

# COMP0086 Summative Assignment

Nov 14, 2022

## Question 1

- (a) Our sample space for images is  $\{0, 1\}^D$ , where each of our  $D$  dimensions can only take binary values,  $D$  being the number of pixels in the image. The exponential family best suited on this sample space is the  $D$ -dimensional multivariate Bernoulli distribution because it shares the same sample space. On the other hand, a  $D$ -dimensional multivariate Gaussian has the sample space  $\mathbb{R}^D$ , which does not match the sample space of our data. To match our data sample space, we might have to define an additional mapping between our data and model spaces, adding unnecessary complexity. Thus it would be inappropriate to model this dataset of images with a multivariate Gaussian.
- (b) For  $\{\mathbf{x}^{(n)}\}_{n=1}^N$ , a data set of  $N$  images, the joint likelihood (assuming images are independently and identically distributed) is the product of  $N$   $D$ -dimensional multivariate Bernoulli distributions, one for each image:

$$P(\{\mathbf{x}^{(n)}\}_{n=1}^N | \mathbf{p}) = \prod_{n=1}^N P(\mathbf{x}^{(n)} | \mathbf{p})$$

Substituting the  $D$ -dimensional multivariate Bernoulli:

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

Taking the logarithm of this, we get the log likelihood:

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

Note that since the logarithm is a monotonically increasing function on  $\mathbb{R}_+$ , the maximisers and minimisers of the likelihood do not change. Thus, to solve for the maximum likelihood estimate,  $\hat{p}_d$ , we can take the derivative of  $\mathcal{L}(\{\mathbf{x}^{(n)}\}_{n=1}^N | \mathbf{p})$  with respect to  $p_d$ , the  $d^{th}$  element of  $\mathbf{p}$ :

$$\begin{aligned} \frac{\partial \mathcal{L}(\{\mathbf{x}^{(n)}\}_{n=1}^N | \mathbf{p})}{\partial p_d} &= \sum_{n=1}^N \left( \frac{x_d^{(n)}}{p_d} - \frac{1 - x_d^{(n)}}{1 - p_d} \right) \\ \frac{\partial \mathcal{L}(\{\mathbf{x}^{(n)}\}_{n=1}^N | \mathbf{p})}{\partial p_d} &= \frac{\sum_{n=1}^N x_d^{(n)}}{p_d} - \frac{\sum_{n=1}^N (1 - x_d^{(n)})}{1 - p_d} \end{aligned}$$

and set the derivative to zero and solve for  $\hat{p}_d$ :

$$\begin{aligned}\frac{\sum_{n=1}^N x_d^{(n)}}{\hat{p}_d} - \frac{\sum_{n=1}^N (1 - x_d^{(n)})}{1 - \hat{p}_d} &= 0 \\ \sum_{n=1}^N x_d^{(n)} - \hat{p}_d \sum_{n=1}^N x_d^{(n)} - \hat{p}_d \cdot N + \hat{p}_d \sum_{n=1}^N x_d^{(n)} &= 0 \\ \hat{p}_d &= \frac{1}{N} \sum_{n=1}^N x_d^{(n)}\end{aligned}$$

Moreover, the second derivative with respect to  $p_d$ :

$$\frac{\partial \mathcal{L}(\{\mathbf{x}^{(n)}\}_{n=1}^N | \mathbf{p})}{\partial p_d^2} = \frac{-\sum_{n=1}^N x_d^{(n)}}{p_d^2} + \frac{-\sum_{n=1}^N (1 - x_d^{(n)})}{(1 - p_d)^2}$$

For a maximum, we need to show that the second derivative is negative. Since  $x_d^{(n)} \in \{0, 1\}$ , in the worst case, of  $N = 1$ , the single pixel  $x_d^{(1)}$  must either be white ( $\sum_{n=1}^N x_d^{(n)} > 0$ ) or black ( $\sum_{n=1}^N 1 - x_d^{(n)} > 0$ ) with the other being zero,  $\frac{\partial \mathcal{L}(\{\mathbf{x}^{(n)}\}_{n=1}^N | \mathbf{p})}{\partial p_d^2} < 0$  will be guaranteed and  $\hat{p}_d$  is a maximum as required for the maximum likelihood estimate.

Because we assume that each pixel is independent (we are taking the product of  $D$  one dimensional Bernoulli distributions), we can express the maximum likelihood for  $\mathbf{p}$  in vectorised form as  $\hat{\mathbf{p}}^{MLE}$ :

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

(c) From Bayes' Theorem:

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

Taking the logarithm:

$$\mathcal{L}(\mathbf{p} | \{\mathbf{x}^{(n)}\}_{n=1}^N) = \mathcal{L}(\{\mathbf{x}^{(n)}\}_{n=1}^N | \mathbf{p}) + \mathcal{L}(\mathbf{p}) - \mathcal{L}(\{\mathbf{x}^{(n)}\}_{n=1}^N)$$

Taking the derivative with respect to  $p_d$ :

$$\frac{\partial \mathcal{L}(\mathbf{p} | \{\mathbf{x}^{(n)}\}_{n=1}^N)}{\partial p_d} = \frac{\partial \mathcal{L}(\{\mathbf{x}^{(n)}\}_{n=1}^N | \mathbf{p})}{\partial p_d} + \frac{\partial \mathcal{L}(\mathbf{p})}{\partial p_d}$$

where  $\frac{\partial \mathcal{L}(\{\mathbf{x}^{(n)}\}_{n=1}^N)}{\partial p_d} = 0$  because it doesn't depend on  $p_d$ .

