iterations:

```
[0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0, 1, 1, 1, 0, 0,
 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 1, 0, 0, 1, 0,
 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 1, 1, 1, 1, 0, 1,
 1, 0, 1, 1, 0, 0, 0, 1, 1, 0, 1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0,
 0, 1, 1, 0, 0, 0, 0, 1, 0, 1, 1, 1, 1, 0, 0, 1, 0, 1, 1, 1, 0, 0,
 1, 1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0,
 0, 1, 0, 0, 0, 1, 1, 0, 1, 1, 0, 1, 0, 1, 1, 0, 1, 0, 0, 1, 0, 1,
 1, 1, 0, 1, 0, 0, 0, 1, 1, 1, 0, 0, 1, 0, 0, 1, 1, 1, 1, 0, 0, 1,
 0, 0, 1, 0, 0, 1, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0,
 0, 1, 0, 1, 1, 1, 0, 1, 1, 0, 0, 1, 1, 0, 1, 0, 0, 1, 0, 1, 1, 0,
 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1, 0, 0, 0,
 1, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1,
 1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1, 0, 1,
 0, 1, 1, 0, 0, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1,
 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1,
 0, 0, 1, 0, 1, 0, 1, 0, 0, 1, 1, 1, 1, 1, 1, 0, 1, 0, 1, 1, 0, 0,
 0, 0, 0, 0, 1, 0, 0, 1, 1, 1, 1, 1, 0, 1, 0, 1, 0, 1, 1, 0, 0, 1,
 1, 0, 0, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1,
 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1,
 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1,
 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0,
 1, 1, 0, 1, 1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 0, 1, 1, 0, 1, 0, 0, 1,
 0, 1, 1, 0, 0, 1, 1, 0, 1, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0,
 1, 0, 0, 0, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1,
 1, 1, 0, 0, 1, 1, 1, 1, 1, 0, 1, 0, 0, 1, 1, 0, 0, 1, 0, 1, 1, 1,
 1, 1, 1, 1, 0, 0, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 1, 0, 1, 0, 1, 0,
 0, 1, 1, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0, 1,
 1, 0, 1, 0, 0, 1, 1, 0, 0, 1, 0, 1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1,
 1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1,
 0, 1, 1, 1, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1,
 0, 1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 1, 1, 1,
 1, 1, 0, 1, 1, 0, 0, 1, 1, 0, 1, 0, 0, 1, 0, 0, 1, 1, 1, 0, 1, 0,
 0, 1, 0, 1, 1, 0, 1, 0, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 1, 1,
 0, 0, 1, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0, 1, 1, 0, 1, 0, 1, 0, 0, 1,
 1, 1, 1, 1, 0, 1, 1, 0, 0, 1, 1, 0, 1, 0, 1, 0, 0, 0, 1, 1, 1, 1,
 1, 1, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 0, 1,
 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 0, 1,
```

```
1, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 1, 0,
1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0,
1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 0, 0, 1, 1, 0, 1, 0, 1, 0, 1, 1, 0,
1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1,
1, 0, 1, 1, 0, 1, 0, 0, 0, 1, 0, 1, 0, 1, 1, 0, 1, 1, 1, 0, 0, 1,
0, 0, 1, 0, 0, 1, 1, 1, 0, 0, 1, 1, 0, 1, 0, 1, 1, 0, 1, 0, 1, 1,
1, 0, 0, 1, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1,
0, 1, 1, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 0, 0,
1, 0, 1, 1, 0, 0, 1, 1, 1, 0]
```

2.4 Question 4

If we treat the first 248 bits of the decoded vector described in section 2.3 as a sequence of 31 ASCII symbols, we recover the following message:

"Happy Holidays! Dmitry&David :)"

3 Mean Field Approximation and Gibbs Sampling

3.1 Exact Inference

Figure 6 is a representation of the 10x10 Ising lattice model (defined in Exercise 6.7 from *Barber, 2012* [1]) as a singly connected factor graph. Given β we have:

$$\begin{aligned}
\phi(x_k, x_l) &= e^{\beta \mathbb{I}[x_k = x_l]}, \text{ where } x_k \text{ and } x_l \text{ are neighbours in the lattice} \\
y_j &= (x_{1,j}, x_{2,j}, \ldots, x_{n,j}),\ x_{i,j} \in \{0, 1\} \\
f_0(y_1) &= \prod_{i=1}^{9} \phi(x_{i,1}, x_{i+1,1}) \qquad (22) \\
f_j(y_j, y_{j+1}) &= \prod_{i=1}^{9} \phi(x_{i,j+1}, x_{i+1,j+1}) \prod_{i=1}^{10} \phi(x_{i,j}, x_{i,j+1}),\ j \in \{1, 2, \ldots, 9\} \qquad (23)
\end{aligned}$$

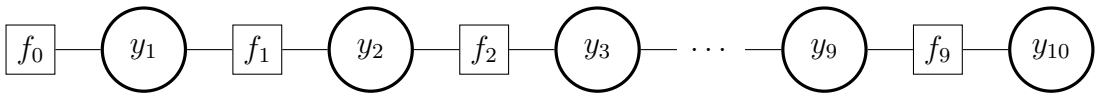
Note, $x_{i,j}$ represents node on row i and column j, and y_j represents column j. We will complete exact inference to calculate the joint probability distribution of the top and bottom right nodes of the 10x10 Ising model. To do this we will use message passing starting at f_0, and passing through the whole network to give a distribution on y_{10}. We will then marginalise out all unwanted variables, only to leave the probability distribution of the top right and bottom right nodes which are $x_{1,10}$ and $x_{10,10}$.

