COMP0080 Graphical Models

Coursework 1 Report

Team 2

Mazin Abdulmahmood (18064052)

Ying-Lun Cheng (17126493)

Department of Computer Science

December 21, 2022

Contents

1 Brief Introduction

This report details our results, derivations and experimentation details for the Graphical Models coursework in which we were very pleased with the outcomes to our results!

2 Part 1: Inference and Learning

2.1 Question 1: Two Earthquakes

a.

Note: *The code for this question is in Appendix Part 1 - Question 1.*

This question we look at predicting the location of two earthquakes at locations denoted S_1 and S_2 given a set of observed sensor values $v \in \{v_1, ..., v_N\}$. Given a spiral coordinate system over the land where $s \in [(e_{x_1}, e_{y_1}), ..., (e_{x_K}, e_{y_K})]$, and K is set to 2000 points, the expected sensor values at each sensor for two explosions at positions i, j is given as:

$$v_i = \frac{1}{d_i^2 + 0.1} + \frac{1}{d_j^2 + 0.1} + \sigma\epsilon_i \tag{1}$$

where, $d_i^2 = (x_i - s_{1x})^2 + (y_i - s_{1y})^2$, denotes the distance from the sensor to any potential explosion point within the coordinate system, and we expect additional noise that is normally distributed with standard deviation, $\sigma = 0.2$, on our observed values. For an observed sensor value we can then say the probability it belonged to a given location within the coordinate system is normally distributed over all expected signal values to the sensor v_i.

$$p(v_i|s_1, s_2) = \frac{1}{\sqrt{2\pi\sigma^2}}\, e^{-\frac{1}{2\sigma^2}\left(v_i - \frac{1}{d_i^2+0.1} + \frac{1}{d_j^2+0.1}\right)} \tag{2}$$

We assume that the observed sensor values are independent given the explosions location and that the explosions location is equally likely for any coordinate point, that is we have a uniform prior $p(s_x, s_y)$. This gives a belief network as shown in Figure 1.

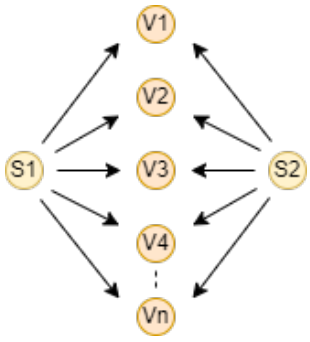

Figure 1: Belief network for two explosions problem.

The joint p.d.f for our simple model is:

$$p(v_1, ..., v_N, s_1, s_2) = p(s_1)p(s_2)\prod_{i=1}^{N} p(v_i|s_1, s_2) \tag{3}$$

For a set of observed sensor values and a target posterior $p(s_1|v_{1,\dots,N})$ we start at the posterior over possible locations and keeping in mind our independence assumptions it breaks down to:

$$p(s_1, s_2|v_{1,\dots,N}) = \frac{p(s_1)p(s_2)\prod_{i=1}^{N} p(v_i|s_1, s_2)}{\sum_{s_1,s_2} p(s_1)p(s_2)\prod_{i=1}^{N} p(v_i|s_1, s_2)}$$

in which we marginalise s_2 out to obtain the target posterior $p(s_1|v_{1,\dots,N})$ while cancelling the uniform priors:

$$p(s_1|v_{1,\dots,N}) = \frac{\prod_{i=1}^{N}\sum_{s_2} p(v_i|s_1, s_2)}{\sum_{s_1,s_2}\prod_{i=1}^{N} p(v_i|s_1, s_2)}$$

This multiplication on our Gaussian likelihood function can lead to very small numbers that may exceed the machine's precision during calculation and so it is advisable to take the log over the posterior Gaussian and transform this back later to a posterior probability once we calculate it.

$$log\left[p(s_1|v_{1,\dots,N})\right] = \frac{log\left[\prod_{i=1}^{N}\sum_{s_2} p(v_i|s_1, s_2)\right]}{log\left[\sum_{s_1,s_2}\prod_{i=1}^{N} p(v_i|s_1, s_2)\right]} = \frac{\sum_{i=1}^{N}\sum_{s_2} log\left[p(v_i|s_1, s_2)\right]}{\sum_{i=1}^{N}\sum_{s_1,s_2} log\left[p(v_i|s_1, s_2)\right]} \quad (4)$$

Carrying this out over the two explosions we take the $argmax_s p(s_1|v_{1,\dots,N})$ as the most likely explosion locations from the observed data and plot this in Figure 2.

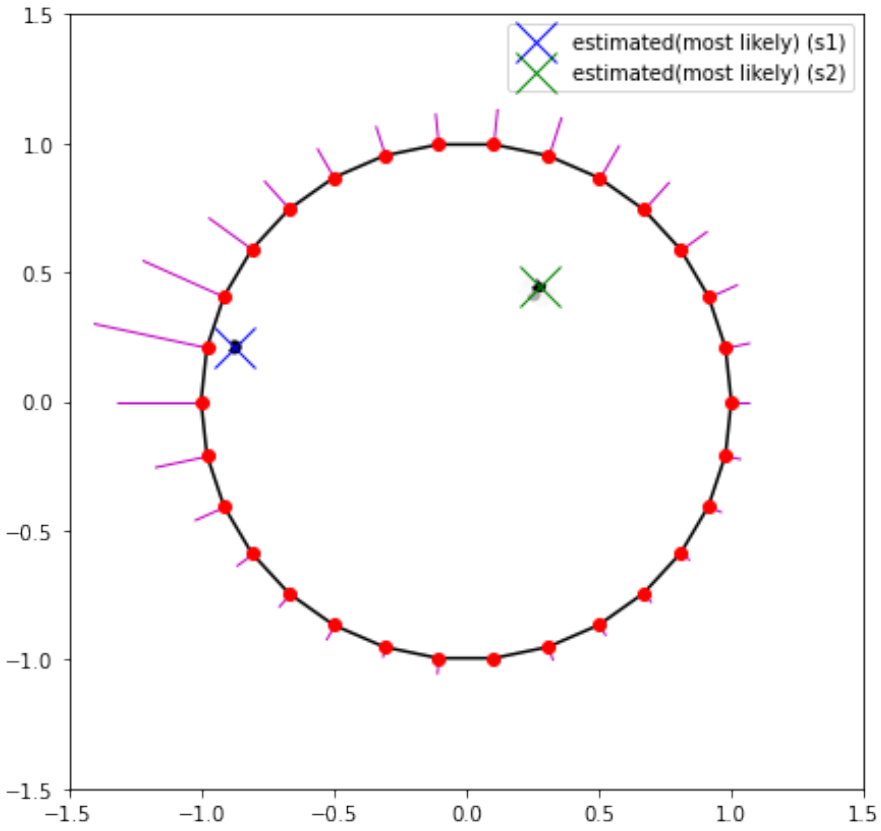

Figure 2: Posterior probability $p(s|v)$ over all coordinate positions where the crosses represent the most likely explosion points given the sensor data (blue = s_1, green = s_2). The red dots represent the sensor locations and the magenta lines represent the magnitude of observed data at each sensor.

In our method we conserve the marginalisation across position s_2 and instead compute a Gaussian likelihood for all combinations of (s_1, s_2) in which the maximum indicates both our positions. Note we plot both possible locations, but for s_1 refer to the blue cross.

b.

Writing out two hypothesis, $\mathcal{H}$, such that $\mathcal{H}_1$ relates the hypothesis that there is only one explosion, $s = (s_1)$, and $\mathcal{H}_2$ that there is two explosions, $s = (s_1, s_2)$, we wish to identify the value of $logp(v_{1,...,N}|\mathcal{H}_2) - logp(v_{1,...,N}|\mathcal{H}_1)$. Taking a look at Equation 4 in the previous section we see this is simply the likelihood there would be two explosions versus just one explosion, which is our numerator. We can recalculate this for the case of one explosion and this breaks down to:

$$\sum_{i=1}^{N} log\left[p(v_i|s_1)\right] \tag{5}$$

For our most probable point, that is the point that would generate and satisfy our hypothesis, we can simply take this log likelihood that we have already calculated in our code and subtract the log likelihoods for $\mathcal{H}_1$ from $\mathcal{H}_2$. We point out here that when identifying the most probable point, we are generating a matrix of Gaussian likelihoods for each location, $s \in [(e_{x_1}, e_{y_1}), ..., (e_{x_K}, e_{y_K})]$, and taking the max likelihood. This gives us

$$logp(v_{1,...,N}|\mathcal{H}_2) - logp(v_{1,...,N}|\mathcal{H}_1) = 742.09$$

c.

