Graphical Models (COMP0080)
Coursework

Team: SciLi

Student Numbers:

21195123 21093512

17019028 21141386

18091452

January 2022

Contents

Part 1

1.1 Two Earthquakes

This section corresponds to question *1.22* in the BRML book. The code written to answer this set of questions can be found in the appendix, 4.1.1.

1.1.1 (a) Obtaining posterior distribution

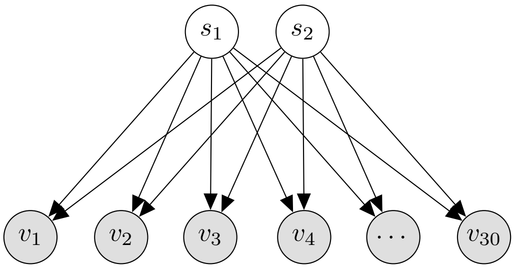

Figure 1.1.1: Graphical model representing the belief network, with two explosions s_2 and s_2

In order to compute the posterior we assume that the prior is uniform. Hence the posterior is proportional to the likelihood. The likelihood in the *2-explosion* case is computed using a normal distribution with mean, $\mu = \frac{1}{d_i(1)^2+0.1}^2 + \frac{1}{d_i(2)^2+0.1}^2 + \sigma\epsilon_i$ and standard deviation $\sigma = 0.2$. However, the $\sigma\epsilon_i$ term is not added to the likelihood term as the noise can already been seen in the observed values in the given data. The $d_i(1)$ and $d_i(2)$ terms correspond to the distance of the i^{th} sensor from explosion 1 and 2. Its equation is shown below.

$$p(v_i|s_1, s_2) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{1}{2\sigma^2}\left(v_i - (\frac{1}{d_i(1)^2+0.1}^2 + \frac{1}{d_i(2)^2+0.1}^2)\right)} \tag{1.1}$$

The aim of this question is to compute the marginal posterior distribution, $p(s_1|\mathbf{v})$. In order to do so we firstly need to compute the total posterior (over s_1 s_2) distribution. Then we'll marginalise out s_2. By applying Bayes theorem and using the independence assumptions made on the belief graph (Figure 1.1.1), we get :

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

When marginalizing out s_2, we get the following marginal posterior.

$$p(s_1|\mathbf{v}) = \frac{\sum_{s2} p(s_1)p(s_2)\prod_{i=1}^{n} p(v_i|s_1, s_2)}{\sum_{s_1,s_2} p(s_1)p(s_2)\prod_{i=1}^{n} p(v_i|s_1, s_2)}$$

Using the assumption made on the uniformity of the prior, $p(s_1) = p(s_2)$. Hence, we get the final posterior equation.

$$p(s_1|\mathbf{v}) = \frac{\sum_{s2}\prod_{i=1}^{n} p(v_i|s_1, s_2)}{\sum_{s_1,s_2}\prod_{i=1}^{n} p(v_i|s_1, s_2)}$$

The resultant posterior image can be seen in figure 1.1.2

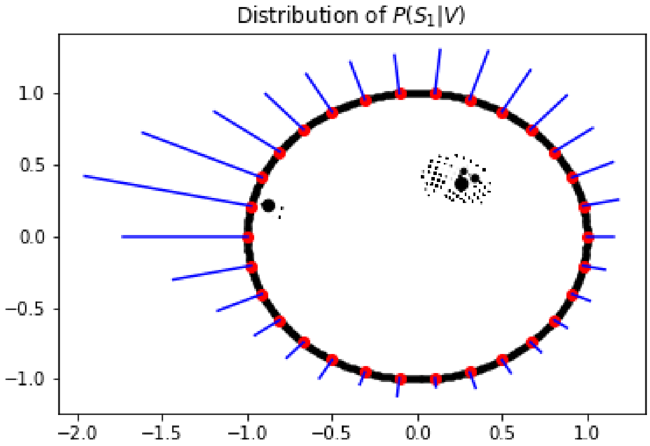

Figure 1.1.2: Posterior distribution given a set of observed values

The posterior distribution for the location of two explosions (darker corresponds to a higher probability). A scale parameter is added to increase the size of points with higher posterior probability (see larger black dots). The observed (noisy) measurements at each sensor are represented by the blue lines (lengths are not to scale).

1.1.2 (b) Evaluating 2 hypotheses

Let $\mathcal{H}_1$ be the hypothesis that we have 1 explosion and $\mathcal{H}_2$ be the hypothesis that we have 2 explosions. The posterior distribution of the belief network is $p(\mathbf{v}|s_k, \mathcal{H}_k)$ where k is the number of explosions. Similarly to tasks above, we carry on the assumption that the prior is uniform. In order to obtain values for $p(\mathbf{v}|\mathcal{H}_k)$, we have to marginalise out s_k. This entails summing over all possible explosion locations and taking the product of the likelihood over all observed values. Hence the likelihood for the *1-explosion* case is,

$$p(\mathbf{v}|\mathcal{H}_1) = \sum_s \prod_v p(\mathbf{v}|s),$$

and the likelihood for the *2-explosion* case is,

$$p(\mathbf{v}|\mathcal{H}_2) = \sum_{s_1} \sum_{s_2} \prod_v p(\mathbf{v}|s_1, s_2).$$

The desired value is shown below.

$$\log p(\mathbf{v}|\mathcal{H}_2) - \log p(\mathbf{v}|\mathcal{H}_2) = 743.2807173246063 \tag{1.2}$$

1.1.3 (c) Hypothesis Evaluation Metric

Once again, assuming that we have no prior preference on both hypotheses, we can say that the quantity shown above relates to the probability that there are two explosions compared to only 1.

Decomposing the expression given in the assignment, we get the following.

$$\log p(\mathbf{v}|\mathcal{H}_2) - \log p(\mathbf{v}|\mathcal{H}_2) = \log(\frac{p(\mathbf{v}|\mathcal{H}_2)}{p(\mathbf{v}|\mathcal{H}_1)})$$

This ratio of logarithmic likelihoods can thus also be seen as a hypothesis evaluation metric. Indeed, the larger this value is, the more likely $\mathcal{H}_2$ is over $\mathcal{H}_1$.

Since the logarithmic function is monotonically increasing for all positive reals and the likelihood values can only have positive real values, we can say that this relationship holds when applying the log: **the larger this value is, the more likely $\mathcal{H}_2$ is over $\mathcal{H}_1$**. In the given scenario, the ratio value is of 743.28, indicating strong evidence for $\mathcal{H}_2$. This would mean that given the observed data the hypothesis that there were 2 explosions is much more likely.

It relates to the Bayes Factor (i.e. the likelihood ratio of the marginal likelihoods of two competing hypotheses), shown below,

$$\frac{p(\mathbf{v}|\mathcal{H}_2)}{p(\mathbf{v}|\mathcal{H}_1)}.$$

Bayes Factor can be interpreted as a measure of the strength of evidence in favor of one theory among two competing theories.

1.1.4 (d) Computational Complexity

If we assumed that there were $\mathcal{K}$ explosions, explain the computational complexity of calculating $\log p(v|\mathcal{H}_K)$ would be of $\mathcal{O}(S^K)$, where S is number of possible earthquake locations within the hypothesis space. All observed locations are independent, hence the observed values at each location must be calculated individually.

1.2 Meeting scheduling

In this section we want to model the delays of a group of N friends, and the probability that they will meet on time to get the train in different scenarios.
We let D_i be the delay of the i-th friend, and assume that all D_i are independent and identically distributed.