First we will find the distribution of $p(y_{10})$ which is given below:

$$p(y_{10}) = \frac{1}{Z} \sum_{y_1, \ldots, y_9} f_0(y_1) \prod_{i=1}^{9} f_i(y_i, y_{i+1}), \text{ where } Z = \sum_{y_1, \ldots, y_{10}} f_0(y_1) \prod_{i=1}^{9} f_i(y_i, y_{i+1})$$

This is very computationally intensive, so we will use message passing by pushing variables outside of as many summations as possible to reduce the number of calculations required. We define our messages below. Note usually messages are passed from variable to factor and then from factor to variable. But because our model is singly connected there is no additional information in messages from variables to factors, so we only consider variable to variable

Figure 6 Ising model represented as a singly connected factor graph



messages:

$$\mu_{y_0 \to y_1}(y_1) = f_0(y_1)$$
$$\mu_{y_j \to y_{j+1}}(y_{j+1}) = \sum_{y_j} f_j(y_j, y_{j+1}) \mu_{y_{j-1} \to y_j}(y_j),\ j \in \{1, 2, \ldots, 9\}$$

This works directly when β is 1, but when it's 0.01 or 4 the messages either blow up to infinity or vanish to zero because f_i is a product of exponentials. So we have to use a similar technique used in *Section 1 - Part B* by using log messages using *Equation 5*. Now we can calculate $\mu_{y_9 \to y_{10}}(y_{10})$. Remember $y_{10} = (x_1, x_2, \ldots, x_{10})$ is a variable of the final column in the 10x10 lattice, so we marginalise out the unwanted variables and normalise to find our desired result:

$$q(x_1, x_{10}) = \sum_{x_2, \ldots, x_9} \mu_{y_9 \to y_{10}}(x_1, x_2, \ldots, x_{10}), \text{ and } Z = \sum_{x_1, x_{10}} q(x_1, x_{10})$$
$$\text{then } p(x_1, x_{10}) = \frac{1}{Z} q(x_1, x_{10})$$

Using [*Appendix 3: Code: 1*], where x_1 and x_{10} are the top right and bottom right nodes respectively in the 10x10 lattice, we calculate the following:

(x_1, x_{10})	$P(x_1, x_{10} \mid \beta = 0.01)$	$P(x_1, x_{10} \mid \beta = 1)$	$P(x_1, x_{10} \mid \beta = 4)$
(0,0)	0.25000	0.28045	0.49965
(1,0)	0.25000	0.21955	0.00035
(0,1)	0.25000	0.21955	0.00035
(1,1)	0.25000	0.28045	0.49965

3.2 Mean Field Approximation

To use a Mean Field Approximation with coordinate ascent, we estimate $P(x) = Z^{-1} e^{\beta \sum_{i<j} \mathbb{I}[x_i = x_j]}$ using a factorised distribution

$$Q(x) = \prod_{n=1}^{N} q_n(x_n)$$

where, in this case for a 10x10 lattice, $N = 100$ for each point on the lattice. To maximise the free energy, we set

$$q_k^*(x_k) \propto e^{<logP(x_k \mid x_{-k})>_{q_{-k}}}$$

Coordinate ascent is used to optimise this value iteratively by updating $q_k^*(x_k)$ one variable at a time, while holding the values of the other variables fixed. This process is repeated until the solution converges.

We can find the how to calculate $q_k^*(x_k)$ by breaking down the expression.

Firstly, we can express $P(x_k \,|\, x_{-k})$ in terms of the neighbours of x_k by pulling out the indicator function of all pairs of neighbours not containing x_k as a factor in the numerator and denominator as follows:

$$\begin{aligned}
P(x_k \,|\, x_{-k}) &= \frac{P(x)}{P(x_{-k})} \\
&= \frac{P(x)}{\sum_{x_k \in [0,1]} P(x)} \\
&= \frac{\frac{1}{Z} e^{\beta \sum_{i>j} \mathbb{I}[x_i = x_j]}}{\frac{1}{Z} e^{\beta \sum_{i,j \neq k} \mathbb{I}[x_i = x_j]} \sum_{x_k} e^{\beta \sum_{k_i \sim k} \mathbb{I}[x_k = x_{k_i}]}} \\
&= \frac{e^{\beta \sum_{k_i \sim k} \mathbb{I}[x_k = x_{k_i}]}}{\sum_{x_k} e^{\beta \sum_{k_i \sim k} \mathbb{I}[x_k = x_{k_i}]}} \\
&= \frac{e^{\beta \sum_{k_i \sim k} \mathbb{I}[x_k = x_{k_i}]}}{K}
\end{aligned}$$

where K is the normalising constant and x_{k_i} are the neighbours of x_k. Thus, the marginal probability of x_k conditioned on all the other nodes on the lattice is actually only dependent on the state of it's neighbours. Taking logs of this we get:

$$\log P(x_k \,|\, x_{-k}) = \beta \sum_{k_i \sim k} \mathbb{I}[x_k = x_{k_i}] - \log K$$

We then need to take the expected value of this with respect to q_{-k}, which, applying the linearity of expectations, gives

$$< \log P(x_k \,|\, x_{-k}) >_{q_{-k}} = \beta \sum_{k_i \sim k} < \mathbb{I}[x_k = x_{k_i}] >_{q_{-k}} - L$$

where L is the expectation $\log K$. Note that we are only considering $x_k = 1$ from here to simplify, because if we find $P(x_k = 1 \,|\, x_{-k})$ we have found the whole conditional distribution: $P(x_k = 0 \,|\, x_{-k}) = 1 - P(x_k = 1 \,|\, x_{-k})$. The expectation of the indicator function gives the following:

$$\begin{aligned}
< \mathbb{I}[x_k = x_{k_i}] >_{q_{-k}} &= \sum_{x_1} \ldots \sum_{x_{k-1}} \sum_{x_{k+1}} \ldots \sum_{x_n} \mathbb{I}[x_{k_i} = x_k] q_{-k}(x) \\
&= \sum_{x_{k_i}} \mathbb{I}[x_{k_i} = x_k] q_{k_i}(x_{k_i}) [\sum_{x_1} \ldots \sum_{x_{k-1}} \sum_{x_{k+1}} \ldots \sum_{x_n} q_{-k,-k_i}(x)] \\
&= q_{k_i}(x_{k_i} = 1)
\end{aligned}$$

as the sum over all states of the non-neighbouring nodes of the joint distribution (excluding x_k and x_{k_i}) equals 1, and the indicator function is equal to 0 when $x_{k_i} = 0$ (as $x_k = 1$). Thus

$$< \log P(x_k = 1|x_{-k}) >_{q_{-k}} = \beta[\sum_{k_i \sim k} q_{k_i}(x_{k_i} = 1)] - L$$

Thus, by finding the exponential of this number, we have

$$q_k^*(x_k = 1) \propto \frac{e^{\beta \sum_{k_i \sim k} q_{k_i}(x_{k_i}=1)}}{L}$$

.