Computing $logp(v_{1,...,N}|\mathcal{H}_2) - logp(v_{1,...,N}|\mathcal{H}_1)$ through log law is equivalent to:

$$log\left[\frac{p(v_{1,...,N}|\mathcal{H}_2)}{p(v_{1,...,N}|\mathcal{H}_1)}\right]$$

Using Bayes theorem we can break down the posterior further:

$$log\left[\frac{\frac{p(\mathcal{H}_2|v_{1,...,N})p(v_{1,...,N})}{p(\mathcal{H}_2)}}{\frac{p(\mathcal{H}_1|v_{1,...,N})p(v_{1,...,N})}{p(\mathcal{H}_1)}}\right] = log\left[\frac{p(\mathcal{H}_2|v_{1,...,N})p(\mathcal{H}_2)}{p(\mathcal{H}_1|v_{1,...,N})p(\mathcal{H}_1)}\right]$$

which for no prior preference, that is $p(\mathcal{H}_1) = p(\mathcal{H}_2)$, we get:

$$log\left[\frac{p(\mathcal{H}_2|v_{1,...,N})}{p(\mathcal{H}_1|v_{1,...,N})}\right] \tag{6}$$

giving us the relation between the hypotheses to each other. The log function is monotonically increasing for all positive values and so a larger argument (larger probability of $\mathcal{H}_2$), validly relates the probability of the two explosions versus one explosion.

d.

The computational complexity increases by $O(SN^K)$ where S is number of stations, N is the number of points in our coordinate system, and K is the number of explosions. This is

because for each increase in the number of explosions we have to solve, for each position in the system, n, where the next explosion could be and so on until we have solved for K explosions. For a set number of sensors this gives an $O(N^K)$ increase in K increase of explosions.

2.2 Question 2: Meeting Scheduling

a.
Given that our target time is 21:05 arrival, for a given friend's delay, D_i, and an independence assumption on each friend arriving late we can model the probability of all our friend's delays as a product (and rule):

$$p(D_1, D_2, ..., D_N) = \prod_{i=0}^{N} p(D_i) \tag{7}$$

where $\forall i, T_0 + D_i < 21:05$ if we wish to make our train. To meet this requirement we note that the given delay probabilities are constant over ranges. We can therefore tabulate the cumulative probability of a delay for the upper-bound of each range, the worst possible time for that delay range. This can be seen in Table 1.

Table 1: Cumulative probability of delay for a given T_0.

T_0	D_{max}	$p(D_{max})$
21:05	$\mathrm{p}(\mathrm{D}_i \leq 0)$	0.70
21:00	$\mathrm{p}(\mathrm{D}_i < 5)$	0.80
20:55	$\mathrm{p}(\mathrm{D}_i < 10)$	0.90
20:50	$\mathrm{p}(\mathrm{D}_i < 15)$	0.97
20:45	$\mathrm{p}(\mathrm{D}_i < 20)$	0.99
<20:45	$\mathrm{p}(\mathrm{D}_i \geq 20)$	1.0

In order to make the train, all of our friends must arrive by 21:05, so we must compute the joint delay cumulative probability for each friend for the worst-case scenario, that being they all arrive by this upper-bound we have on T_0 with probability of 90%. This is simply equal to

$$p(D_{1,2,...,N} < 21:05 - T_0) = p(D < 21:05 - T_0)^N$$

where this probability should be ≥ 0.9 as then we are 90% sure our friends will make it by 21:05. For example for 3 friend's the probability they all arrive by 21:05 for a $T_0 = 20:45$ is $p(D_i < 20)^3 = 0.99^3 = 0.97$. This calculation is repeated for each delay, working backwards from the earliest T_0, giving us:

Table 2: Joint probability of delays for a given T_0.

	N		
T_0	3	5	10
20:45	0.97	0.95	0.90
20:50	0.91	0.86	0.74
20:55	0.73	-	-
21:00	-	-	-
21:05	-	-	-

The red times show that the delay for that given T_0 means that we are below 90% certainty of all our friends making it by 21:05 and so the time in the above cell indicates the latest we can set T_0. This is summarised as below

$$N = 3 : T_0 = 20 : 50$$

$$N = 5 : T_0 = 20 : 45$$

$$N = 10 : T_0 = 20 : 45$$

b.

We now have new data on how punctual our friends are, Z_i. So far we have assumed all our friends will be punctual to their designated delay time frames, $Z_i = 1$, but now we have uncertainty on these time ranges in that some friends may not be as punctual, $Z_i = 0$. We update the model with these new cumulative probabilities as seen in Table 3.

Table 3: Updated cumulative probability of delay given a T_0.

T_0	D_{max}	$p(D_{max}\|Z_i = 1)$	$p(D_{max}\|Z_i = 0)$
21:05	$\mathrm{p}(\mathrm{D}_i \leq 0)$	0.70	0.50
21:00	$\mathrm{p}(\mathrm{D}_i < 5)$	0.80	0.70
20:55	$\mathrm{p}(\mathrm{D}_i < 10)$	0.90	0.80
20:50	$\mathrm{p}(\mathrm{D}_i < 15)$	0.97	0.90
20:45	$\mathrm{p}(\mathrm{D}_i < 20)$	0.99	0.95
<20:45	$\mathrm{p}(\mathrm{D}_i \geq 20)$	1.0	1.0

We also have a prior that our friends are punctual with certainty $p(Z_i = 1) = \frac{2}{3}$, and that they are not punctual with certainty $p(Z_i = 0) = \frac{1}{3}$. If we ask our friends to arrive at the same T_0 that we calculated in Part A, we may have a lower probability than 90% in making the train.

Currently we can say that using the T_0's above for each N amount of friends, we are $\geq$90% certain we will make the train, meaning we are $\leq$10% certain we will miss the train, that is:

$$\overline{p(train)} = 1 - p(D_{1,2,\ldots,N} < 21 : 05 - T_0)$$

We will abbreviate $21:05 - T_0$ as T^*. This probability assumes $Z_i = 1$, but now we have two possible states, we have to account for this by de-marginalising over Z_i:

$$= 1 - \left[\sum_{Z_i} p(D < T^*, Z_i) \right]^N$$

Using Bayes Theorem:

$$= 1 - \left[\sum_{Z_i} p(D < T^* | Z_i)\, p(Z_i) \right]^N$$

This means that we consider the cases that our friends may not be punctual, as well as punctual through the 'or rule' (the summation over Z_i). Using this model we calculate the probability that we miss the train using the same T_0 as:

$$N = 3, T_0 = 20:50:1 - (0.97 * \frac{2}{3} + 0.90 * \frac{1}{3})^3 = 0.15$$
$$N = 5, T_0 = 20:45:1 - (0.99 * \frac{2}{3} + 0.95 * \frac{1}{3})^5 = 0.11$$
$$N = 10, T_0 = 20:45:1 - (0.99 * \frac{2}{3} + 0.95 * \frac{1}{3})^1 0 = 0.21$$

As seen, in all cases due to our updated prior beliefs, we are now more than 10% certain that we may miss the train.

c.