We know from (b):

$$\frac{\partial \mathcal{L}(\{\mathbf{x}^{(n)}\}_{n=1}^N | \mathbf{p})}{\partial p_d} = \frac{\sum_{n=1}^N x_d^{(n)}}{p_d} - \frac{\sum_{n=1}^N (1 - x_d^{(n)})}{1 - p_d}$$

For the second term  $\frac{\partial \mathcal{L}(\mathbf{p})}{\partial p_d}$ , we start with  $P(\mathbf{p})$ , assuming each pixel to have an independent prior:

$$P(\mathbf{p}) = \prod_{d=1}^D P(p_d)$$

Assuming a Beta prior on each  $p_d$ :

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

Taking the logarithm:

$$\mathcal{L}(\mathbf{p}) = \sum_{d=1}^D -\log(B(\alpha, \beta)) + (\alpha-1) \log p_d + (\beta-1) \log(1-p_d)$$

Taking the derivative with respect to  $p_d$ :

$$\frac{\partial \mathcal{L}(\mathbf{p})}{\partial p_d} = \frac{(\alpha-1)}{p_d} - \frac{(\beta-1)}{1-p_d}$$

Since we are only concerned with  $p_d$ , we are only left with a single element of the summation pertaining to  $p_d$ .

Combining, we have have an expression for  $\frac{\partial \mathcal{L}(\mathbf{p}|\{\mathbf{x}^{(n)}\}_{n=1}^N)}{\partial p_d}$ :

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

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

To find the maximum a posteriori (MAP) estimate  $\hat{p}_d$  set  $\frac{\partial \mathcal{L}(\mathbf{p}|\{\mathbf{x}^{(n)}\}_{n=1}^N)}{\partial p_d} = 0$  and solve:

$$0 = \frac{(\alpha-1) + \sum_{n=1}^N x_d^{(n)}}{\hat{p}_d} - \frac{(\beta-1) + \sum_{n=1}^N (1-x_d^{(n)})}{1-\hat{p}_d}$$

$$0 = (1-\hat{p}_d)(\alpha-1) + (1-\hat{p}_d) \left( \sum_{n=1}^N x_d^{(n)} \right) - \hat{p}_d(\beta-1) - \hat{p}_d \left( \sum_{n=1}^N (1-x_d^{(n)}) \right)$$

$$0 = (\alpha - \alpha\hat{p}_d + \hat{p}_d - 1) + \left( \sum_{n=1}^N x_d^{(n)} - \hat{p}_d \sum_{n=1}^N x_d^{(n)} \right) - (\hat{p}_d\beta - \hat{p}_d) - \left( \hat{p}_d \cdot N - \hat{p}_d \sum_{n=1}^N x_d^{(n)} \right)$$

Cancelling the  $\hat{p}_d \sum_{n=1}^N x_d^{(n)}$  terms:

$$0 = \alpha - \alpha\hat{p}_d + \hat{p}_d - 1 + \sum_{n=1}^N x_d^{(n)} - \hat{p}_d\beta + \hat{p}_d - \hat{p}_d \cdot N$$

$$0 = \hat{p}_d(2 - \alpha - \beta - N) + \alpha - 1 + \sum_{n=1}^N x_d^{(n)}$$

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

To show that this is a maximum, the second derivative is:

$$\frac{\partial^2 \mathcal{L}(\mathbf{p} | \{\mathbf{x}^{(n)}\}_{n=1}^N)}{(\partial p_d)^2} = \frac{(1 - \alpha) - \sum_{n=1}^N x_d^{(n)}}{(p_d)^2} + \frac{(1 - \beta) - \sum_{n=1}^N (1 - x_d^{(n)})}{(1 - p_d)^2}$$

. For a maximum, we need  $\frac{\partial^2 \mathcal{L}(\mathbf{p} | \{\mathbf{x}^{(n)}\}_{n=1}^N)}{(\partial p_d)^2} < 0$  meaning that we need at least one of the strict inequalities  $\alpha < 1 - \sum_{n=1}^N x_d^{(n)}$  or  $\beta < 1 - \sum_{n=1}^N (1 - x_d^{(n)})$  to be satisfied, where the other can be  $\leq$ . The Beta distribution requires  $\alpha > 0$  and  $\beta > 0$  so this requirement will always be satisfied (in the worst case of a single image, either  $x_d^{(1)} = 1$  or  $1 - x_d^{(1)} = 1$ ).

Due to independence of our likelihood and priors for each dimension, we can express the maximum a priori for  $\mathbf{p}$  in vectorised form as  $\hat{\mathbf{p}}^{MAP}$ :

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

(d&e) The Python code for MLE and MAP:

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3
4
5 def _compute_maximum_likelihood_estimate(x: np.ndarray) -> np.ndarray:
6     """
7     Calculates MLE of images
8     :param x: numpy array of shape (N, D)
9     :return: MLE estimate
10    """
11    return np.mean(x, axis=0)
12
13
14 def _compute_maximum_a_priori_estimate(
15     x: np.ndarray, alpha: float, beta: float
16 ) -> np.ndarray:
17     """
18     Calculates MAP estimate of images
19     :param x: numpy array of shape (N, D)
20     :param alpha: param of prior distribution
21     :param beta: param of prior distribution
22     :return: MAP estimate
23    """
24
25    n, _ = x.shape
26    return (alpha - 1 + np.sum(x, axis=0)) / (n + alpha + beta - 2)
27
28
29 def d(x: np.ndarray, figure_path: str, figure_title: str) -> None:
30     """
31     Produces answers for question 1d
32     :param x: numpy array of shape (N, D)
33     :param figure_path: path to store figure
34     :param figure_title: figure title
35     :return:
36    """
37    maximum_likelihood = _compute_maximum_likelihood_estimate(x)
38    plt.figure()
39    plt.imshow(
40        np.reshape(maximum_likelihood, (8, 8)),
41        interpolation="None",
42    )
43    plt.colorbar()
44    plt.axis("off")
45    plt.title(figure_title)
46    plt.savefig(figure_path)
47
48
49 def e(
50     x: np.ndarray, alpha: float, beta: float, figure_path: str, figure_title: str
51 ) -> None:
52     """
53     Produces answers for question 1e
54     :param x: numpy array of shape (N, D)
55     :param alpha: param of prior distribution
56     :param beta: param of prior distribution
57     :param figure_path: path to store figure
58     :param figure_title: figure title
59     :return:
60    """
61    maximum_a_priori = _compute_maximum_a_priori_estimate(x, alpha, beta)
62    plt.figure()
63    plt.imshow(
64        np.reshape(maximum_a_priori, (8, 8)),
65        interpolation="None",
66    )
67    plt.colorbar()
68    plt.axis("off")
69    plt.title(figure_title)
70    plt.savefig(f"{figure_path}.png")
71
72    maximum_likelihood = _compute_maximum_likelihood_estimate(x)
73    plt.figure()
74    plt.imshow(
75        np.reshape(maximum_a_priori - maximum_likelihood, (8, 8)),
76        interpolation="None",
77    )
78    plt.colorbar()
79    plt.axis("off")
80    plt.title(f"MAP vs MLE")
81    plt.savefig(f"{figure_path}-mle-vs-map.png")
```

src/solutions/q1.py

Displaying the learned parameters:

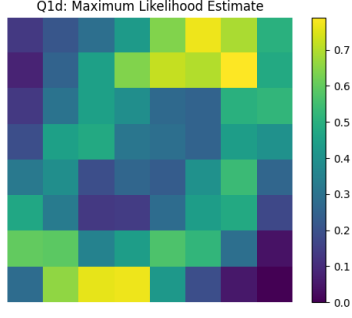


Figure 1: ML parameters

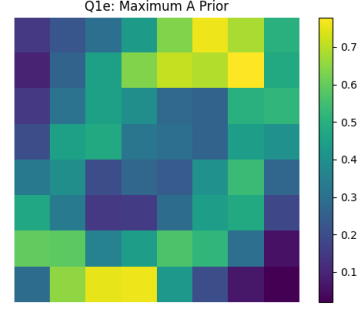


Figure 2: MAP parameters

Comparing the equations:

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

and

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

As the number of data points increases,  $\hat{\mathbf{p}}^{MAP}$  approaches  $\frac{1}{N} \sum_{n=1}^N \mathbf{x}^{(n)}$ ,  $\hat{\mathbf{p}}^{MLE}$ . This makes sense because as our data set gets bigger, we are less reliant on our prior. However, if a specific pixel in all of the images of our data set are white or all black, the MLE for that pixel would either be 1 or 0. This may not be representative of our intuitions about images, as there should be some non-zero probability of a pixel being black or white. By introducing an appropriate prior we can ensure that the probability of that pixel will never be exactly zero or one. In our case, with a Beta(3,3) prior on each pixel, our parameter values are biased to be closer to 0.5 and to never be at the extremities 0 and 1. We can see this in Figure 2 where the range of our parameters is smaller than the range of Figure 1 and doesn't include zero. Figure 3 visualises  $\hat{\mathbf{p}}^{MAP} - \hat{\mathbf{p}}^{MLE}$  and we can see that for likelihoods greater than 0.5 in the MLE, the MAP has a lower value and for likelihoods less than 0.5, the MAP has a higher value, confirming our intuitions.

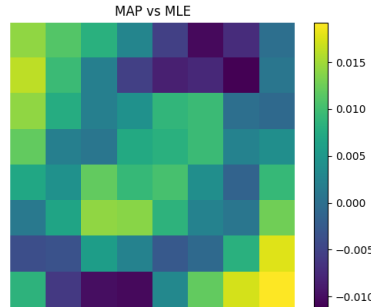


Figure 3:  $\hat{\mathbf{p}}^{MAP} - \hat{\mathbf{p}}^{MLE}$

Priors can also help ensure numerical stability during calculations. The logarithm of zero is negative infinity, so having if the MLE is zero it can be problematic for log-likelihoods calculations whereas MAP can ensure non-zero probabilities. Interestingly, when  $\alpha = \beta = 1$ ,  $\hat{\mathbf{p}}^{MLE} = \hat{\mathbf{p}}^{MAP}$ . This is when the prior is a uniform distribution and so there is uniform bias on the location of  $\mathbf{p}$  and we recover the MLE.

On the other hand, a mis-specified prior can be problematic, as the estimated parameters might be skewed by the prior and not properly represent the underlying data generating process, this can result in parameter estimates that are worse than using the MLE if our data set is limited.

## Question 2

When all D components are generated from a Bernoulli distribution with  $p_d = 0.5$ , we have the likelihood function for model  $M_1$ :

$$P(\mathbf{x}^{(n)}|\mathbf{p}^{(1)} = [0.5, 0.5, \dots, 0.5]^T, M_1) = \prod_{n=1}^N \prod_{d=1}^D (0.5)^{x_d^{(n)}} (0.5)^{1-x_d^{(n)}}$$