1.2.1 (a) Simple scenario

In the first scenario, we wish to find the latest meeting time for N friends $T_0(N)$ that still guarantees to catch the train at 21:05 with probability at least 0.9.
Let d^* denote the maximum delay each friend can have while still catching the train on time, with $d^* \in \{0, 5, 10, 15, 20\}$.
We are given the cumulative distribution function of the delays in minutes:

$$\mathrm{F}_D(D_i) = \begin{cases} 0.7 & : D_i \leq 0 \\ 0.8 & : D_i < 5 \\ 0.9 & : D_i < 10 \\ 0.97 & : D_i < 15 \\ 0.99 & : D_i < 20 \\ 1 & : D_i \to +\infty \end{cases}$$

$T_0(N)$ can then be indicated as follows:

$$T_0(N) := 21{:}05 - d^*(N) \tag{1.3}$$

In the situation at hand, the event "N friends arrive on time" is the intersection of N independent events "Friend i arrives on time", since if only one friend is late, we miss the train. Moreover, we require this intersection of events to have probability at least 0.9. Expressing these pieces of information in probability form:

$$\prod_{i=1}^{N} \mathrm{P}(D_i < d^*) \geq 0.9 \tag{1.4}$$

Given that the meeting time is the same for all friends, we want to calculate one d^* for each N friends condition. Using our *iid* assumption it also follows that $\mathrm{P}(D_i < d^*)$ will be the same for every friend, therefore we re-write 1.4 as:

$$\begin{aligned} \mathrm{P}(D < d^*)^N &\geq 0.9 \\ \mathrm{P}(D < d^*) &\geq \sqrt[N]{0.9} \end{aligned} \tag{1.5}$$

The solution for $d^*(N)$ can then conveniently be expressed in terms of the CDF as:

$$\min_{d^*} \mathrm{F}_D(d^*) \geq \sqrt[N]{0.9} \tag{1.6}$$

Expressed in this form, the results of using equations 1.3 and 1.6 to solve the problem can be simply read out the CDF. This is summarised in the following table:

N	$\sqrt[N]{0.9}$	d^*	$T_0(N)$
3	0.9654	15	20:50
5	0.9791	20	20:45
10	0.9895	20	20:45

Table 1.2.1: Latest possible meeting times for a group of N friends.

1.2.2 (b) Being non-punctual

In the second scenario, we refine our model for the delays by introducing the unobserved binary random variable Z_i, representing if friend i is punctual or not. Z_i-s are also independent of all i-s, and have probabilities $\mathrm{P}(Z_i = \text{punctual}) = \frac{2}{3}$ and $\mathrm{P}(Z_i = \text{not punctual}) = \frac{1}{3}$.

In this redefined scenario, the cumulative distribution functions of the delays (in minutes) given Z are:

$$\mathrm{F}_D(D_i|Z_i = \text{punctual}) = \begin{cases} 0.7 & : D_i \leq 0 \\ 0.8 & : D_i < 5 \\ 0.9 & : D_i < 10 \\ 0.97 & : D_i < 15 \\ 0.99 & : D_i < 20 \\ 1 & : D_i \to +\infty \end{cases}$$

$$\mathrm{F}_D(D_i|Z_i = \text{not punctual}) = \begin{cases} 0.5 & : D_i \leq 0 \\ 0.7 & : D_i < 5 \\ 0.8 & : D_i < 10 \\ 0.9 & : D_i < 15 \\ 0.95 & : D_i < 20 \\ 1 & : D_i \to +\infty \end{cases}$$

We are now interested in the probability of missing the train if we use the answers from Table 1.2.1 in this situation. Lets first note that, since we miss the train even if only one friend is not on time:

$$\begin{gathered} \mathrm{P}(\text{miss}) = 1 - \mathrm{P}(\text{all on time}) \\ \mathrm{P}(\text{miss}) = 1 - \prod_{i=1}^{N} \mathrm{P}(D_i < d^*) \end{gathered} \tag{1.7}$$

So, we require that each friend arrives before the maximum delay calculated. This is now dependent on Z_i, i.e. for each friend we have $\mathrm{P}(D_i \leq d^*|Z_i)$. Retrieving $\mathrm{P}(D_i < d^*)$ using the sum and product rule:

$$\begin{gathered} \mathrm{P}(D_i < d^*) = \sum_z \mathrm{P}(D_i < d^*, Z_i) \\ \mathrm{P}(D_i < d^*, Z_i) = \mathrm{P}(D_i < d^*|Z_i) \cdot \mathrm{P}(Z_i) \\ \implies \mathrm{P}(D_i < d^*) = \sum_z \mathrm{P}(D_i < d^*|Z_i) \cdot \mathrm{P}(Z_i) \end{gathered}$$

Putting it together with 1.7, thus solving for N friends:

$$\begin{aligned}\mathrm{P}(\text{miss}) &= 1 - \prod_{i=1}^{N}\sum_{z}\mathrm{P}(D_i < d^*|Z_i)\cdot\mathrm{P}(Z_i)\\ &= 1 - \left(\sum_{z}\mathrm{P}(D_i < d^*|Z_i)\cdot\mathrm{P}(Z_i)\right)^N\end{aligned} \tag{1.8}$$

gives us an expression which we can now directly evaluate by reading out the values of the CDFs and $\mathrm{P}(Z_i)$. For example, for $N = 3$, $T_0(N) = 20:50$, $d^* = 15$:

$$\begin{aligned}\mathrm{P}(\text{miss}) &= 1 - \left(\frac{2}{3}\cdot\mathrm{F}_D(d^*|Z_i = \text{punctual}) + \frac{1}{3}\cdot\mathrm{F}_D(d^*|Z_i = \text{not punctual})\right)^3\\ &= 1 - \left(\frac{2}{3}\cdot 0.97 + \frac{1}{3}\cdot 0.9\right)^3\\ &= 0.1526\end{aligned}$$

And similarly for $N = 5$ and $N = 10$. We can summarise the results in the following table:

N	$T_0(N)$	**P(miss)**
3	20:50	0.1516
5	20:45	0.1113
10	20:45	0.2103

Table 1.2.2: Probability of missing the train using the results from Table 1.2.1 if friends can be non-punctual.

1.2.3 (c) Bonus

For $N = 5$, $T_0 = $ 20:45, $d^* = 20$ we missed the train. We now want to calculate what is the posterior probability that k friends were not punctual.
Let $K \in \{0, 1, 2, 3, 4, 5\}$ be a random variable denoting the number of non-punctual friends. Using Bayes rule, we have:

$$\mathrm{P}(K \mid \text{miss}) = \frac{\mathrm{P}(\text{miss} \mid K)\cdot\mathrm{P}(K)}{\mathrm{P}(\text{miss})} \tag{1.9}$$

Let us derive all the terms one at a time.

The prior: $\mathbf{P}(K)$

Each friend has the same probability of being not punctual, and one friend being not punctual does not influence the others. Therefore, the punctuality of each friend can be modelled as a Bernoulli trial, and the punctuality of k friends follows a binomial distribution. This will have the form:

$$\mathrm{P}(K = k) = \binom{N}{k}p^k(1-p)^{N-k} \tag{1.10}$$

with $p = P(Z_i = \text{not punctual}) = \frac{1}{3}$ and $N = 5$.
Evaluating $\forall\ k \in \{0, 1, 2, 3, 4, 5\}$:

k	0	1	2	3	4	5
$P(K = k)$	0.132	0.329	0.329	0.165	0.041	0.004

Table 1.2.3: Prior probabilities of k friends being not punctual

The likelihood: $\mathbf{P}(\text{miss} \mid K)$

We now need to calculate the likelihood of missing the train given that k friends were not punctual. As done before, we will derive this from the complementary event, i.e. catching the train with everyone on time given that k friends were not punctual (since all the information we already have is relative to this event). We first note that this event is itself the intersection of two events:

- k friends who are still on time even if non-punctual
- $N-k$ friends who are on time and punctual

$P(\text{on time}|K=k)$ will therefore be:

$$P(\text{on time}|K=k) = \prod_{i=1}^{k} P(D_i < d^*|Z_i = \text{not punctual}) \ \cdot \prod_{i=1}^{N-k} P(D_i < d^*|Z_i = \text{punctual})$$

Given that in this case $d^* = 20$, $N = 5$, and using again the iid assumption, $P(\text{miss}|K=k)$ becomes:

$$\begin{aligned} P(\text{miss}|K=k) &= 1 - (P(D_i < 20|Z_i = \text{not punct.}))^k \cdot (P(D_i < 20|Z_i = \text{punct.}))^{N-k} \\ &= 1 - (0.95)^k \cdot (0.99)^{5-k} \end{aligned} \tag{1.11}$$

Evaluating $\forall\ k \in \{0,1,2,3,4,5\}$:

k	0	1	2	3	4	5
$P(\text{miss}\|K=k)$	0.049	0.087	0.124	0.159	0.193	0.226

Table 1.2.4: Likelihoods of missing the train given that k friends were not punctual

The marginal likelihood: P(miss)

The marginal likelihood in this case has the form:

$$\begin{aligned} P(\text{miss}) &= \sum_k P(\text{miss}|K=k) \cdot P(K=k) \\ &= 0.1113 \end{aligned}$$

Which, as we would expect, matches the result obtained in part 1.2.2.