Given that we miss the train, the posterior distribution over the number of friends that were not punctual is generated via Bayes theorem:

$$p(N_L = n | \overline{train}) = \frac{p(\overline{train} | N_L = n)\, p(N_L = n)}{p(\overline{train})} \tag{8}$$

$p(N_L = n)$ relates the prior on the number of friends, n, being late/non-punctual (L and non-punctual will be used interchangeably). Given $Z_i \in \{0, 1\}$, we have a binomial distribution as each friend can be punctual or not punctual for N amount of friends (Equation 7 AND rule). We therefore have a probability of $P_L = 1 - P_{NL}$ over N trials which suits the binomial distribution. Therefore the probability of n friends being late/non-punctual is:

$$p(N_L = n) = \binom{N}{n} p_L^n \, (1 - p_L)^{N-n} \tag{9}$$

We note for all previous probabilities we have essentially been using an altered binomial for the AND rule products but we used $n = N$, that is, we assumed the worst case of all trials releasing the same outcome.

For $N = 5$ and our probabilities $P_L = \frac{1}{3}$, $\overline{P_L} = \frac{2}{3}$ we essentially wish to compute the probability of lateness for N trials over all combinations of friends, this is given in Table 4.

Table 4: Probability of n friends being non-punctual.

n	$p(N_l = n)$
0	0.13
1	0.33
2	0.33
3	0.16
4	0.04
5	$4.16e^{-3}$

Note that we calculate the base case of $n = 0$ as even if we assumed all friends were punctual, as in the first part, we were only 95% certain we would still make the train, even when no one was late. Therefore, even if all our friends are punctual we could still miss the train and so must consider the posterior over this base case.

We now have the prior $p(N_L = n)$, we will move onto the normalising constant in the denominator. We can expand this by de-marginalising over the number of late friends and using Bayes rule to match the likelihood function as such:

$$p(\overline{train}) = \sum_n p(\overline{train}, N_L = n) = \sum_n p(\overline{train}|N_L = n)\, p(N_L = n)$$

We see it is simply normalising over all cases of the number of late friends.

Finally, we can rewrite our likelihood in a more familiar form. We note the probability of missing the train given n friends being late is just a joint over the probability of missing the train but n friends are non-punctual, so we can use our AND rule equation from above once again:

$$p(\overline{train}) = 1 - \left[\sum_{Z_i} p(D < T^*|Z_i)\, p(Z_i)\right]^N$$

but now we are not considering all N friends being one state or the other, but only n of them being in a certain state. For example, in the case of $n = 1$, the probability of missing the train is $p(D < T^*|Z_i = 0)^1 \cap p(D < T^*|Z_i = 1)^4$, which is the AND rule that 1 friend is late AND 4 friends are not late. Therefore, for $N_L = n$ we can generalise this and the likelihood function can be reformulated as:

$$p(\overline{train}|N_L = n) = 1 - p(D < T^*|Z_i = 0)^n \cdot p(D < T^*|Z_i = 1)^{N-n} \tag{10}$$

This is similar form to our binomial, but we do not consider a combination of all trials through the 'Choose' function as our probabilities are arbitrary for the independence assumption on each friend being delayed and is not affected by ordering. The final posterior is therefore:

$$p(N_L = n|\overline{train}) = \frac{\left[1 - p(D < T^*|Z_i = 0)^n \cdot p(D < T^*|Z_i = 1)^{N-n}\right]\; p(N_L = n)}{\sum_n p(\overline{train}|N_L = n)\, p(N_L = n)} \tag{11}$$

We use Python to compute this and calculate the posteriors given in Table 5.

Table 5: Probability of n friends being non-punctual given we missed the train.

n	$p(N_L = n\|\overline{train})$
0	0.06
1	0.26
2	0.37
3	0.23
4	0.07
5	0.01

2.3 Question 3: Three Weather Stations

A and B

Note: *The code for this question is in Appendix Part 1 - Question 3.*

Taking a look at our dataset we have 500 sequences, v, that we wish to attribute a weather station, h, to. Examining our data we see that our dataset contains values $v \in V, \{0, 1, 2\}$ and for our 3 weather stations we can also mark them as $h \in H\, \{0, 1, 2\}$. In our dataset we have, V (500), sequences each of dimension, L, $v^{1,...,T} = \{v_1^t, v_2^t, ..., v_L^t\}$. We believe that each sequence, v^t, is modelled by a first-order Markov chain with each sequence associated with one hidden state, the station that generated it, h. Figure 3 shows this.

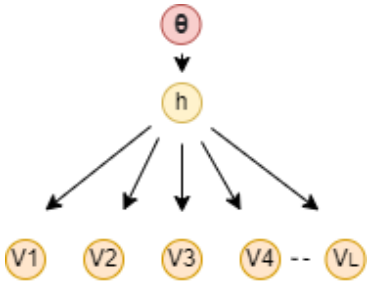

Figure 3: Hidden Markov Model for Weather-Station problem (in practice θ imposes dependencies for all nodes).

With HMM probability decomposition as such:

$$p(v|\theta) = \sum_{h \in H} p(h_k|\theta) p(v|h_k, \theta) \tag{12}$$

$$where\ p(v|h_k, \theta^k) = p(v_1^t|h_k, \theta^k) \prod_{l=1}^{L} p(v_l^t|v_{l-1}^t, h_k, \theta^k)$$

Each component of our model is also controlled by parameters, θ, which we wish to estimate from the dataset using Expectation-Maximisation Baum-Welch algorithm. The parameters of our model follow typical HMM parameters:

$$\theta^k \begin{cases} \pi_k : \pi_k = p(h = k|\theta^k) \\ P_{i,j} : P_{i,j} = p(v_l = i|v_{l-1} = j, h = k, \theta^k)\ i, j \in \{0, 1, 2\} \\ Q_i : Q_i = p(v_1 = i|h = k, \theta^k) \end{cases}$$

where π is the global prior probability parameter that a data sequence could be generated from a given weather station,k , $p(h = k)$. $P_{i,j}$ is the transition matrix of a given observed data point, v, transitioning into a new value and has an associated probability for each possible change (i.e 3x3 in this case). Finally, Q_i is the initial state probability for a sequence under a given weather station, h, and is $p(v_1 = i|h = k)$. Each weather station will have their own set of estimated parameters so we will have 3 sets of θ.

Expectation Step

We maximise the likelihood $p(v|\theta)$ by defining a lower bound on the log likelihood and then proceed to increase this bound iteratively. The LB is a function of our parameters, θ, and an auxiliary variational distribution, $q(h)$, in which $q(h)$ is the probability distribution over our hidden state, h. For a set θ we update our variational distribution by optimising the free-energy of our w.r.t q giving us

$$q_{new} = argmax_q \, log\left[p(v|\theta_{old})\right] = \left\langle \frac{p(v,h|\theta_{old})}{q(h)} \right\rangle_{q(h)}$$

which is optimal when setting the free-energy to the posterior, $p(h|v,\theta)$, for minimum KL divergence distance.

Maximisation Step

We now optimise for our new parameters, θ_{new}, w.r.t to our newly calculated $q(h)_{new}$ by obtaining parameters that correspond to the maximum expected bound:

$$\theta_{new} = argmax_\theta \left\langle log\, p(v,h|\theta) \right\rangle_{q(h)} + H(q)$$

$$= \theta_{new} = argmax_\theta \left\langle log\, p(v,h|\theta) \right\rangle_{q(h)}$$

Where $q(h) = p(h|v,\theta_{old})$ and we drop $H(q)$ as we are optimising w.r.t a fixed variational distribution and so the term does not contribute once differentiated to the maximal parameters. Using Baum-Welch our free-energy can be thought of as the form:

$$F^{'}(\theta,\theta_s) = \sum_h log\, p(v,h|\theta)\, p(v,h|\theta_s))$$

and

$$p(v,h|\theta) = p(v|h,\theta)p(h|\theta) \overset{log}{\rightarrow} log\, p(v|h,\theta) + log\, p(h|\theta)$$

