COMP0086

Probabilistic and Unsupervised Learning Coursework

October 2024

University College London

Contents

1 Models for binary vectors

1.1 Question 1a

A multivariate Gaussian would not be an appropriate model because of the nature of the data. The dataset contains discrete data, describing the pixels with binary values (0 or 1). In contrast, the multivariate normal distribution is designed to model continuous data, with values that can be any real number. The Gaussian distribution would predict a range of values between 0 and 1, which aren't valid pixel values for the images encoded in binary. On the other hand, the Bernoulli distribution has the same sample space as the dataset, and thus would be a better choice of model to describe it.

1.2 Question 1b

For a D-dimensional multivariate Bernoulli distribution, the likelihood of observing a single image $\mathbf{x}^{(n)}$ given the parameter vector $\mathbf{p}$ is given by:

$$P(\mathbf{x}^{(n)}|\mathbf{p}) = \prod_{d=1}^{D} p_d^{x_d^{(n)}} (1 - p_d)^{1 - x_d^{(n)}}$$

Knowing that the N images are modeled as i.i.d. samples from the distribution, the likelihood function can be written as:

$$L(\mathbf{p}) = \prod_{n=1}^{N} P(\{\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(N)}\}|\mathbf{p}) = \prod_{n=1}^{N} \prod_{d=1}^{D} p_d^{x_d^{(n)}} (1 - p_d)^{1 - x_d^{(n)}}$$

We take the logarithm of the likelihood function to obtain the log-likelihood:

$$\log L(\mathbf{p}) = \sum_{n=1}^{N} \sum_{d=1}^{D} \left[x_d^{(n)} \log(p_d) + (1 - x_d^{(n)}) \log(1 - p_d) \right]$$

This transforms the products into sums, and doesn't move the extrema as the logarithm is a monotonically increasing function. To find the maximum likelihood estimate, we take the derivative of the log-likelihood with respect to each parameter p_d and set it equal to zero:

$$\frac{\partial}{\partial p_d} \log L(\mathbf{p}) = \sum_{n=1}^{N} \left[\frac{x_d^{(n)}}{p_d} - \frac{1 - x_d^{(n)}}{1 - p_d} \right] = 0$$

Rearranging the above equation gives us:

$$\sum_{n=1}^{N} x_d^{(n)} \cdot (1 - p_d) = \sum_{n=1}^{N} (1 - x_d^{(n)}) \cdot p_d$$

Which we can simplify to get:

$$\sum_{n=1}^{N} x_d^{(n)} = p_d \cdot N$$

Thus, the maximum likelihood estimate for p_d is:

$$p_d = \frac{1}{N} \sum_{n=1}^{N} x_d^{(n)}$$

Or in vector form:

$$\mathbf{p}_{ML} = \frac{1}{N} \sum_{n=1}^{N} \mathbf{x}^{(n)}$$

1.3 Question 1c

To find the Maximum a posteriori (MAP) estimate, we use Bayes theorem to incorporate prior beliefs. In our case, Bayes theorem states that:

$$P(\mathbf{p}|\{\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(N)}\}) = \frac{P(\{\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(N)}\}|\mathbf{p}) \cdot P(\mathbf{p})}{P(\{\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(N)}\})}$$

To find the MAP estimate, we maximise the posterior:

$$
\begin{aligned}
\mathbf{p}_{\text{MAP}} &= \arg\max_{\mathbf{p}} P(\mathbf{p} \mid \{\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(N)}\}) \\
&= \arg\max_{\mathbf{p}} P(\{\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(N)}\} \mid \mathbf{p}) \cdot P(\mathbf{p}) \\
&= \arg\max_{\mathbf{p}} \left(\log P(\{\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(N)}\} \mid \mathbf{p}) + \log P(\mathbf{p})\right)
\end{aligned}
$$

As before, the likelihood of observing N independent images is:

$$
P(\mathbf{x}^{(n)}|\mathbf{p}) = \prod_{n=1}^{N} \prod_{d=1}^{D} p_d^{x_d^{(n)}} (1 - p_d)^{1 - x_d^{(n)}}
$$

And the corresponding log-likelihood:

$$
\log P(\mathbf{x}^{(n)}|\mathbf{p}) = \sum_{n=1}^{N} \sum_{d=1}^{D} \left[x_d^{(n)} \log(p_d) + (1 - x_d^{(n)}) \log(1 - p_d) \right]
$$

We know that the prior for each p_d is given as a Beta distribution by:

$$
P(p_d) = \frac{1}{B(\alpha, \beta)} p_d^{\alpha - 1} (1 - p_d)^{\beta - 1}
$$

Since the priors on each p_d are independent, the prior for $\mathbf{p}$ is:

$$
P(\mathbf{p}) = \prod_{d=1}^{D} P(p_d) = \prod_{d=1}^{D} \frac{1}{B(\alpha, \beta)} p_d^{\alpha - 1} (1 - p_d)^{\beta - 1}
$$

We get the log-prior by taking the logarithm, which yields:

$$
\log P(\mathbf{p}) = \sum_{d=1}^{D} \left[(\alpha - 1) \log(p_d) + (\beta - 1) \log(1 - p_d) \right] + \text{constant}
$$

The constants won't change the position of the extrema so they aren't very important to the calculation. The log of the posterior is:

$$
\log P(\mathbf{p}|\mathbf{x}^{(n)}) \propto \log P(\mathbf{x}^{(n)}|\mathbf{p}) + \log P(\mathbf{p})
$$

We can then substitute the expressions for $\log P(\mathbf{x}^{(n)}|\mathbf{p})$ and $\log P(\mathbf{p})$ to get:

$$\log P(\mathbf{p} \mid \mathbf{x}^{(n)}) \propto \sum_{d=1}^{D} \left[\sum_{n=1}^{N} \left(x_d^{(n)} \log(p_d) + (1 - x_d^{(n)}) \log(1 - p_d) \right) \right.$$
$$\left. + (\alpha - 1) \log(p_d) + (\beta - 1) \log(1 - p_d) \right]$$

To maximize this, we take the derivative with respect to p_d and set it to zero:

$$\frac{\partial}{\partial p_d} \log P(\mathbf{p}|\mathbf{x}^{(n)}) = \frac{\sum_{n=1}^{N} x_d^{(n)}}{p_d} - \frac{\sum_{n=1}^{N} (1 - x_d^{(n)})}{1 - p_d} + \frac{\alpha - 1}{p_d} - \frac{\beta - 1}{1 - p_d} = 0$$

Rearranging the terms yields:

$$\sum_{n=1}^{N} x_d^{(n)} + (\alpha - 1) = p_d \left(N + \alpha + \beta - 2 \right)$$

Solving for p_d gives the MAP estimate for p_d:

$$p_d = \frac{\sum_{n=1}^{N} x_d^{(n)} + \alpha - 1}{N - 2 + \alpha + \beta}$$

We can write the MAP estimate in vectorised form:

$$\mathbf{p}_{MAP} = \frac{\sum_{n=1}^{N} \mathbf{x}^{(n)} + \alpha - 1}{N - 2 + \alpha + \beta}$$

because the images were modelled as independently and identically distributed samples.

1.4 Question 1d

Listing 1 shows the code that was written to find the Maximum Likelihood estimate of the parameters, as well as display Figure 1.

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

# We load the data into a numpy array
x = np.loadtxt('binarydigits.txt')

```

```
# We compute the maximum likelihood estimate of the parameters of the multivariate Bernoulli
maximum_likelihood = np.mean(x, axis=0)

# We plot the learnt parameters with matplotlib
plt.figure()
plt.title("Maximum Likelihood estimate of the parameters")
plt.imshow(np.reshape(maximum_likelihood, (8, 8)), cmap='inferno')
plt.colorbar()
plt.show()
```

Listing 1: Python code used to compute ML estimates and display figure 1

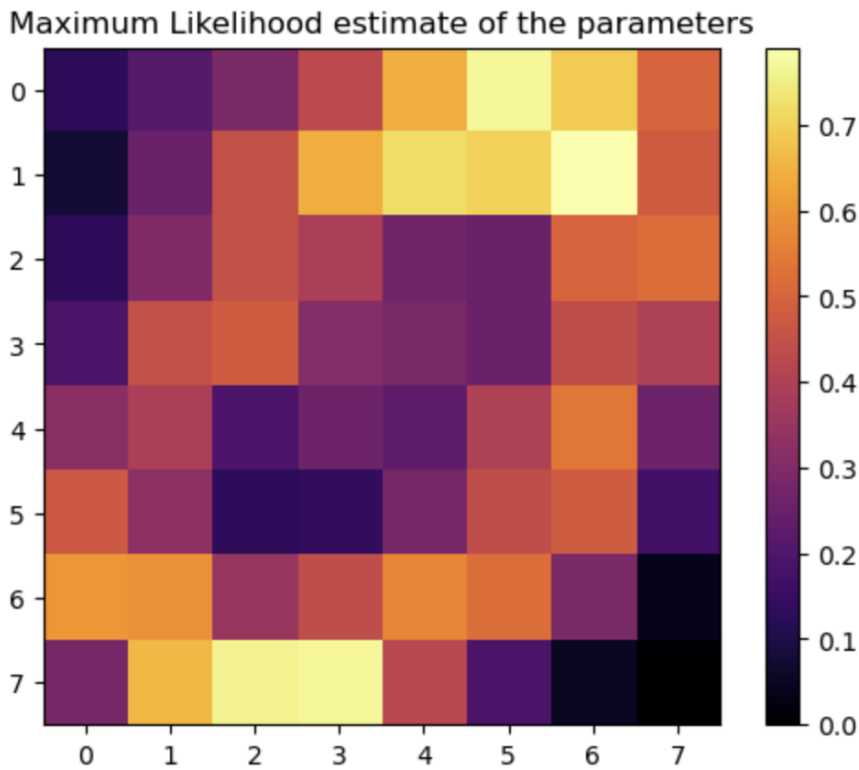


Figure 1: Maximum Likelihood estimate of the parameters

1.5 Question 1e

Listing 2 shows the code that was written to find the MAP estimate of the parameters, as well as display Figure 2.

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

# We load the data into a numpy array
x = np.loadtxt('binarydigits.txt')
```

```python

# We define values of alpha and beta
alpha = 3
beta = 3

# We calculate the MAP estimate of the parameters
maximum_a_posteriori = (alpha - 1 + np.sum(x, axis=0)) / (x.shape[0] + alpha + beta - 2)

# We plot the learned parameters
plt.figure()
plt.title("Maximum a posteriori estimate of the parameters")
plt.imshow(np.reshape(maximum_a_posteriori, (8, 8)), cmap='inferno')
plt.colorbar()
```

Listing 2: Python code used to compute MAP estimates and display figure 2

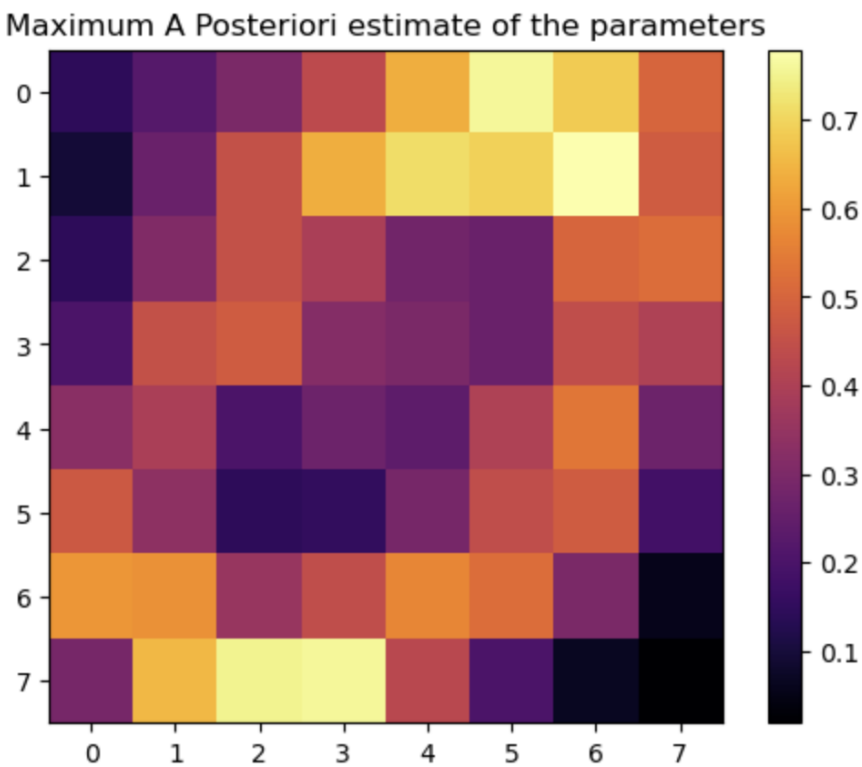


Figure 2: Maximum A Posteriori estimate of the parameters

From figures 1 and 2, we see that both methods find similar estimates of the parameters. However, introducing a prior makes the MAP estimate less certain about its predictions. This is not obvious from figures 1 and 2, so an easy way to show this is by plotting the difference between the MAP and ML estimates:

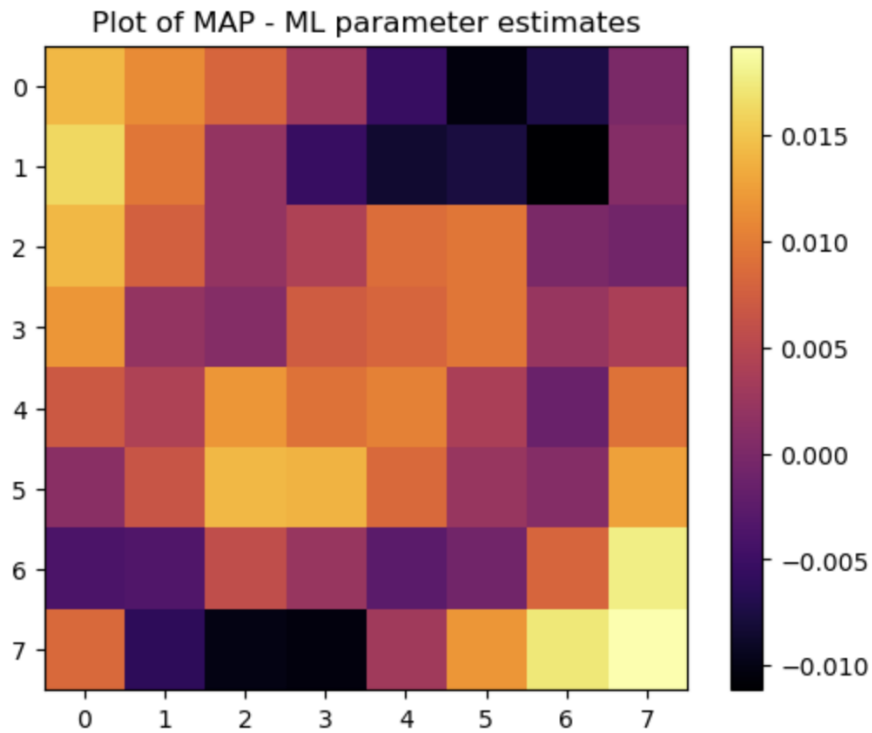


Figure 3: Plot of MAP-ML parameter estimates

We see that the difference is negative when parameters estimates are higher, meaning the MAP is smaller than the ML, and it is positive for lower parameter estimates, meaning the MAP is bigger than the ML. The MAP solution thus gives higher probability to cases it hasn't seen before, and this uncertainty will slowly decrease as more examples are given, since the prior will has less and less impact. The MAP estimate is better because it allows us to have some uncertainty which reflects the quantity of data the model has seen.