The posterior: P(K | miss)

At this point, we already have calculated everything we need to evaluate the posterior. Using equation 1.9, we obtain:

k	0	1	2	3	4	5
$P(K=k\|\text{miss})$	0.0580	0.2585	0.3675	0.2361	0.0715	0.0084

Table 1.2.5: Posterior probability that k friends out of N were not punctual, given that we missed the train

It is evident how knowing that we missed the train changed the distribution of the number of friends being not punctual, assigning higher probability to higher values of k. Most importantly, it is now much more unlikely that all friends were punctual (i.e. that 0 friends were not punctual). We can better visualise this in the following picture:

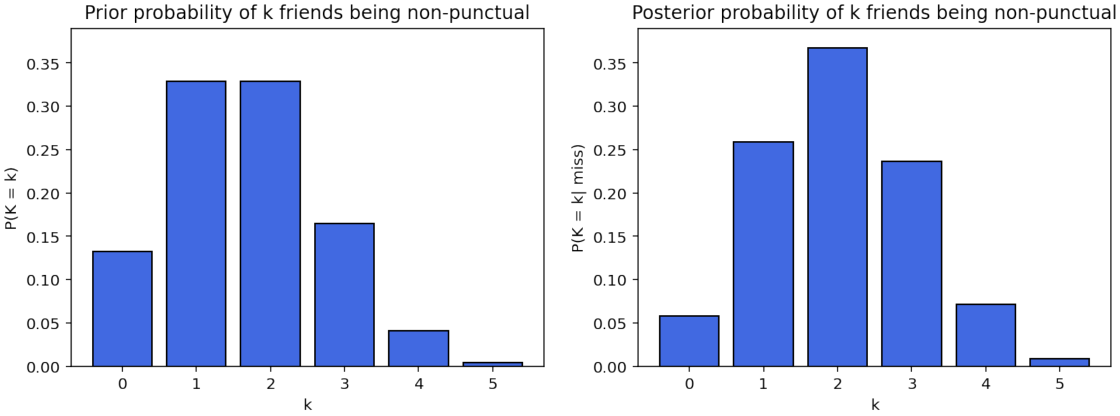

Figure 1.2.1: Prior distribution of the probability that k friends are going to be non-punctual, compared with the posterior distribution of the probability that k friends are non-punctual given that we missed the train.

1.3 Three Weather Stations

1.3.1 (a) The model

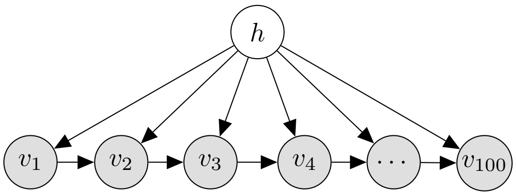

Figure 1.3.1: Markov model for the "three weather stations" problem

In this section we use unsupervised learning to cluster a data sequence to three weather stations $h = \{1, 2, 3\}$ by assuming the data for each weather station follows a first order Markov chain. There are three states within the data sequence defined as $v = \{0, 1, 2\}$. The transition between different states can be described by a probability M_{ij}, which can be summarised into a transition matrix M shown below:

$$\mathbf{M^h} = \begin{pmatrix} P_{0,0} & P_{1,0} & P_{2,0} \\ P_{0,1} & P_{1,1} & P_{2,1} \\ P_{0,2} & P_{1,2} & P_{2,2} \end{pmatrix} \tag{1.12}$$

$$p\left(x_{t+1} = i \mid x_t = j, W = h\right) = M_{ij}^h \tag{1.13}$$

t is a time pointer running from 1 to 100, indicating the current location within the data sequence. In this question, we have three transition matrices M^1, M^2, and M^3 describing different transitioning patterns within data recorded at weather stations 1,2,3. The first condition of Markov chain is described as π.

Expectation maximisation

At its core, the EM algorithm is an iterative method for maximum likelihood estimation of the parameters of statistical models. In maximum likelihood, we learn and revisit the estimation of the unknown parameters by maximizing the marginal likelihood of the observed data, shown below. θ represents the current parameter set.

$$p(v \mid \theta) = \sum_h p(v, h \mid \theta) \tag{1.14}$$

By assuming that each sequence follows a first order Markov chain, we define each model to have:

$$p\left(v_{1:T}\right)=\sum_{h=1}^{H} p(h) p\left(v_{1:T} \mid h\right)=\sum_{h=1}^{H} p(h) \prod_{t=1}^{T} p\left(v_t \mid v_{t-1}, h\right). \tag{1.15}$$

In the expectation step, a log-likelihood function is evaluated using the current transition matrix.

$$\underset{\theta}{\operatorname{argmax}} \log(\mathrm{P}(v \mid \theta)) \tag{1.16}$$

In the maximisation step, we try to maximise the 'energy' term with respect to θ.

$$E=\sum_{n=1}^{N}\left\langle\log p\left(v_{1:T}^{n}, h\right)\right\rangle_{p^{\text{old}}\left(h \mid v_{1:T}^{n}\right)}=\sum_{n=1}^{N}\left\{\langle\log p(h)\rangle_{p^{\text{old}}\left(h \mid v_{1:T}^{n}\right)}+\sum_{t=1}^{T}\left\langle\log p\left(v_t \mid v_{t-1}, h\right)\right\rangle_{p^{\text{old}}\left(h \mid v_{1:T}^{n}\right)}\right\} \tag{1.17}$$

Finally, p will be updated in the following way:

$$p^{\text{new}}\left(h \mid v^n\right) \propto p\left(v^n \mid h, \theta_{v \mid h}^{\text{old}}\right) p\left(h \mid \theta_h^{\text{old}}\right) \tag{1.18}$$

1.3.2 (b) Result

$p(h)$ represents the weighting of the unobserved variables h , it is found as below:

$$p(h)=\left(\begin{array}{c} 0.5038 \\ 0.1920 \\ 0.2997 \end{array}\right) \tag{1.19}$$

The transition matrices for three weather stations are:

$$M^1=\left(\begin{array}{ccc} 0.0705 & 0.1368 & 0.5108 \\ 0.4947 & 0.2259 & 0.4372 \\ 0.4348 & 0.6374 & 0.0520 \end{array}\right) \tag{1.20}$$

$$M^2=\left(\begin{array}{ccc} 0.3887 & 0.2406 & 0.5453 \\ 0.1485 & 0.5155 & 0.0176 \\ 0.4628 & 0.2440 & 0.4371 \end{array}\right) \tag{1.21}$$

$$M^3=\left(\begin{array}{ccc} 0.0597 & 0.1301 & 0.4176 \\ 0.1395 & 0.3238 & 0.4278 \\ 0.8008 & 0.5461 & 0.1546 \end{array}\right) \tag{1.22}$$

The initial stage of the first order Markov chain is found to have distribution as below:

$$\pi_1=\left[\begin{array}{c} 0.1921 \\ 0.4193 \\ 0.3886 \end{array}\right] \tag{1.23}$$

$$\pi_1=\left[\begin{array}{c} 0.4612 \\ 0.3312 \\ 0.2076 \end{array}\right] \tag{1.24}$$

$$\pi_3=\left[\begin{array}{c} 0.4271 \\ 0.5728 \\ 0 \end{array}\right] \tag{1.25}$$