Since all sequences are i.i.d we break down the likelihood into a product over all sequences which by log product rule becomes a sum:

$$= \sum_{t=1}^{T} \left[log\, p(v^t|h^t,\theta) + log\, p(h^t|\theta)\right]$$

Which we can plug in the decomposition for $p(v^t|h^t,\theta)$ from Equation 12 and also substitute our parameters in giving us a new free-energy:

$$= \sum_{t=1}^{T} log\, p(v_1^t|h^t,\theta) + \sum_{t=1}^{T}\sum_{l=2}^{L} log\, p(v_l^t|v_{l-1}^t,h^t,\theta) + \sum_{t=1}^{T} log\, p(h^t|\theta)$$

$$F^{'}(\theta, \theta_s) = \sum_{h \in H} p(h, v|\theta_s) \left[\sum_{t=1}^{T} \sum_{i \in V} log\, Q^{h}_{v_l=i} + \sum_{t=1}^{T} \sum_{l=2}^{L} \sum_{i,j \in V} log\, P^{h}_{v_l^t=i, v_{l-1}^t=j} + \sum_{t=1}^{T} log\, \pi^{t}_{h} \right]$$

Optimising w.r.t to our parameters θ's over our old parameters θ_{old} and approximating our posterior over our hidden states to be active if and only if it belongs to that weather station, which we use an indicator Dirac spike function to model, $\mathbb{I}[h^t = h^*]$, we get the following iterative updates similar to those in the Bernouli notes:

$$\pi^{new}_{h'} \equiv p^{new}(h = h'|\theta_{old}) \propto \sum_{t=0}^{T} p(h = h'|v^t, \theta_{old}) = \frac{\sum_{t=0}^{T} p(h = h'|v^t, \theta_{old})}{\sum_{h \in H} \sum_{t=0}^{T} p(h = h|v^t, \theta_{old})} \tag{13}$$

$$Q^{h',new}_{v_l=i} \equiv p(v_1 = i|h = h', \theta_{old}) = \frac{\sum_{t=0}^{T} \mathbb{I}[v_1^t = i] p(h = h'|v^t, \theta_{old})}{\sum_{k \in V} \sum_{t=0}^{T} \mathbb{I}[v_1^t = k] p(h = h'|v^t, \theta_{old})} \tag{14}$$

$$P^{h',new}_{i,j} = p(v_l = i|v_{l-1} = j, h = h', \theta^k) = \frac{\sum_{t=0}^{T} \sum_{l=1}^{L} \mathbb{I}[v_l^t = i] \mathbb{I}[v_{l-1}^t = j] p(h = h'|v^t, \theta_{old})}{\sum_{k,l \in V} \sum_{t=0}^{T} \sum_{l=1}^{L} \mathbb{I}[v_l^t = k] \mathbb{I}[v_{l-1}^t = l] p(h = h'|v^t, \theta_{old})} \tag{15}$$

Carrying out these updates we get the following parameters:

Global prior probability over our latent variables was found to be:

$$\pi_{h=0} = 0.197$$
$$\pi_{h=1} = 0.283$$
$$\pi_{h=2} = 0.520$$

The initial state distribution where each column is for $v_1 = col$ (i.e first column for each result is $v_1 = 0$), was found to be:

$$Q^{h=0}_{v_1} = [0.427,\ 0.573,\ 0]$$
$$Q^{h=1}_{v_1} = [0.238,\ 0.511,\ 0.251]$$
$$Q^{h=2}_{v_1} = [0.509,\ 0.367,\ 0.124]$$

And finally the transition probability distribution where each $[i, j] = p(v_l = j|vl - 1 = i$ (please carefully note the ordering of i,j):

$$P^{h=0}_{i,j} = \begin{bmatrix} 0.387 & 0.148 & 0.465 \\ 0.240 & 0.516 & 0.244 \\ 0.547 & 0.018 & 0.435 \end{bmatrix}$$

$$P^{h=1}_{i,j} = \begin{bmatrix} 0.058 & 0.140 & 0.802 \\ 0.130 & 0.322 & 0.548 \\ 0.423 & 0.426 & 0.151 \end{bmatrix}$$

$$P^{h=2}_{i,j} = \begin{bmatrix} 0.069 & 0.495 & 0.436 \\ 0.136 & 0.226 & 0.638 \\ 0.513 & 0.435 & 0.052 \end{bmatrix}$$

And the identified log likelihood of the data for these parameters was:

$$log - likelihood = -73.671$$

Computing the posterior over the first 10 data points we get:

$$p(h|v_{1,\ldots,10},\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0.996 & 0.004 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{bmatrix}$$

Where the model seems very certain for most posteriors and their respective weather-station clusters.

C.

The learned parameters depend on the initialised values as the EM algorithm guarantees convergence but only to a local maximum of the likelihood and therefore depending on the initial values and the climb of the iterations, we could end up and different maximas each time. For this we could initialise our parameters randomly and perform CV type training until we find a likelihood that maximises over the previous initialisations. We choose to randomly initialise our parameters, however, as this is a heavily researched area but recommendations could be to initialise based on simple fitting of the data to a basic model and iterate using those converged parameters derived from that simpler model.

Sometimes the EM algorithm will completely output the wrong parameters for our clusters and will predict overlapping of points (matching likelihoods to different h) into different regions of points that are well outside of the expected regions they should be in. This anomaly happens if the generated data is very sparsely distibuted in which we may get numerical errors or overflows. Some initialisations could mean infinitely large values to our likelihoods or likelihoods well below the precision of the machine giving us NaN values. Reading up further, when we have outliers in the data or we have repitition with our data this could cause 'collapsing' of the EM algorithm. Outliers may cause our parameters to try to model that outlier instead of our data which we did not account for measurement error in our model which we noticed occurs when using EM in other clustering situations like Gaussian clustering.

Lastly, one issue encountered was that the clusters would sometimes 'switch places' that is running multiple times sometimes the results for $h = 1$ would switch with $h = 2$ which we assume is just arbitrary cluster numbering or due to the overlapping points problem we saw before. This was resolved by a simple re-run of the algorithm on new initialisations.

d.

If we initialise our parameters to be uniform over V, H then we would initialise with prior probability of $\frac{1}{3}$ for all. The problem with initialising with a flat probability distribution

is that it leads to a similar 'flat' output for our updates, over all the updates which in turn means unchanging parameters as they all scale with each other. We can prove this as such, taking x to be the equal prior for all distributions we get:

$$\pi_{h'}^{new} = \frac{\sum_{t=0} x}{\sum_{h \in H} \sum_{t=0} p(h = h | v^t, \theta_{old})} = \frac{Tx}{\sum_{t=0} 1} = x$$

$$Q_{v_l=i}^{h',new} = \frac{\sum_{t=0}^{T} \mathbb{I}[v_1^t = i]x}{\sum_{k \in V} \sum_{t=0}^{T} \mathbb{I}[v_1^t = k]x} = \frac{n_i x}{Tx} = \frac{n_i}{T}$$

$$P_{i,j}^{h',new} = \frac{\sum_{t=0}^{T} \sum_{l=1}^{L} \mathbb{I}[v_l^t = i]\mathbb{I}[v_{l-1}^t = j]x}{\sum_{k,l \in V} \sum_{t=0}^{T} \sum_{l=1}^{L} \mathbb{I}[v_l^t = k]\mathbb{I}[v_{l-1}^t = l]x} = \frac{n_{i,j} x}{\sum_{t=0}^{T} (L-1)x} = \frac{n_{i,j}}{T(L-1)}$$

where n_i is the number of times $v_1 = i$ for that Q_i, $n_{i,j}$ is the number of transitions matching $P_{i,j}$. Combining our updates over all states of h' we see these simply just give an overall constant update to all parameters:

$$\frac{\pi_{h'}^{new} Q_{v_l=i}^{h',new} P_{i,j}^{h',new}}{\sum_{h'} \pi_{h'}^{new} Q_{v_l=i}^{h',new} P_{i,j}^{h',new}} = \frac{1}{3}$$

Giving a constant for the next round of updates, and so we maintain the same uniform distributions for over all parameters and hence the EM never actually converges and so is not a good idea.