2 Model selection

2.1 Question 2a

In this model, all components are generated from a Bernoulli distribution with $p_d = 0.5$. Plugging this into the Bernoulli likelihood equation:

$$P(\{\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(N)}\}|\mathbf{p}^{(n)} = (0.5, 0.5, \ldots, 0.5)) = \prod_{n=1}^{N}\prod_{d=1}^{D}(0.5)^{x_d^{(n)}}(0.5)^{1-x_d^{(n)}}$$

Since this is a Bernoulli distribution, x_d can only take the values 0 or 1. If $x_d = 0$, the the likelihood simplifies to:

$$P(\{\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(N)}\}|\mathbf{p}^{(n)} = (0.5, 0.5, \ldots, 0.5)) = \prod_{n=1}^{N}\prod_{d=1}^{D}(0.5)^{0}(0.5)^{1-0} = \prod_{n=1}^{N}\prod_{d=1}^{D}(0.5)$$

On the other hand if $x_d = 1$, the the likelihood simplifies to:

$$P(\{\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(N)}\}|\mathbf{p}^{(n)} = (0.5, 0.5, \ldots, 0.5)) = \prod_{n=1}^{N}\prod_{d=1}^{D}(0.5)^{1}(0.5)^{1-1} = \prod_{n=1}^{N}\prod_{d=1}^{D}(0.5)$$

This means that no matter the value of x_d we have the same likelihood function. Therefore the likelihood function for this model is:

$$P(\{\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(N)}\}|\mathbf{p}^{(n)}) = \prod_{n=1}^{N}\prod_{d=1}^{D}(0.5) = (0.5)^{N \cdot D}$$

2.2 Question 2b

In this model, all components are generated from Bernoulli distributions with unknown, but identical p_d. Plugging this into the Bernoulli likelihood equation:

$$P(\{\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(N)}\}|\mathbf{p}^{(n)} = (p_d, p_d, \ldots, p_d)) = \prod_{n=1}^{N}\prod_{d=1}^{D}(p_d)^{x_d^{(n)}}(1-p_d)^{1-x_d^{(n)}}$$

To find the relative probability, we have to integrate over all possible values of p_d, which on our case is just one variable (since all values of p_d are all identical). This yields:

$$P(\{\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(N)}\}) = \int_0^1 \prod_{n=1}^{N} \prod_{d=1}^{D} (p_d)^{x_d^{(n)}} (1-p_d)^{1-x_d^{(n)}} \, dp_d$$

By using power rules, we can change the products into sums, which will make a known integral form appear:

$$\begin{aligned}
P(\{\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(N)}\}) &= \int_0^1 (p_d)^{\sum_{n=1}^{N} \sum_{d=1}^{D} x_d^{(n)}} (1-p_d)^{\sum_{n=1}^{N} \sum_{d=1}^{D} (1-x_d^{(n)})} \, dp_d \\
&= \int_0^1 (p_d)^{\sum_{n=1}^{N} \sum_{d=1}^{D} x_d^{(n)}} (1-p_d)^{\sum_{n=1}^{N} \sum_{d=1}^{D} - \sum_{n=1}^{N} \sum_{d=1}^{D} x_d^{(n)}} \, dp_d \\
&= \int_0^1 (p_d)^{y-1} (1-p_d)^{z-1} \, dp_d \\
&= \frac{\Gamma(\sum_{n=1}^{N} \sum_{d=1}^{D} x_d^{(n)} + 1) \cdot \Gamma(\sum_{n=1}^{N} \sum_{d=1}^{D} - \sum_{n=1}^{N} \sum_{d=1}^{D} x_d^{(n)} + 1)}{\Gamma(\sum_{n=1}^{N} \sum_{d=1}^{D} + 2)} \\
&= \frac{\Gamma(\sum_{n=1}^{N} \sum_{d=1}^{D} x_d^{(n)} + 1) \cdot \Gamma(N \cdot D - \sum_{n=1}^{N} \sum_{d=1}^{D} x_d^{(n)} + 1)}{\Gamma(N \cdot D + 2)}
\end{aligned}$$

In the final few steps of the computation above, we have used the Beta function integral property:

$$B(y, z) = \int_0^1 t^{y-1} (1-t)^{z-1} \, dt$$

as well as the following substitutions:

$$y = \sum_{n=1}^{N} \sum_{d=1}^{D} x_d^{(n)} + 1$$

$$z = \sum_{n=1}^{N} \sum_{d=1}^{D} - \sum_{n=1}^{N} \sum_{d=1}^{D} x_d^{(n)} + 1$$

2.3 Question 2c

In this model, each component is Bernoulli distributed with separate, unknown p_d. The likelihood function thus becomes:

$$P(\{\mathbf{x}^{(1)},\ldots,\mathbf{x}^{(N)}\}|\mathbf{p}^{(n)}=(p_1,p_2,p_3,\ldots,p_D)) = \prod_{n=1}^{N}\prod_{d=1}^{D}(p_d)^{x_d^{(n)}}(1-p_d)^{1-x_d^{(n)}}$$

To find the relative probability, we have to integrate over all possible values of p_d, which gives:

$$P(\{\mathbf{x}^{(1)},\ldots,\mathbf{x}^{(N)}\}) = \int_0^1\cdots\int_0^1\prod_{n=1}^{N}\prod_{d=1}^{D}(p_d)^{x_d^{(n)}}(1-p_d)^{1-x_d^{(n)}}dp_1dp_2\ldots dp_D$$

We proceed as we did previously:

$$\begin{aligned}
P(\{\mathbf{x}^{(1)},\ldots,\mathbf{x}^{(N)}\}) &= \int_0^1\cdots\int_0^1\prod_{n=1}^{N}\prod_{d=1}^{D}(p_d)^{x_d^{(n)}}(1-p_d)^{1-x_d^{(n)}}\,dp_1dp_2\ldots dp_D \\
&= \prod_{d=1}^{D}\int_0^1\prod_{n=1}^{N}(p_d)^{x_d^{(n)}}(1-p_d)^{1-x_d^{(n)}}\,dp_d \\
&= \prod_{d=1}^{D}\int_0^1(p_d)^{\sum_{n=1}^{N}x_d^{(n)}}(1-p_d)^{\sum_{n=1}^{N}(1-x_d^{(n)})}\,dp_d \\
&= \prod_{d=1}^{D}\int_0^1(p_d)^{y-1}(1-p_d)^{z-1}\,dp_d \\
&= \prod_{d=1}^{D}\frac{\Gamma(\sum_{n=1}^{N}x_d^{(n)}+1)\cdot\Gamma(\sum_{n=1}^{N}(1-x_d^{(n)})+1)}{\Gamma(\sum_{n=1}^{N}+2)}
\end{aligned}$$

Where in the second step, we have taken out the product over d so that we could evaluate each integral with one value of p_d. Then we used the trick of putting the product over n in the exponent and using the following substitutions:

$$y = \sum_{n=1}^{N}x_d^{(n)}+1$$

$$z = \sum_{n=1}^{N}-\sum_{n=1}^{N}x_d^{(n)}+1$$

to make the Beta function integral appear.

Now we can find the posterior probabilities of each of the three models having generated the data in "binarydigits.txt". Bayes rule states that:

$$P(M_i|\{\mathbf{x}^{(1)},\ldots,\mathbf{x}^{(N)}\}) = \frac{P(\{\mathbf{x}^{(1)},\ldots,\mathbf{x}^{(N)}\}|M_i)P(M_i)}{P(\{\mathbf{x}^{(1)},\ldots,\mathbf{x}^{(N)}\})}$$

The evidence can be written as a sum over our three models, and rewriting the denominator as: $P(\{\mathbf{x}^{(1)},\ldots,\mathbf{x}^{(N)}\}) = P(\{\mathbf{x}^{(1)},\ldots,\mathbf{x}^{(N)}\}|M_i) \cdot P(M_i)$, we have:

$$P(M_i|\{\mathbf{x}^{(1)},\ldots,\mathbf{x}^{(N)}\}) = \frac{P(\{\mathbf{x}^{(1)},\ldots,\mathbf{x}^{(N)}\}|M_i)}{\sum_{i=1}^{3} P(\{\mathbf{x}^{(1)},\ldots,\mathbf{x}^{(N)}\}|M_i)}$$

The posterior probabilities of each of the three models having generated the data in binary-digits.txt is registered in the tablle below:

Model	**Posterior Probability**
Model 1	9.14×10^{-255}
Model 2	1.43×10^{-188}
Model 3	1.00×10^{0}

Table 1: Posterior Probabilities for Each Model

These number (while they don't seem to add up to 1 because of numerical precision) are sensible, since models 1 and 2 would predict that all the pixels have the same probability of being black or white. For example, if all $p_d = 0.5$, this means that the colours are just random and that there is no underlying pattern to the data, which is obviously false. The third model's probability is very close to one since we assume that one of the models did generate the data, so its probability is just 1 minus the probability that models 1 and 2 generated the data.

The code used to calculate these results can be found in Listing 3.

```
import numpy as np
from scipy.special import betaln, logsumexp

# We load the binary digit data
x = np.loadtxt('binarydigits.txt')
n, d = x.shape

# We calculate the log likelihoods for each model
# Model 1: p_d = 0.5
log_pd_m1 = n * d * np.log(0.5)

# Model 2: unknown identical p_d
# We count the total number of 1s
k = np.sum(x).astype(int)
log_pd_m2 = betaln(k + 1, n * d - k + 1)

# Model 3:  separate, unknown p_d
```

```python
k_d = np.sum(x, axis=0).astype(int)
log_pd_m3 = np.sum(betaln(k_d + 1, n - k_d + 1))

# We combine log likelihoods for each model
log_pd_total = np.array([log_pd_m1, log_pd_m2, log_pd_m3])

# We calculate posterior log probabilities for each model
log_pm_data = log_pd_total - logsumexp(log_pd_total)

# We go back to linear space by taking the exponential
posterior_probs = np.exp(log_pm_data)

for i, prob in enumerate(posterior_probs, start=1):
    print(f"Model {i}: Posterior Probability = {prob:.2e}")
```

Listing 3: Python code used to compute the posterior probabilities of each model having generated the data in binarydigits.txt

3 EM for Binary Data

3.1 Question 3a

We are looking to model the binary data using a mixture of K multivariate Bernoulli distributions. The likelihood function for this type of distribution is:

$$P(\{\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(N)}\}|\pi, \mathbf{P}) = \prod_{d=1}^{D} \prod_{n=1}^{N} p_{kd}^{x_d^{(n)}} \cdot (1 - p_{kd})^{1-x_d^{(n)}}$$

where the parameters are:

1. $\pi = (\pi_1, \pi_2, \ldots, \pi_K)$, representing the mixing proportion for the k-th component. We have $0 \leq \pi_k \leq 1$ and $\sum_{k=1}^{K} \pi_k = 1$.
2. $\mathbf{P}$, which is a matrix containing the Bernoulli parameters p_{kd}. They represents the probability that a pixel d has the value 1 with a mixture component k.

For a single image, the equation for the likelihood function becomes:

$$P(\mathbf{x}^{(n)}|\pi, \mathbf{P}) = \sum_{k=1}^{K} \pi_k \prod_{d=1}^{D} p_{kd}^{x_d^{(n)}} \cdot (1 - p_{kd})^{1-x_d^{(n)}}$$

This is the likelihood function for a binary image.

3.2 Question 3b

The responsibility is defined by the following expression:

$$r_{nk} = P(s^{(n)} = k|\mathbf{x}^{(n)}, \pi, \mathbf{P})$$

Bayes theorem states that:

$$r_{nk} = \frac{P(\mathbf{x}^{(n)}|s^{(n)} = k, \pi, \mathbf{P})P(s^{(n)} = k|\pi, \mathbf{P})}{P(\mathbf{x}^{(n)}|\pi, \mathbf{P})}$$

Since the latent variable is not dependent on $\mathbf{P}$, we have: $P(s^{(n)} = k|\pi, \mathbf{P}) = P(s^{(n)} = k|\pi)$. Additionally, we are told that $P(s^{(n)} = k|\pi) = \pi_k$, which lets us write the numerator as:

$$P(\mathbf{x}^{(n)}|s^{(n)} = k, \mathbf{P}) \cdot \pi_k$$

To evaluate the other expression, we take the likelihood function for a mixture component k and a corresponding probability of choosing component k, π_k:

$$P(\mathbf{x}^{(n)}|s^{(n)} = k, \pi, \mathbf{P}) = \prod_{d=1}^{D} (p_{kd}^{x_d^{(n)}} (1 - p_{kd})^{1-x_d^{(n)}})$$

We know an expression for the denominator $P(\mathbf{x}^{(n)}|\pi, \mathbf{P})$ from question 3a:

$$P(\mathbf{x}^{(n)}|\pi, \mathbf{P}) = \sum_{j=1}^{K} \pi_j \prod_{d=1}^{D} \left(p_{jd}^{x_d^{(n)}} (1 - p_{jd})^{1-x_d^{(n)}} \right)$$

Therefore the responsibility is equal to:

$$r_{nk} = \frac{\pi_k \prod_{d=1}^{D} \left(p_{kd}^{x_d^{(n)}} (1 - p_{kd})^{1-x_d^{(n)}} \right)}{\sum_{j=1}^{K} \pi_j \prod_{d=1}^{D} \left(p_{jd}^{x_d^{(n)}} (1 - p_{jd})^{1-x_d^{(n)}} \right)}$$

3.3 Question 3c

We aim to maximize the expected log-joint probability:

$$\arg\max_{\pi, \mathbf{P}} \left\langle \sum_{n=1}^{N} \log P(\mathbf{x}^{(n)}, s^{(n)}|\pi, \mathbf{P}) \right\rangle_{q(\{s^{(n)}\})}$$

where $q(\{s^{(n)}\})$ is the distribution over the latent variables given in the previous question. Firstly, we expand the expected jog-joint distribution for each data point when $s^{(n)} = k$:

$$\log P(\mathbf{x}^{(n)}, s^{(n)} = k|\pi, \mathbf{P}) = \log \left(P(\mathbf{x}^{(n)}|s^{(n)} = k, \mathbf{P}) \pi_k \right)$$

where we used $P(s^{(n)} = k|\pi) = \pi_k$ as before. Writing out the distribution and using the log properties gives:

$$\log P(\mathbf{x}^{(n)}, s^{(n)} = k|\pi, \mathbf{P}) = \log \pi_k + \sum_{d=1}^{D} \left[x_d^{(n)} \log p_{kd} + (1 - x_d^{(n)}) \log(1 - p_{kd}) \right]$$

We now calculate the expectation value of this for all data points. This means we sum over n, and the expectation computation will be a sum over k since the latent variable is discrete. This gives:

$$\sum_{n=1}^{N} \sum_{k=1}^{K} r_{nk} \left[\log \pi_k + \sum_{d=1}^{D} \left(x_d^{(n)} \log p_{kd} + (1 - x_d^{(n)}) \log(1 - p_{kd}) \right) \right]$$

The goal now is to take the derivative with respect π and $\mathbf{P}$, set them to zero, and solve for π and $\mathbf{P}$. These will be the parameters that maximise the expectation value. We will start by maximising with respect to π. Because we are subject to the constraint $\sum_{k=1}^{K} \pi_k = 1$, we can maximise using a Lagrange multiplier λ and a Lagrange function defined as follows:

$$\mathcal{L} = \sum_{k=1}^{K} \left(\sum_{n=1}^{N} r_{nk} \right) \log \pi_k + \lambda \left(1 - \sum_{k=1}^{K} \pi_k \right)$$