The following matrix contains the posterior distribution of which stations these data sequences come from. Only the first 10 rows are printed.

$$P(h \mid V, \theta) = \begin{bmatrix} 0.0000 & 1.0000 & 0.0000 \\ 0.9998 & 0.0000 & 0.0002 \\ 0.0000 & 0.0000 & 1.0000 \\ 0.0000 & 0.0000 & 1.0000 \\ 0.0000 & 0.0000 & 1.0000 \\ 1.0000 & 0.0000 & 0.0000 \\ 0.0000 & 1.0000 & 0.0000 \\ 0.0000 & 0.0000 & 1.0000 \\ 0.0000 & 0 & 1.0000 \\ 0.0000 & 0.0000 & 1.0000 \end{bmatrix} \tag{1.26}$$

1.3.3 (c) Initial guess for parameters

As we discussed, given an initialisation of parameters, iterative E-M steps can be carried until convergence. It is commonly accepted that the initialised parameters of mixture models can seriously affect solutions. One of the reasons is that mixture models tend to have a log-likelihood function with more than one local maximum (i.e. it is non-convex), and it is generally true even for the most simplified model with an observed variable and a hidden variable. Thus, depending on the initial choices of parameters, successive EM steps can possibly converge to a local maximum instead of a global maximum. Exercise-51 in BSML reemphasizes the importance of initial guesses by providing an example in which a 0.05 difference in initialised parameter results in 30% disparity in log-likelihood.

An effective way to avoid converging to a log-likelihood local maximum is to try randomly chosen parameters and record their converged value. When a sufficient number of parameters has been tested out ,their highest log-likelihood value can be taken as a global maximum with increasing confidence.

Our strategy here is to randomly pick our initialisation number between [0,1], and normalise each column to have sum of 1. For example, when initialising the $p(h)$ distribution, we define a 3-by-1 matrix and fill it with random numbers between [0,1]. Say now we have our chosen p(h) as:

$$p(h) = \begin{pmatrix} 0.7250 \\ 0.2903 \\ 0.5001 \end{pmatrix} \tag{1.27}$$

Then one of our function will do the following step to normalised $p(h)$ so the sum of each column equals 1. Several random initialisations have been run and found to provide highly similar results.

$$p(h) = \begin{pmatrix} 0.7250/(0.7259 + 0.2903 + 0.5001) \\ 0.2903/(0.7259 + 0.2903 + 0.5001) \\ 0.5001/(0.7259 + 0.2903 + 0.5001) \end{pmatrix} \tag{1.28}$$

The normalisation makes sense because the sum $p(h1)..p(h3)$ should be equal to 1 according to probability rules.

Computational issue In the implementation phase, we mainly adapted the 'demoMixMarkov' toolbox provided in the book with slight modification to suit our dataset. The original author of the code uses the observed states v directly as an index because he was dealing with states {1,2,3,4}, where the elements are exactly the same as their index numbers. However, this is not valid in this case because we define our set of states as {0,1,2}. Since MATLAB does not take any non-positive index(for example, 0), we had to modify our states to be {1,2,3}.

1.3.4 (d) Uniform guess

On an intuitive sense, uniform guesses on initial parameters can eventually improve its estimates by accommodating an increasing number of evidence/observations. However, it will fail to distinguish between individual models since they have been assigned with equal weights initially and the same update will be applied to all transition matrices throughout each iteration. By assuming uniformity at the beginning, we actually train this algorithm to learn a single Markov chain distribution, not a mixture of three as wanted and expected.

When uniform initialisation was applied to the EM algorithm, the learnt parameters were found to be:

$$
\begin{gathered}
M = \begin{pmatrix} 0.1598 & 0.1445 & 0.4870 \\ 0.3060 & 0.2835 & 0.3502 \\ 0.5343 & 0.5720 & 0.1628 \end{pmatrix} \\
\mathrm{p(h)} = \begin{pmatrix} 0.3740 \\ 0.4040 \\ 0.2220 \end{pmatrix}
\end{gathered} \tag{1.29}
$$

Now, due to the reason demonstrated before, the transition matrix does not differentiate between the three weather stations which means they share the same M. This departures from our initial objective of using the EM algorithm: identify and cluster different sequential patterns and conduct accurate classification. Effectively, when such uniformity is assumed, each sequence will have the same opportunity to be classified in each cluster.

Ideally, there are some initialisation techniques to apply even when prior understanding of parameters is limited. Random values, as we did in (c), is shown to produce different transition matrices, thus it can successfully cluster our data. Other existing techniques to achieve this include iterative constrained EM algorithm, k-means, and Agglomerative hierarchical clustering.

Part 2

2.1 LDPC codes

Solutions to 2.1.1, 2.1.3, and 2.1.4 were generated using Python. Code snippets are available in section 4.2 in the Appendix.

2.1.1 Generator Matrix

Given a parity check matrix:

$$H = \begin{bmatrix} 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 & 0 \end{bmatrix} \tag{2.30}$$

The systematic coding matrix is:

$$G = \begin{bmatrix} 1 & 1 & 0 \\ 1 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.31}$$

And $\hat{H}$ is:

$$\hat{H} = \begin{bmatrix} 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 \end{bmatrix} \tag{2.32}$$

2.1.2 Factor-Graph

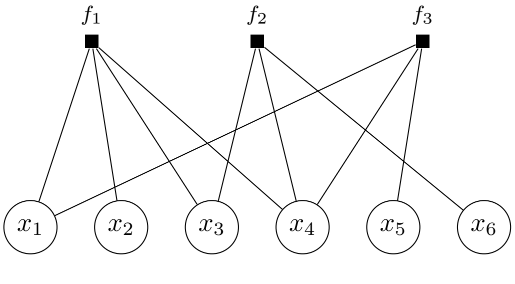

Figure 2.1.1: Factor graph of matrix H

The distribution corresponding to H is:

$$p(x) = f_1(x_1, x_2, x_3, x_4) f_2(x_3, x_4, x_6) f_3(x_1, x_4, x_5) \tag{2.33}$$

The updates for the messages are:

$$\mu_{f_m \to x_n}(x_n) = \sum_{x'_n : n' \epsilon N(m) \backslash n} f_m(x_n, x'_n) \prod_{x'_n \epsilon N(m) \backslash n} \mu_{x'_n \to f_m}(x'_n) \tag{2.34}$$

$$\mu_{x_n \to f_m}(x_n) = \prod_{m' \epsilon N(n) \setminus m} \mu_{f'_m \to x_n}(x_n) \tag{2.35}$$

2.1.3 Decoding

The decode function was successful and took 8 iterations to converge. Our output vector is a 1000 x 1 array as shown below:

```
        [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, 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, 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]
```

2.1.4 Message

The original message reads: Happy Holidays! Dimitry&David :)

Part 3

3.1 Exact Inference

In line with the instructions, the problem was treated as a singly connected graph by collapsing the lattice column-wise. This means that each node cluster is actually a collection of 10 nodes, whose probability distribution can be given by:

$$P(X_1, X_2 \cdots X_{10}) \propto \phi(X_1)\phi(X_2)\cdots\phi(X_{10}) \tag{3.36}$$

Where the variables $X_1 \cdots X_{10}$ are the nodes in the respective row. For a lattice size of $n = 10$ the singly connected graph has 10 node clusters. With this determined, it is possible to use message passing to determine the probability distributions of each node cluster.

First, all possible permutations for a single node cluster are found. This is identical for each of the node clusters, since they all have the same number of identical binary nodes. Using this configuration, the probability distribution is found using the Ising model, that is:

$$P(X_1, X_2 \cdots X_{10}) \propto e^{\beta(X_1 == X_2 + X_2 == X_3 \cdots X_9 == X_{10})} \tag{3.37}$$

Then, a probability distribution over all possible configurations for any pair of nodes were found (using the same Ising model). This is a matrix $\underline{\underline{C}}$ of size $2^{10} \times 2^{10}$.