3 Part 2: LDPC Codes

3.1 Question 1: Systematic encoding

Note: *The code for this question is in Appendix Part 2 - Question 1.*

We are given a parity check matrix H and the goal is to return two matrices: $\hat{H}$ and G. $\hat{H}$ is the echelon form of H and this could be done by Gaussian elimination. Furthermore, $\hat{H}$ could be express in the form $[I, P]$ where I is the identity matrix. The systematic encoding matrix G could be created by stacking P vertically above another identity matrix. To examine whether G is created correctly, we can check if $\hat{H}G = 0$. Note that all operations are calculated by $\mathbb{F}_2$. Results are shown below.

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

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

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

3.2 Question 2: Factor graph

The factor graph of H is given in Figure 4. Also, we can get,

$$x_1 + x_2 + x_3 + x_4 = 0$$

$$x_3 + x_4 + x_6 = 0$$

$$x_1 + x_4 + x_5 = 0$$

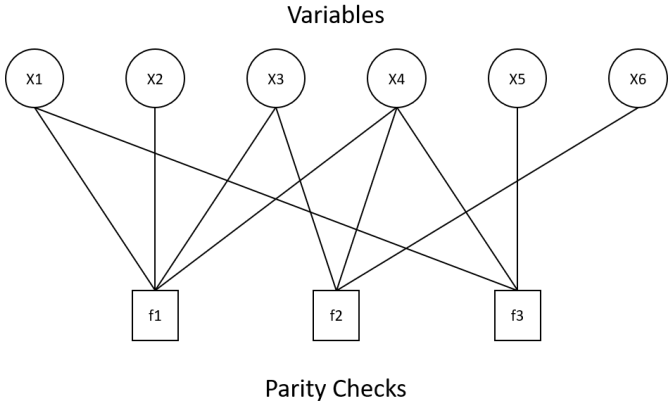

Figure 4: Factor graph

3.3 Question 3: LDPC-decoder

Note: *The code for this question is in Appendix Part 2 - Question 3 and 4.*

The LDPC decoder we built is based on the equations given by the tutorial [1]. In the very first iteration of belief propagation (iteration 0), we pass on the log-likelihood of the received word y. Since we are using Binary Symmetric Channel (BSC) with a noise ratio of p, the initial message of node values equal zero will simply be $\ln(1-p) - \ln p$, and the node values equal one will be the same value but with opposite sign.
After the initial setup, any follow-up iteration contains two steps. First, a check node sends a likelihood to message node according to eq 16. Second, a message node sends a sum of log-likelihood from other check nodes (not include the current check node) to the check node according to eq 17.

$$\ln L\left(x_1 \oplus \cdots \oplus x_l | y_1 \cdots y_l|\right) = \ln \frac{1 + (\prod_{i=1}^{l} \tanh(l_i/2))}{1 - (\prod_{i=1}^{l} \tanh(l_i/2))} \tag{16}$$

$$\ln L\left(x | y_1 \cdots y_d|\right) = \sum_{i=1}^{d} \ln L\left(x | y_1\right) \tag{17}$$

In previous section we mentioned that $\hat{H}G = 0$ can examine whether the message is correctly decoded. Therefore, we simply check our decoded message with the parity check matrix and see if all the values are 0 (Following $\mathbb{F}_2$ operation). If decoded correctly, return

the values and stop the iterations, otherwise continue the propagation until reach the maximum number. It took us 9 iterations to fully decode the message (not counting the initial iteration 0).

3.4 Question 4: Recover the English message

Note: *The code for this question is in Appendix Part 2 - Question 3 and 4.*

To recover the message, we slice the first 248 digits into 31 sets of 8 digits. The 8 digits can be transformed to an integer number, which can be further transform into an ASCII symbol. With 31 sets of 8 digits, we can get 31 ASCII symbols. The English message we got was:
Happy Holidays! Dmitry&David :)

4 Part 3: Mean Field Approximation and Gibbs Sampling

4.1 Question 1: Exact Inference

Note: *The code for this question is in Appendix Part 3 - Question 1.*

Considering a binary variable Markov Random Field based on the Ising model, we perform exact inference in a naive manner by clustering columns within the field into a cluster variable X_t, as shown in Figure 5.

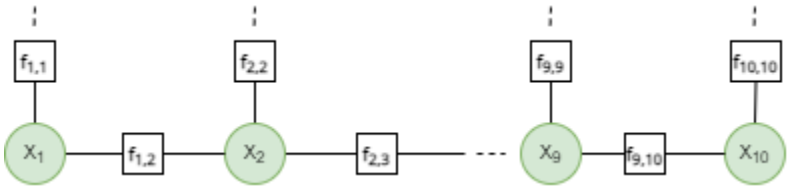

Figure 5: Clustered Ising Model.

Before clustering, each variable in the lattice is binary and therefore has 2 possible states denoted $\{0, 1\}$. By clustering over columns the new variable X_t will have 2^{10} states, corresponding to the different configurations across the column values.

Once clustered, message passing that will occur going down the the column will travel through column factors, denoted by matching indices in our Figure (e.g $f_{1,1}$ for column 1) meaning for each combination of possible column variables, we would have a product over the factors between those column variables, which follow $\phi(x_i, x_j) = e^{\beta \mathbb{I}[x_i = x_j]}$ giving us 2^{10} different possible column factor values seen at $f_{1,1}$, for example.

Factors denoted with different indices (e.g $f_{1,2}$) denote factors across the column travelling to the right. For two columns, we could have row-wise combinations of up to $2^{10} \times 2^{10}$ combinations, therefore for each possible pairing we could have a different product over the row-wise factors as like we did with the column factors.

Computing $p(x_{1,10}, x_{10,10})$

We require a marginal probability over variable X_{10} as this is what contains our target column variables. During message passing, we only obtain this marginal through the product over all incoming messages into X_{10}, starting from X_1. Here we have a combination of variable-to-factor messaging and factor-to-variable messaging. Beginning at $f_{1,1} \to X_1$ we would typically multiply the factor by a product over all previous incoming messages and marginalise, however, it is on its own and therefore we have:

$$\mu_{f_{1,1} \to X_1}(X_1) = f_{1,1}$$

We then propagate from $X_1 \to f_{1,2}$, which is a variable-to-factor propagation and is a product over all incoming messages into X_1, which is just:

$$\mu_{X_1 \to f_{1,2}}(X_1) = \mu_{f_{1,1} \to X_1}(X_1) = f_{1,1}$$

We encounter a different case for the propagation from $f_{1,2} \to X_2$, which as we know we multiply the messages accumulated at $f_{1,2}$ by the factor itself, marginalise over those incoming messages, then propagate:

$$\mu_{f_{1,2} \to X_2}(X_2) = \sum_{X_1} f_{1,2} \times \mu_{X_1 \to f_{1,2}}(X_1)$$

which we then iterate this procedure over all factors and variables until arriving at X_{10} in which the marginal is as we described, a product over incoming messages:

$$p(X_{10}) = \mu_{f_{9,10} \to X_{10}}(X_{10})\, \mu_{f_{10,10} \to X_{10}}(X_{10}) = \sum_{X_9} f_{9,10} \times \mu_{X_8 \to f_{8,9}}(X_8) \times f_{10,10}$$

in which we obtain $p(x_{1,10}, x_{10,10}) = Z^{-1} \prod_{10,1} \phi_{10}(x_{10}, x_1)$.

We compute this probability of being in the same state over the messages using the code shown in (Part 3 Question 1 in the Appendix) and obtain the results in Table 6.

Table 6: Probability of $x_{1,10} = x_{10,10}$ in the Ising model using Exact Inference.