This is then normalised using

$$\begin{aligned} q_k^*(x_k = 1) &= \frac{\frac{e^{\beta \sum_{k_i \sim k} q_{k_i}(x_{k_i}=1)}}{L}}{\frac{e^{\beta \sum_{k_i \sim k} q_{k_i}(x_{k_i}=0)}}{L} + \frac{e^{\beta \sum_{k_i \sim k} q_{k_i}(x_{k_i}=1)}}{L}} \\ &= \frac{e^{\beta \sum_{k_i \sim k} q_{k_i}(x_{k_i}=1)}}{e^{\beta \sum_{k_i \sim k} q_{k_i}(x_{k_i}=0)} + e^{\beta \sum_{k_i \sim k} q_{k_i}(x_{k_i}=1)}} \\ &= \frac{e^{\beta \sum_{k_i \sim k} q_{k_i}(x_{k_i}=1)}}{e^{\beta \sum_{k_i \sim k} 1 - q_{k_i}(x_{k_i}=1)} + e^{\beta \sum_{k_i \sim k} q_{k_i}(x_{k_i}=1)}} \end{aligned}$$

Thus the L falls out. Therefore, to update the estimated marginal distributions, we just need to look at the marginal distributions of the neighbours of x_k, normalising as we go.

The initial values of q_k are set randomly, and the nodes are updated in order. After 10,000 iterations, the joint distribution of $x_{1,10}$ and $x_{10,10}$ is estimated as a product of the marginal distributions

$$q(x_{1,10}, x_{10,10}) = q(x_{1,10})q(x_{10,10})$$

The results for $\beta = 0.01$ behave much in the same way as in question 3.1, as the interaction between neighbours becomes negligible, and therefore there is an equal chance of all outcomes. For this case, $q(x_{1,10}) = q(x_{10,10}) = 0.5$ and thus, the joint distribution is

$X = (x_{1,10}, x_{10,10})$	$q(X\|\beta = 0.01)$
(0,0)	0.25
(1,0)	0.25
(0,1)	0.25
(1,1)	0.25

as before.

However, when $\beta = 1$ and $\beta = 4$, the updated values of $q_k^*(x_k)$ encourage interaction between neighbours, and the algorithm appears to get stuck in local minima of the KL divergence. This can be seen as the resulting joint distribution is not symmetrical as it is in question 3.1, and is dependent on the random initial

distributions of each node.

When $\beta = 1$, both marginal distributions converge to either 0.8454 or 0.1546 in any combination; explicitly there are four possible convergences:

$$q(x_{1,10}) = 0.8454,\ q(x_{10,10}) = 0.8454$$
$$q(x_{1,10}) = 0.8454,\ q(x_{10,10}) = 0.1546$$
$$q(x_{1,10}) = 0.1546,\ q(x_{10,10}) = 0.8454$$
$$q(x_{1,10}) = 0.1546,\ q(x_{10,10}) = 0.1546$$

Therefore the joint distribution converges to one of four options:

$X = (x_{1,10}, x_{10,10})$	$q(X\|\beta = 1)$	$q(X\|\beta = 1)$	$q(X\|\beta = 1)$	$q(X\|\beta = 1)$
(0,0)	0.0239	0.7147	0.1307	0.1307
(1,0)	0.1307	0.1307	0.0239	0.7147
(0,1)	0.1307	0.1307	0.7147	0.0239
(1,1)	0.7147	0.0239	0.1307	0.1307

While there is some symmetry in the joint distribution, it shows no preference for the lattice to be in the same state.

This is similar, but more extreme, when $\beta = 4$. In this case, the marginal distributions take on the values 0.00034 and 0.99966 which give the following joint distributions.

$X = (x_{1,10}, x_{10,10})$	$q(X\|\beta = 4)$	$q(X\|\beta = 4)$	$q(X\|\beta = 4)$	$q(X\|\beta = 4)$
(0,0)	0.99932	0.00000	0.00034	0.00034
(1,0)	0.00034	0.00034	0.99932	0.00000
(0,1)	0.00034	0.00034	0.00000	0.99932
(1,1)	0.00000	0.99932	0.00034	0.00034

We can see from the table that the joint distribution strongly favours one combination state of $x_{1,10}$ and $x_{10,10}$; however the nodes do not necessarily have to be the same state. This contradicts the results found in 3.1 and the intuition on the behaviour of the lattice when β is high.

3.3 Gibbs Sampling

Gibbs sampling is a Markov Chain Monte Carlo (MCMC) method used to generate samples from a multivariate distribution. It works by iteratively sampling from the full conditional distribution of each variable, given the current values of the other variables. This process is repeated until the samples converge to the stationary distribution of the Markov chain, which approximates the target distribution.

Once again, we use the conditional distribution $P(x_k|x_{-k})$ to estimate the joint distribution $P(x)$. We found in question 3.2 that

$$P(x_k = 1|x_{-k}) = \frac{e^{\beta[\sum_{k_i \sim k} \mathbb{I}[x_k = x_{k_i}]]}}{\sum_{x_k} e^{\beta[\sum_{k_i \sim k} \mathbb{I}[x_k = x_{k_i}]]}}$$
$$= \frac{e^{\beta(\#\text{ neighbours} = 1)}}{e^{\beta(\#\text{ neighbours} = 1)} + e^{\beta(\#\text{ neighbours} = 0)}}$$

where x_{k_i} are the neighbours of x_k.

First, we initialise the state of the lattice randomly, and sample from the variables one at a time in a random order, calculating $P(x_k|x_{-k})$ based on the values of the neighbours. The state of x_k is then set to 1 with this probability.

The first 1000 samples are burned, and the remaining samples for $x_{1,10}$ and $x_{10,10}$ are saved. The joint distribution is then calculated by counting the number of occurrences of each combination state, and dividing by the total number of samples.

Once again, the joint distribution of $x_{1,10}$ and $x_{10,10}$ behave in the expected way for $\beta = 0.01$, resulting in the approximately uniform distribution (when rounded to two decimal places):

$X = (x_{1,10}, x_{10,10})$	$q(X\|\beta = 0.01)$
(0,0)	0.248
(1,0)	0.252
(0,1)	0.251
(1,1)	0.249

This agrees with both question 3.1 and 3.2, indicating each corner node has an equal chance of being in state 0 or 1.

When $\beta = 1$, the joint distribution converges symmetrically and identically to question 3.1 (correct to two decimal places):

$X = (x_{1,10}, x_{10,10})$	$q(X\|\beta = 1)$
(0,0)	0.281
(1,0)	0.216
(0,1)	0.224
(1,1)	0.279

However, when $\beta = 4$, the lattice converges to the same state - whether that state is 0 or 1 is dependent on the initial values. Therefore, there are two distinct results (correct to 3 decimal places):

This could be anticipated when sampling from the conditional probability, as it is very likely that the node will adjust to match it's neighbours, therefore, with enough iterations, the whole lattice will converge.