Following this, message passing was used to pass messages from the left most node to the right most node, and vice versa. To pass messages from the left most node, the following equation was used:

$$\underline{\mu_t} = \underline{\underline{C}} \cdot \underline{\mu}_{t-1} \tag{3.38}$$

Where $\underline{\mu}_{t-1}$ is a vector determining the probability distribution of the previous node cluster. Since all the nodes in the lattice have the same variables, most of this calculation was vectorised (except for the sequential component).

Table 3.1.1: Joint probability distribution $P(x_{1,10}, x_{10,10})$ with exact inference, $\beta = 4$

	$P(x_{1,10} = 0)$	$P(x_{1,10} = 1)$
$P(x_{10,10} = 0)$	0.343	0.157
$P(x_{10,10} = 1)$	0.157	0.343

Table 3.1.2: Joint probability distribution $P(x_{1,10}, x_{10,10})$ with exact inference, $\beta = 1$

	$P(x_{1,10} = 0)$	$P(x_{1,10} = 1)$
$P(x_{10,10} = 0)$	0.25	0.25
$P(x_{10,10} = 1)$	0.25	0.25

Table 3.1.3: Joint probability distribution $P(x_{1,10}, x_{10,10})$ with exact inference, $\beta = 0.01$

	$P(x_{1,10} = 0)$	$P(x_{1,10} = 1)$
$P(x_{10,10} = 0)$	0.25	0.25
$P(x_{10,10} = 1)$	0.25	0.25

3.2 Mean Field Approximation

A random matrix $\underline{\underline{q}}$ of shape $(2, 10, 10)$ is created that represents the probabilities that each node in the lattice take a value of 0 or 1 respectively. Then coordinate ascent is formed to find the $\underline{\underline{q}}^\star$ that minimises the error (defined as being less than a pre-specified threshold ϵ), or until a certain number of iterations pass (e.g. a brute force stop to the optimisation). Doing this loop, the new iteration of q is found by:

$$\underline{\underline{q}}^{(k)} \leftarrow e^{E[\underline{\underline{q}}^{(k-1)}]} \tag{3.39}$$

Where E is the expectation. Given the optimal $\underline{\underline{q}}^\star$, the joint probability distribution was found by summing over all the possible values that each node could take, while considering the likelihood of it's neighbours as well. The final probability distribution was then found by normalising the sums.

Table 3.2.1: Joint probability distribution $P(x_{1,10}, x_{10,10})$ using Mean Field Approximation with a maximum of 100 iterations, $\beta = 4$

	$P(x_{1,10} = 0)$	$P(x_{1,10} = 1)$
$P(x_{10,10} = 0)$	0.391	0.250
$P(x_{10,10} = 1)$	0.250	0.109

Table 3.2.2: Joint probability distribution $P(x_{1,10}, x_{10,10})$ using Mean Field Approximation with a maximum of 100 iterations, $\beta = 1$

	$P(x_{1,10} = 0)$	$P(x_{1,10} = 1)$
$P(x_{10,10} = 0)$	0.25	0.25
$P(x_{10,10} = 1)$	0.25	0.25

Table 3.2.3: Joint probability distribution $P(x_{1,10}, x_{10,10})$ using Mean Field Approximation with a maximum of 100 iterations, $\beta = 0.01$

	$P(x_{1,10} = 0)$	$P(x_{1,10} = 1)$
$P(x_{10,10} = 0)$	0.199	0.259
$P(x_{10,10} = 1)$	0.241	0.301

3.3 Gibbs Sampling

In Gibbs Sampling, the Markov Random Field is instantiated in each run. Then, for each node of the instance, the probability distribution is estimated using the potentials from it's neighbours. If the estimated probability of the node taking a value in {0, 1} is higher than a randomly sampled probability, then the node is relabelled to take that value. Then, using a counter, the estimated joint probability of $x_{1,10}$ and $x_{10,10}$ is updated based on the updated Markov Random Field instance. After all the runs are complete, the joint probability matrix has counts for each pair of values {0, 1}, {1, 1}, {1, 0}, {0, 0}. This is normalised to provide a probability distribution.

In order to make sure that this process is random, the node order is shuffled. An extra hyperparameter is used to repeat the updating process, making sure that the value appended to the estimated joint probability suffers from less noise. In experiments, this led to better results.

Table 3.3.1: Joint probability distribution $P(x_{1,10}, x_{10,10})$ using Gibbs Sampling with 10000 runs, $\beta = 4$

	$P(x_{1,10} = 0)$	$P(x_{1,10} = 1)$
$P(x_{10,10} = 0)$	0.332	0.1769
$P(x_{10,10} = 1)$	0.1696	0.3215

Table 3.3.2: Joint probability distribution $P(x_{1,10}, x_{10,10})$ using Gibbs Sampling with 10000 runs, $\beta = 1$

	$P(x_{1,10} = 0)$	$P(x_{1,10} = 1)$
$P(x_{10,10} = 0)$	0.241	0.251
$P(x_{10,10} = 1)$	0.247	0.262

Table 3.3.3: Joint probability distribution $P(x_{1,10}, x_{10,10})$ using Gibbs Sampling with 10000 runs, $\beta = 0.01$

	$P(x_{1,10} = 0)$	$P(x_{1,10} = 1)$
$P(x_{10,10} = 0)$	0.251	0.247
$P(x_{10,10} = 1)$	0.250	0.252

Appendix

4.1 Part 1

4.1.1 Two earthquakes problem

The code snippet below was written in order to answer questions in part 1.1.

```
import numpy as np
import sys
np.set_printoptions(threshold=sys.maxsize)
import matplotlib.pyplot as plt
from itertools import product
import time
import os
from scipy.stats import norm

print(’Answering question 1!’)

data = np.loadtxt(’EarthquakeExerciseData.txt’)
scaled_data = 1 + data/max(data) # for plotting purposes
# adding 1 to the obvs value to ensure outward projection, (cos and sin values of the )

def value(x_true,y_true,x_sensor,y_sensor):
    return 1/(0.1+ (x_sensor-x_true)**2 + (y_sensor-y_true)**2)


# Building spherical 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

# building the spherical coordinate system as done in setup file
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)

# Computing station locations

# getting positions of the stations
N=len(data) # number of stations
sd=0.2 # standard deviation of the sensor noise
x_sensor = np.zeros(N)
y_sensor = np.zeros(N)
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)

# Building hypothesis space

# Building hypothesis space
# earthquakes are said to be independent, hence the value function is called independently at every station
```

```
print('Building the hypothesis space...')
build = time.time()
hypothesis = np.zeros((S,N))
for s in range(S):
    for i in range(N):
        hypothesis[s][i]=value(x[s],y[s],x_sensor[i],y_sensor[i])

v = np.zeros((S,S, N))
for sensor in range(N):
    for j in range(S):
        for k in range(S):
            v[j,k,sensor] = hypothesis[j,sensor] + hypothesis[k,sensor]

print('Took',time.time() - build,'seconds to build hypothesis space for 2 earthquakes')

# Computing/Plotting posterior distribution

# Determine likelihood, p(location|observed sensor values), given these sensor values
like = np.ones((S,S))
for sensor in range(N):
    like[:, :] *= norm.pdf(v[:,:,sensor], data[sensor], sd)
double_sum = np.sum(like,axis = 0) # summing along one direction in order to marginalise out s2
double_scale = double_sum/double_sum.max()

# Plotting posterior distribution

plt.figure()
# plotting posterior distribution of s1
plt.scatter(x, y, s = double_scale*30,zorder = 0, color = "black")

# plottling perimeter
for theta in np.arange(0,2*np.pi,0.01):
    plt.plot(np.cos(theta),np.sin(theta),".",color=[0,0,0])

# plotting station locations
for sensor in range(N):
    plt.plot(x_sensor[sensor],y_sensor[sensor],"o",color=[1,0,0])

# plotting observed values at each sensor (spikes shown in fig (1.3))
for sensor in range(N):
    scale =1
    theta_sensor = 2*np.pi*sensor/N
    base = [x_sensor[sensor],((scaled_data[sensor])*np.cos(theta_sensor))]
    top = [y_sensor[sensor], ((scaled_data[sensor])*np.sin(theta_sensor))]
    plt.plot(base, top,"-",color='b')


plt.title("Distribution of $P(S_{1}|V)$")
plt.savefig('posterior.png')
plt.show()


print('Answering question 2!')

# Computing likelihood for both situations

# for 1 explosion
like_single = np.ones(S)
for sensor in range(N):
    like_single *= norm.pdf(hypothesis[:,sensor], data[sensor], sd)  # using 1 explosion hypothesis
single_sum = like_single.sum()

log_comp = np.log(double_sum.sum()) - np.log(single_sum)
print('The value to report is:',log_comp)
```