β	$p(x_{1,10} = x_{10,10})$
0.01	0.500
1	0.562
4	0.999

4.2 Question 2: Mean Field Approximation and coordinate ascent

Note: *The code for this question is in Appendix Part 3 - Question 2.*

In mean-field approximation we wish to find the best distribution q that maximises an Evidence Lower Bound (ELBO) by factorising our joint distribution $p(X)$ into a product of factored $q(X)$'s which minimise the KL divergence between the p and q (Equation 18), that we can then optimise via coordinate ascent w.r.t to different variables iteratively. We say we have converged when $ELBO_{t+1} - ELBO_t < \epsilon$, some stopping criterion.

$$Q(x) = \prod_{n=1}^{N} q(x_n) : \mathbb{KL}(\prod_{n=1}^{N} q(x_n) \,||\, p) \rightarrow min_{q_1,q_2,...,q_N} \tag{18}$$

From the notes we define the free-energy (ELBO) to be

$$ELBO(q) = E\left[log\, p(x)\right] - E\left[log\, q(x)\right]$$

We can sub in our distributions to get

$$E_q\left[log\, Z^{-1} \prod_{i>j} e^{\beta \mathbb{I}[x_i = x_j]}\right] - E_{q_i}\left[log \prod_n q(x_i)\right]$$

The first term can be decomposed using log laws into:

$$E_q\left[log\, Z^{-1} \prod_{i>j} e^{\beta \mathbb{I}[x_i = x_j]}\right] = E_q\left[\sum_{i>j} \beta \mathbb{I}[x_i = x_j] - log\, Z\right] = \sum_{i>j} E_q\left[\beta \mathbb{I}[x_i = x_j]\right] - log\, Z$$

The second term also decomposes into:

$$E_{q_i}\left[log \prod_n q(x_i)\right] = \sum_n E_{q_i}\left[log\, q(x_i)\right]$$

For an overall free-energy, we can drop the normalising term as it is not respective of our q distribution giving:

$$= \sum_{i>j} E_q\left[\beta \mathbb{I}[x_i = x_j]\right] - \sum_n E_{q_i}\left[log\, q(x_i)\right]$$

Considering the bound as a function of $q(x_k)$, where k could be any node in the Ising model lattice, such that we can break down the free energy, we first see that the right hand side becomes:

$$\sum_i E_{q_i}\left[log\, q(x_i)\right] = E_k\left[E_{-k}\left[\sum_i log\, q(x_i)\right]\right] = E_k\left[log\, q(x_k) + E_{-k}\left[\sum_{i \neq k} log\, q(x_i)\right]\right]$$

which is equal to an expectation over $q(x_k)+1-q(x_k)$, as the expectation of the distribution over the remaining is simply $i \neq k$. Doing the same to the left side, the bound we want to maximise is then:

$$argmax_{q_k(x_k)}\left(E_{q_k}\left[E_{-k}\left[\beta \mathbb{I}[x_k = x_{-k \cap nei(k)}]\right]\right] - E_k\left[log\, q_k(x_k)\right]\right) \tag{19}$$

Reducing the left hand side in the same manner (take note that $-k \cap nei(k)$ is a limitation placed due to the Markov random-field dependency on neighbouring particles when calculating $p(X)$, hence translates to 'set of not k intersection with set of neighbours to k'):

$$= argmax_{q_k(x_k)}\left(\sum_{i \in nei(k)} \beta\, E_{q_k}\left[\mathbb{I}[x_k = x_i]\right] - E_k\left[log\, q(x_k)\right]\right) \tag{20}$$

Calculating the expected over k on our indicator function (1 if the argument holds, else 0) and using independence relationship we get:

$$E_{q_k}\left[\mathbb{I}[x_k = x_i]\right] = q(x_k = x_i) = q(x_k = 1)q(x_i = 1) + q(x_k = 0)q(x_i = 0)$$

$$= q(x_k)q(x_i) + (1 - q(x_k))(1 - q(x_i))$$

Where we shorten notation and $q(x_k) = q(x_k = 1)$ and we will use this decomposition in order to compute $\mathbb{I}[x_{1,10} = x_{10,10}]$. Plugging this into Equation 20, differentiating w.r.t q_k and setting to 0 to find the maximum and recalling Eq. 19, we get:

$$= \frac{d}{dq_k}\left(\sum_{i \in nei(k)} \beta\ \left[q(x_k)q(x_i) + (1 - q(x_k))(1 - q(x_i))\right] - E_k\left[log\, q(x_k)\right]\right) = 0$$

$$= \beta \sum_{i \in nei(x_k)} (2q(x_i) - 1) - \frac{d}{dq_k}\ \left(E_k\left[log\, q_k(x_k) + 1 - E_k\left[log(1 - q(x_k))\right]\right]\right) = 0$$

Using expectation equation as the sum over its argument multiplied by the probability of its argument, the RHS reduces to:

$$= \beta \sum_{i \in nei(x_k)} (2q(x_i) - 1) - log\, q(x_k) + 1 + log(1 - q(x_k)) - 1 = 0$$

Solving for $q(x_k)$:

$$\rightarrow q(x_k)^{t+1} = \frac{1}{1 + e^{-\beta \sum_{i \in nei(x_k)} (2q(x_i) - 1)}} \tag{21}$$

Which is the mean function of a binary logistic regression model, hence the approximation is that of a binary model so we can assume this will be close to the true distribution but we compute the values and obtain:

Table 7: Probability of $x_{1,10} = x_{10,10}$ in the Ising model using CAVI.

β	$p(x_{1,10} = x_{10,10})$
0.01	0.500
1	0.739
4	0.999

We note that the extremities of the 'coolness' parameter for $\beta = 0.01, 4$ correspond to similar values as the exact inference method meaning that even our approximation distribution can compute these correctly as at low temperature the nodes can be in either state being they are far from each other but at a high temperature there is large high temperature expansion and therefore the two nodes have a high probability of being in the same state.

4.3 Question 3

Note: *The code for this question is in Appendix Part 3 - Question 3.*

Gibbs sampling is a way to simplify complicated distribution problems. For our 10×10 lattices, it is difficult to get the whole distribution directly, but the conditionals are easy to compute since every node has neighbour nodes to perform the calculations. Recall that we have $P(x) = Z^{-1} \prod_{i>j} \phi(x_i, x_j)$ with $\phi(x_i, x_j) = e^{\beta \mathbb{I}[x_i = x_j]}$. The procedure of Gibbs sampling will then be,
1. Randomly select a coordinate from the lattice.
2. Calculate the conditional by getting the neighbour nodes (up, down, right, left, take extra care for corner cases which have less than 4 neighbours) of that coordinate and compute $\phi(x_i, x_j) = e^{\beta \mathbb{I}[x_i = x_j]}$.
3. Generate a random number and compare with the conditional calculated above. Flip the node if the randomly generated number is smaller than the conditional.
4. Repeat above steps until the distribution is the sample we are looking for (this repeating step is called burn-in).

The concept is that we start out by a randomly sampled lattice. Then we update one coordinate according to its neighbours (this coordinate could be chosen at random or in turn). The more coordinates we update, the closer the distribution will match our requirements. The process that we turn a random distribution into a desired distribution is called burn-in. The reason that this method works is due to the fact that Gibbs sampling satisfies detailed balance. Assume the probability of picking a node $x_{i,j}$ is $\pi_{i,j}$, the transition probability will be,

$$T(x \to x') = \pi_{i,j} p(x'_{i,j} | x_{\setminus i,j})$$

Then we can get,

$$T(x \to x') p(x) = \pi_{i,j} p(x'_{i,j} | x_{\setminus i,j}) p(x_{i,j} | x_{\setminus i,j}) p(x_{\setminus i,j})$$

Similarly,

$$T(x' \to x) p(x) = \pi_{i,j} p(x_{i,j} | x'_{\setminus i,j}) p(x'_{i,j} | x'_{\setminus i,j}) p(x'_{\setminus i,j})$$

Since $x'_{\setminus i,j} = x_{\setminus i,j}$ the above two transition probability are the same. Therefore, continually updating coordinates will eventually lead to a state that satisfies the proposed distribution.

Gibbs sampling is a method that continuously updates the lattice. Thus, to get the probability table for $P(x_{1,10}, x_{10,10})$, we could directly get the results by re-running the burnt-in Gibbs sampling model over and over for reasonable amount of times and record the results. This could be quite slow and computational heavy, however, that is the nature of Gibbs sampling.