$X = (x_{1,10}, x_{10,10})$	$q(X\|\beta = 4)$	$q(X\|\beta = 4)$
(0,0)	1.000	0.000
(1,0)	0.000	0.000
(0,1)	0.000	0.000
(1,1)	0.000	1.000

This can be seen in the joint distribution of $x_{1,10}$ and $x_{10,10}$, as either both nodes are in state 0 or both nodes are in state 1.

Appendix A

Code for Q1

1 Earthquake Calculations - Part A and B

code date: 22.12.23 - 16:51

```
import numpy as np
import stat as stats
import matplotlib.pyplot as plt
import matplotlib.colors as colors

# Load earthquake data
loc = "earthquake files-20221127\EarthquakeExerciseData.txt"
data_earthquake = np.loadtxt(loc)

# assuming sensor standard deviation is 0.2 (this is the same as example 1.12 in
sensor_sd = 0.2


def earthquake_exercise_setup(station_num=30, spiral_points_num=2000, angular_spira
    """Setup the spiral world coordinates of the earthquake model, returning worl
    This function is based on the earthquakeExerciseSetup.jl function provided fr
    converting it to Python and changing some of the outputs

    Args:
        station_num (int, optional): Number of sensors on the surface of the worl
        spiral_points_num (int, optional): Number of discretized spiral world poi
        angular_spiral_rate (int, optional): How many world points to step throug

    Returns:
        world (ndarray): spiral_points_num x 2 array. Element i is (x,y) coordina
        sensors (ndarray): station_num array x 2. Element i is (x,y) coordinates
```

```
    """

    # explosion detector (using spiral coordinate system)
    # define the coordinate system:
    S = spiral_points_num # number of points on the spiral
    rate = angular_spiral_rate # angular rate of spiral

    # array to hold world coordinates (x, y)
    world = np.zeros((S, 2))

    # iteratively work around the spiral filling in cartesian location values in
    for s in range(S):
        r = s / S
        theta = rate * 2 * np.pi * r
        world[s, 0] = r * np.cos(theta)
        world[s, 1] = r * np.sin(theta)

    # define the locations of the detection stations on the surface
    # Also define what value on each sensor would be generated by an explostion a
    N = station_num # number of stations

    # array to hold sensor locations
    sensors = np.zeros((N, 2))

    for sensor in range(N):
        # initialise x, y values for each sensor
        theta_sensor = 2 * np.pi * sensor / N
        sensors[sensor, 0] = np.cos(theta_sensor)
        sensors[sensor, 1] = np.sin(theta_sensor)

    return world, sensors


def calculate_likelihood_exponentials(world, sensors, sensor_data, sensor_sd, is_tw
    """This calculates the likelihood contribution from an explosion at the i-th
    It only calculates the power of the exponential for numerical accuracy
    If we are calculating for two explosions, then there is an additonal returned

    Args:
        world (ndarray): Sx2 array, i-th element is (x,y) coordinates of i-th wor
        sensors (ndarray): Nx2 array, i-th element is (x,y) coordinates of i-th s
        sensor_data (ndarray): N array, i-th element is observed signal from i-th
```

```
        sensor_sd (float): standard deviation of error in observed signal from ea
        is_two_explosion (bool, optional): bool flag indicating whether calculati

    Returns:
        F (ndarray): SxN array (SxSxN if two explosions). Element [i,j] is the lo
    """

    S = world.shape[0]
    N = sensors.shape[0]

    # Calculate distances between world point i and sensor j at index [i, j]
    dist = np.zeros((S, N))
    for sensor in range(N):
        dist[:, sensor] = np.linalg.norm(world - sensors[sensor], axis=1)

    # array of theoretical signal from a single explosion. At [i, j] is signal at
    world_signal = 1 / (dist**2 + 0.1)

    # Calculate for 2 explosions:
    if is_two_explosion:
        F = np.zeros(shape=(S, S, N))
        for s1 in range(S):
            for s2 in range(S):
                F[s1, s2, :] = (sensor_data - 0.5 * world_signal[s1, :] - 0.5 * wor

    # calculate for 1 explosion:
    else:
        F = np.zeros(shape=(S, N))
        for s1 in range(S):
            F[s1, :] = (sensor_data - world_signal[s1, :])**2

    factor = -1 / (2 * pow(sensor_sd, 2))
    F = factor * F
    return F


def log_sum_of_exponential_components(F, sum_axis=0):
    """given an array F of exponential componenets, return log(np.exp(F).sum(axis
    This uses a method of finding the maximum component and removing it from the
    If calculated directly numerically it would often go to +/-inf

    Args:
```

```
        F (ndarray): array of exponential components
        sum_axis (int): axis to complete the sum along. Defaults to 0
    """
    # find the maximum term. Then when computing take the max outside the log sum
    F_max = F.max(axis=sum_axis, keepdims=True)
    log_sum = F_max + np.log(np.exp(F - F_max).sum(axis=sum_axis, keepdims=True))

    # squeeze to remove the axis that has been summed over
    return log_sum.squeeze()


def show_earth_likelihood(sensor_locations, world, like, sensor_data=None, axes=No
    """Graphically show the likelihood of the earthquake model, with markers for

    Args:
        sensor_locations (ndarray): Nx2 array, i-th element is (x,y) coordinates
        world (ndarray): Sx2 array, i-th element is (x,y) coordinates of i-th wor
        like (ndarray): S array, likelihood (or log_likelihood) at each world poi
        sensor_data (ndarray, optional):N array, i-th element is observed signal
        axes (matplotlib: Axes, optional): Axes to plot onto. Defaults to None.
        visual_scale (float, optional): plot the top visual_scale percentage like
        is_two_explosions (bool, optional): bool flag indicating whether calculat
        title (str, optional): Title to include on plot. Defaults to 'Title'.
    """
    # size of visible signal bars
    radial_scale = 0.05

    # create axes if not given any
    if axes is None:
        __, axes = plt.subplots()
    axes.set_title(title)

    draw_circle = plt.Circle((0, 0), 1,fill=False)

    axes.set_aspect(1)
    axes.add_artist(draw_circle)

    colormap = plt.cm.Greys #or any other colormap

    # take top visual_scale % of likelihoods that aren't -inf
    is_not_inf_index = np.logical_not(np.isinf(like))
    sorted_like = np.sort(like[is_not_inf_index])
```

```
    N = int(sorted_like.size) - 1
    visual_min = sorted_like[N - int(N * visual_scale)]

    normalize = colors.Normalize(vmin=visual_min, vmax=like.max())
    axes.scatter(world[:,0], world[:,1], c=like, s=5, cmap=colormap, norm=normalize

    if sensor_data is not None:
        # generate vectors pointing out from centre representing visible signal
        radial_signal = sensor_locations * radial_scale * sensor_data.reshape((-1,
        for sensor in np.arange(sensor_locations.shape[0]):
            x = [sensor_locations[sensor, 0], radial_signal[sensor, 0]]
            y = [sensor_locations[sensor, 1], radial_signal[sensor, 1]]
            axes.plot(x, y, c='red')

    # plot the sensors onto the earth
    axes.scatter(sensor_locations[:,0], sensor_locations[:,1], c='black', s=15)

    # plot the max likelihood point
    max_like_index = like.argmax()
    axes.scatter(world[max_like_index,0], world[max_like_index,1], c='red', marker=

    # if 2 explosions plot the second local maximum
    if is_two_explosions:
        # visually looking at plot we can see the second local maximum is in the
        world_right_index = world[:,0] > 0
        like_world_right = like[world_right_index]
        world_right = world[world_right_index]
        max_2_like_index = like_world_right.argmax()
        axes.scatter(world_right[max_2_like_index,0], world_right[max_2_like_index

    axes.legend(loc='lower left')
    axes.axis('off')


# Pull it all together to calculate Answer
# setup world and sensors
world, sensors = earthquake_exercise_setup()

# calculate the likelihood exponential components for both 1 explosion (H1) and 2
like_exponentials_H1 = calculate_likelihood_exponentials(world, sensors, data_earth
like_exponentials_H2 = calculate_likelihood_exponentials(world, sensors, data_earth
```

```
# For H1 the loglikelihood is just a simple sum
log_like_H1 = like_exponentials_H1.sum(axis=1)

# But for H2 the likelihood is a sum of poructs, therefore we sum over the sensor
# and then use the log_sum function to calculate the second sum in a numerically
log_like_H2 = log_sum_of_exponential_components(like_exponentials_H2.sum(axis=2), s

# plot the likelihoods
fig, axs = plt.subplots(1, 2)
show_earth_likelihood(sensors, world, log_like_H2, data_earthquake, axes=axs[0], vi
show_earth_likelihood(sensors, world, log_like_H1, data_earthquake, axes=axs[1], vi
fig.tight_layout()

# Save solution figure
plt.savefig('Q1.1.a_explosion_posterior.png', dpi=300, bbox_inches='tight')


# ***** Part B ******

# Continuing the assumption that priors p(s1) and p(s2) are constant over all wor
p_s1 = 1 / world.size
p_s2 = p_s1

log_prob_H1 =  np.log(p_s1) + log_sum_of_exponential_components(like_exponentials_H
log_prob_H2 =  np.log(p_s1) + np.log(p_s2) + log_sum_of_exponential_components(like

# Print solutions
answer = log_prob_H2 - log_prob_H1
print(f"log(p(v|H2))= {log_prob_H2}")
print(f"log(p(v|H1)) = {log_prob_H1}")
print(f"log(p(v|H2)) - log(p(v|H1)) = {answer}")
```