We now take the derivative of $\mathcal{L}$ with respect to π_k, set it to zero and solve for π_k:

$$\frac{\partial \mathcal{L}}{\partial \pi_k} = \frac{\sum_{n=1}^{N} r_{nk}}{\pi_k} - \lambda = 0$$

$$\pi_k = \frac{\sum_{n=1}^{N} r_{nk}}{\lambda}$$

Using the constraint $\sum_{k=1}^{K} \pi_k = 1$:

$$\sum_{k=1}^{K} \pi_k = \sum_{k=1}^{K} \frac{\sum_{n=1}^{N} r_{nk}}{\lambda} = \frac{N}{\lambda} = 1$$

We find that $\lambda = N$, and the updated mixing coefficients are:

$$\pi_k = \frac{\sum_{n=1}^{N} r_{nk}}{N}$$

Now to maximise with respect to p_{kd}. Since constants won't affect where the maximum is, we only take into account terms containing p_{kd}:

$$\sum_{n=1}^{N} r_{nk} \left[x_d^{(n)} \log p_{kd} + (1 - x_d^{(n)}) \log(1 - p_{kd}) \right]$$

Like we did for π_k, we take the derivative with respect to p_{kd}, set it to zero and solve for p_{kd}:

$$
\begin{aligned}
&\frac{\partial}{\partial p_{kd}} \left(\sum_{n=1}^{N} r_{nk} \left[x_d^{(n)} \log p_{kd} + (1 - x_d^{(n)}) \log(1 - p_{kd}) \right] \right) = 0 \\
&\Rightarrow \sum_{n=1}^{N} r_{nk} \left(\frac{x_d^{(n)}}{p_{kd}} - \frac{1 - x_d^{(n)}}{1 - p_{kd}} \right) = 0 \\
&\Rightarrow \frac{1}{p_{kd}(1 - p_{kd})} \left(\sum_{n=1}^{N} r_{nk} x_d^{(n)} (1 - p_{kd}) - \sum_{n=1}^{N} r_{nk} (1 - x_d^{(n)}) p_{kd} \right) = 0 \\
&\Rightarrow \sum_{n=1}^{N} r_{nk} x_d^{(n)} (1 - p_{kd}) = \sum_{n=1}^{N} r_{nk} (1 - x_d^{(n)}) p_{kd} \\
&\Rightarrow (1 - p_{kd}) \sum_{n=1}^{N} r_{nk} x_d^{(n)} = p_{kd} \sum_{n=1}^{N} r_{nk} (1 - x_d^{(n)}) \\
&\Rightarrow \left(\sum_{n=1}^{N} r_{nk} x_d^{(n)} \right) = p_{kd} \left(\sum_{n=1}^{N} r_{nk} x_d^{(n)} + \sum_{n=1}^{N} r_{nk} (1 - x_d^{(n)}) \right) \\
&\Rightarrow \sum_{n=1}^{N} r_{nk} x_d^{(n)} = p_{kd} \sum_{n=1}^{N} r_{nk}
\end{aligned}
$$

Finally, solving for p_{kd}:

$$
p_{kd} = \frac{\sum_{n=1}^{N} r_{nk} x_d^{(n)}}{\sum_{n=1}^{N} r_{nk}}
$$

3.4 Question 3d

As suggested by the hint, I've introduced weak priors on $\mathbf{P}$ and π. I've chosen a uniform Dirichlet distribution for the prior of π, meaning all components are assumed to be equally likely before observing any data. I've chosen a Beta distribution $\mathbf{P}$, with both α and β parameters set to 1 (making it a uniform distribution) to indicate no prior knowledge. I've implemented the EM algorithm for a mixture of K multivariate Bernoullis and run it for $K = \{2, 3, 4, 7, 10\}$. The plots of the log posteriors as a function of the iteration number for each value of K are shown below.

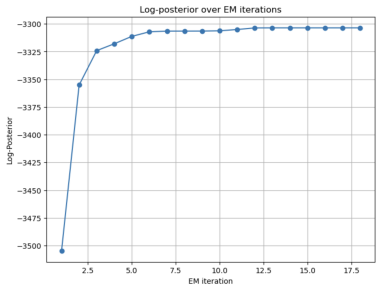


Figure 4: K = 2, Convergence step = 17

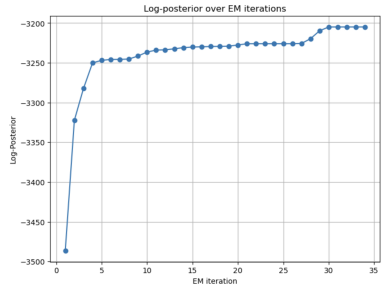


Figure 5: K = 3, Convergence step = 33

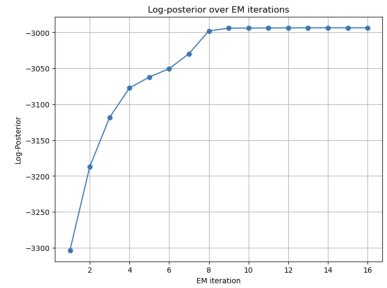


Figure 6: K = 4, Convergence step = 15

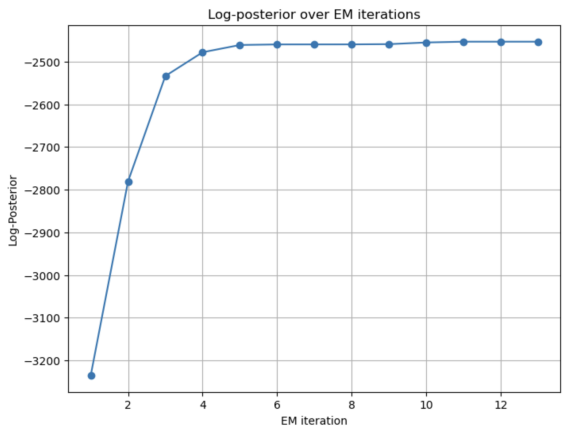


Figure 7: K = 7, Convergence step = 12

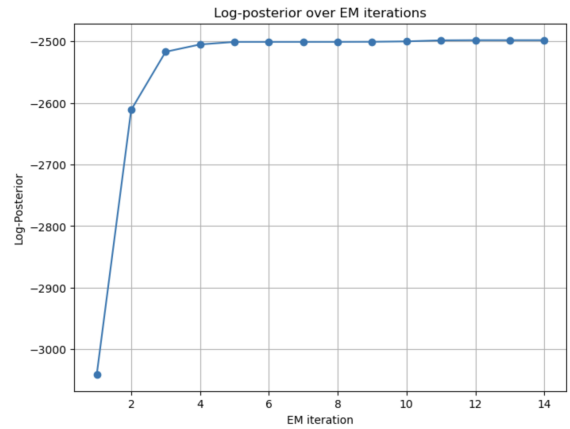


Figure 8: K = 10, Convergence step = 13

The parameters π and $\mathbf{P}$ are shown in tables 2 and 3 respectively:

K	**Values of π_k**
2	0.5804, 0.4196
3	0.2700, 0.1300, 0.6000
4	0.4402, 0.2810, 0.1090, 0.1698
7	0.1600, 0.0900, 0.1600, 0.2300, 0.0500, 0.0600, 0.2500
10	0.1400, 0.0500, 0.1800, 0.1600, 0.0300, 0.0300, 0.0300, 0.0800, 0.2000, 0.1000

Table 2: Table showing the values of π_k for different K

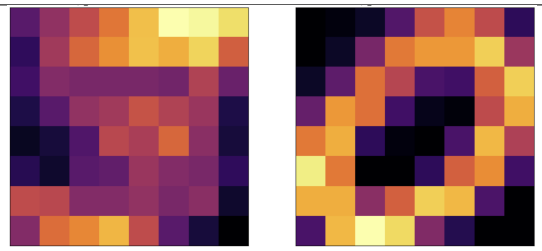
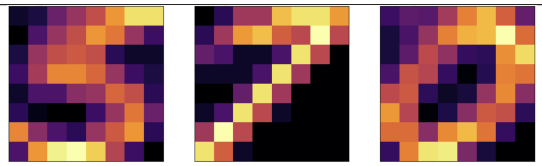
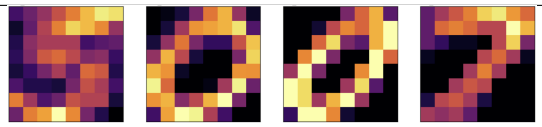
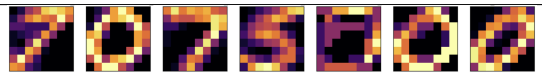
K	Optimised Bernoulli parameters
2	
3	
4	
7	
10	

Table 3: Table showing the optimised Bernoulli parameters for different K values

The code used to run the EM algorithm and plot the graphs can be found in Listing 4, and the code used to display the parameters can be found in Listing 5:

```
import torch
import torch.nn.functional as F
from torch.distributions import Dirichlet
from typing import List, Tuple
from scipy.special import betaln
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np

def init_params(k: int, d: int, epsilon: float = 1e-6):
    """
    This function randomly initialises the model parameters (log_pi and log_p_matrix)

    INPUT:
    k       : Number of mixture components
    d       : Dimensionality of the data (number of features)
    epsilon : Small constant used to avoid numerical instability


    OUTPUT:
    log_pi       : Log mixing proportions, Shape: (1, k)
    log_p_matrix : Log probability of the Bernoulli parameters, Shape: (d, k)
    """
    # We initialise the mixing proportions using a Dirichlet distribution
```

```python
    # The .unsqueeze method adds a dimension to the pi vector (goes from shape (k) to (1, k))
    pi = Dirichlet(torch.ones(k)).sample().unsqueeze(0)
    log_pi = torch.log(pi)

    # We initialize the Bernoulli parameters
    # I've encountered numerical errors due to taking the log of 0 later on in the code
    # The clamping with the epsilon variable is here to avoid that
    p_initial = torch.rand(d, k)
    p_initial = torch.clamp(p_initial, min=epsilon, max=1 - epsilon)
    log_p_matrix = torch.log(p_initial)

    return log_pi, log_p_matrix

def compute_log_one_minus_p_matrix(log_p_matrix: torch.Tensor, epsilon: float = 1e-6) -> torch.Tensor:
    """
    This function computes log(1 - p) from log(p)

    INPUT:
    log_p_matrix     : Log probabilities of p, Shape: (d, k)
    epsilon          : Small constant used to avoid numerical instability


    OUTPUT:
    log_one_minus_p_matrix: Log probabilities of (1 - p), Shape: (d, k)
    """
    # The espilon is just here to avoid numerical instability (taking the log of 0)
    log_one_minus_p_matrix = torch.log1p(-torch.exp(log_p_matrix) + epsilon)
    return log_one_minus_p_matrix

def compute_log_pi_repeated(log_pi: torch.Tensor, n: int) -> torch.Tensor:
    """
    This function repeats log_pi n times along the first dimension

    INPUT:
    log_pi   : Log mixing proportions, Shape: (1, k)
    n        : Number of repetitions

    OUTPUT:
    Repeated log_pi, Shape: (n, k)
    """
    return log_pi.expand(n, -1)

def compute_log_e_step(x: torch.Tensor, log_p_matrix: torch.Tensor, log_one_minus_p_matrix: torch.Tensor, log_pi: torch.Tensor) -> torch.Tensor:
    """
    This function computes the log responsibilities for each data point and mixture component

    INPUT:
    x                       : Data matrix, Shape: (n, d)
    log_p_matrix            : Log probabilities of Bernoulli parameters, Shape: (d, k)
    log_one_minus_p_matrix  : Log probabilities of (1 - p), Shape: (d, k)
    log_pi                  : Log mixing proportions, Shape: (1, k)

    OUTPUT:
    log_r_responsibilities  : Log responsibilities (shape: (n, k)
    """
```

```python
    # We compute the log-probability of each data point belonging to each component
    log_component_p = x @ log_p_matrix + (1 - x) @ log_one_minus_p_matrix

    # We add the log of mixing proportions to get the unnormalized log-responsibilities
    log_r_unnormalised = log_component_p + log_pi

    # We normalise the log-responsibilities across components to ensure they sum to 1
    log_r_normaliser = torch.logsumexp(log_r_unnormalised, dim=1, keepdim=True)

    # We subtract the normaliser to get the final log responsibilities
    log_r_responsibilities = log_r_unnormalised - log_r_normaliser
    return log_r_responsibilities

def compute_log_pi_hat(log_responsibility: torch.Tensor) -> torch.Tensor:
    """
    This function updates the log mixing proportions.

    INPUT:
    log_responsibility: Log responsibilities, Shape: (n, k)

    OUTPUT:
    log_pi_hat = Updated log_pi, Shape: (1, k)
    """
    # We compute the sum of the log-responsibilities across all data points for each component
    #   and normalise the log mixing proportions by subtracting log(n)
    log_pi_hat = torch.logsumexp(log_responsibility, dim=0, keepdim=True) - torch.log(torch.tensor([n], dtype=log_responsibility.dtype))
    return log_pi_hat

def compute_log_p_matrix_hat(x: torch.Tensor, log_responsibility: torch.Tensor, alpha_parameter: float, beta_parameter: float, epsilon: float = 1e-6,) -> torch.Tensor:
    """
    This function updates the log Bernoulli parameters.

    INPUT:
    x                   : Data matrix, Shape: (n, d)
    log_responsibility  : Log responsibilities, Shape: (n, k)
    alpha_parameter     : Alpha parameter of the Beta prior
    beta_parameter      : Beta parameter of the Beta prior
    epsilon             : Small constant used to avoid numerical instability

    OUTPUT:
    log_p_matrix_hat = Updated log_p_matrix, Shape: (d, k)
    """
    # We go from log space back to linear space
    responsibility = torch.exp(log_responsibility)

    # We compute the numerator
    numerator = x.T @ responsibility + (alpha_parameter - 1)

    # We compute teh denominator
    denominator = torch.sum(responsibility, dim=0) + (alpha_parameter + beta_parameter - 2)
    denominator = denominator.unsqueeze(0)

    # we compute the updated probability matrix
    p_matrix_hat = numerator / denominator
    # Again, we use espilon to prevent taking log(0)
    p_matrix_hat = torch.clamp(p_matrix_hat, min=epsilon, max=1 - epsilon)
```

```python

    log_p_matrix_hat = torch.log(p_matrix_hat)
    return log_p_matrix_hat

def compute_log_likelihood(x: torch.Tensor, log_p_matrix: torch.Tensor, log_one_minus_p_matrix: torch.Tensor, log_pi: torch.Tensor) -> float:
    """
    This function computes the total log-likelihood of the data given the model parameters

    INPUT:
    x                       : Data matrix, Shape: (n, d)
    log_p_matrix            : Log probabilities of p, Shape: (d, k)
    log_one_minus_p_matrix  : Log probabilities of (1 - p), Shape: (d, k)
    log_pi:                  Log mixing proportions, Shape: (1, k)

    OUTPUT:
    log_likelihood          : Total log-likelihood, Shape: (scalar)
    """
    # We calculate the log probabilities
    log_component_probabilities = x @ log_p_matrix + (1 - x) @ log_one_minus_p_matrix
    log_pi_repeated = compute_log_pi_repeated(log_pi, n)
    log_probabilities = log_component_probabilities + log_pi_repeated

    # We compute the log-likelihood for each data point by summing over components
    log_likelihood = torch.sum(torch.logsumexp(log_probabilities, dim=1))
    return log_likelihood.item()

def compute_log_prior(log_p_matrix: torch.Tensor, log_one_minus_p_matrix: torch.Tensor, alpha_parameter: float, beta_parameter: float) -> float:
    """
    This function omputes the log-prior probability of the model parameters

    INPUT:
    log_p_matrix            : Log probabilities of p, Shape: (d, k)
    log_one_minus_p_matrix  : Log probabilities of (1 - p), Shape: (d, k)
    alpha_parameter         : Alpha parameter of the Beta prior
    beta_parameter          : Beta parameter of the Beta prior

    OUTPUT:
    log_prior               : Total log-prior, Shape: (scalar)
    """
    # We find the log of the beta function normalisation constant
    beta_ln = betaln(alpha_parameter, beta_parameter)
    # We calculate the log-prior
    log_prior = beta_ln * log_p_matrix.numel() + torch.sum((alpha_parameter - 1) * log_p_matrix + (beta_parameter - 1) * log_one_minus_p_matrix)
    return log_prior.item()

def compute_unnormalised_log_posterior_likelihood(x: torch.Tensor, log_p_matrix: torch.Tensor, log_one_minus_p_matrix: torch.Tensor, log_pi: torch.Tensor, alpha_parameter: float, beta_parameter: float) -> float:
    """
    This function computes the unnormalised log-posterior probability of the model parameters

    INPUT:
    x                       : Data matrix, Shape: (n, d))
    log_p_matrix            : Log probabilities of p, Shape: (d, k))
    log_one_minus_p_matrix  : Log probabilities of (1 - p), Shape: (d, k))
```

```python
    log_pi                    : Log mixing proportions, Shape: (1, k))
    alpha_parameter           : Alpha parameter of the Beta prior
    beta_parameter            : Beta parameter of the Beta prior

    OUTPUT:
    log_posterior             : Unnormalized log-posterior, Shape: (scalar)
    """
    log_likelihood = compute_log_likelihood(x, log_p_matrix, log_one_minus_p_matrix, log_pi)
    log_prior = compute_log_prior(log_p_matrix, log_one_minus_p_matrix, alpha_parameter, beta_parameter)
    log_posterior = log_likelihood + log_prior
    return log_posterior

def run_expectation_maximisation(x: torch.Tensor, log_pi: torch.Tensor, log_p_matrix: torch.Tensor, alpha_parameter: float, beta_parameter: float, max_number_of_steps: int, epsilon: float) -> Tuple[torch.Tensor, torch.Tensor, torch.Tensor, List[float]]:
    """
    This is the main function that runs the EM algorithm

    INPUT:
    x                    : Data matrix, Shape: (n, d))
    log_pi               : Initial log mixing proportions, Shape: (1, k))
    log_p_matrix         : Initial log Bernoulli parameters, Shape: (d, k))
    alpha_parameter      : Alpha parameter of the Beta prior
    beta_parameter       : Beta parameter of the Beta prior
    max_number_of_steps  : Maximum number of EM iterations
    epsilon              : Small constant representing the convergence threshold

    OUTPUT:
    Updated log_pi, log_p_matrix, log_responsibility, and log_posteriors over iterations
    """
    log_posteriors = []
    for step in range(max_number_of_steps):
        log_one_minus_p_matrix = compute_log_one_minus_p_matrix(log_p_matrix)

        # We perform the E-step
        log_responsibility = compute_log_e_step(x, log_p_matrix, log_one_minus_p_matrix, log_pi)

        # We perform the M-step
        log_pi = compute_log_pi_hat(log_responsibility)
        log_p_matrix = compute_log_p_matrix_hat(x, log_responsibility, alpha_parameter, beta_parameter)

        # Recompute log_one_minus_p_matrix after updating log_p_matrix
        log_one_minus_p_matrix = compute_log_one_minus_p_matrix(log_p_matrix)

        # Compute log-posterior using the UPDATED log_one_minus_p_matrix
        log_posterior = compute_unnormalised_log_posterior_likelihood(
            x, log_p_matrix, log_one_minus_p_matrix, log_pi, alpha_parameter, beta_parameter
        )
        log_posteriors.append(log_posterior)

        # As required, we use another constant epsilon to check for convergence and end the loop early
        if step > 0 and abs(log_posteriors[-1] - log_posteriors[-2]) < epsilon:
            print(f"Converged at iteration {step}")
            break