Table 8: Probability of $x_{1,10} = x_{10,10}$ in the Ising model using Gibbs sampling.

β	$p(x_{1,10} = x_{10,10})$
0.01	0.563
1	0.753
4	0.991

References

[1] Amin Shokrollahi. Ldpc codes: An introduction. 2002.

5 Appendix

5.1 Part 1 Inference and Learning

5.1.1 Question 1: Explosions

```
import numpy as np
import matplotlib.pyplot as plt

def val_fcn(x, y, xs, ys):
    ''' Distance function for 1 explosion case. '''
    dist = np.square(x-xs) + np.square(y-ys)

    return 1/(dist + 0.1)

def val_fcn_2d(x, y, x2, y2, xs, ys, sd):
    ''' Distance function for 2 explosion case. '''
    dist = np.square(x-xs) + np.square(y-ys)
    dist2 = np.square(x2-xs) + np.square(y2-ys)

    return 1/(dist + 0.1) + 1/(dist2 + 0.1)

# Set up our coordinate system ----------------------
S=2000 # number of points on the spiral
rate=25 # angular rate of spiral
sd=0.2 # standard deviation of the sensor Gaussian noise
x=np.zeros(S)
y=np.zeros(S)

for s in range(S):
    theta=rate*2*np.pi*s/S
    r=s/S
    x[s]=r*np.cos(theta)
    y[s]=r*np.sin(theta)
N=30 # number of stations

x_sensor=np.zeros(N)
y_sensor=np.zeros(N)
v=np.zeros((S, N)) # v for 1 explosion
v2=np.zeros((S, S, N)) # v for 2 explosions

# Compute sensor locations in spiral system
for sensor in range(N):
    theta_sensor=2*np.pi*sensor/N
    x_sensor[sensor]=np.cos(theta_sensor)
    y_sensor[sensor]=np.sin(theta_sensor)

# Compute a matrix of expected sensor values for each pair of
   coordinate locations
```

```
for sensor in range(N):
    for i in range(S):
        v[:, sensor] = val_fcn(x, y, x_sensor[sensor], y_sensor[sensor
    ]) # 1D
        v2[i, :,sensor]=val_fcn_2d(x[i],y[i], x,y ,x_sensor[sensor],
    y_sensor[sensor], sd) # 2D

# Load our observed data
data = np.loadtxt('EarthquakeExerciseData.txt')

# Compute log likelihood----------------------------
logp = np.zeros(S) # 1D
logp_2d = np.zeros((S, S)) # Log posterior over each combination of
    locations (2D)

for s in range(S):
    logp[s] = np.sum(-0.5*(data - v[s, :])**2/(sd**2)) # log version of
     gaussian prob means no exp
    logp_2d[s, :] = np.sum(-0.5*(data - v2[s, :, :])**2/(sd**2), axis
    =1)

p = np.exp(logp-np.max(logp)) # Convert normalised log likelihood to
    normal likelihood (Scale to 0-1 by minusing max)
p /= np.sum(p) # Normalise for all probabilities
p_2d = np.exp(logp_2d-np.max(logp_2d)) # 2 Explosion case
p_2d /= np.sum(p_2d)

# Plt output posteriors-----------------------------
maxind2, maxp2 = np.unravel_index(p_2d.argmax(), p_2d.shape), np.max(
    p_2d) # get argmax

# Log difference in hypothesis
np.max(logp_2d) - np.max(logp)
```

Listing 1: Explosion code

5.1.2 Question 2: Weather-Stations

```
import numpy as np

data = np.genfromtxt('meteo1.csv', delimiter=' ', dtype=np.int32)

T, L = data.shape
H, V = 3, 3
T_L = np.zeros((T, V, V)) # transition matrix
T_I = np.zeros((T, V)) # Initial matrix

# Compute transitions and initial states of our data
for i in range(T):
    for j in range(L):
        if j == 0:
            T_I[i, data[i, j]] += 1 # Initial state
        else:
            T_L[i, data[i, j-1], data[i, j]] += 1 # Transition from l
    -> l+1
```

```
17
18 # Initialise parameters
19 pi = np.abs(np.random.normal(size=(H,1)))
20 pi /= np.sum(pi) # Normalise
21 P = np.abs(np.random.normal(size=(H, V, V))) # Transition prob
22 for i in range(H): P[i,...] /= np.sum(P[i,...], axis=1).reshape(3,1) #
       Normalise
23 Q = np.abs(np.random.normal(size=(H, V))) # Initial prob
24 Q /= np.sum(Q, axis=1).reshape(3,1)
25
26 # EM
27 likelihood = []
28 for _ in range(10):
29     # Expectation----------
30     loglik=0
31     lph_old = np.zeros((T, H)) # p(h=h|v,theta_old)
32     for t in range(T):
33         lph_old[t, :] = np.log(pi).squeeze() + np.log(Q[:, data[t, 0]])
       .squeeze()
34         for l in range(1, L):
35             lph_old[t, :] = lph_old[t, :] + np.squeeze(np.log(P[:, data
       [t, l-1], data[t, l]]))
36     ph_old = np.exp(lph_old-np.max(lph_old, axis=1).reshape(T, 1)) #
       Condexp from BRML toolbox
37     ph_old /= np.sum(lph_old, axis=1).reshape(T,1) # Condp from BRML
       toolbox
38     loglik += np.log(np.sum(np.exp(lph_old)))
39
40     # Updates using eqs-------------
41     # Pi_new
42     pi_new = np.zeros_like(pi)
43     for h in range(H):
44         pi_new[h] = np.sum(ph_old[:, h])
45     pi_new /= np.sum(pi_new)
46
47     # P_new
48     P_new = np.zeros_like(P)
49     for h in range(H):
50         for t in range(T):
51             P_new[h, :, :] += ph_old[t, h]*T_L[t, :, :]
52         P_new[h, ...] /= np.sum(P_new[h,...], axis=1).reshape(3,1)
53
54     # Q_new
55     Q_new = np.zeros_like(Q)
56     for h in range(H):
57         for i in range(V):
58             Q_new[h, i] += np.sum(np.multiply(ph_old[:, h],T_I[:, i]))
59     Q_new /= np.sum(Q_new, axis=1).reshape(3,1)
60
61     # Final update theta
62     pi, P, Q = np.copy(pi_new), np.copy(P_new), np.copy(Q_new)
63     # Store likelihood for these params
64     likelihood.append(loglik)
65
66 final_likelihood = likelihood[-1]
```

```
# Posterior under new params
post = np.zeros((10, H))
for i in range(10):
    for h in range(H):
        post[i, h] = pi[h] * ph_old[i, h]
post /= np.sum(post, axis=1).reshape(10, 1)
```

Listing 2: EM algorithm code

5.2 Part 2 LDPC codes

5.2.1 Question 1

```
import numpy as np

def systematic_encoding(H):
    """
    Decompose H to the echelon form so that H = [I P], we assume all H must be full rank
    Later create G based on H_hat and identity matrix
    """
    # first we get the number of rows of H
    H_row, H_col = H.shape
    H_hat = np.copy(H)

    for i in range(H_row):
        for j in range(H_row):
            # make sure diagonal H_hat[i, i] = 1
            if H_hat[i, i] == 0:
                for k in range(H_row):
                    if H_hat[k, i] == 1 and k > i:
                        # Make H_hat[i, i] = 1 by swapping
                        tempH = np.copy(H_hat[k])
                        H_hat[k] = H_hat[i]
                        H_hat[i] = tempH
                        break

            # eliminate all other 1 of the column, but don't compare the same row
            if i != j:
                if H_hat[j, i] == H_hat[i, i]:
                    H_hat[j, :] = H_hat[j] + H_hat[i]

                    # make 1+1=2 into 1+1=0
                    H_hat = np.where(H_hat == 2, 0, H_hat)

    P = H_hat[:, H_row:]
    I = np.eye(H_col - H_row)
    G = np.vstack((P, I))

    return H_hat, G

# create the given H and get H_hat and 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 = systematic_encoding(H)
```

```
41
42 print(f"""Initial H, \n {H} \n
43 H_hat, \n {H_hat} \n
44 G, \n {G} \n""")
```