2 Weather stations

```
import numpy as np
from numpy.typing import NDArray

# global parameters
V = 3  # number of observable states
H = 3  # number of Markov models
```

```python
def condp(data: NDArray) -> NDArray:
    """Return a conditional distribution from a data array

    All columns sum to 1

    Args:
        data: An array of data

    Returns:
        NDArray: the conditional distribution
    """
    return data / np.sum(data, axis=0)


def condexp(logp: NDArray) -> NDArray:
    """Compute the exponential of the log probability

    Args:
        logp: log probability

    Returns:
        NDArray: conditional distribution
    """
    return condp(np.exp(logp - np.max(logp, 0)))


def uniform_initialisation() -> tuple[NDArray, NDArray, NDArray]:
    """Uniformly initialize the parameters

    Returns:
        tuple[NDArray, NDArray, NDArray]: uniform p(h),
            p(v_1|h), p(v_t|v_{t - 1},h)
    """
    ph = np.full((H, 1), 1 / H)
    pv1gh = np.full((V, H), 1 / V)
    pvgvh = np.full((V, V, H), 1 / V)

    return ph, pv1gh, pvgvh


def random_initialisation() -> tuple[NDArray, NDArray, NDArray]:
    """Randomly initialize the parameters
```

```
    Taken from a uniform distribution

    Returns:
        tuple[NDArray, NDArray, NDArray]: random p(h),
            p(v_1|h), p(v_t|v_{t-1},h)
    """
    uniform = np.random.uniform
    ph = uniform(size=(H, 1))
    pv1gh = uniform(size=(V, H))
    pvgvh = uniform(size=(V, V, H))

    return condp(ph), condp(pv1gh), condp(pvgvh)


def prior_pvgvh(data: NDArray) -> NDArray:
    """Set p(v_t|v_{t-1},h) from the data

    Computes the empirical distribution from the data

    Returns:
        NDArray: a (V,V,H) matrix for the conditional distribution
            of p(v_t|v_{t-1},h) based on the data
    """
    pvgvh = np.zeros((V, H))
    for h in range(H):
        r, c = np.where(data[:, :-1] == h)
        n = data[(r, c + 1)]
        nc = np.array([np.count_nonzero(n == i) for i in range(V)])
        pvgvh[:, h] = nc

    pvgvh /= np.sum(pvgvh, axis=0)
    return np.broadcast_to(pvgvh, (V, V, H))


def run_em(
    data: NDArray,
    ph: NDArray,
    pv1gh: NDArray,
    pvgvh: NDArray,
    max_iters: int = 20,
) -> tuple:
```

```
    """Run the EM algorithm and return learned params

    Args:
        data: the sequence data
        ph: the initialised p(h)
        pv1gh: the initialised p(v_1|h)
        pvgvh: the initialised p(v_t|v_{t-1}, h)
        max_iters: a maximum number of iterations to run the algorithm

    Returns:
        tuple: a tuple of the following data
            ph - learned p(h)
            pv1gh - learned p(v_1|h)
            pvgvh - learned p(v_t|v_{t-1}, h)
            phgv - the calculated posterior p(h|v)
            llik - the log likelihoods at each step of the algorithm
    """
    max_iters = 20
    llik = []
    for iter in range(max_iters):
        # e-step
        ph_stat = np.zeros((H, 1))
        pv1gh_stat = np.zeros((V, H))
        pvgvh_stat = np.zeros((V, V, H))
        loglik = 0

        ph_old = []
        for n in range(data.shape[0]):
            T = data.shape[1]

            lph_old = np.log(ph) + np.log(pv1gh[data[n, 0]])[:, np.newaxis]

            for t in range(1, T):
                lph_old += np.log(pvgvh[data[n, t], data[n, t - 1]])[:, np.newaxis]

            ph_old.append(condexp(lph_old))
            loglik += np.log(np.sum(np.exp(lph_old)))

            ph_stat += ph_old[n]

            pv1gh_stat[data[n, 0]] += np.squeeze(ph_old[n])
```

```
            for t in range(1, T):
                pvgvh_stat[data[n, t], data[n, t - 1]] += np.squeeze(ph_old[n])

        llik.append(loglik)

        # m-step
        ph = condp(ph_stat)
        pv1gh = condp(pv1gh_stat)
        pvgvh = condp(pvgvh_stat)

        print(f"end of iteration: {iter}, current loglik: {loglik:.5f}")

    phgv = np.array(ph_old)

    return ph, pv1gh, pvgvh, phgv, llik


data = np.loadtxt("meteo1.csv").astype(int)

# uniform initialisation
# ph_0, pv1gh_0, pvgvh_0 = uniform_initialisation()

# uniform_initialisation with prior generated from the data
# ph_0, pv1gh_0, _ = uniform_initialisation()
# pvgvh_0 = prior_pvgvh(data)

# random initialisation
ph_0, pv1gh_0, pvgvh_0 = random_initialisation()

# random_initialisation with prior generated from the data
# ph_0, pv1gh_0, _ = random_initialisation()
# pvgvh_0 = prior_pvgvh(data)

ph, pv1gh, pvgvh, phgv, llik = run_em(
    data=data,
    ph=ph_0,
    pv1gh=pv1gh_0,
    pvgvh=pvgvh_0,
)

print("")
print("")
```

```
print("----- learned parameters -----")

print("\np(h):")
print(np.around(ph, 4))

print("\np(v_1|h):")
print(np.around(pv1gh, 4))

pvgvh = np.around(pvgvh, 4)
print("\np(v_t|v_{t-1},h):")
print("h = 1:")
print(pvgvh[:, :, 0])
print("h = 2:")
print(pvgvh[:, :, 1])
print("h = 3:")
print(pvgvh[:, :, 2])

print(f"\nlog likelihood for these parameters: {llik[-1]:.4f}")

print("\nposterior for first 10 rows:")
print(np.around(phgv, 4)[:10])
```