4.1.2 Meeting Scheduling

```python
#####################################
# Imports
####################################

import numpy as np
import matplotlib.pyplot as plt
import torch
import torch.distributions as dist

#####################################
# Calculating the prior
#####################################

dist1 = dist.Binomial(5, (1/3))
calc_prior_not_punct = lambda x: dist1.log_prob(torch.Tensor([x])).exp().item()
prior_not_punct = np.array([calc_prior_not_punct(k) for k in range(6)])

#####################################
# Calculating the likelihood
#####################################

def calc_lik_not_punct(k):
    lik = 1 - (((0.99)**(5-k) ) * ((0.95)**(k)))
    return lik
lik_not_punct = np.array([calc_lik_not_punct(k) for k in range(6)])

#####################################
# Calculating the posterior
#####################################

def calc_posterior(prior_not_punct, lik_not_punct):
    posterior = prior_not_punct * lik_not_punct / np.sum(prior_not_punct * lik_not_punct)
    return posterior
posterior = calc_posterior(prior_not_punct, lik_not_punct)

#####################################
# Plotting
#####################################

priors = [0.132, 0.329, 0.329, 0.165, 0.041, 0.004]
posteriors = posterior
f, (ax1, ax2) = plt.subplots(1, 2, figsize = (12, 4))
ax1.set_title("Prior probability of k friends being non-punctual")
ax1.set_xlabel(’k’)
ax1.set_ylabel(’P(K = k)’)
ax1.set_ylim(0, 0.39)
ax1.bar(list(range(6)), priors, color = ’royalblue’, edgecolor = ’black’);
ax2.set_title("Posterior probability of k friends being non-punctual")
ax2.set_ylim(0, 0.39)
ax2.set_xlabel(’k’)
ax2.set_ylabel(’P(K = k| miss)’)
ax2.bar(list(range(6)), posteriors, color = ’royalblue’, edgecolor = ’black’);
```

4.1.3 Weather Stations

```matlab
function [ph,pv1gh,pvgvh,loglikelihood,phgv]=mixMarkov(v,V,H,opts)
%MIXMARKOV EM training for a mixture of Markov Models
%[ph,pv1gh,pvgvh,loglikelihood,phgv]=mixMarkov(v,V,H,opts)
%
% Inputs:
% v : cell array of the data. v{1} contains the first data sequence, v{2} the next etc.
% V : the number of visible states of the data
% H : number of mixture componetns.
% opts.plotprogress=1 to display log likelihood
% opts.maxit : the number of iterations of the EM algorithm
%
% Outputs:
```

```
% ph : learned p(h)
% pv1gh : p(v(1)|h)
% pvgvh : p(v(t)|v(t-1),h)
% loglikelihood : log likelihood of all the sequences (assuming iid)
% phgv : posterior probability p(h|v) for each sequence
% See also demoMixMarkov.m


%ph= condp(rand(H,1)); % random parameter initialisation
%pv1gh=condp(rand(V,H));
%pvgvh=condp(rand(V,V,H));

ph=ones(H,1)/3;
pv1gh=ones(V,H)/3;
pvgvh=condp(ones(V,V,H));

for emloop=1:opts.maxit
  % E-step
  ph_stat=zeros(H,1);
  pv1gh_stat =zeros(V,H);
  pvgvh_stat =zeros(V,V,H);
  loglik=0;
  for n=1:length(v)
    T = length(v{n});
    lph_old =log(ph)+log(pv1gh(v{n}(1),:)');
    for t=2:T
      lph_old=lph_old+squeeze(log(pvgvh(v{n}(t),v{n}(t-1),:)));
    end
    ph_old{n}=condexp(lph_old);
    loglik = loglik + logsumexp(lph_old,ones(H,1)); % avoids numerical underflow
    % collect statistics for M-step:
    ph_stat=ph_stat + ph_old{n};
    pv1gh_stat(v{n}(1),:)=pv1gh_stat(v{n}(1),:) + ph_old{n}';
    for t=2:T
      pvgvh_stat(v{n}(t),v{n}(t-1),:) = squeeze(pvgvh_stat(v{n}(t),v{n}(t-1),:))+ph_old{n};
    end
  end
  llik(emloop)=loglik;
  if opts.plotprogress; plot(llik); title('log likelihood'); drawnow; end
  % M-step
  ph = condp(ph_stat);
  pv1gh = condp(pv1gh_stat);
  pvgvh = condp(pvgvh_stat);
end
loglikelihood=llik(end); phgv=ph_old;
disp(ph)
disp(pvgvh)
disp(pv1gh)
disp(llik)

function demoMixMarkov_modified
%DEMOMIXMARKOV demo of training a mixture of Markov models
load v

for n=1:500 % convert the characters into (arbitrary) numerical values (from 1 to 4)
    for m=1:100
        v{n}(m)=v{n}(m)+1;
    end
end

opts.maxit=50; opts.plotprogress=1;
[ph,pv1gh,pvgvh,loglikelihood,phgv]=mixMarkov(v,3,3,opts);
save('phgv.mat','phgv')

% For each sequence find the highest posterior hidden class:
c=ones(1,100);
class2sequences=[]; class1sequences=[]; class3sequences=[];
for n=1:100
  if phgv{n}(1)<0.33; c(n)=1;
    class1sequences=vertcat(class1sequences,v{n});
    elseif phgv{n}(1)>0.66; c(n)=3;
    class3sequences=vertcat(class3sequences,v{n});
    else
        class2sequences=vertcat(class2sequences,v{n});
  end
end

function pnew=condp(pin,varargin)
%CONDP Make a conditional distribution from an array
% pnew=condp(pin,<distribution indices>)
%
% Input : pin  -- a positive matrix pin
% Output:  matrix pnew such that sum(pnew,1)=ones(1,size(p,2))
%
% The optional specifies which indices form the distribution variables.
% For example:
% r=rand([4 2 3]); p=condp(r,[3 1]);
% p is now an array of the same size as r, but with sum(sum(p,3),1) equal
% to 1 for each of the dimensions of the 2nd index.
m=max(pin(:));
if m>0
    p = pin./m;
else
    p=pin+eps;% in case all unnormalised probabilities are zero
end
if nargin==1
 pnew=p./repmat(sum(p,1),size(p,1),1);
else
    allvars=1:length(size(pin));
    sizevars=size(pin);
    distvars=varargin{1};
    condvars=setdiff(allvars,distvars);
    newp=permute(pin,[distvars condvars]);
    newp=reshape(newp,prod(sizevars(distvars)),prod(sizevars(condvars)));
    newp=newp./repmat(sum(newp,1),size(newp,1),1);
    pnew=reshape(newp,sizevars([distvars condvars]));
    pnew=ipermute(pnew,[distvars condvars]);
end

function anew=logsumexp(a,varargin)
%LOGSUMEXP Compute log(sum(exp(a).*b)) valid for large a
% logsumexp(a,<b>)
% if b is missing it is assumed to be 1
% example: logsumexp([-1000 -1001 -998],[1 2 0.5])
if nargin==1
    b=ones(size(a));
else
    b=varargin{1};
    if isscalar(b); b=b*ones(size(a)); end
end
amax=max(a); A =size(a,1);
anew = amax + log(sum(exp(a-repmat(amax,A,1)).*b));
```