Listing 3: parity check matrix and systematic encoding matrix

5.2.2 Question 3 and 4

```
1  import numpy as np
2
3  def initialization(y, p):
4      """
5      Refer to "LDPC Codes: An Introduction" p.8
6      Create a log-likelihood of the received word
7      """
8      P = np.copy(y)
9
10     # log likelihood
11     p0 = np.log(1-p) - np.log(p)
12     p1 = -p0
13     log_like = np.where(P == 0, p0, p1)
14
15     return log_like
16
17
18 def BeliefProp_calculation(H, message, log_like):
19     """
20     refer to "LDPC Codes: An Introduction" p.8
21     Pass message from message node to check node
22     """
23     H_row, H_col = H.shape
24     check2msg = np.zeros(H.shape)
25
26     for row in range(H_row):
27         index = np.where(H[row] == 1)[0]
28         for idx in index:
29             tanh_part = np.prod(np.tanh(message[row,:][index]/2)) / np.
    tanh(message[row,:][idx]/2)
30             check2msg[row, idx] = np.log((1+tanh_part)/(1-tanh_part))
31
32     msg2check = log_like + np.sum(check2msg, axis=0)
33
34     result = np.where(msg2check > 0, 0, 1)
35
36     return check2msg, result
37
38
39 def BeliefProp_update(H, message, log_like, check2msg):
40     """
41     Update the message
42     """
43     new_message = np.copy(message)
44
45     for col in range(new_message.shape[1]):
```

```
        index = np.where(H[:, col] == 1)[0]
        for idx in index:
            result = np.sum(check2msg[:, col][index]) - check2msg[:, col][idx]
            new_message[idx,col] = log_like[col] + result

    return new_message


def LDPC(H, y, p, max_iter):
    """
    LDPC-decoder based on Belief Propagation for Binary Symmetric Channel
    """
    iters = 0
    output = -1

    # This is considered round 0 (iter 0)
    log_like = initialization(y, p)
    message = np.copy(H)

    while iters < max_iter:
        iters += 1
        # Loopy Belief Propagation
        check2msg, decode = BeliefProp_calculation(H, message, log_like)
        message = BeliefProp_update(H, message, log_like, check2msg)

        # Examining the decoded message
        check_decode = np.where(H@decode % 2 == 0, 0, H@decode)
        if np.sum(check_decode) == 0:
            output = 0
            return decode, output, iters

    return decode, output, iters

    def recover_ASCII(ascii):
    """
    Recover first 248 bits of the message using ASCII code
    """
    to_str = ''.join(str(s) for s in ascii[0:248])
    store_str = []
    for i in range(0, 248, 8):
        symbol = chr(int(to_str[i:i+8], 2))
        store_str.append(symbol)
    ori_message = "".join(str(c) for c in store_str)

    return ori_message

H1 = np.loadtxt("H1.txt", delimiter = " ")
y1 = np.loadtxt("y1.txt", delimiter = " ")

decode, output, iters = LDPC(H1, y1, 0.1, 20)
print(output, iters)

```

```
recover = recover_ASCII(decode)
print(recover)
```

Listing 4: LDPC-decoder and recovering original English message

5.3 Part 3 Mean Field Approximation and Gibbs Sampling

5.3.1 Question 1

```
import numpy as np
import itertools

N = 10 # N
vals = [0,1] # Binary values
beta_vals = [0.01, 1, 4]
X = np.array(list(map(list, itertools.product(vals, repeat=N)))) # Get combination matrix

for beta in beta_vals: # Can possibly use log since values are very large and require normalising frequently
    # f_1,1 -> X1
    # Get identical neighbours
    f_11 = np.zeros(X.shape[0])
    for i in range(X.shape[0]): # For each neighbouring component in 1024
        for j in range(X.shape[1]-1):
            if X[i, j] == X[i, j+1]:
                f_11[i] += 1
    f_11 = np.exp(beta*f_11) # Convert to potential form

    marginal_copy = f_11.copy() # Base case of previous variable marginal
    for m in range(10): # Over all variables
        f_m = np.zeros((X.shape[0], X.shape[0])) # e.g f_11 -> X1
        for i in range(X.shape[0]): # For each combination of the 1024 x1024 states count
            for j in range(X.shape[0]):
                matches = np.sum(X[i, :] == X[j, :])
                f_m[i, j] = marginal_copy[j] * np.exp(beta*matches) # Match the j to f_11 for consistency
        f_m = np.sum(f_m, axis=1) # Marginalise over X variable
        marginal_copy = np.multiply(f_11, f_m) # Update this variables marginal via multiply prev msg with its own potential
        marginal_copy /= np.sum(marginal_copy)

    # Compute p(x_1 10=x_10 10)
    # Over all passed messages use OR rule to sum cases of matches of 1,10 and 10,10 for all 2^10 states
    x_10 = np.sum(marginal_copy[np.where(X[:, 0]==X[:, 9])])
    print(x_10)
```

Listing 5: Exact Inference

5.3.2 Question 2

```
N = 10 # N
vals = [0,1] # Binary values
beta_vals = [0.01, 1, 4]
directions = [[0,1], [0,-1], [1,0], [-1,0]] # Neighbour check index offsets

for beta in beta_vals:
    grid = np.random.rand(N,N)
    for _ in range(20): # Epoch cycles
        for i in range(grid.shape[0]): # Iterate over lattice
            for j in range(grid.shape[1]):
                neighbour_vals = []
                for d in directions:
                    if 0 <= i + d[0] < grid.shape[0] and 0 <= j + d[1] < grid.shape[1]: # Check validity of neighbours index
                        neighbour_vals.append(grid[i+d[0], j+d[1]]) # Append neighbour values
                grid[i, j] = 1/(1 + np.exp(-beta*np.sum(2*np.array(neighbour_vals) - 1))) # Compute new iteration
    # q(x_1,10)*q(x_10,10) = I[xk=xi] from derivations
    p_10 = grid[0, 9]*grid[9,9] + (1-grid[0, 9])*(1-grid[9, 9])
    print(f'P(x)={p_10}')
```

Listing 6: Mean-Field CAVI Approximation code

5.3.3 Question 3

```
import numpy as np

def conditionals(lattice, i, j, beta):
    """decide whether to flip the node by considering the 4 closest neighbours"""
    neighbour = [[i+1, j], [i, j+1], [i-1, j], [i, j-1]]
    same_neighbour = 0

    for k in range(4):
        # check if neighboors are in the lattice
        if 10>neighbour[k][0]>0 and 10>neighbour[k][1]>0:
            # check if neighboors are the same with the node
            if lattice[neighbour[k][0], neighbour[k][1]]==lattice[i, j]:
                same_neighbour += 1

    return np.exp(-beta * same_neighbour)

def gibbs_sampling(lattice, beta, runs):
    """perform gibbs sampling on a lattice for a given beta and runs"""
    copy_lattice = np.copy(lattice)
    for _ in range(runs):
        # randomly pick a node
        i, j = np.random.randint(10, size=2)
        threshold = conditionals(copy_lattice, i, j, beta)

        # decide whether flip the node or not
        if np.random.rand() < threshold:
```

```
            copy_lattice[i, j] = copy_lattice[i, j] * -1

    return copy_lattice

beta_values = [0.01, 1, 4]

for beta in beta_values:
    # randomly initialize the lattice
    initialize = np.random.choice([-1, 1], (10, 10))

    # burn-in the lattice to get a Ising model
    burn_in = gibbs_sampling(initialize, beta, runs=2000)
    count_same = 0
    samples = 5000

    # rerun the Ising model several times to get the resulting
    probability distribution
    for _ in range(samples):
        Ising = gibbs_sampling(burn_in, beta, runs=50)
        if Ising[0, 9] == Ising[9, 9]:
            count_same += 1

    print(f"beta = {beta}, probability of being the same: {count_same/
    samples}")
```

Listing 7: Gibbs sampling