Appendix B

Code for Q2

1 Encoder matrix

```
import numpy as np
from numpy.typing import NDArray


def echelon(m: NDArray) -> NDArray:
    """Return the echelon version of a matrix

    Args:
        m: the matrix to be echeloned

    Returns:
        NDArray: an echelon version of the given matrix
    """
    if 0 in m.shape:
        # one of the rows or cols is zero, no more actions to take
        return m

    # check if first column is all zero
    if not np.any(m[:, 0]):
        # apply echelon on matrix minus first col
        m_sub = echelon(m[:, 1:])
        # append first col
        return np.c_[m[:, 0], m_sub]

    # find nonzero element indices in first col
    nonzero_idx = np.where(m[:, 0] != 0)[0]

    # if first nonzero element is not the first element,
```

```
    # i.e. first element is 0
    first_nonzero = nonzero_idx[0]
    if first_nonzero != 0:
        # add first nonzero row, making the first element 1
        m[0] = (m[0] + m[first_nonzero]) % 2

    # add the first row to the rows with nonzero elements
    # to make them 0
    remaining_nonzero_idx = nonzero_idx[nonzero_idx != 0]
    m[remaining_nonzero_idx] = (m[remaining_nonzero_idx] + m[0]) % 2

    # run echelon on the matrix minus first row
    m_sub = echelon(m[1:])

    # add the first row back and return
    return np.vstack((m[0], m_sub))


def rref(m: NDArray) -> NDArray:
    """Return the rref of a given matrix

    Args:
        m: the matrix to be rref

    Returns:
        NDArray: the rref version of m
    """
    m_ech = echelon(m)

    for i in reversed(range(m_ech.shape[0])):
        row = m_ech[i]

        # if row is all 0s, ignore
        if not np.any(row):
            continue

        # find the leading one col
        leading_one_col = np.argmax(row)

        # find indices of ones above the leading one and add leading one row
        ones = np.where(m_ech[:i, leading_one_col])[0]
        m_ech[ones] = (m_ech[ones] + m_ech[i]) % 2
```

```
    return m_ech


def create_encoder_matrix(h: NDArray) -> tuple[NDArray, NDArray]:
    """Generate an encoder matrix for a parity check matrix

    Args:
        h: a parity check matrix

    Returns:
        tuple[NDArray, NDArray]: a tuple of the rref equivalent
            form of h, and the generator matrix
    """
    H_rref = rref(h)

    m, n = H_rref.shape
    k = n - m

    P = H_rref[:, m:]
    H_hat = np.c_[P, np.identity(m)].astype(int)
    G = np.r_[np.identity(k), P].astype(int)

    return H_hat, G

H = np.array([
    [1, 1, 1, 1, 0, 0],
    [0, 0, 1, 1, 0, 1],
    [1, 0, 0, 1, 1, 0],
])

H_hat, G = create_encoder_matrix(H)

# generate some random ts and assert H_hat @ G @ t = 0
ts = np.random.randint(0, 2, size=9).reshape((3, 3))
for t in ts:
    assert not np.any((H_hat @ G @ t) % 2)
```