```

```python
    return log_pi, log_p_matrix, log_responsibility, log_posteriors

def plot_log_posteriors(log_posteriors: List[float], title: str = "Log-posterior over EM iterations") -> None:
    """
    This function plots the log-posteriors over EM iterations

    INPUT:
    log_posteriors  : List of log-posterior values.
    title           : Title of the plot.

    OUTPUT:
    Plot of the log-posterior over EM iterations
    """
    plt.figure(figsize=(8, 6))
    plt.plot(range(1, len(log_posteriors) + 1), log_posteriors, marker='o')
    plt.title(title)
    plt.xlabel('EM iteration')
    plt.ylabel('Log-Posterior')
    plt.grid(True)
    plt.show()

# We choose the inputs to the function
k = 3
max_number_of_steps = 100
data_np = np.loadtxt('binarydigits.txt')
x = torch.tensor(data_np, dtype=torch.float32)
n, d = x.shape
epsilon = 1e-4

# We initialise the parameters
log_pi, log_p_matrix = init_params(k, d)

# We set the prior parameters for the Beta distribution to 1 (this choice is exlpained in the submitted pdf file)
alpha_parameter = 1.0
beta_parameter = 1.0

# We run EM algorithm
log_pi, log_p_matrix, log_responsibility, log_posteriors = run_expectation_maximisation(
    x,
    log_pi,
    log_p_matrix,
    alpha_parameter,
    beta_parameter,
    max_number_of_steps,
    epsilon,
)

plot_log_posteriors(log_posteriors)

### Code to display the Log-posterior against the EM iteration for different values of K and the parameters ###

# We initialise the values of K to test and display
K_values = [2, 3, 4, 7, 10]

# Store log-likelihoods and parameters for each K
log_likelihoods_per_k = {}
```

```python
parameters_per_k = {}

# Run the EM algorithm for each K
for k in K_values:
    print(f"K = {k}")
    log_pi, log_p_matrix = init_params(k, d)

    # We run the EM algorithm
    log_pi, log_p_matrix, log_responsibility, log_posteriors = run_expectation_maximisation(x, log_pi, log_p_matrix, alpha_parameter, beta_parameter, max_number_of_steps, epsilon=epsilon)

    # Store the log-likelihoods and parameters
    log_likelihoods_per_k[k] = log_posteriors
    parameters_per_k[k] = {'log_pi': log_pi.detach(), 'log_p_matrix': log_p_matrix.detach()}

    # Plot the log-likelihood over iterations
    plot_log_posteriors(log_posteriors)

    # Display the parameters found
    print(f"Parameters for K = {k}:")
    print("Mixing proportions (pi):")
    print(torch.exp(log_pi))
    print("Bernoulli parameters (p_matrix):")
    print(torch.exp(log_p_matrix))
```

Listing 4: Python code running the EM algorithm for a mixture of K multivariate Bernoullis

```python
def visualise_p_matrices(log_p_matrices: List[np.ndarray], log_pis: List[np.ndarray], ks: List[int], figure_title: str, figure_path: str) -> None:
    """
    This function lets us visualise the P matrices contained in the parameters_per_k variable

    INPUT:
    log_p_matrices  : List of log probability matrice, Shape: (d, k)
    log_pis         : List of log mixing proportions, each array has shape (1, k)
    ks              : List of k values for each P matrix (2, 3, 4, 7 10)
    figure_title    : Title for the figures
    figure_path     : File path to save the figures

    OUTPUT:
    The visual representation of the P matrices
    """
    # We find the number of P matrices to show and the maximum k value
    n = len(ks)
    m = np.max(ks)

    # We set up the plot dimensions
    fig = plt.figure()
    fig.set_figwidth(15)
    fig.set_figheight(10)

    for i, (log_p_matrix, log_pi, k) in enumerate(zip(log_p_matrices, log_pis, ks)):
        # We calculate P matrix and mixing proportions
        p_matrix = np.exp(log_p_matrix).reshape(8, 8, k)
        pi_values = np.exp(log_pi)

        for j in range(k):
```

```
            ax = plt.subplot(n, m, m * i + j + 1)
            ax.imshow(p_matrix[:, :, j], cmap="inferno", interpolation="none")
            ax.tick_params(axis="both", which="both", bottom=False, left=False)
            ax.xaxis.set_ticklabels([])
            ax.yaxis.set_ticklabels([])
            ax.set_title(f"pi_{j}: {np.round(pi_values[0, j], 2)}")
            if j == 0:
                ax.set_ylabel(f"k={k}")

    fig.suptitle(figure_title)
    plt.savefig(figure_path)
    plt.show()

ks = [2, 3, 4, 7, 10]

# We display as 8x8 images the Bernoulli parameters
for i in ks:
    log_pi = np.array(parameters_per_k[i]['log_pi'])
    log_p_matrix = np.array(parameters_per_k[i]['log_p_matrix'])
    ks = [i]
    log_p_matrices = [log_p_matrix]
    log_pis = [log_pi]
    figure_title = "test"
    figure_path = f"path_{i}.png"
    visualise_p_matrices(log_p_matrices, log_pis, ks, figure_title, figure_path)

ks = [2, 3, 4, 7, 10]
# We display the pi_k in the linear space
for i in ks:
    print(torch.exp(parameters_per_k[i]['log_pi']))
```

Listing 5: Python code used to display the parameters

3.5 Question 3e

I ran the algorithm 3 times (test runs 1, 2 and 3) with randomly chosen initial conditions, and the resulting parameters are shown below in tables 4 to 7.

K	π_k **on run 1**	π_k **on run 2**	π_k **on run 3**
2	0.5804, 0.4196	0.4099, 0.5901	0.7600, 0.2400
3	0.3399, 0.2200, 0.4401	0.4002, 0.2998, 0.2999	0.6900, 0.1800, 0.1300
4	0.1500, 0.4103, 0.1300, 0.3097	0.1615, 0.3193, 0.2200, 0.2992	0.1000, 0.3700, 0.2900, 0.2400
7	0.2200, 0.1600, 0.0800, 0.0300, 0.1400, 0.2500, 0.1200	0.0501, 0.1600, 0.1299, 0.2600, 0.1500, 0.0300, 0.2200	0.0200, 0.0700, 0.2600, 0.1806, 0.1100, 0.3100, 0.0494
10	0.0100, 0.0700, 0.2001, 0.1190, 0.1600, 0.2099, 0.0300, 0.0100, 0.1510, 0.0400	0.0700, 0.0400, 0.0900, 0.1685, 0.0900, 0.1800, 0.0900, 0.0800, 0.1109, 0.0806	0.1200, 0.0300, 0.0200, 0.0600, 0.1300, 0.0400, 0.0600, 0.2700, 0.1100, 0.1600

Table 4: Table showing the values of π_k for different K on the 3 test runs

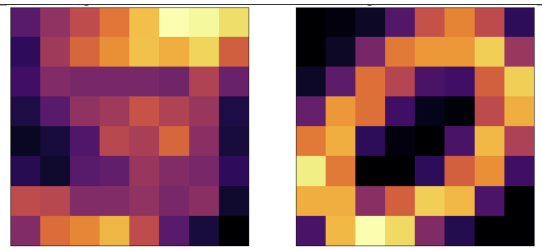
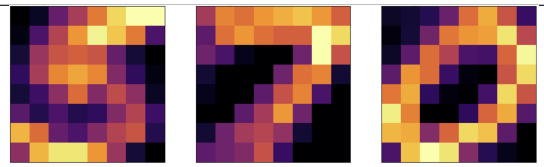
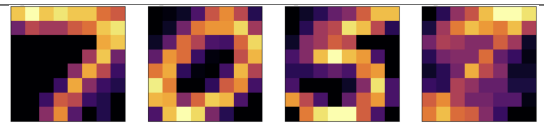
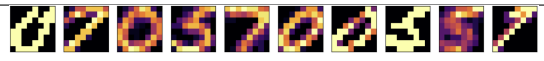
K	**Optimised Bernoulli parameters**
2	
3	
4	
7	
10	

Table 5: Table showing the optimised Bernoulli parameters for the first test run

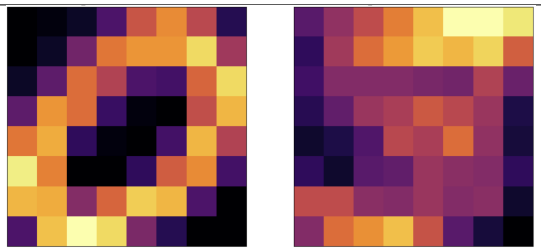
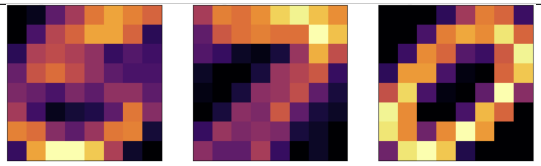
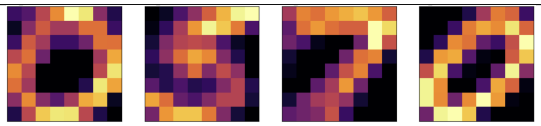
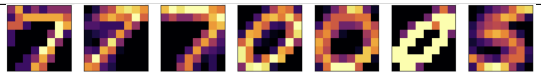
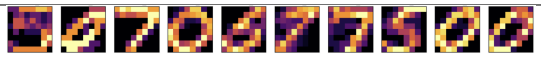
K	Optimised Bernoulli parameters
2	
3	
4	
7	
10	

Table 6: Table showing the optimised Bernoulli parameters for the second test run

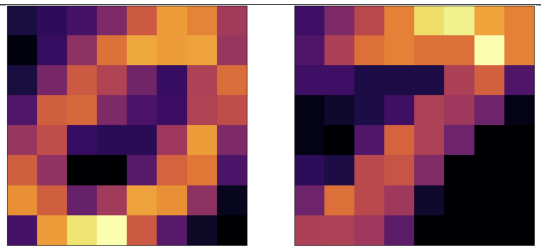
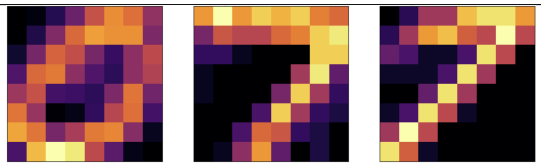
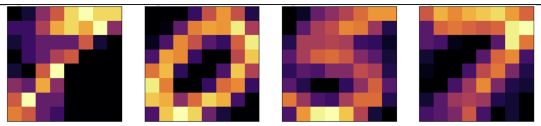
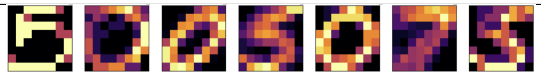
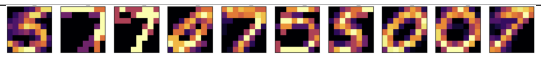
K	Optimised Bernoulli parameters
2	
3	
4	
7	
10	

Table 7: Table showing the optimised Bernoulli parameters for the third test run

Looking at the results above, we see that there is a similarity to the solutions found. For example, in the solution obtained in question 3d and the first two test runs, we find the same patterns. When $K = 3$, all three runs show the numbers $5, 7, 0$ in the same order. A similar pattern can be found for $K = 2$, where the numbers $5, 0$ appear every time, up to a permutation since test run 2 shows $0, 5$ instead. We see that as the value of K gets higher, we lose this similarity. For example, comparing the Bernoulli parameters for test runs 1 and 3 when $K = 10$, they have different numbers of $0, 5$ and 7s and the numbers are in very different orders. The similarity in the solutions does seem to depend on K.

The algorithm works well in general. From tables 5 to 7, we see that it correctly finds clusters of the numbers 0, 5 and 7. However, its performance is very dependent on the value of K. If we look at the $K = 2$ case of our test runs, we see that the algorithm finds two clusters, one which is clearly identifiable as the 0, and the other seems to be a mix of the 5 and 7, resulting in the image seeming blurry.This makes sense, as the three numbers would require one cluster each, so we would need at least $K = 3$. Now looking at the other end of the spectrum, when $K = 10$ we see that some of the images look to be only 2 colours. This is because the probability for this cluster is really small (e.g. $\pi_1 = 0.01$ for the test run 1), which results in loosely defined clusters. The probability is must better distributed among the clusters for $K = 3$, where each π_k is closer to 30%. With this in mind, one way to improve the model is to change our belief on what the ideal number of clusters is, which we've seen empirically is at least 3. If we could incorporate prior knowledge of the data into the Dirichlet and Beta distributions we defined (such as the fact that we don't like clusters with very low probability), we could improve the model.