4.2 LDPC Codes

4.2.1 Generator Matrix

The code snippet below was written in order to answer part 2.1.

```
#import libraries
import numpy as np
import matplotlib.pyplot as plt
import math

def rref(H):

    ''' return reduce row echelon form'''
    m, n = H.shape
    # k = n-m

    i = 0
    j = 0
    r = 0

    H_hat = H.copy()

    while True:

        if i >= m or j >= n:
            break

        # print(H_hat)

        #check if entry = 0
        if H_hat[i,j]== 0:

            #find next row with non zero entry to swap
            r = i

            while r < m and H_hat[r,j] == 0 :
                r += 1

            #if no nonzero found skip to next column
            if r == m:
                j+=1
                break

        #swap row
        zero_row = H_hat[i,:].copy()
        nonzero = H_hat[r,:].copy()
        H_hat[i,:] = nonzero
        H_hat[r,:] = zero_row

        # print(H_hat)

        #perform row elimination
        for row in range(m):
            if row == i:
                continue
            #if entry does not = 0, add nonzero row, remember this is in F2 (so mod 2 addition)
            if H_hat[row,j] != 0:
                H_hat[row,:] = (H_hat[row,:] + H_hat[i,:]) % 2

        # print(H_hat)

        i+=1
        j+=1

    return H_hat

def gen_G(H):
    ''' return generator matrix G and H_hat, the row reduced form of H'''
    m, n = H.shape
    k = n-m

    H_hat = rref(H)
    # print("H_hat:\n",H_hat)
    P = H_hat[:, m:n]
    # print("P:\n",P)
    G = np.vstack((P, np.identity(k)))
    # print("G\n",G)

    return H_hat, G


#initialize H
H = np.array([[1, 1, 1, 1, 0, 0],[0, 0, 1, 1, 0, 1],[1, 0, 0, 1, 1, 0]], dtype=int)

H_hat, G = gen_G(H)

print("H_hat:\n",H_hat, "\nG:\n", G)
```

4.2.2 Decoding & Message

The code snippet below was used to answer part 2.3 and 2.4. An Introduction to LDPC Codes[1] was used as a reference.

```
#import libraries
import numpy as np
import matplotlib.pyplot as plt
import math

#note that H1.txt and y1.txt must be saved to the same directory as this script

#import y1
file_y = open("y1.txt")
y_content = file_y.read()
file_y.close()
y = y_content.split("\n")
# print(y1)
y = y[:-1]
y1 = np.array([int(i) for i in y])


#import H1
file_h = open("H1.txt")
H_content = file_h.read()
H1 = H_content.split("\n")
H1 = H1[:-1]
H = [np.array(i.split(), dtype=int) for i in H1]
h1 = np.vstack(H)


def decode(H, y, p=.1, max_iter = 20):

    success = -1
    m, n = H.shape

    #list of factor checks
    B = []
    for i in range(m):
        B.append(np.where(H[i]==1)[0])
    # print("Bit Check:",B)

    #probability y given x

    #if true val is 1
    P_y_x1 = np.zeros(len(y))
    #if true val is 0
    P_y_x0 = np.zeros(len(y))

    for i in range(len(y)):
        P_y_x1[i] = p**((y[i]+1)%2) * (1-p)**((y[i]+1+1)%2)
        P_y_x0[i] = p**(y[i]) * (1-p)**((y[i]+1)%2)

    #initialize
    M = np.zeros((m,n))

    # print(M)

    for i in range(m):
        for j in range(n):
            #for efficiency: take difference and use log liklihood:
            # then u(0) - u(1) -> log(u(0))-log(u(1)) = log(u(0)/u(1))
            M[i,j] = np.log([P_y_x0[j]/P_y_x1[j]])

    # print("Initial Message:\n",M)


    for iter in range(max_iter):

        #factor to variable
        M_f = np.zeros((m,n))

        for j in range(len(B)):

            new_row = np.zeros(n)
            # print("\nrow:", j)

            #get values of M for all indexes where h==1 for the jth row
            prod = M[j,(B[j])]
            # print(prod)

            for i in range(len(prod)):

                # print("\nindex:", B[j][i])

                #take product of all incoming messages excludng the ith bit
                to_prod = np.hstack((prod[0:i],prod[i+1:len(prod)]))
                #note: from ref use fact that tanh(1/2log(1-p/p)) = 1-2p
                product = np.prod(np.tanh(to_prod/2))
                to_log = (1+product)/(1-product)
                # print("log:",to_log)

                new_row[B[j][i]] = np.log(to_log)

            # print(j, new_row)

            M_f[j,:] = new_row
            # M[j,:] = new_row

        # print(M_f)

        #calc log liklihood of posterior
        posterior = np.sum(M_f, axis = 0)

        for i in range(n):
            # calc log like
            posterior[i] += np.log([P_y_x0[i]/P_y_x1[i]])

        # print("\nposterior",posterior)

        #update prediction
        z = np.zeros(n)
        for i in range(n):
            if posterior[i]>0:
                z[i]=0
            else:
                z[i]=1

        # print("\nprediction:", z)

        #test, dont forget mod 2
        test = np.matmul(H, z.T)% 2

        #check if parity conditions met or reached max iterations
        if iter == max_iter or np.all(test == 0):
            success = 0
            break

        #if not continue
        else:

            #update variable to factor
            for i in range(n):
                for j in range(m):
                    M[j,i] = posterior[i] - M_f[j,i]

    return success, iter+1, z

#decode the message
success,iter,decoded = decode(h1,y1, p=.1, max_iter=20)
print(success, "\niterations:",iter, "\nDecoded vector:\n",decoded)

#print the original message in ascii:

#translate
x = np.reshape(decoded.astype(int), (125,8))
x = x.astype(str)
```

```
lst = x.tolist()

b = []
for i in range(len(lst)):
    b.append(’’.join(map(str, lst[i])))

message = []
for i in range(31):
    message.append(chr(int(b[i],2)))

print(’’.join(message))
```

4.3 Exact and Approximate Inference

4.3.1 Exact Inference

```
# Script name: Task3Q1.py
import numpy as np
import plac
import potential_functions
from helpers import get_permutations, pass_messages_to_right,
    pass_messages_to_left, get_joint_binary_distribution
from functools import partial


@plac.annotations(
    lattice_size=("size of square lattice", ’positional’, None, int),
    beta=("temperature of potential function", ’positional’, None, float),
    ids=("ids of vsriables to be joined over", ’positional’, None, int),
)
def main(lattice_size, beta, *ids, potential_function=’ising_potential’):
    """Perform exact inference

    :param lattice_size: the size of square lattice
    :param beta: temperature of potential function
    :param ids: ids of the variables whose joint dist is found
    :param potential_function: string name for the potential function to use
    :return: discrete joint probability distribution
    """

    potential = partial(getattr(potential_functions, potential_function), beta=
    beta)

    n = lattice_size
    single_row_perms = get_permutations([0, 1], repeat=n)
    N = single_row_perms.shape[0]
    assert N == 2 ** n

    # Find initial probabilities for single row / column
    perm_left = single_row_perms[:, :-1]
    perm_right = single_row_perms[:, 1:]
    potentials = potential(perm_left, perm_right)
    initial_potentials = np.prod(potentials, axis=1)
    initial_probas = initial_potentials / np.sum(initial_potentials)

    # determine all potentials
    lattice_perms = get_permutations(single_row_perms, single_row_perms)
    lattice_config = np.prod(potential(lattice_perms[:, 0, :], lattice_perms[:, 1,
     :]), axis=1)
    lattice_config = lattice_config.reshape(-N, N)
    lattice_probas = lattice_config / np.sum(lattice_config, axis=0)

    l_to_r_msg = pass_messages_to_right(lattice_probas, initial_probas, n)
    r_to_l_msg = pass_messages_to_left(lattice_probas, n)

    posterior = l_to_r_msg[n - 1, :] * r_to_l_msg[n - 1, :]
    joint_dist = get_joint_binary_distribution(n, posterior, *ids)

    print(joint_dist)