When all D components are generated from Bernoulli distributions with unknown, but identical,  $p_d$ , we have the likelihood function for model  $M_2$ :

$$P(\mathbf{x}^{(n)}|\mathbf{p}^{(2)} = [p_d, p_d, \dots, p_d]^T, M_2) = \prod_{n=1}^N \prod_{d'=1}^D p_d^{x_{d'}^{(n)}} (1 - p_d)^{1-x_{d'}^{(n)}}$$

When each component is Bernoulli distributed with separate, unknown  $p_d$ , we have the likelihood function for model  $M_3$ :

$$P(\mathbf{x}^{(n)}|\mathbf{p}^{(3)} = [p_1, p_2, \dots, p_D]^T, M_3) = \prod_{n=1}^N \prod_{d=1}^D p_d^{x_d^{(n)}} (1 - p_d)^{1-x_d^{(n)}}$$

For each model  $M_i$ , we can marginalise out  $\mathbf{p}^{(i)}$  to get  $P(\{\mathbf{x}^{(n)}\}_{n=1}^N|M_i)$ :

$$P(\{\mathbf{x}^{(n)}\}_{n=1}^N|M_i) = \int_0^1 \dots \int_0^1 P(\{\mathbf{x}^{(n)}\}_{n=1}^N|p_d, M_i) P(p_d|M_i) dp_1 \dots dp_D$$

where  $d = 1, \dots, D$  and  $\{\mathbf{x}^{(n)}\}_{n=1}^N$  is our data set.

Given that the prior of any unknown probabilities is uniform, i.e.  $P(p_d|M_i) = 1$ . We can simplify:

$$P(\{\mathbf{x}^{(n)}\}_{n=1}^N|M_i) = \int_0^1 \dots \int_0^1 P(\{\mathbf{x}^{(n)}\}_{n=1}^N|p_d, M_i) dp_1 \dots dp_D$$

For  $M_1$ , we have that all pixels have probability 0.5:

$$P(\{\mathbf{x}^{(n)}\}_{n=1}^N|M_1) = \int_0^1 \dots \int_0^1 \prod_{n=1}^N \prod_{d=1}^D (0.5)^{x_d^{(n)}} (1 - 0.5)^{1-x_d^{(n)}} d\theta_1 \dots d\theta_D$$

We can remove the integrals and knowing that either  $x_d^{(n)}$  or  $1 - x_d^{(n)}$  will be 1 and the other zero, we can simplify  $(0.5)^{x_d^{(n)}} (1 - 0.5)^{1-x_d^{(n)}}$  to 0.5:

$$P(\{\mathbf{x}^{(n)}\}_{n=1}^N|M_1) = \prod_{n=1}^N \prod_{d=1}^D (0.5)$$

$$P(\{\mathbf{x}^{(n)}\}_{n=1}^N|M_1) = (0.5)^{N \cdot D}$$

For  $M_2$ , we have that all pixels share some probability  $p_d$  so we only need to integrate over a single variable  $p_d$ :

$$P(\{\mathbf{x}^{(n)}\}_{n=1}^N|M_2) = \int_0^1 \prod_{n=1}^N \prod_{d'=1}^D p_d^{x_{d'}^{(n)}} (1 - p_d)^{1-x_{d'}^{(n)}} dp_d$$



Changing the products to sums:

$$P(\{\mathbf{x}^{(n)}\}_{n=1}^N | M_2) = \int_0^1 p_d^{\sum_{n=1}^N \sum_{d'=1}^D x_{d'}^{(n)}} (1 - p_d)^{\sum_{j=1}^N \sum_{d'=1}^D 1 - x_{d'}^{(j)}} dp_d$$

Rewriting:

$$P(\{\mathbf{x}^{(n)}\}_{n=1}^N | M_2) = \int_0^1 (p_d)^K (1 - p_{d'=1})^{N \cdot D - K} dp_d$$

where  $K = \sum_{n=1}^N \sum_{d'=1}^D x_{d'}^{(n)}$ .

This integral is the beta function:

$$P(\{\mathbf{x}^{(n)}\}_{n=1}^N | M_2) = \frac{K!(N \cdot D - k)!}{(N \cdot D + 1)!}$$

For  $M_3$ , we need an integral for each  $p_d$ :

$$P(\{\mathbf{x}^{(n)}\}_{n=1}^N | M_3) = \int_0^1 \dots \int_0^1 \prod_{n=1}^N \prod_{d=1}^D p_d^{x_d^{(n)}} (1 - p_d)^{1 - x_d^{(n)}} dp_1 \dots dp_D$$

We can separate the integrals to only contain the relevant  $p_d$ :

$$P(\{\mathbf{x}^{(n)}\}_{n=1}^N | M_3) = \prod_{d=1}^D \left( \int_0^1 \prod_{n=1}^N p_d^{x_d^{(n)}} (1 - p_d)^{1 - x_d^{(n)}} dp_d \right)$$

Changing the products to sums:

$$P(\{\mathbf{x}^{(n)}\}_{n=1}^N | M_3) = \prod_{d=1}^D \left( \int_0^1 p_d^{\sum_{n=1}^N x_d^{(n)}} (1 - p_d)^{\sum_{n=1}^N 1 - x_d^{(j)}} dp_d \right)$$

In this case, we have the product of integrals where each evaluates to a beta function:

$$P(\{\mathbf{x}^{(n)}\}_{n=1}^N | M_3) = \prod_{d=1}^D \frac{K_d!(N - K_d)!}{(N + 1)!}$$

where  $K_d = \sum_{n=1}^N x_d^{(n)}$ .

The posterior probability of a model  $M_i$  can be expressed:

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

We only have three models, so in this case the normalisation  $P(\{\mathbf{x}^{(n)}\}_{n=1}^N)$  can be expressed as a sum:

$$P(M_i | \{\mathbf{x}^{(n)}\}_{n=1}^N) = \frac{P(\{\mathbf{x}^{(n)}\}_{n=1}^N | M_i) P(M_i)}{\sum_{i \in \{1,2,3\}} P(\{\mathbf{x}^{(n)}\}_{n=1}^N | M_i) P(M_i)}$$

Given that  $P(M_i) = \frac{1}{3}$  for all  $i \in \{1, 2, 3\}$ :

$$P(M_i | \{\mathbf{x}^{(n)}\}_{n=1}^N) = \frac{P(\{\mathbf{x}^{(n)}\}_{n=1}^N | M_i)}{\sum_{i \in \{1,2,3\}} P(\{\mathbf{x}^{(n)}\}_{n=1}^N | M_i)}$$

$i$	$P(M_i \{\mathbf{x}^{(n)}\}_{n=1}^N)$
1	1E-1924
2	1E-1858
3	$1-(1\text{E-}1924)-(1\text{E-}1858)$

Table 1: Posterior Probabilities

Calculating the posterior probabilities of each of the three models having generated the data in binarydigits.txt using python, we can show the values in the Table 1:

We can see that for models specified to have the same parameter value for all pixels like  $M_1$  is very unlikely with the given data set. This makes sense because it is specifying models where the image is essentially blank (a uniform shade), which is not reflective of our digit images. Moreover,  $M_1$  specifies a specific value of 0.5 for all the parameters whereas  $M_2$  specifies any value for all the parameters as long as it's the same. So the model  $M_1$  is a subset of the models specified in  $M_2$  and we can see this reflected in our probabilities when  $P(M_2|\{\mathbf{x}^{(n)}\}_{n=1}^N) > P(M_1|\{\mathbf{x}^{(n)}\}_{n=1}^N)$ .

The Python code for calculating the posterior probabilities of the three models:

```

1 import numpy as np
2 import pandas as pd
3 from scipy.special import betaln, logsumexp
4
5
6 def _log-p-d-given-m1(x: np.ndarray) -> float:
7     """
8     Calculates log likelihood of model 1
9     :param x: numpy array of shape (N, D)
10    :return: log likelihood
11    """
12    n, d = x.shape
13    return n * d * np.log(0.5)
14
15
16 def _log-p-d-given-m2(x: np.ndarray):
17     """
18     Calculates log likelihood of model 2
19     :param x: numpy array of shape (N, D)
20     :return: log likelihood
21     """
22    n, d = x.shape
23    k = np.sum(x, axis=0).astype(int)
24    return betaln(np.sum(k) + 1, n * d - np.sum(k) + 1)
25
26
27 def _log-p-d-given-m3(x: np.ndarray):
28     """
29     Calculates log likelihood of model 3
30     :param x: numpy array of shape (N, D)
31     :return: log likelihood
32     """
33    n, - = x.shape
34    k = np.sum(x, axis=0).astype(int)
35    return logsumexp(betaln(k + 1, n - k + 1))
36
37
38 def _log-p-model-given-data(x) -> np.ndarray:
39     """
40     Calculates posterior log likelihood of models given image data
41     :param x: numpy array of shape (N, D)
42     :return: posterior log likelihood
43     """
44    log-p-d-given-m = np.array(
45        [
46            _log-p-d-given-m1(x),
47            _log-p-d-given-m2(x),
48            _log-p-d-given-m3(x),
49        ]
50    )
51    log-p-m-given-data = log-p-d-given-m - logsumexp(log-p-d-given-m)
52    return log-p-m-given-data
53
54
55 def c(x: np.ndarray, table_path: str) -> None:
56     """
57     Produces answers for question 2c
58     :param x: numpy array of shape (N, D)
59     :param table_path: path to store table posterior likelihoods
60     :return:
61     """
62    log-p-m-given-data = _log-p-model-given-data(x)
63    df = pd.DataFrame(
64        data=np.array(
65            [
66                np.arange(len(log-p-m-given-data)).astype(int) + 1,
67                [f"1E{int(x/np.log(10))}" for x in log-p-m-given-data[:-1]]
68                + [
69                    f"1-{'-'.join([f'(1E{int(x/np.log(10))})' for x in log-p-m-given-data[:-1]])}"
70                ],
71            ],
72        ).T,
73        columns=["Model", "P(M,i|D)"],
74    )
75    df.set_index("Model", inplace=True)
76    df.to_csv(table_path)

```

src/solutions/q2.py

### Question 3

- (a) The likelihood for a model consisting of a mixture of  $K$  multivariate Bernoulli distributions can be expressed as the product across  $N$  data points:

$$P(\{\mathbf{x}^{(n)}\}_{n=1}^N|\theta) = \prod_{i=1}^N P(x_i|\theta)$$

where  $\{\mathbf{x}^{(n)}\}_{n=1}^N$  is our data set with  $\mathbf{x}^{(n)} \in \mathbb{R}^{D \times 1}$  and  $\theta = \{\pi, \mathbf{P}\}$ ,  $\pi = [\pi_1, \dots, \pi_K] \in \mathbb{R}^{K \times 1}$  our mixing proportions ( $0 \leq \pi_k \leq 1$ ;  $\sum_k \pi_k = 1$ ) and  $\mathbf{P} \in \mathbb{R}^{D \times K}$  the  $K$  Bernoulli parameter vectors with elements  $p_{kd}$  denoting the probability that pixel  $d$  takes value 1 under mixture component  $k$ . We also assume the images are iid and that the pixels are independent of each other within each component distribution.

For each  $P(\mathbf{x}^{(n)}|\theta)$ :

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

The log-likelihood  $\mathcal{L}(\mathbf{x}^{(n)}|\theta)$  can be expressed in matrix form:

$$\mathcal{L}(\mathbf{x}^{(n)}|\theta) = \log \sum_{k=1}^K \pi_k \exp \left( \mathbf{x}^{(n)} \log(\mathbf{P}_k) + (1 - \mathbf{x}^{(n)}) \log(1 - \mathbf{P}_k) \right)$$

which can be further vectorised using Python scipy's *logsumexp* operation.

Moreover, the log-likelihood  $\mathcal{L}(\{\mathbf{x}^{(n)}\}_{n=1}^N|\theta)$  can be expressed:

$$\mathcal{L}(\{\mathbf{x}^{(n)}\}_{n=1}^N|\theta) = \sum_{i=1}^N \left( \log \sum_{k=1}^K \pi_k \exp \left( \mathbf{x}^{(n)} \log(\mathbf{P}_k) + (1 - \mathbf{x}^{(n)}) \log(1 - \mathbf{P}_k) \right) \right)$$

- (b) We know that:

$$P(A|B) \propto P(B|A)P(A)$$

Thus,

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

where  $s^{(n)} \in \{1, \dots, K\}$  a discrete hidden variable with  $P(s^{(n)} = k|\mathbf{x}^{(n)}, \pi) = \pi_k$ . Note that  $P(s^{(n)} = k|\mathbf{x}^{(n)}, \pi) = P(s^{(n)} = k|\mathbf{x}^{(n)}, \pi, \mathbf{P})$  as  $s^{(n)}$  isn't dependent on  $\mathbf{P}$ .

Let  $P(s^{(n)} = k|\mathbf{x}^{(n)}, \pi, \mathbf{P}) \propto P(s^{(n)})$  be the unnormalised responsibility  $\tilde{r}_{nk}$ . Using the mixture for component  $k$ ,  $\pi_k$  and the likelihood function of component  $k$ :

$$\tilde{r}_{nk} = \pi_k \prod_{d=1}^D (p_{kd})^{\mathbf{x}_d^{(n)}} (1 - p_{kd})^{1 - \mathbf{x}_d^{(n)}}$$

Normalising across the components:

$$r_{nk} = \frac{\tilde{r}_{nk}}{\sum_{j=1}^K \tilde{r}_{nj}}$$

we have calculated  $P(s^{(n)} = k | \mathbf{x}^{(n)}, \pi, \mathbf{P})$  for the E step of an EM algorithm.

Moreover,

$$\log \tilde{r}_{nk} = \log \pi_k + \sum_{d=1}^D \left( \mathbf{x}_d^{(n)} \log(p_{kd}) + (1 - \mathbf{x}_d^{(n)}) \log(1 - \exp(\log(p_{kd}))) \right)$$

and

$$\log r_{nk} = \log \tilde{r}_{nk} - \log \sum_{j=1}^K \exp(\log \tilde{r}_{nj})$$

which can be vectorised as  $\log \mathbf{r}_n$  calculated with  $\log \pi$  and  $\log \mathbf{P}$  using Python scipy's *logsumexp* operation.

(c) We know that the expectation log joint can be expressed:

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

Let this quantity be  $E$ . Each term of  $E$  can be expressed:

$$q(s^{(n)}) = \mathbf{r}_n$$

and

$$\log P(\mathbf{x}^{(n)}, s^{(n)} | \pi, \mathbf{P}) = \log[P(\mathbf{x}^{(n)} | s^{(n)}, \pi, \mathbf{P}) P(s^{(n)} | \pi, \mathbf{P})]$$

which is the vectorised version of  $\log \tilde{r}_{nk}$  from part (b) so:

$$\log P(\mathbf{x}^{(n)}, s^{(n)} | \pi, \mathbf{P}) = \log(\pi) + \log(\mathbf{P})^T \mathbf{x}^{(n)} + \log(1 - \mathbf{P})^T (1 - \mathbf{x}^{(n)})$$

Combining:

$$E = \sum_n \mathbf{r}_n^T [\log(\pi) + \log(\mathbf{P})^T \mathbf{x}^{(n)} + \log(1 - \mathbf{P})^T (1 - \mathbf{x}^{(n)})]$$

To maximise with respect to  $\pi$  and  $\mathbf{P}$  for the M step, we want to take the derivative, set to zero, and solve for  $\hat{\pi}$  and  $\hat{\mathbf{P}}$ .

For the  $k^{th}$  element of  $\pi$ :

$$\frac{\partial E}{\partial \pi_k} = \sum_n r_{nk} \frac{1}{\pi_k}$$

The second derivative:

$$\frac{\partial E}{(\partial \pi_k)^2} = \sum_n r_{nk} \frac{-1}{(\pi_k)^2}$$

is always negative because  $r_{nk} \geq 0$ ,  $\sum_n r_{nk} = 1$ ,  $\pi_k \geq 0$ , and  $\sum_n \pi_k = 1$ , ensuring a maximum in the next step.

We can calculate the maximiser with:

$$\frac{\partial E}{\partial \pi_k} + \lambda = 0$$

where  $\lambda$  is a Lagrange multiplier ensuring that the mixing proportions sum to unity.

Thus,

$$\hat{\pi}_k = \frac{\sum_n r_{nk}}{N}$$

For the  $dk^{th}$  element of  $\mathbf{P}$ :

$$\frac{\partial E}{\partial \mathbf{P}_{dk}} = \sum_n r_{nk} \frac{\partial}{\partial \mathbf{P}_{dk}} [\mathbf{x}_d^{(n)} \log \mathbf{P}_{dk} + (1 - \mathbf{x}_d^{(n)}) \log(1 - \mathbf{P}_{dk})]$$

Simplifying:

$$\frac{\partial E}{\partial \mathbf{P}_{dk}} = \sum_n r_{nk} \left( \frac{\mathbf{x}_d^{(n)}}{\mathbf{P}_{dk}} - \frac{1 - \mathbf{x}_d^{(n)}}{1 - \mathbf{P}_{dk}} \right)$$

Similar to Question 1, we can see that taking second derivative, the term in the brackets will always be less than zero and with  $r_{nk} \geq 0$  and  $\sum_n r_{nk} = 1$ , the second derivative will always be negative. This ensures that we have a maximum in the next step.