2 Decoding

```
import numpy as np
from numpy.typing import NDArray
```

```python
def initialise_probs(H: NDArray, y: NDArray, p: float) -> NDArray:
    """Initialise probabilities

    The log-likelihood of bit node v conditioned on its observed value.
    These act as the first messages to be passed from bit to check node.

    Args:
        H: a parity check matrix
        y: the word to decode
        p: the noise ratio

    Returns:
        NDArray: a matrix of initialised probabilities
    """
    log_lik = np.log((1 - p) / p)
    y_lik = np.where(y == 0, log_lik, -log_lik)
    return H * y_lik


def bit_to_check(m_v: NDArray, p_mat: NDArray) -> NDArray:
    """Pass messages from bit nodes to check nodes

    Args:
        m_v: the initial messages from bit to check
        p_mat: a matrix of the current probabilities

    Returns:
        NDArray: an updated matrix of probabilities
    """
    # initialise the return matrix as the initial bit to check messages
    new_p_mat = m_v.copy()

    # iterate through columns (bit nodes)
    for idx, col in enumerate(p_mat.T):
        # find the nonzero elements (the check nodes incident to this bit node)
        non_zeros = np.nonzero(col)
        # calculate the updated probabilities
        col_sums = np.sum(col[non_zeros]) - col[non_zeros]
        # add to the initial messages for this bit node
        new_p_mat.T[idx][non_zeros] += col_sums
```

```
    return new_p_mat


def check_to_bit(p_mat: NDArray) -> NDArray:
    """Pass messages from check nodes to bit nodes

    Args:
        p_mat: a matrix of the current probabilities

    Returns:
        NDArray: an updated matrix of probabilities
    """
    new_p_mat = np.zeros_like(p_mat)

    # iterate through the rows (check nodes)
    for idx, row in enumerate(p_mat):
        # find the nonzero elements (the bit nodes incident to this check node)
        non_zeros = np.nonzero(row)
        # calculate the updated message
        row_tanh = np.prod(np.tanh(row[non_zeros] / 2))
        prod_tanh = row_tanh / np.tanh(row[non_zeros] / 2)
        # update the row in the return matrix with the updated probabilities
        new_p_mat[idx][non_zeros] = np.log((1 + prod_tanh) / (1 - prod_tanh))

    return new_p_mat


def decode(
    H: NDArray,
    y: NDArray,
    p: float,
    max_iters: int = 20,
) -> tuple[NDArray, int, int]:
    """Decode a word using a parity check matrix

    Implements the Loopy Belief Propagation for Binary Symmetric Channel
    algorithm, as per LDPC Codes: An Introduction (Shokrollahi) and Information
    Theory, Inference, and Learning Algorithms (MacKay)

    Args:
        H: a parity check matrix
```

```
        y: the word to decode
        p: the noise ratio
        max_iters: a maximum number of iterations to reach convergence

    Returns:
        tuple[NDArray, int, int]: a tuple of the decoded message, a return code
            denoting whether the algorithm converged to an answer, and
            the number of iterations needed
    """
    return_code = -1
    # start off with decoded as the same as the received word
    decoded = y
    # tracking the number of iterations
    i = -1

    # initialise the probabilities matrix to be updated by the algorithm
    p_mat = initialise_probs(H, y, p)
    # freeze the initialised probabilities to be used as the first round of messa
    # from bit to check nodes
    m_v = p_mat.copy()

    # iterate through the algorithm
    for i in range(max_iters):
        # next round of message passing from check to bit nodes
        p_mat = check_to_bit(p_mat)

        # generate a tentative decoding
        decoded = np.where(np.sum(p_mat, axis=0) < 0, 1, 0)
        # see if decoded is a valid codeword
        if not np.any(H @ decoded % 2):
            # decoded is a valid codeword, we can halt the algorithm
            return_code = 0
            break

        # we don't have a valid codeword, proceed to next round of messages from
        # check nodes
        p_mat = bit_to_check(m_v, p_mat)

    return decoded, return_code, i


H = np.loadtxt("H1.txt")
```

```
y = np.loadtxt("y1.txt")

decoded, success, num_iterations = decode(H, y, 0.1)

if success == 0:
    print("---- SUCCESSFUL DECODING -----")

    original_msg = bytearray(np.packbits(decoded[:248])).decode().strip("\x00")
    print(f"decoded message: {original_msg}")
    print(f"number of iterations needed: {num_iterations + 1}")
else:
    print("---- unsuccessful decoding -----")
```

Appendix C

Code for Q3

1 Exact Inference

code date: 22.12.23 - 16:56

```
import numpy as np

def generate_potential(B):
    """Given Beta B, return a 2x2 array containing possible potential values

    Args:
        B (float): Beta constant used in potential function

    Returns:
        2x2 array: Contains all values of potential. Element (i,j) is phi(x1=i, x
    """
    potentials = np.ones((2,2))
    potentials[0,0] = np.exp(B)
    potentials[1,1] = potentials[0, 0]
    return potentials


def generate_column_states(n):
    """given the dimension n, return an array of all possible states for a binary

    Args:
        n (int): dimension of column

    Return:
        y : binary array of shape 2^n x n
    """
```

```
    y = np.zeros((pow(2, n), n)).astype(int)

    for state_index in range(pow(2, n)):
        state_binary = bin(state_index)[2:]

        for bit_index, bit_value in enumerate(reversed(state_binary)):
            y[state_index, bit_index] = bit_value

    return y


def f_0(y1, potential):
    """given binary column y of dim n, return the potential of y under f_0

    Args:
        y1 (n-array): n dimensional array (this is the first column in the nxn ar
        potential : 2x2 array of potential

    Return:
        message: return the message from this factor
    """

    factor = 1
    for row in range(y1.size - 1):
        factor *= potential[y1[row], y1[row+1]]

    return factor


def f_0_vectorize(column_states, potential):
    """given an array of all possible column states, return an array of all poten
    This is vectorized version of previously defined function f_0

    Args:
        column_states (nxm array): array of m different states of binary column
        potential (2x2 array): all possible values of potential
    """
    m = column_states.shape[0]
    f0 = np.zeros(m)

    for y_i in range(m):
```

```
        f0[y_i] = f_0(column_states[y_i], potential)

    return f0


def f_i(y1, y2, potential):
    """given column 1 and 2 vectors, calculate the shared potential between these

    Args:
        y1 (n-array): column y1, binary array
        y2 (n-array): column y2, binary array
        potential (2x2 array): potential to be applied to all neighbours cells

    Return:
        message: return the message from y1 and y2
    """
    factor = 1

    # calculate potentials going down the 2nd column and shared edges between y1
    for row in range(y1.size-1):
        factor *= potential[y2[row], y2[row+1]]
        factor *= potential[y1[row], y2[row]]
    # above for loop misses out last row - calculate that outside loop
    factor *= potential[y1[-1], y2[-1]]

    return factor


def f_i_vectorize(column_states, potential):
    """given an array of all possible column states, return an array of all poten
    This is a vectorized version of function f_i previously defined

    Args:
        column_states (nxm array): array of m different binary states of length n
        potential (2x2 array): 2x2 array containing the possible potential values
    """
    m = column_states.shape[0]
    fi = np.zeros((m,m))

    for y_i in range(m):
        for y_j in range(m):
            fi[y_i, y_j] = f_i(column_states[y_i], column_states[y_j], potential)
```

```
    return fi


def caclulate_desired_prob_dist(n=10, B=1):
    """Given size of lattice (n) and Beta value (B), print the probability
    distribution of the top right and bottom right binary nodes

    Args:
        n (int, optional): Size of nxn lattice. Defaults to 10.
        B (int, optional): Beta value used in Potential. Defaults to 1.
    """
    column_states = generate_column_states(n)
    potential = generate_potential(B)

    # calculate transition potential matrixes of all possible states
    f0 = f_0_vectorize(column_states, potential)
    fi = f_i_vectorize(column_states, potential)

    # calculate log_messages passing through lattice columns
    log_message = np.log(f0)
    log_message_max = log_message.max()
    for col in range(1, n):
        log_message = log_message_max + np.log(np.exp(log_message - log_message_max
        log_message_max = log_message.max()

    # calculate log_Z (total potential of model)
    log_Z = log_message_max + np.log(np.exp(log_message - log_message_max).sum())

    # marginalise over all unwanted variables in final column
    log_message_x1_xn = np.zeros((2,2))
    n = column_states.shape[1]
    for state_index, state in enumerate(column_states):
        i = state[0]
        j = state[n-1]
        log_message_x1_xn[i,j] += np.exp(log_message[state_index] - log_message_max
    log_message_x1_xn = np.log(log_message_x1_xn)
    log_message_x1_xn += log_message_max

    # calculate the final join distribution
    joint_dist = np.exp(log_message_x1_xn - log_Z)
```

```
    # print results
    print(f"\nn: {n}, Beta: {B}")
    print(f"log(Z): {log_Z}")
    print(f"joint_dist of top right and bottom left nodes: \n{joint_dist}")



# Pull it all together to calculate the desired solution
inputs = np.array([[10, 0.01], [10, 1], [10, 4]])
for input in inputs:
    caclulate_desired_prob_dist(n=int(input[0]), B=input[1])
    print("---------------")
```