3.6 Bonus: Question 3f

Our log likelihood are calculated using the natural logarithm, which is of base e. To encode the log-likelihood in bits, we need to convert from (base e) to base 2, which we can do this by dividing the log-likelihood by log(2):

$$\text{Log-likelihood in bits} = \frac{\text{log-likelihood}}{\log(2)}$$

The naive way to encode the binary data is to use the 0s and 1s that correspond to the pixels as the encoding. Each binary image has D pixel, so for a dataset with N binary images the length of the encoding would be:

$$\text{Naïve encoding length} = N \times D \text{ bits}$$

Gzip is a compression algorithm that finds patterns in data to reduce its storage size. If the log likelihood in bits is lower than the gzip encoding length, this shows that the EM algorithm capture the structure of the data well and is a better compression algorithm. If gzip works better than the EM model, it could be because the EM algorithm doesn't capture all the patterns contained the binarydigits.txt.

A difference between the two could come about because of the difference in approach by both methods. The EM model tries to understand the underlying probability distribution, while gzip compresses regardless of the probabilistic background. This can make for better compression if the model that we've made can't represent all the patterns. This would be the case for low values of K. For example, if we compare the $K = 2$ and $K = 4$ Bernoulli parameters for the second test run (in table 6), we see that the $K = 4$ parameters capture the difference between a round 0 and a slanted one, which we don't see in $K = 2$.

3.7 Bonus: Question 3g

We must consider both the costs of the data and the parameters, which makes the total cost:

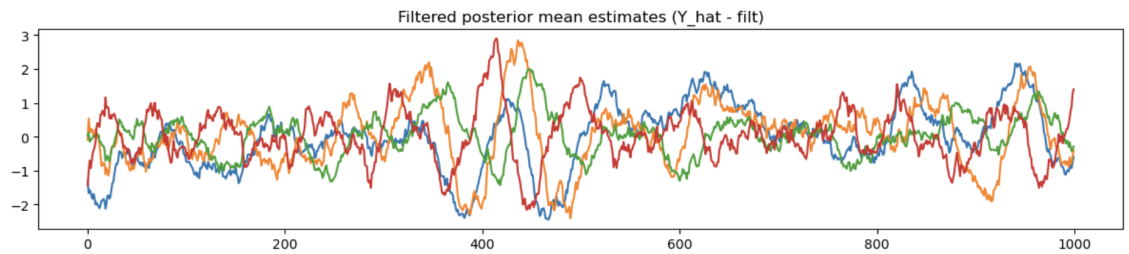
$$\text{Total cost} = \text{Cost of the model parameters} + \text{Cost of the data given a certain model}$$

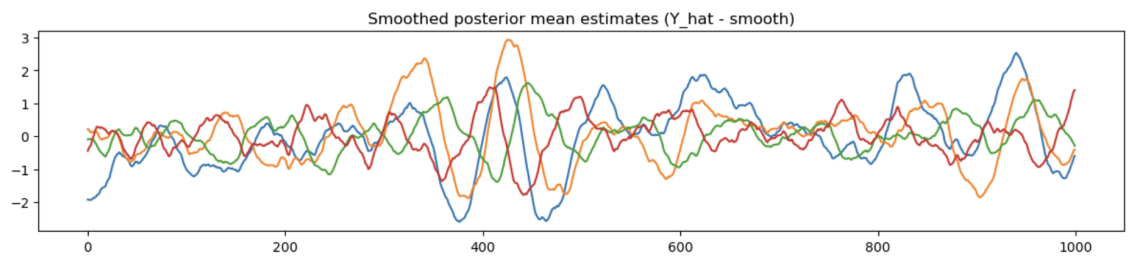
This sets our model apart from the gzip algorithm, which doesn't need to take into account parameter cost since it compresses regardless of the probabilistic background. As K increases, our model becomes more complex and requires more parameters, which would increase the total cost of storing the model. However, if K is small the model might not understand all patterns in the data as discussed in the previous question, which would result in a higher cost of the data. Therefore they must be an optimal K that minimises the total cost.

4 Bonus: LGSSMs, EM and SSID

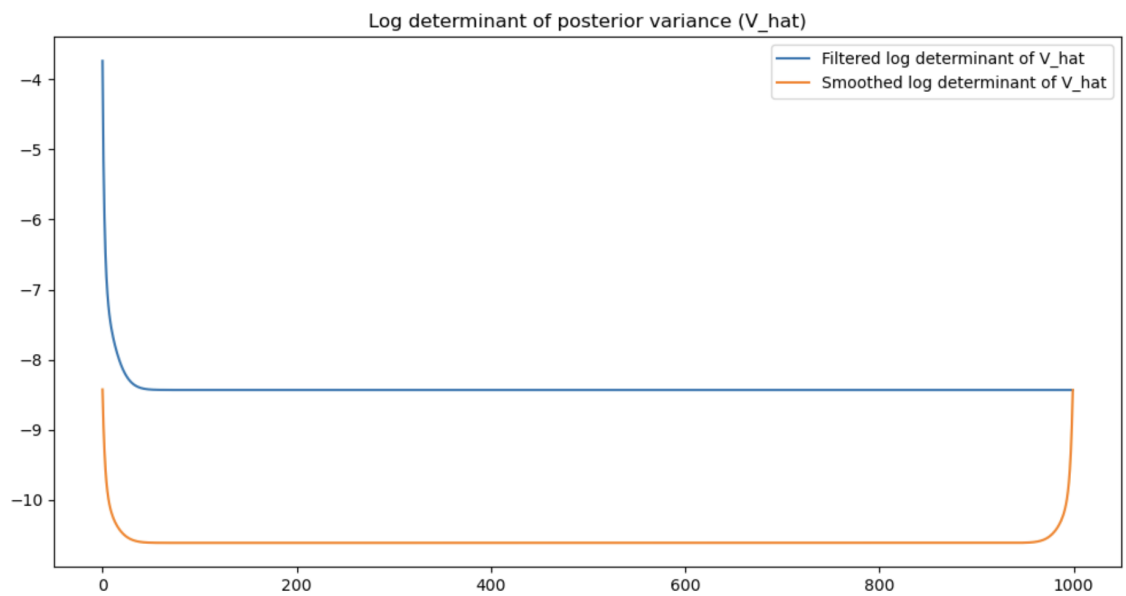
4.1 Question 4a

The figures below show the Kalman filter and Kalman smoother respectively.





The behaviour of Y on both plots is similar, as we can see that both plots have similar shapes, however they differ in their uncertainty. This is linked to how both were computed: the filtered estimates were generated using only the data up to each time step. Each estimate depends only on the information available up to that time, resulting in higher uncertainty. On the other hand, the smoothed estimates uses data from the entire time series, which makes more accurate state estimates. The log determinant of posterior variances for the Kalman filter and smoother were also calculated and displayed in the figure below.



The log determinant of the posterior variance is a measure of the uncertainty in the state estimates. We see from the figure above that the log determinant for the filtered estimates is higher than the one for the smoothed estimates, which is what we would expect. They have similar shapes near the start of the time series as the algorithm has little information to work with. The Kalman filter only has the initial state estimate and the initial observation, and the Kalman smoother starts lower because it has access to future observations. The two plots differ at the end, where the Kalman smoother uncertainty increases as it can no longer look at the future observations, which makes it lose confidence in its estimates, and join the Kalman filter (as both can only see the past observations now). The code used to to compute and display the figures in this question can be found in Listing 6:

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

def run_ssm_kalman(X, y_init, Q_init, A, Q, C, R, mode='smooth'):
    """
    Calculates kalman-smoother estimates of SSM state posterior.
    :param X:        data, [d, t_max] numpy array
    :param y_init:   initial latent state, [k,] numpy array
    :param Q_init:   initial variance, [k, k] numpy array
    :param A:        latent dynamics matrix, [k, k] numpy array
    :param Q:        innovariations covariance matrix, [k, k] numpy array
    :param C:        output loading matrix, [d, k] numpy array
    :param R:        output noise matrix, [d, d] numpy array
    :param mode:     'forw' or 'filt' for forward filtering, 'smooth' for also backward
        filtering
    :return:
    y_hat:       posterior mean estimates, [k, t_max] numpy array
    V_hat:       posterior variances on y_t, [t_max, k, k] numpy array
    V_joint:     posterior covariances between y_{t+1}, y_t, [t_max, k, k] numpy array
    likelihood: conditional log-likelihoods log(p(x_t|x_{1:t-1})), [t_max,] numpy array
```

```python
    """
    d, k = C.shape
    t_max = X.shape[1]

    # dimension checks
    assert np.all(X.shape == (d, t_max)), "Shape of X must be (%d, %d), %s provided" % (d, t_max, X.shape)
    assert np.all(y_init.shape == (k,)), "Shape of y_init must be (%d,), %s provided" % (k, y_init.shape)
    assert np.all(Q_init.shape == (k, k)), "Shape of Q_init must be (%d, %d), %s provided" % (k, k, Q_init.shape)
    assert np.all(A.shape == (k, k)), "Shape of A must be (%d, %d), %s provided" % (k, k, A.shape)
    assert np.all(Q.shape == (k, k)), "Shape of Q must be (%d, %d), %s provided" % (k, k, Q.shape)
    assert np.all(C.shape == (d, k)), "Shape of C must be (%d, %d), %s provided" % (d, k, C.shape)
    assert np.all(R.shape == (d, d)), "Shape of R must be (%d, %d), %s provided" % (d, k, R.shape)

    y_filt = np.zeros((k, t_max))  # filtering estimate: \hat(y)_t^t
    V_filt = np.zeros((t_max, k, k))  # filtering variance: \hat(V)_t^t
    y_hat = np.zeros((k, t_max))  # smoothing estimate: \hat(y)_t^T
    V_hat = np.zeros((t_max, k, k))  # smoothing variance: \hat(V)_t^T
    K = np.zeros((t_max, k, X.shape[0]))  # Kalman gain
    J = np.zeros((t_max, k, k))  # smoothing gain
    likelihood = np.zeros(t_max)  # conditional log-likelihood: p(x_t|x_{1:t-1})

    I_k = np.eye(k)

    # forward pass

    V_pred = Q_init
    y_pred = y_init

    for t in range(t_max):
        x_pred_err = X[:, t] - C.dot(y_pred)
        V_x_pred = C.dot(V_pred.dot(C.T)) + R
        V_x_pred_inv = np.linalg.inv(V_x_pred)
        likelihood[t] = -0.5 * (np.linalg.slogdet(2 * np.pi * (V_x_pred))[1] +
                                x_pred_err.T.dot(V_x_pred_inv).dot(x_pred_err))

        K[t] = V_pred.dot(C.T).dot(V_x_pred_inv)

        y_filt[:, t] = y_pred + K[t].dot(x_pred_err)
        V_filt[t] = V_pred - K[t].dot(C).dot(V_pred)

        # symmetrise the variance to avoid numerical drift
        V_filt[t] = (V_filt[t] + V_filt[t].T) / 2.0

        y_pred = A.dot(y_filt[:, t])
        V_pred = A.dot(V_filt[t]).dot(A.T) + Q

    # backward pass

    if mode == 'filt' or mode == 'forw':
        # skip if filtering/forward pass only
        y_hat = y_filt
        V_hat = V_filt
```

```python
        V_joint = None
    else:
        V_joint = np.zeros_like(V_filt)
        y_hat[:, -1] = y_filt[:, -1]
        V_hat[-1] = V_filt[-1]

        for t in range(t_max - 2, -1, -1):
            J[t] = V_filt[t].dot(A.T).dot(np.linalg.inv(A.dot(V_filt[t]).dot(A.T) + Q))
            y_hat[:, t] = y_filt[:, t] + J[t].dot((y_hat[:, t + 1] - A.dot(y_filt[:, t])))
            V_hat[t] = V_filt[t] + J[t].dot(V_hat[t + 1] - A.dot(V_filt[t]).dot(A.T) - Q).dot(J[t].T)

        V_joint[-2] = (I_k - K[-1].dot(C)).dot(A).dot(V_filt[-2])

        for t in range(t_max - 3, -1, -1):
            V_joint[t] = V_filt[t + 1].dot(J[t].T) + J[t + 1].dot(V_joint[t + 1] - A.dot(V_filt[t + 1])).dot(J[t].T)

    return y_hat, V_hat, V_joint, likelihood

# We start by defining the matrices as given in the question
A = 0.99 * np.array([
    [np.cos(2 * np.pi / 180), -np.sin(2 * np.pi / 180), 0, 0],
    [np.sin(2 * np.pi / 180), np.cos(2 * np.pi / 180), 0, 0],
    [0, 0, np.cos(2 * np.pi / 90), -np.sin(2 * np.pi / 90)],
    [0, 0, np.sin(2 * np.pi / 90), np.cos(2 * np.pi / 90)]
])

Q = np.eye(4) - A.dot(A.T)
C = np.array([
    [1, 0, 1, 0],
    [0, 1, 0, 1],
    [1, 0, 0, 1],
    [0, 1, 0, 1],
    [0.5, 0.5, 0.5, 0.5]
])
R = np.eye(5)

# We load data and transpose X such that we have Shape: (d, T)
X = np.loadtxt('ssm_spins.txt').T

# We initialise the state and covariance
y_init = np.zeros(4)
Q_init = np.eye(4)

# We call the function contained in the ssm_kalman.py for filtering
y_hat_filt, V_hat_filt, V_joint_filt, likelihood_filt = run_ssm_kalman(X, y_init, Q_init, A, Q, C, R, mode='filt')

# We call the function contained in the ssm_kalman.py for smoothing
y_hat_smooth, V_hat_smooth, V_joint_smooth, likelihood_smooth = run_ssm_kalman(X, y_init, Q_init, A, Q, C, R, mode='smooth')

# We plotting y_hat and estimates for filtering and smoothing
plt.figure(figsize=(12, 8))
plt.subplot(3, 1, 2)
plt.plot(y_hat_filt.T)
plt.title("Filtered posterior mean estimates (Y_hat - filt)")

```

```python
plt.subplot(3, 1, 3)
plt.plot(y_hat_smooth.T)
plt.title("Smoothed posterior mean estimates (Y_hat - smooth)")
plt.tight_layout()
plt.show()