```

```

if __name__ == ’__main__’:
    plac.call(main)
```

4.3.2 Mean Field Approximation

```
# Script name: Task3Q2.py
import numpy as np
import plac
import potential_functions
from helpers import get_permutations, expected_value, BinaryMRF, get_probas
from functools import partial


@plac.annotations(
    lattice_size=("size of square lattice", ’positional’, None, int),
    beta=("temperature of potential function", ’positional’, None, float),
    ids=("ids of vsriables to be joined over", ’positional’, None, int)
)
def main(lattice_size, beta, *ids, iterations=100, potential_function=’ising_potential’, epsilon=0.00001):
    """Perform approximate inference using mean field approximation

    :param lattice_size: the size of square lattice
    :param beta: temperature of potential function
    :param ids: ids of the variables whose joint dist is found
    :param iterations: determines the max number of iterations for optimisation
    :param potential_function: string name for the potential function to use
    :return: discrete joint probability distribution

    """
    potential = partial(getattr(potential_functions, potential_function), beta=beta)
    n = lattice_size
    shape = (n, n)

    q = np.zeros((2, *shape))
    q[0, ...] = np.random.rand(*shape)
    q[1, ...] = 1 - q[0, ...]

    iter_nr = 0
    err = np.float(’inf’)
    G = BinaryMRF(n, potential).G
    while err > epsilon:
        q_ = np.exp(expected_value(G, q, potential))
        q_ = q_ / q_.sum(axis=0)

        err = np.linalg.norm(q-q_)
        q = q_
        iter_nr += 1
        if iter_nr > iterations:
            break

    probas = get_probas(G, q, potential, *ids)
    print(probas)


if __name__ == ’__main__’:
    plac.call(main)
```

4.3.3 Gibbs Sampling

```
# Script name: Task3Q3.py
import numpy as np
from tqdm import tqdm
import plac
from helpers import get_permutations, BinaryMRF
import potential_functions
from functools import partial
```

```


@plac.annotations(
    lattice_size=("size of square lattice", 'positional', None, int),
    beta=("temperature of potential function", 'positional', None, float),
    ids=("ids of vsriables to be joined over", 'positional', None, int)
)
def main(lattice_size, beta, *ids, runs=10000, iterations=1, potential_function='ising_potential'):
    probas = np.zeros((len(ids), len(ids)))
    n = lattice_size
    potential = partial(getattr(potential_functions, potential_function), beta=beta)

    for run in tqdm(range(runs)):
        mrf_instance = BinaryMRF(n, potential)
        nodes = [node for node in mrf_instance.G.nodes()]

        for iter_nr in range(iterations):
            np.random.shuffle(nodes)

            for node_id, (i, j) in enumerate(nodes):
                probas_ = np.zeros(2)
                for label in [0, 1]:
                    mrf_instance.label[i, j] = label
                    probas_[label] = mrf_instance.get_probas((i, j))

                probas_ = probas_ / probas_.sum()

                # reset value based on probability
                mrf_instance.label[i, j] = 1 if probas_[label] >= np.random.rand() else 0

        x1, x2 = ids
        probas[mrf_instance.label[x1, 9], mrf_instance.label[x2, 9]] += 1

    probas = probas / np.sum(probas)

    print(probas)


if __name__ == '__main__':
    plac.call(main)
```

4.3.4 Shared Code

```
# Script name: helpers.py
import numpy as np
import itertools
from networkx.generators.lattice import grid_graph


def get_permutations(*arr, **kwargs):
    return np.asarray(list(itertools.product(*arr, **kwargs)))


def expected_value(G, q, partial_potential):
    """
    :param partial_potential: pre-initialised potential, ready to be called
    """
    potentials = partial_potential(
        np.array([
            [0, 0],
            [1, 1]
        ]),
        np.array([
            [0, 1],
            [0, 1]
        ])
    )
    potentials = np.log(potentials.sum(axis=1)/4)
    expected_val = np.zeros(q.shape)
    for node in G.nodes():
        for neighbor in G.neighbors(node):
            expected_val[:, node[0], node[1]] += (potentials * q[:, neighbor[0], neighbor[1]])
    return expected_val


class BinaryMRF:
    """Class to create a Binary Markov Random Field
    """

    def __init__(self, n, partial_potential):
        self.label = np.random.randint(0, 2, (n, n))
        self.G = grid_graph(dim=(n, n))
        self.potentials = self.create_binary_potential_matrix(partial_potential)

    def get_probas(self, node):
        proba = 1
        for neighbor in self.G.neighbors(node):
            neighbor_val = self._get_node_label(neighbor)
            proba *= self.potentials[self._get_node_label(node), neighbor_val]
        return proba

    def create_binary_potential_matrix(self, partial_potential):
        perm = get_permutations([0, 1], repeat=2)
        perm_left = perm[:, :-1]
        perm_right = perm[:, 1:]
        potentials = partial_potential(perm_left, perm_right)
        return potentials.reshape(-2, 2)

    def _get_node_label(self, node):
        return self.label[node[0], node[1]]


def get_probas(G, q, potential, *ids):
    """gets joint probability for the ids given
    """

    probas = np.ones((len(ids), len(ids)))

    x1, x2 = ids

    single_row_perms = get_permutations([0, 1], repeat=10)

    for m in range(2):
        for n in range(2):
            for row_id in range(single_row_perms.shape[0]):
                x = single_row_perms[row_id, :4]
                for counter, (x_, (i, j), X) in enumerate(
                        zip(x, [n for i in ids for n in G[(i, 9)]], [x1, x1, x2, x2])
                ):
                    val = m if counter < 2 else n
                    probas[m, n] += q[x_, i, j] * potential(x_, val) * q[val, X, 9]

    probas = probas / np.sum(probas)

    return probas


def pass_messages_to_right(lattice_probas, initial_probas, n):
    """Passes messages from left node to rest of n-1 nodes

    :param lattice_probas: probabilities of all configurations
    :param initial_probas: probabilities of left node
    :param n: number of nodes
    """

    alpha = np.zeros((n, lattice_probas.shape[0]))
    alpha[0, :] = initial_probas

    for t in range(1, n):
        alpha[t] = lattice_probas @ alpha[t - 1]

    return alpha


def pass_messages_to_left(lattice_probas, n):
    """Passes msgs from right node to rest of n-1 nodes

    :param lattice_probas: probabilities of all configs
    :param n: number of nodes
    """
    beta = np.zeros((n, lattice_probas.shape[0]))
    beta[n - 1, :] = np.ones(lattice_probas.shape[0])

    for t in range(n - 2, -1, -1):
        beta[t] = beta[t+1] @ lattice_probas

    return beta


def get_joint_binary_distribution(n, posterior, *ids):
    """Gets joint binary distribution

    :param n: number of nodes
    :param posterior: posterior
    :param ids: positions of variables whose joint is to be found
    """

    if len(ids) != 2:
        raise TypeError(’Function only supports binary joint distributions’)

    probas = np.zeros((len(ids), len(ids)))
    x1, x2 = ids

    # Obtain all permutations
    single_row_perms = get_permutations([0, 1], repeat=n)

    for i in range(single_row_perms.shape[0]):
        probas[
            single_row_perms[i, x1],
            single_row_perms[i, x2]
        ] += posterior[i]

    return probas
```

```
# Script name: potential_functions.py
import numpy as np
from helpers import expected_value, expectation
from networkx.generators.lattice import grid_graph


def ising_potential(xi, xj, beta):
    return np.exp((beta*(xi == xj)))
```

References

1. Ryan, W. E. An Introduction to LDPC Codes. *Coding and Signal Processing for Magnetic Recording Systems Computer Engineering Series.* http://tuk88.free.fr/LDPC/ldpcchap.pdf (2004).