2 Mean Field Approximation

```
import numpy as np
import math

b = 1 #set beta

#initialize the lattice containing random values of q(x)
lattice = np.random.rand(10, 10) #set random initial values for marginal dist

def neighbours(lattice,x,y):
    '''
    Function that returns neighbour values in a list of given point on lattice
    Input:
    lattice (array) : mxn array
    x (int) : x position on lattice < m
    y (int) : y position on lattice < n
    Output:
    neighbours (list) : list of values of the neighbours on the lattice
    '''
    m = lattice.shape[0]
    n = lattice.shape[1]
    neighbours = []

    if x > 0:
        neighbours.append(lattice[x-1,y])
    if x < (m - 1):
        neighbours.append(lattice[x+1,y])
    if y > 0:
```

```
        neighbours.append(lattice[x,y-1])
    if y < (n - 1):
        neighbours.append(lattice[x,y+1])
    return neighbours

#define q_1
def q_1(lattice, x, y):
    q_1 = math.exp(b*sum(neighbours(lattice, x, y)))
    return q_1

#define q_0
def q_0(lattice, x, y):
    n_0 = [(1 - n) for n in neighbours(lattice, x, y)]
    q_0 = math.exp(b*sum(n_0))
    return q_0

#iterate through the lattice working through each node xk updating q(xk)
for iter in range(1000):
    for y in range(10):
        for x in range(10):
            lattice[x,y] = q_1(lattice,x,y)/(q_1(lattice,x,y) + q_0(lattice,x,y))

print(lattice[0,9], lattice[9,9])

#joint dist
dist_09 = np.array([1-lattice[0,9], lattice[0,9]])
dist_99 = np.array([1-lattice[9,9], lattice[9,9]])
joint_dist = np.array([[dist_09[0]*dist_99[0], dist_09[0]*dist_99[1]], [dist_09[1]*
print(joint_dist)
```

3 Gibbs Sampling

```
import numpy as np
import math
import random

b = 4#set beta

#initialize the lattice random values
lattice = np.random.binomial(1, 0.5, size=(10, 10))

#set number of iterations and initial number of samples to burn
```

```
iter = 10000
burn = 1000

#create an array to store the samples - only interested in the values at positior
samples = np.zeros((iter, 2))

def neighbours(lattice,x,y):
    '''
    Function that returns neighbour values in a list of given point on lattice
    Input:
    lattice (array) : nxn array
    x (int) : x position on lattice < n
    y (int) : y position on lattice < n
    Output:
    neighbours (list) : list of values of the neighbours on the lattice
    '''
    neighbours = []
    m = lattice.shape[0]
    n = lattice.shape[1]
    if x > 0:
        neighbours.append(lattice[x-1,y])
    if x < m-1:
        neighbours.append(lattice[x+1,y])
    if y > 0:
        neighbours.append(lattice[x,y-1])
    if y < n-1:
        neighbours.append(lattice[x,y+1])
    return neighbours

#define conditional probability
def cond_prob(lattice, x, y):
    p = (math.exp(b*sum(neighbours(lattice, x, y))))/(math.exp(b*sum(neighbours(lat
    return p

#iterate through the sampling process
for i in range(iter + burn):
    #move through the nodes in a random order
    indices = [(row, col) for row in range(10) for col in range(10)] # Generate a
    random.shuffle(indices) # Shuffle the list of indices

    for x, y in indices:
        # Calculate the conditional probability based on the neighbours
```

```
            p = cond_prob(lattice, x, y)
            # Sample from the full conditional distribution of lattice[x,y], given th
            lattice[x,y] = np.random.binomial(1, p)
        if i > burn: # Store the samples of both corners for samples after burn perio
            samples[i - burn,0] = lattice[0,9]
            samples[i - burn,1] = lattice[9,9]

p_0_0 = np.count_nonzero(np.all(samples==np.array([0,0]), axis=1)) / iter
p_0_1 = np.count_nonzero(np.all(samples==np.array([0,1]), axis=1)) / iter
p_1_0 = np.count_nonzero(np.all(samples==np.array([1,0]), axis=1)) / iter
p_1_1 = np.count_nonzero(np.all(samples==np.array([1,1]), axis=1)) / iter

joint_dist = np.array([[p_0_0, p_0_1], [p_1_0, p_1_1]])
print(joint_dist)
```

References

1. Barber, D. *Bayesian Reasoning and Machine Learning* (Cambridge University Press, 2012).
2. MacKay, D. J. *Information Theory, Inference, and Learning Algorithms* ISBN: 9780521642989 (Cambridge University Press, 2005).
3. Shokrollahi., A. *LDPC codes: An introduction* (2002).