# We plot log determinant of V_hat to show the uncertainty
logdet_filt = [np.linalg.slogdet(V)[1] for V in V_hat_filt]
logdet_smooth = [np.linalg.slogdet(V)[1] for V in V_hat_smooth]

plt.figure(figsize=(12, 6))
plt.plot(logdet_filt, label='Filtered log determinant of V_hat')
plt.plot(logdet_smooth, label='Smoothed log determinant of V_hat')
plt.title("Log determinant of posterior variance (V_hat)")
plt.legend()
plt.show()
```

Listing 6: Python code used to answer question 4a

5 Decrypting Messages with MCMC

5.1 Question 5a

The formula for the ML estimates of the transition probabilities $p(s_i = \alpha \mid s_{i-1} = \beta) \equiv \psi(\alpha, \beta)$ are simply the number of times the pair (α, β) occurs over the total number occurrences of β:

$$\psi(\alpha, \beta) = \frac{N_{(\alpha,\beta)}}{N_{\beta,total}}$$

Below is the table showing the transition probabilities $\psi(\alpha, \beta)$ for all the characters. The table is too big for the text to be readable on the page. I suggest either zooming in on the computer or running the code given at the end of the question.

	,	a	p	i	s	r	m	n	d	e	w	o	t	h	l	u	f	y	b	k	g	x	0	9	1	3	2	4	8	5
k	1.44e-01	7.36e-04	1.76e-03	9.25e-02	1.28e-01	3.08e-02	2.51e-02	8.04e-02	2.38e-04	2.55e-03	6.33e-03	1.66e-03	1.93e-03	6.80e-02	2.35e-03	1.39e-03	5.20e-03	8.54e-04	0.00e-00	2.79e-03	1.17e-03	3.09e-02	9.66e-05	4.03e-04	1.42e-02	7.15e-02	2.05e-02	8.72e-03	4.16e-02	1.02e-02
x	2.15e-01	4.09e-03	3.66e-04	1.81e-01	8.38e-02	2.63e-03	2.40e-03	2.05e-02	4.92e-02	1.31e-01	1.74e-01	1.86e-03	8.09e-02	1.08e-01	2.48e-03	1.14e-01	1.77e-01	2.95e-02	2.19e-04	8.49e-02	2.12e-03	1.47e-02	1.26e-01	1.14e-03	1.38e-01	1.78e-03	9.15e-04	0.00e-00	0.00e-00	0.00e-00
r	1.72e-02	4.17e-03	3.12e-02	2.05e-03	1.27e-03	1.86e-02	1.05e-02	3.58e-02	1.97e-02	8.08e-04	1.73e-03	3.08e-03	1.08e-02	8.42e-03	8.19e-04	1.71e-02	3.20e-02	0.00e-00	1.32e-03	1.12e-03	5.75e-03	8.66e-05	1.66e-02	2.72e-05	4.64e-03	3.15e-03	2.62e-03	0.00e-00	1.39e-02	1.26e-02
m	1.18e-01	3.63e-03	2.03e-04	9.93e-02	3.48e-02	1.51e-02	5.44e-02	1.24e-01	1.72e-02	2.06e-03	7.96e-02	1.15e-02	2.17e-02	5.47e-02	7.95e-03	6.04e-02	2.50e-02	2.08e-02	3.84e-02	7.26e-02	1.52e-03	3.09e-02	9.25e-02	4.05e-03	6.47e-02	2.42e-02	1.05e-02	5.24e-02	6.40e-03	2.70e-02
n	2.19e-01	4.21e-02	2.27e-04	1.05e-01	1.06e-01	6.51e-02	1.00e-01	6.12e-02	1.25e-02	7.78e-02	8.44e-02	1.06e-03	6.91e-02	1.52e-02	3.81e-02	6.19e-02	2.52e-02	2.78e-04	1.28e-03	1.00e-03	1.52e-03	1.19e-01	1.15e-02	1.27e-02	1.00e-03	1.78e-02	0.00e-00	0.00e-00	0.00e-00	0.00e-00
d	6.76e-02	2.61e-02	1.28e-03	1.09e-02	2.19e-02	2.59e-02	3.31e-02	1.56e-01	1.23e-01	1.49e-02	4.57e-02	6.76e-03	9.63e-02	1.12e-01	1.52e-02	4.81e-03	7.51e-03	1.79e-01	0.00e-00	4.55e-02	6.61e-03	2.31e-02	8.08e-03	5.85e-02	4.61e-03	2.58e-02	2.04e-02	0.00e-00	1.46e-02	0.00e-00
h	4.28e-02	6.93e-03	1.28e-02	4.66e-01	1.75e-01	4.05e-03	6.93e-03	7.05e-03	7.04e-03	1.54e-02	1.08e-02	3.64e-04	8.19e-02	1.52e-02	1.78e-02	3.55e-01	6.34e-01	1.77e-03	0.00e-00	1.36e-02	6.13e-04	1.42e-03	1.78e-01	3.15e-03	4.08e-04	2.90e-02	0.00e-00	0.00e-00	0.00e-00	0.00e-00
i	4.28e-02	1.51e-03	4.33e-02	9.33e-02	6.25e-02	1.28e-02	8.86e-02	5.41e-02	1.51e-01	1.70e-01	9.70e-02	2.93e-03	3.16e-02	1.77e-02	1.38e-02	5.01e-02	6.04e-02	2.72e-02	4.28e-03	8.62e-02	7.93e-02	2.55e-02	1.72e-03	9.17e-03	2.79e-02	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00
l	1.27e-02	2.01e-03	7.05e-02	9.35e-02	1.37e-02	3.34e-02	9.01e-02	2.51e-02	1.73e-02	1.51e-03	1.13e-03	1.18e-02	9.73e-03	1.90e-02	2.92e-02	1.43e-02	9.40e-02	0.00e-00	2.25e-02	3.09e-02	5.80e-03	1.05e-01	1.73e-02	9.03e-03	2.51e-02	3.03e-03	0.00e-00	0.00e-00	1.82e-02	6.47e-02
u	7.66e-03	7.51e-02	6.11e-02	1.08e-02	2.11e-01	2.91e-01	1.60e-03	2.00e-03	2.52e-02	1.01e-01	1.02e-03	9.27e-01	1.77e-01	2.96e-01	1.24e-01	3.17e-01	1.31e-01	4.05e-02	0.00e-00	3.11e-01	1.77e-01	5.09e-01	1.38e-03	1.64e-02	7.74e-01	7.48e-01	3.94e-02	3.47e-02	0.00e-00	0.00e-00
p	2.38e-02	4.17e-03	8.56e-02	1.74e-01	2.16e-02	1.41e-01	3.36e-02	2.64e-02	5.24e-03	1.22e-03	2.28e-03	6.21e-04	1.51e-02	1.04e-01	1.42e-02	1.22e-01	0.00e-00	1.84e-03	5.61e-03	3.98e-02	5.74e-02	4.04e-03	6.66e-02	2.51e-02	1.18e-03	0.00e-00	0.00e-00	0.00e-00	2.90e-02	0.00e-00
s	2.76e-02	9.04e-03	1.24e-02	2.57e-02	1.34e-02	1.28e-03	1.97e-02	9.00e-03	7.98e-04	1.09e-04	1.54e-02	1.52e-04	1.57e-03	2.45e-02	1.24e-01	2.67e-02	9.94e-03	7.77e-05	4.00e-04	3.41e-02	2.03e-02	1.71e-03	1.52e-01	1.13e-03	7.14e-04	0.00e-00	6.38e-04	0.00e-00	0.00e-00	0.00e-00
y	1.36e-03	6.36e-02	2.58e-02	7.28e-03	1.41e-02	1.21e-02	3.90e-02	1.14e-02	1.83e-03	7.66e-04	1.64e-03	8.77e-04	1.14e-02	2.44e-02	1.75e-02	2.79e-03	1.03e-02	3.11e-04	0.00e-00	1.84e-03	3.09e-02	2.60e-04	7.32e-02	1.56e-02	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00
e	1.76e-03	1.61e-03	9.86e-02	2.42e-02	2.84e-02	6.76e-02	6.22e-02	7.39e-02	1.24e-01	1.51e-02	1.78e-02	6.44e-03	3.99e-02	6.17e-02	3.63e-02	1.91e-02	2.32e-02	1.27e-01	0.00e-00	4.56e-02	2.32e-02	1.05e-02	2.52e-02	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00
,	4.21e-04	7.54e-03	1.68e-02	8.44e-03	1.31e-02	1.94e-02	1.95e-02	2.00e-03	1.64e-02	1.25e-02	0.00e-00	1.85e-02	3.72e-02	4.45e-03	2.86e-02	5.24e-02	4.80e-03	0.00e-00	0.00e-00	2.13e-02	6.79e-02	1.11e-03	1.13e-02	1.46e-02	7.51e-02	2.46e-01	2.73e-02	0.00e-00	0.00e-00	0.00e-00
q	1.91e-03	0.00e-00	3.95e-05	0.00e-00	1.67e-05	9.55e-04	1.01e-03	2.48e-03	1.63e-03	1.68e-05	6.22e-05	1.76e-04	1.57e-03	1.14e-02	1.63e-02	3.90e-05	5.34e-03	3.11e-05	0.00e-00	3.64e-05	0.00e-00	8.65e-05	0.00e-00	3.99e-04	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00
d	1.63e-01	1.99e-03	1.64e-02	6.05e-05	1.27e-04	9.25e-02	3.03e-02	2.99e-02	1.06e-02	4.17e-03	7.39e-02	1.94e-03	1.38e-01	1.65e-02	1.66e-03	1.51e-02	2.23e-02	0.00e-00	9.64e-04	1.87e-04	2.49e-05	6.35e-04	6.64e-03	2.23e-04	4.84e-03	5.74e-03	0.00e-00	0.00e-00	0.00e-00	0.00e-00
,	1.99e-04	2.51e-03	2.22e-03	6.08e-03	1.09e-02	1.49e-02	1.41e-02	3.88e-05	7.55e-04	9.16e-03	6.27e-03	1.01e-05	1.19e-02	2.11e-02	1.95e-03	2.06e-02	2.38e-02	3.04e-03	0.00e-00	7.60e-03	2.67e-02	4.67e-02	6.06e-04	1.29e-04	1.53e-02	3.78e-02	1.53e-03	2.33e-02	6.48e-03	1.79e-01
m	8.46e-05	2.11e-03	2.32e-02	6.05e-04	1.80e-03	2.27e-02	2.66e-02	3.41e-02	1.79e-02	3.44e-04	1.45e-03	1.24e-03	1.14e-02	5.96e-02	1.51e-03	2.27e-02	2.14e-02	0.00e-00	1.86e-03	2.29e-02	4.62e-02	2.02e-03	1.74e-01	2.23e-02	4.08e-03	4.17e-01	0.00e-00	0.00e-00	2.41e-02	3.67e-02
w	1.86e-04	7.94e-03	1.28e-02	8.64e-04	6.45e-03	1.28e-02	2.78e-03	7.11e-02	3.15e-01	1.15e-02	3.23e-03	7.05e-03	1.56e-02	8.25e-03	5.26e-02	2.14e-02	2.60e-02	2.95e-04	0.00e-00	1.66e-03	9.90e-04	5.68e-02	6.06e-04	1.24e-02	1.08e-02	2.21e-02	0.00e-00	0.00e-00	6.20e-03	0.00e-00
o	8.56e-05	1.28e-05	1.73e-03	0.00e-00	3.95e-03	1.28e-04	6.59e-04	2.27e-04	1.36e-03	0.00e-00	4.64e-04	2.14e-05	1.63e-04	2.46e-05	8.09e-04	7.93e-05	4.23e-05	1.15e-03	0.00e-00	0.00e-00	0.00e-00	7.79e-04	8.66e-05	1.24e-04	2.50e-04	0.00e-00	0.00e-00	0.00e-00	8.58e-02	0.00e-00
,	1.99e-05	1.93e-03	9.28e-05	1.77e-04	1.95e-04	1.17e-04	4.43e-05	1.28e-04	3.52e-03	0.00e-00	4.94e-03	1.07e-03	7.37e-05	6.38e-04	9.98e-04	1.12e-04	0.00e-00	1.64e-04	1.02e-03	1.97e-03	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	2.13e-02	7.84e-04	0.00e-00	0.00e-00
,	1.63e-05	6.38e-05	2.95e-05	1.10e-04	2.05e-04	1.72e-04	3.99e-04	0.00e-00	1.83e-04	2.24e-04	0.00e-00	1.24e-04	0.00e-00	4.34e-05	1.15e-03	1.56e-05	7.77e-05	0.00e-00	3.46e-04	7.50e-04	2.58e-02	0.00e-00	2.22e-04	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00
h	2.25e-05	2.23e-03	1.83e-04	1.52e-04	1.60e-04	8.94e-04	1.28e-03	1.98e-05	1.29e-04	7.72e-04	1.16e-03	0.00e-00	8.42e-03	1.31e-04	4.16e-04	2.01e-03	1.96e-02	4.46e-05	0.00e-00	3.04e-03	1.39e-04	3.22e-03	1.44e-04	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00
h	1.53e-05	2.52e-03	2.25e-03	8.42e-04	1.29e-03	8.24e-03	1.05e-02	2.61e-02	2.52e-01	3.78e-02	4.44e-03	1.53e-03	1.06e-02	6.55e-02	1.41e-03	1.99e-02	2.75e-01	1.36e-03	0.00e-00	1.65e-04	5.14e-03	3.04e-02	3.95e-04	1.91e-02	2.52e-02	3.09e-03	0.00e-00	0.00e-00	1.65e-02	5.24e-02
j	1.99e-05	6.59e-05	4.94e-05	8.86e-05	2.12e-04	2.36e-04	2.36e-04	0.00e-00	2.85e-04	1.42e-04	0.00e-00	2.52e-04	4.79e-04	1.74e-05	1.09e-04	2.95e-04	4.66e-05	0.00e-00	0.00e-00	2.29e-04	2.22e-04	7.64e-04	6.38e-04	1.05e-02	0.00e-00	0.00e-00	0.00e-00	0.00e-00	4.28e-04	0.00e-00
d	6.53e-05	2.67e-03	1.68e-02	4.48e-05	1.45e-02	7.36e-03	1.05e-02	1.75e-02	2.52e-02	1.52e-03	1.68e-03	1.41e-03	1.36e-03	8.60e-04	2.24e-03	9.00e-02	2.36e-03	5.66e-02	0.00e-00	1.29e-02	1.79e-02	1.17e-03	0.00e-00	2.72e-03	3.66e-03	2.51e-03	0.00e-00	0.00e-00	0.00e-00	0.00e-00
p	3.25e-05	2.97e-03	2.12e-02	5.83e-02	7.65e-04	1.23e-02	1.23e-03	2.20e-02	4.98e-03	2.59e-03	4.38e-03	2.76e-03	1.11e-03	2.16e-02	1.66e-02	1.05e-03	1.56e-02	4.23e-02	0.00e-00	9.03e-04	1.61e-02	2.98e-03	1.66e-02	0.00e-00	0.00e-00	0.00e-00	5.93e-04	2.65e-02	1.75e-02	3.55e-03
h	3.25e-05	1.28e-04	1.73e-02	4.24e-04	8.61e-04	1.86e-03	2.24e-03	4.40e-02	7.36e-02	5.51e-03	1.68e-03	1.51e-03	1.31e-02	3.75e-02	5.18e-03	1.39e-03	2.68e-03	1.46e-02	0.00e-00	1.33e-02	1.51e-02	5.45e-03	6.03e-03	1.53e-02	0.00e-00	0.00e-00	9.82e-04	6.36e-02	8.74e-03	1.93e-03
f	1.99e-05	1.17e-03	6.38e-04	8.42e-04	1.57e-03	1.73e-03	1.98e-03	1.28e-03	1.54e-03	1.48e-03	6.68e-03	1.01e-03	3.18e-02	7.51e-04	2.71e-02	2.67e-03	1.63e-03	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	2.93e-03	0.00e-00	0.00e-00	0.00e-00	0.00e-00
f	1.96e-05	5.00e-04	8.46e-03	2.24e-03	1.38e-02	1.14e-02	3.06e-02	3.70e-02	2.14e-02	2.21e-02	1.51e-02	3.23e-02	8.93e-02	1.21e-02	2.56e-03	5.23e-02	3.04e-02	4.23e-03	0.00e-00	4.14e-02	2.66e-02	1.76e-03	2.55e-03	0.00e-00	0.00e-00	1.42e-03	7.05e-02	8.74e-04	7.28e-04	2.66e-02
l	1.63e-05	6.74e-03	2.65e-02	2.75e-04	1.94e-03	1.86e-02	1.78e-03	0.00e-00	2.45e-04	1.45e-03	7.77e-04	0.00e-00	1.98e-03	1.66e-02	8.42e-04	2.60e-02	1.55e-02	1.48e-03	0.00e-00	1.25e-03	1.65e-03	4.78e-03	1.78e-01	5.26e-03	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00
d	1.63e-05	2.99e-03	1.76e-02	4.24e-04	8.61e-04	1.86e-03	2.24e-03	4.40e-02	7.36e-02	5.51e-03	1.68e-03	1.51e-03	1.31e-02	3.75e-02	5.18e-03	1.39e-03	2.68e-03	1.46e-02	0.00e-00	1.33e-02	1.51e-02	5.45e-03	6.03e-03	1.53e-02	0.00e-00	0.00e-00	9.82e-04	6.36e-02	8.74e-03	1.93e-03
d	0.00e-00	2.99e-03	8.66e-03	2.28e-03	1.79e-03	1.28e-03	2.56e-03	0.00e-00	1.28e-03	1.28e-03	1.28e-03	1.28e-03	1.28e-03	1.28e-03	1.28e-03	1.28e-03	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00
j	0.00e-00	1.99e-06	8.65e-04	2.28e-05	1.99e-05	1.48e-05	2.66e-05	0.00e-00	1.66e-05	0.00e-00	2.79e-04	6.73e-05	1.17e-04	6.37e-04	9.74e-05	2.17e-03	1.52e-05	0.00e-00	9.11e-05	4.97e-05	2.36e-04	1.65e-03	0.00e-00	0.00e-00	0.00e-00	5.48e-04	0.00e-00	0.00e-00	2.66e-02	2.29e-02
a	0.00e-00	0.00e-00	2.52e-04	0.00e-00	0.00e-00	1.99e-02	0.00e-00	4.56e-04	1.79e-03	0.00e-00	1.64e-05	1.25e-05	1.22e-05	7.42e-04	0.00e-00	0.00e-00	6.66e-04	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	1.95e-02	0.00e-00	0.00e-00
3	0.00e-00	0.00e-00	0.00e-00	0.00e-00	1.52e-03	0.00e-00	0.00e-00	2.72e-05	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	2.77e-03	0.00e-00	0.00e-00	0.00e-00	0.00e-00	1.62e-02	6.47e-02	7.28e-04	8.51e-02
5	0.00e-00	0.00e-00	0.00e-00	0.00e-00	4.42e-04	0.00e-00	0.00e-00	1.55e-05	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	2.54e-02	0.00e-00	1.28e-05	0.00e-00
2	0.00e-00	0.00e-00	0.00e-00	0.00e-00	1.28e-06	0.00e-00	0.00e-00	3.66e-05	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	1.52e-03	2.32e-02	1.36e-02	0.00e-00
0	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	3.19e-06	0.00e-00	1.84e-06	0.00e-00	0.00e-00	0.00e-00	0.00e-00	1.63e-05	6.14e-06	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	1.62e-05	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00
9	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	6.76e-04	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00
7	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	1.17e-05	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	2.25e-02	5.78e-03	0.00e-00	0.00e-00
8	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	1.28e-05	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	4.26e-03	1.78e-03	2.60e-02	0.00e-00
4	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	1.75e-05	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	2.50e-03	1.78e-03	0.00e-00	2.13e-02
6	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	3.26e-05	0.00e-00	0.00e-00	0.00e-00	2.11e-05	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	4.42e-03	1.78e-03	0.00e-00	0.00e-00
-	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	2.88e-06	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00
*	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	1.17e-05	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	0.00e-00	2.13e-02	1.89e-05	0.00e-00	0.00e-00

Below is the table showing the invariant probability distribution. The probabilities are given in descending order.

Symbol	Probability	Symbol	Probability
space	0.165211	x	0.001303
e	0.100472	!	0.001260
t	0.072515	?	0.001006
a	0.064910	j	0.000825
o	0.060963	-	0.000586
n	0.059082	;	0.000367
i	0.055083	:	0.000315
h	0.053595	z	0.000766
s	0.052255	q	0.000748
r	0.047498	(	0.000214
d	0.037947	)	0.000214
l	0.030960	1	0.000114
u	0.020655	*	0.000090
c	0.019650	8	0.000056
m	0.019776	2	0.000042
w	0.018999	0	0.000049
f	0.017609	5	0.000015
g	0.016464	6	0.000016
y	0.014834	3	0.000018
p	0.014485	9	0.000010
,	0.012797	7	0.000012
b	0.011110	’	0.000002
.	0.009906	"	0.000007
v	0.008630	=	0.000001
k	0.006550	/	0.000003

Table 8: Table showing the invariant probability distribution

The code use to load the text, learn the transition statistics, estimate the transition probabilities, the stationary distribution, and display both tables can be found in Listing 7:

```
import numpy as np
from collections import Counter, defaultdict
import pandas as pd

# We start by loading War and Peace, the symbols list and the encypted message using the
    functions below:

def load_symbols(file_path):
```

```python
    with open(file_path, 'r', encoding='utf-8') as file:
        symbols = []
        for line in file:
            symbol = line.rstrip()  # Use rstrip() to remove newline but not internal spaces
            # I've had to add the line below because I couldn't get the space to appear when loading the symbols
            if symbol == "":
                symbols.append(" ")
            else:
                symbols.append(symbol)
    return symbols

def load_encrypted_message(file_path):
    with open(file_path, 'r', encoding='utf-8') as file:
        encrypted_message = file.read().strip()
    return encrypted_message

def load_war_and_peace(file_path):
    with open(file_path, 'r', encoding='utf-8') as file:
        text = file.read()
        # I've truncated the text such that the text starts at the start of the book
        # I've made sure only symbols that exist in the symbols.txt file remained in the text
        text = text[7473:].lower()
        filtered_text = ''.join(char.lower() for char in text if char.lower() in symbols)
    return filtered_text

def calculate_transition_statistics(text):
    # We count occurrences of each symbol and each symbol pair
    single_counts = Counter(text)
    pair_counts = Counter(zip(text, text[1:]))

    # We sum to find the total symbols count
    total_symbols = sum(single_counts.values())

    # We calculate the stationary distribution
    stationary_dist = {char: count / total_symbols for char, count in single_counts.items()}

    # We calculate transition probabilities
    transition_probs = defaultdict(dict)
    for (char1, char2), count in pair_counts.items():
        transition_probs[char1][char2] = count / single_counts[char1]

    return stationary_dist, transition_probs

# We load the three files
symbols = load_symbols('symbols.txt')
encrypted_message = load_encrypted_message('message.txt')
war_and_peace_text = load_war_and_peace('war_and_peace.txt')

# We get the invariant distribution and the transition probailities
probabilities = calculate_transition_statistics(war_and_peace_text)
invariant_distrib = probabilities[0]
transition_prob = probabilities[1]

# We creating stationary distribution table
invariant_df = pd.DataFrame(list(invariant_distrib.items()), columns=['Symbol', 'Probability'])

```

```python
# We create an NxN transition probability matrix DataFrame from the transition_prob dictionary
transition_df = pd.DataFrame(transition_prob).fillna(0)

# Get the table into Latex
transition_df.to_latex("transition_table.tex", index=True, float_format="%.2e")
```

Listing 7: Python code used to answer question 5a

5.2 Question 5b

Since we are assuming that we have a uniform prior distribution over all possible permutations, the first choice in the permutation is chosen independently of the others. However, as soon as a particular permutation σ is selected, each mapping $\sigma(s)$ is dependent on the other mappings in that permutation (since each symbol (original text) must map to a unique symbol (encrypted text)). So the latent variables are not independent.

The joint probability of observing an encrypted sequence $e_1 e_2 \cdots e_n$ given a specific permutation σ is determined by the probability of the next character given the previous one. We know that the first one we choose can be modelled as sampled from the stationary distribution, and that the rest will be the conditional probabilities of each character based on the preceding character. We can express this as:

$$p(e_1 e_2 \cdots e_n|\sigma) = p(e_1|\sigma) \prod_{i=2}^{n} p(e_i|e_{i-1}, \sigma)$$

Now using the transition probability $\psi(\alpha, \beta)$ for transitions from symbol β to α (given by the Markov model), we can rewrite this as:

$$p(e_1 e_2 \cdots e_n|\sigma) = p(\sigma^{-1}(e_1)) \prod_{i=2}^{n} \psi(\sigma^{-1}(e_i), \sigma^{-1}(e_{i-1}))$$

where we have used the σ^{-1} to denote the act of decrypting.

5.3 Question 5c

The proposal probability $S(\sigma \to \sigma')$ is the probability of proposing a move from the current permutation σ to a new permutation σ'. We would do this by choosing two symbols s and s'randomly and swapping their encrypted symbols $\sigma(s)$ and $\sigma(s')$. Since they are chosen randomly from 53 different symbols, each pair (s, s') has an equal chance of being selected and there are $\binom{53}{2} = \frac{53(53-1)}{2}$ possible ways to choose them. Thus, the probability $\frac{2}{53(52-1)} = 7.26\times$

10^{-4}. Therefore, the proposal probability $S(\sigma \to \sigma')$ doesn't depend on the permutations σ and σ', only on the symbols (s, s') being chosen randomly.

In the Metropolis-Hastings algorithm, the acceptance probability kernel is given by:

$$A(x_{i+1}|x_i) := \min\left\{1, \frac{q(x_i|x_{i+1})p(x_{i+1})}{q(x_{i+1}|x_i)p(x_i)}\right\}$$

Now in our case, for a proposed move from σ to σ' it becomes:

$$A(\sigma \to \sigma') = \min\left(1, \frac{p(e_1e_2\cdots e_n|\sigma')S(\sigma' \to \sigma)}{p(e_1e_2\cdots e_n|\sigma)S(\sigma \to \sigma')}\right)$$

Since $S(\sigma \to \sigma') = S(\sigma' \to \sigma)$ in this symmetric proposal setup, the acceptance probability simplifies to:

$$A(\sigma \to \sigma') = \min\left(1, \frac{p(e_1e_2\cdots e_n|\sigma')}{p(e_1e_2\cdots e_n|\sigma)}\right)$$

5.4 Question 5d

The implemented Metropolis-Hastings sampler is shown in Listing 8.

```python
from collections import Counter, defaultdict
import random
import numpy as np

def load_symbols(file_path):
    """
    This function reads the symbols.tkt file and loads it in Python as a list

    INPUT:
    file_path    : File path of the symbols.txt document

    OUTPUT:
    symbols      : A list of strings containing all the symbols in the symbols.txt file
    """
    with open(file_path, 'r', encoding='utf-8') as file:
        symbols = []
        for line in file:
            symbol = line.rstrip()
            # I've had to add the line below because I couldn't get the space to appear when loading the symbols
            if symbol == "":
                symbols.append(" ")
            else:
                symbols.append(symbol)
    return symbols

def load_encrypted_message(file_path):
    """
    This function loads the encrypted message into Python as a str
```

```python

    INPUT:
    file_path           : File path of the message.txt document

    OUTPUT:
    encrypted_message   : A string containing the encrypted message
    """
    with open(file_path, 'r', encoding='utf-8') as file:
        encrypted_message = file.read().strip()
    return encrypted_message

def load_war_and_peace(file_path):
    """
    This function reads the war_and_peace.txt file and outputs

    INPUT:
    file_path       : File path of the war_and_peace.txt document

    OUTPUT:
    filtered_text   : The War and Peace processed text, to include only lower case and symbols contained in the symbols.txt file
    """
    with open(file_path, 'r', encoding='utf-8') as file:
        text = file.read()
        # I've truncated the text to start at the beginning of the book
        text = text[7473:].lower()
        # We filter out characters not in the symbols list
        filtered_text = ''.join(char.lower() for char in text if char.lower() in symbols)
    return filtered_text

def calculate_transition_statistics(text):
    """
    This function calculates the stationary distribution and the transition probabilities

    INPUT:
    text                : The large corpus of English text (in our case, the war_and_peace_text variable)

    OUTPUT:
    stationary_dist     : A dictionary containg the probabilities of the stationary distribution
    transition_probs    : A dictionary containing all the transition probabilities
    """
    # We count occurrences of each symbol and each symbol pair
    single_counts = Counter(text)
    pair_counts = Counter(zip(text, text[1:]))

    # We sum to find the total symbols count
    total_symbols = sum(single_counts.values())

    # We calculate the stationary distribution
    stationary_dist = {char: count / total_symbols for char, count in single_counts.items()}

    # We calculate transition probabilities
    transition_probs = defaultdict(dict)
    for (char1, char2), count in pair_counts.items():
        transition_probs[char1][char2] = count / single_counts[char1]

    return stationary_dist, transition_probs
```

```python


def generate_initial_permutation_based_on_frequencies(encrypted_message):
    """
    This function generates an initial decryption key

    INPUT:
    encrypted_message    : The encrypted message contained in message.txt

    OUTPUT:
    initial_permutation  : The initial permutation
    """
    # We generate the initial permutation by matching frequencies between the encypted and training texts
    # We count the frequency of each symbol in each text
    encrypted_counts = Counter(encrypted_message)
    encrypted_symbols = list(encrypted_counts.keys())

    war_and_peace_counts = Counter(war_and_peace_text)
    war_and_peace_symbols = list(war_and_peace_counts.keys())

    # We sort the symbols by frequency
    encrypted_symbols_sorted = [symbol for symbol, count in encrypted_counts.most_common()]
    war_and_peace_symbols_sorted = [symbol for symbol, count in war_and_peace_counts.most_common()]

    # We map the encrypted symbols to most frequent symbols in the training text for our initial permutation
    initial_permutation = {}
    for enc_sym, dec_sym in zip(encrypted_symbols_sorted, war_and_peace_symbols_sorted):
        initial_permutation[enc_sym] = dec_sym

    # The method above might not map every symbol, since they might not be present in the encrypted text
    # The code below handles any remaining symbols by assigning them arbitrarily from the remaining symbols
    all_symbols = set(symbols)
    unmapped_symbols = all_symbols - set(initial_permutation.values())
    unmapped_encrypted_symbols = set(encrypted_message) - set(initial_permutation.keys())
    for enc_sym, dec_sym in zip(unmapped_encrypted_symbols, unmapped_symbols):
        initial_permutation[enc_sym] = dec_sym

    return initial_permutation

def decrypt_text(encrypted_message, permutation):
    """
    This function decrypts the encrypted message using the provided permutation

    INPUT:
    encrypted_message : The encrypted message contained in message.txt
    permutation       : The current permutation

    OUTPUT:
    decrypted_message : The decrypted message
    """
    decrypted_message = ''.join([permutation.get(char, char) for char in encrypted_message])
    return decrypted_message

def calculate_posterior_probability(encrypted_message, permutation, transition_probs,
```

```
    stationary_dist):
    """
    This function calculates the log posterior probability of the decrypted message given the current permutation

    INPUT:
    encrypted_message   : The encrypted message contained in message.txt
    permutation         : The current permutation
    transition_probs    : The transition probability dictionary
    stationary_dist     : The stationary distribution probability dictionary

    OUTPUT:
    log_prob            : The log posterior probability of the decrypted message
    """
    # We decrypt the message using the function defined previously
    decrypted_message = decrypt_text(encrypted_message, permutation)
    log_prob = 0.0

    # As suggetsed in the problem, we add the log probability of the first character from the stationary distribution
    first_char = decrypted_message[0]
    if first_char in stationary_dist and stationary_dist[first_char] > 0:
        log_prob += np.log(stationary_dist[first_char])
    else:
        # We add a small probability for any unseen characters
        log_prob += np.log(1e-10)

    # We compute the log probabilities of transitions
    for i in range(1, len(decrypted_message)):
        prev_char = decrypted_message[i - 1]
        curr_char = decrypted_message[i]
        if prev_char in transition_probs and curr_char in transition_probs[prev_char]:
            prob = transition_probs[prev_char][curr_char]
            if prob > 0:
                log_prob += np.log(prob)
            else:
                log_prob += np.log(1e-10)
        else:
            # Similarly to before, we add a small probability for unseen transitions
            log_prob += np.log(1e-10)
    return log_prob

# We load the three files
symbols = load_symbols('symbols.txt')
encrypted_message = load_encrypted_message('message.txt')
war_and_peace_text = load_war_and_peace('war_and_peace.txt')

# We calculate the transition statistics
stationary_dist, transition_probs = calculate_transition_statistics(war_and_peace_text)

# We initialise the Metropolis-Hastings parameters
N = len(encrypted_message)
iterations = 10000
report_every = 100
initial_permutation = generate_initial_permutation_based_on_frequencies(encrypted_message)
best_permutation = initial_permutation.copy()
best_prob = -np.inf

# The following code runs the un the Metropolis-Hastings sampling
```

```python
current_permutation = initial_permutation.copy()
for iteration in range(iterations):
    # We propose a new permutation by swapping two symbols
    s, s_prime = random.sample(current_permutation.keys(), 2)
    proposed_permutation = current_permutation.copy()
    proposed_permutation[s], proposed_permutation[s_prime] = proposed_permutation[s_prime], proposed_permutation[s]

    # We calculate the posterior probabilities of the current and proposed permutations
    current_prob = calculate_posterior_probability(encrypted_message, current_permutation, transition_probs, stationary_dist)
    proposed_prob = calculate_posterior_probability(encrypted_message, proposed_permutation, transition_probs, stationary_dist)

    # We calculate acceptance probability
    acceptance_probability = min(1, np.exp(proposed_prob - current_prob))

    # We decide whether to accept the proposed permutation
    if random.uniform(0, 1) < acceptance_probability:
        current_permutation = proposed_permutation
        current_prob = proposed_prob

    # We update the best permutation found so far
    if current_prob > best_prob:
        best_permutation = current_permutation.copy()
        best_prob = current_prob

    # We report the decryption every 'report_every' iterations
    if iteration % report_every == 0:
        decrypted_text = decrypt_text(encrypted_message[:60], current_permutation)
        print(f"Iteration {iteration}: {decrypted_text}")
```

Listing 8: Python code showing implemented MH smapler

When run, the code above yields the following decryption of the first 60 symbols after every 100 iterations:

Iteration	**Text**
0	on lw whunges tnm lhse furnestpre wetsi lw ytades gtfe le ih
100	on d, ,iungew tna diwe hurnewtpre ,etws d, mtlyew gthe de si
200	an d, ,iongew uns diwe yolnewuple ,euwr d, muthew guye de ri
300	in d, ,oungep ans dope yulnepable ,eapr d, mathep gaye de ro
400	in m. .oungep ans mope yulnepable .eapr m. dathep gaye me ro
500	in m. .oungep ans mope yulnepable .eapr m. dathep gaye me ro
600	in my youngep ans mope .ulnepable yeapr my dathep ga.e me ro
700	in my youngep and mope .ulnepable yeaps my rathep ga.e me so
800	in my younger and more .ulnerable years my pather ga.e me so
900	in my younger and more zulnerable years my father gaze me so
1000	in my younger and more zulnerable years my father gaze me so
1100	in my younger and more vulnerable years my father gave me so

1200	in my younger and more vulnerable years my father gave me so
1300	in my younger and more vulnerable years my father gave me so
1400	in my younger and more vulnerable years my father gave me so
1500	in my younger and more vulnerable years my father gave me so
1600	in my younger and more vulnerable years my father gave me so
1700	in my younger and more vulnerable years my father gave me so
1800	in my younger and more vulnerable years my father gave me so
1900	in my younger and more vulnerable years my father gave me so
2000	in my younger and more vulnerable years my father gave me so

I've displayed no more than 2000 iterations in the table above as the first 60 characters of the message didn't change after this. The chain converged reasonably quickly to a sensible message, and this could be due to the way we initialised the permutation. Following the hint, the permutation wasn't initialised at random and was instead obtained by looking at the frequency of every character in the encrypted message and the war and peace text. By matching the most frequent characters from both, we used the natural frequency distribution of English letters to make an informed guess at the mapping of encrypted symbols. The advantage of this is obvious for our case, however, an issue can arise when the encrypt is too small to determine meaning statistics on the frequency of the characters. Even then, it is better than a random guess as it increases the initial likelihood of correct decryption. Another possible ways to smartly initialise the permutation is to run it many times with random parameters, compute the log-likelihood for each, and keep the best one.

5.5 Question 5e

A Markov chain is called ergodic if it is possible to reach any state from any other state, given enough time, and it has a unique stationary distribution. If some transition probabilities $\psi(\alpha, \beta)$ are zero, it means that certain transitions between states are not allowed under the current model. This would stop the chain from being ergodic as it would prevent certain states reaching others. To restore ergodicity, we can add a small probability to every transition, which will make sure that all states can be reached. This is done and clearly commented in Listing 8.

5.6 Question 5f

Using only symbol probabilities would significantly weaken the algorithm for two main reason. The first is the lack of context. By mapping the symbols in both the English and

encrypted text based on their overall frequency, we are assuming that the characters are independent of each other, which we've seen to not be true in the transition probability matrix. Composite sounds such as "th", "kn" or "oo" for example make certain sequences of letters more probable. This method would work if you assumed that all the characters have distinct frequencies and the both the English and encrypted texts are large (and diverse) enough to show the correct frequency of the characters. Needless to say, this is impractical and very hard to verify. The second reason arises from cases where we don't have representative frequencies for the symbols, as many have similar frequencies, which would make deciphering smaller phrases impossible. High-frequency symbols might be mapped correctly, but without transitions giving information on the structure of the language, low-frequency symbols are difficult to decrypt accurately.

Second-order Markov chains would introduce the dependence on the last two symbols (instead of one) to predict the next. In theory, this would be even better than the first-order Markov chain, as the model would have an even better description of the language's structure. The issue arises when considering the implementation of this method. The transition matrix would go from having 53^2 to 53^3 entries, which makes it a lot heavier to work with computationally. On top of that, we would need a much larger corpus of English text to have accurate transition probabilities for rare sequences of characters. In essence, it is theoretically a good idea to use second-order Markov chains, but in practice it will make difficult to obtain a good transition probability matrix, will make the convergence of the sampler slower (because of the increase in complexity), and will increase computation costs and data requirements.

Having two different symbols map to the same encrypted one would create issues when decrypting. The first of the loss of information, as there would be no way for the Markov chain to distinguish and separate the two original symbols, leading to a decrypted message which won't read like correct English. Additionally, the log probability function relies on modelling the decryption process as a permutation of symbols. If we didn't have one-to-one mapping and the function could not uniquely determine the English from the encrypted text. There will then be ambiguity for the sampler, as multiple plaintext sequences could correspond to the same encrypted text, which might stop it from converging quickly to a solution.

Similarly to what we said for second-order Markov chains, using this method to decrypt Chinese would lead to significant issues computationally. If Chinese has more than 1000 characters, then the transition probability matrix would have 1000^2 entries, which is very difficult to store and use in the sampling. Moreover, this would require an enormous amount of text to get accurate transition probabilities, as some characters would be very rare. The set of all possible permutations is so large that it would take the MC sampler a very long time

to converge too. In conclusion, I don't think this method would work to decrypt Chinese symbols.

6 Bonus: Implementing Gibbs sampling for LDA

Not attempted

7 Optimization

7.1 Question 7a

We want to solve for the local extrema of $f(x, y) = x+2y$ subject to the constraint $y^2+xy = 1$ using Lagrange multipliers. This methods finds the extrema of $f(x, y)$ with the constraint $g(x, y) = 0$, so first, we will rewrite the constraint as:

$$g(x, y) = y^2 + xy - 1$$

with $g(x, y) = 0$, to match the notation. The Lagrange multiplier method uses the fact that, at any point where the function f reaches a local extremum on the constraint curve, the gradients of $f(x, y)$ and $g(x, y)$ are aligned. Since we're constrained to the curve defined by $g(x, y) = 0$, any movement along the constraint doesn't change $g(x, y)$. So, at the extrema, the gradients of $f(x, y)$ and $g(x, y)$ must be proportional, meaning they point in the same or opposite directions. Mathematically this translates to:

$$f(x, y) = \lambda g(x, y)$$
$$f(x, y) - \lambda g(x, y) = 0$$

Intuitively from the expression above, we define the Lagrange function as:

$$\mathcal{L}(x, y, \lambda) = f(x, y) - \lambda g(x, y) = x + 2y - \lambda(y^2 + xy - 1) = 0$$

It follows that the partial derivatives with respect to x, y and λ of the Lagrange function would be equal to 0. This gives:

$$\frac{\partial \mathcal{L}}{\partial x} = 1 - \lambda y = 0.$$
$$\frac{\partial \mathcal{L}}{\partial y} = 2 - \lambda(2y + x) = 0.$$
$$\frac{\partial \mathcal{L}}{\partial \lambda} = -(y^2 + xy - 1) = 0.$$

Which results in the following system of equations:

$$\begin{cases} 1 - \lambda y = 0, \\ 2 - \lambda(2y + x) = 0, \\ y^2 + xy = 1. \end{cases}$$

We can now solve this system of equations. From the first equation we have:

$$\lambda = \frac{1}{y}$$

Substituting this into the second equation gives:

$$2 - \frac{1}{y}(2y + x) = 0 \Rightarrow 2 - 2 - \frac{x}{y} = 0 \Rightarrow x = 0.$$

Substituting $x = 0$ into the constraint equation:

$$y^2 + 0 \cdot y = 1 \Rightarrow y^2 = 1 \Rightarrow y = \pm 1$$

The solution to the system of equation is:

$$\begin{cases} \lambda = \mp 1 \\ x = 0 \\ y = \pm 1 \end{cases}$$

This tells us that there are two local extrema:

1. When $\lambda = 1$, the extrema is at $(x = 0, y = -1)$
2. When $\lambda = -1$, the extrema is at $(x = 0, y = 1)$

7.2 Question 7b

7.2.1 Question 7b i

Newton's method is used as an iterative technique to find a root of a function. We want to find x such that $x = \ln(a)$. Using the definition of the natural logarithm, we have:

$$\exp(x) = a$$

We can rewrite this equation as a root-finding problem:

$$f(x, a) = \exp(x) - a = 0$$

Therefore the function $f(x, a)$ we are looking for is:

$$f(x, a) = \exp(x) - a$$

7.2.2 Question 7b ii

The update equation for Newton's algorithm is given by:

$$x_{n+1} = x_n - \frac{f(x_n, a)}{f'(x_n, a)}$$

where $f'(x_n, a)$ is the derivative of $f(x_n, a)$ with respect to x. For our case:

$$\frac{\partial f(x, a)}{\partial x} = \frac{\partial}{\partial x}(\exp(x) - a) = \exp(x)$$

Substituting in the function we found in part (i) into the update equation, we get:

$$x_{n+1} = x_n - \frac{exp(x_n) - a}{exp(x_n)}$$

8 Bonus: Eigenvalues as solutions of an optimization problem

8.1 Question 8a

The Extreme Value Theorem states that a continuous function reaches its maximum and minimum on a compact set. We are trying to solve the optimisation problem:

$$x^* := \arg\max_{x \in \mathbb{R}^n} R_A(x)$$

But since $\mathbb{R}^n$ isn't compact, we need to search a compact subset. We use the hint that is given, which states we should at the unit sphere:

$$S = \{x \in \mathbb{R}^n \mid \|x\| = 1\},$$

which is a compact set. We can do this because $R_A(x)$ doesn't change if we scale x, as is shown below:

$$R_A(cx) = \frac{(cx)^T A(cx)}{(cx)^T(cx)} = \frac{c^2 x^T A x}{c^2 x^T x} = \frac{x^T A x}{x^T x} = R_A(x)$$

In this compact set, $R_A(x)$ becomes:

$$R_A(x) = \frac{q_A(x)}{\|x\|^2} = q_A(x)$$

as $\|x\|^2 = 1$ in the unit sphere. Thanks to this, we see that $R_A(x)$ is continuous, and thus satisfies all the requirements of the Extreme Value Theorem. $\sup_{x \in \mathbb{S}} R_A(x)$ is attained, we now need to show that this is also true for $x \in \mathbb{R}^n$. Let $z = \frac{x}{\|x\|}$. Substituting this into our equation for $R_A(x)$ yields:

$$\begin{aligned}
R_A(x) &= \frac{x^T A x}{\|x\|^2} \\
&= \frac{x^T A x}{x^T x} \\
&= \left(\frac{x}{\|x\|}\right)^T A \left(\frac{x}{\|x\|}\right) \\
&= z^T A z \\
&= R_A(z)
\end{aligned}$$

This equality shows that the superior in the compact set S is the same as the one in $\mathbb{R}^n$, so we have shown that $\sup_{x \in \mathbb{R}^\ltimes} R_A(x)$ is attained.

8.2 Question 8b

We are told that eigenvectors $\xi_1, \ldots, \xi_n$ form an orthonormal basis and we any vector x can be represented through that orthonormal basis as:

$$x = \sum_{i=1}^{n} (\xi_i^T x)\, \xi_i$$

With this in mind, we expand our expression using the eigenbasis, looking at the nominator first then the denominator. We substitute $x = \sum_{i=1}^{n} (\xi_i^T x)\, \xi_i$ into $x^T A x$:

$$x^T A x = \left(\sum_{i=1}^{n} (\xi_i^T x)\, \xi_i \right)^T A \left(\sum_{j=1}^{n} (\xi_j^T x)\, \xi_j \right)$$

Using linearity of inner product:

$$x^T A x = \sum_{i=1}^{n} \sum_{j=1}^{n} (\xi_i^T x)(\xi_j^T x)\, \xi_i^T A \xi_j$$

As ξ_i are eigenvectors of A with $A\xi_j = \lambda_j \xi_j$, we have $\xi_i^T A \xi_j = \lambda_j \delta_{ij}$. Therefore we have:

$$x^T A x = \sum_{i=1}^{n} (\xi_i^T x)^2 \lambda_i$$

We proceed similarly when looking at the denominator:

$$x^T x = \left(\sum_{i=1}^{n} (\xi_i^T x)\, \xi_i \right)^T \left(\sum_{j=1}^{n} (\xi_j^T x)\, \xi_j \right) = \sum_{i=1}^{n} (\xi_i^T x)^2$$

Putting everything back together gives:

$$R_A(x) = \frac{x^T A x}{x^T x} = \frac{\sum_{i=1}^{n} (\xi_i^T x)^2 \lambda_i}{\sum_{i=1}^{n} (\xi_i^T x)^2}$$

The question tells us that the eigenvalues of A $\lambda_1 \geq \cdots \geq \lambda_n$ are enumerated in decreasing size, meaning λ_1 is the largest. We can write:

$$R_A(x) = \frac{x^T A x}{x^T x} \leq \frac{\sum_{i=1}^{n} (\xi_i^T x)^2 \lambda_1}{\sum_{i=1}^{n} (\xi_i^T x)^2} \leq \lambda_1$$

as required.

8.3 Question 8c

No other eigenvectors have the eigenvalue λ_1, since we have made sure that x isn't one of the eigenvectors with degenerate eigenvalue λ_1. This means the inequality $\lambda_1 \geq \cdots \geq \lambda_n$ becomes strict $\lambda_1 > \cdots > \lambda_n$. Using the final step of our calculation in the previous question, we see that the solution arises:

$$\begin{aligned}
R_A(x) &= \frac{x^T A x}{x^T x} \\
&= \frac{\sum_{i=1}^n (\xi_i^T x)^2 \lambda_i}{\sum_{i=1}^n (\xi_i^T x)^2} \\
&< \frac{\lambda_1 \sum_{i=1}^n (\xi_i^T x)^2}{\sum_{i=1}^n (\xi_i^T x)^2} \\
&< \lambda_1
\end{aligned}$$