Setting the derivative to zero:

$$\frac{\sum_n \mathbf{x}_d^{(n)} r_{nk}}{\mathbf{P}_{dk}} - \frac{\sum_n r_{nk} - \sum_n \mathbf{x}_d^{(n)} r_{nk}}{1 - \mathbf{P}_{dk}} = 0$$

Solving for  $\hat{\mathbf{P}}_{dk}$ :

$$\hat{\mathbf{P}}_{dk} \sum_n r_{nk} - \hat{\mathbf{P}}_{dk} \sum_n \mathbf{x}_d^{(n)} r_{nk} = \sum_n \mathbf{x}_d^{(n)} r_{nk} - \hat{\mathbf{P}}_{dk} \sum_n \mathbf{x}_d^{(n)} r_{nk}$$

Thus,

$$\hat{\mathbf{P}}_{dk} = \frac{\sum_n \mathbf{x}_d^{(n)} r_{nk}}{\sum_n r_{nk}}$$

We have the maximizing parameters for the expected log-joint

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

thus obtaining an iterative update for the parameters  $\pi$  and  $\mathbf{P}$  in the M-step of EM. For numerical stability, we can compute the maximisation step for the MAP of  $\mathbf{P}, \hat{\mathbf{P}}_{dk}^{MAP}$  by solving:

$$\frac{\partial E'}{\partial \mathbf{P}_{dk}} = 0$$

where

$$E' = \sum_{n=1}^N q(s^{(n)}) \log P(\mathbf{P} | \pi, \mathbf{x}^{(n)}, s^{(n)})$$

and from Bayes':

$$\log P(\mathbf{P} | \pi, \mathbf{x}^{(n)}, s^{(n)}) = \log P(\mathbf{x}^{(n)}, s^{(n)} | \pi, \mathbf{P}) + \log P(\mathbf{P}) - \log P(\mathbf{x}^{(n)}, s^{(n)} | \pi)$$

Assuming an independent Beta prior on each pixel of each component:

$$\log P(\mathbf{P}) = \sum_{k=1}^K \sum_{d=1}^D -\log(B(\alpha, \beta)) + (\alpha - 1) \log \mathbf{P}_{dk} + (\beta - 1) \log(1 - \mathbf{P}_{dk})$$

and

$$\frac{\partial \log P(\mathbf{P})}{\partial \mathbf{P}_{dk}} = \frac{(\alpha - 1)}{\mathbf{P}_{dk}} - \frac{(\beta - 1)}{1 - \mathbf{P}_{dk}}$$

Thus, the derivative can be expressed as:

$$\frac{\partial E'}{\partial \mathbf{P}_{dk}} = \sum_n \left( r_{nk} \left( \frac{\partial \log P(\mathbf{x}^{(n)}, s^{(n)} | \pi, \mathbf{P})}{\partial \mathbf{P}_{dk}} + \frac{\partial \log P(\mathbf{P})}{\partial \mathbf{P}_{dk}} \right) \right)$$

Substituting the appropriate expressions:

$$\frac{\partial E'}{\partial \mathbf{P}_{dk}} = \sum_n \left( r_{nk} \left( \frac{\mathbf{x}_d^{(n)}}{\mathbf{P}_{dk}} - \frac{1 - \mathbf{x}_d^{(n)}}{1 - \mathbf{P}_{dk}} + \frac{(\alpha - 1)}{\mathbf{P}_{dk}} - \frac{(\beta - 1)}{1 - \mathbf{P}_{dk}} \right) \right)$$

Simplifying:

$$\frac{\partial E'}{\partial \mathbf{P}_{dk}} = \frac{\sum_n r_{nk}(\alpha - 1 + \mathbf{x}_d^{(n)})}{\mathbf{P}_{dk}} - \frac{\sum_n r_{nk}(\beta - \mathbf{x}_d^{(n)})}{1 - \mathbf{P}_{dk}}$$

For a maximum, we see that we need  $\alpha > \mathbf{x}_d^{(n)} - 1$  or  $\beta < \mathbf{x}_d^{(n)}$ , both of which are satisfied knowing that  $\alpha > 0$  and  $\beta > 0$ . Setting  $\frac{\partial E'}{\partial \mathbf{P}_{dk}} = 0$  we can calculate  $\hat{\mathbf{P}}_{dk}^{MAP}$ :

$$\sum_n r_{nk}(\alpha - 1 + \mathbf{x}_d^{(n)}) - \hat{\mathbf{P}}_{dk} \sum_n r_{nk}(\alpha - 1 + \mathbf{x}_d^{(n)}) = \hat{\mathbf{P}}_{dk} \sum_n r_{nk}(\beta - \mathbf{x}_d^{(n)})$$

$$\hat{\mathbf{P}}_{dk}^{MAP} = \frac{\sum_n r_{nk}(\mathbf{x}_d^{(n)} + \alpha - 1)}{(\alpha + \beta - 1)(\sum_n r_{nk})}$$

As a sense check, we can see when setting  $\alpha = 1$  and  $\beta = 1$  we recover  $\hat{\mathbf{P}}_{dk}^{MLE}$  as we would expect.

(d) Plotting the posterior likelihood as a function of the iteration number:

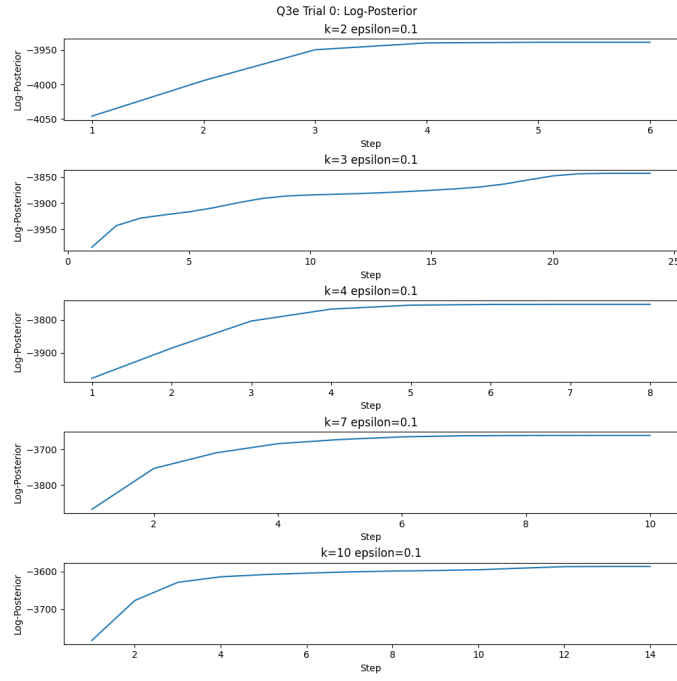


Figure 4: Log Likelihood vs Iteration Number

where *epsilon* is the stopping condition for the posterior posterior converges.



Displaying the parameters found for  $K$  in  $\{2, 3, 4, 7, 10\}$ :

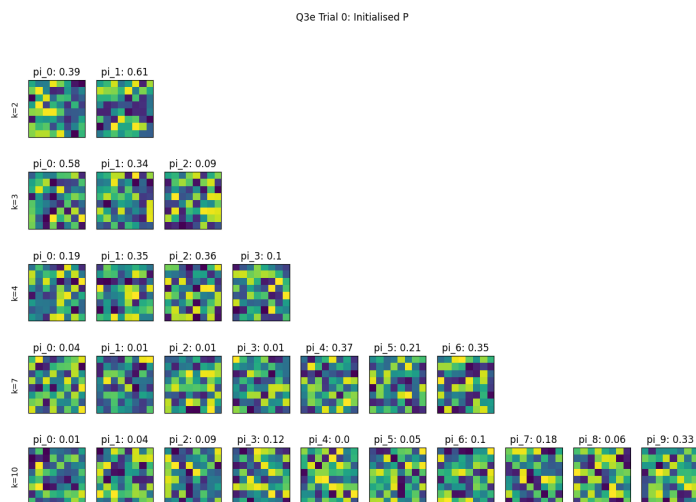


Figure 5: Randomly initialised parameters

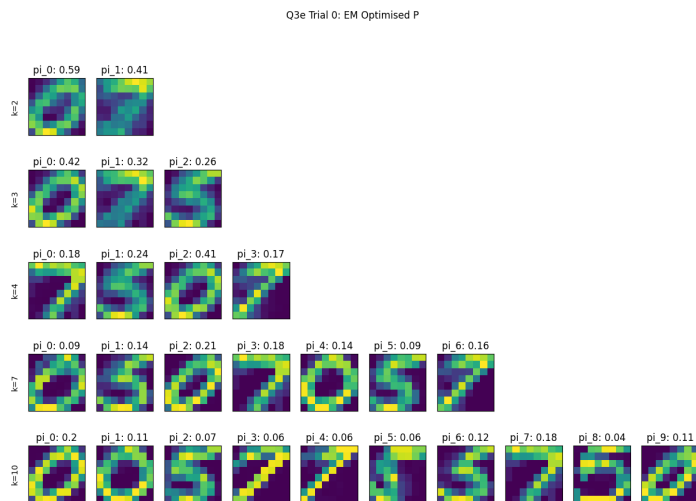


Figure 6: EM optimised parameters

The Python code for the EM algorithm:

```

1 from dataclasses import dataclass
2 from typing import List, Tuple
3
4 import matplotlib.pyplot as plt
5 import numpy as np
6 from scipy.special import logsumexp
7 from sklearn.manifold import TSNE
8
9 from src.constants import DEFAULT_SEED
10
11
12 @dataclass
13 class Theta:
14     """
15     Data class containing the model parameters
16     log-pi: the logarithm of the mixing proportions (1, k)
17     log-p-matrix: the logarithm of the probability where the (i,j)th element is the probability that
18                  pixel j takes value 1 under mixture component i (d, k)
19     """
20
21     log-pi: np.ndarray
22     log-p-matrix: np.ndarray
23
24     @property
25     def pi(self) -> np.ndarray:
26         """
27         Calculates the mixing proportions
28         :return: vector of mixing proportions (1, k)
29         """
30         return np.exp(self.log-pi)
31
32     @property
33     def p-matrix(self) -> np.ndarray:
34         """
35         Calculates the Bernoulli parameters
36         :return: matrix Bernoulli parameters (d, k)
37         """
38         d, k = self.log-p-matrix.shape
39         image_dimension = int(np.sqrt(d))
40         return np.exp(self.log-p-matrix).reshape(image_dimension, image_dimension, -1)
41
42     @property
43     def log-one-minus-p-matrix(self) -> np.ndarray:
44         """
45         Compute log(1-P) where P=exp(log-p-matrix)
46         :return: an array of the same shape as log-p-matrix (d, k)
47         """
48         log-of-one = np.zeros(self.log-p-matrix.shape)
49         stacked_sum = np.stack((log-of-one, self.log-p-matrix))
50         weights = np.ones(stacked_sum.shape)
51         weights[1] = -1 # scale p matrix by -1 for subtraction
52         return np.array(logsumexp(stacked_sum, b=weights, axis=0))
53
54     def log-pi-repeated(self, n: int):
55         """
56         Repeats the log-pi vector n times along axis 0
57         :param n: number of repetitions
58         :return: an array of shape (n, k)
59         """
60         return np.repeat(self.log-pi, n, axis=0)
61
62
63 def _init_params(k: int, d: int) -> Theta:
64     """
65     Random initialisation of theta parameters (log-pi and log-p-matrix)
66     :param k: Number of components
67     :param d: Image dimension (number of pixels in a single image)
68     :return: theta: the parameters of the model
69     """
70     return Theta(
71         log-pi=np.log(np.random.dirichlet(np.ones(k), size=1)),
72         log-p-matrix=np.log(np.random.uniform(low=0, high=1, size=(d, k))),
73     )
74
75
76 def _compute_log_component_p_x_i_given_theta(x: np.ndarray, theta: Theta) -> np.ndarray:
77     """
78     Compute the unweighted probability of each mixing component for each image
79     :param x: the image data (n, d)
80     :param theta: the parameters of the model
81     :return: an array of the unweighted probabilities (n, k)
82     """
83     return x @ theta.log-p-matrix + (1 - x) @ theta.log-one-minus-p-matrix
84
85
86 def _compute_log_p_x_i_given_theta(x: np.ndarray, theta: Theta) -> np.ndarray:
87     """
88     Computes the log likelihood of each image in the dataset x
89     :param x: the image data (n, d)
90     :param theta: the parameters of the model
91     :return: log-p-x-i-given_theta: a log likelihood array containing the log likelihood of each image (n
92             ,1)
93     """
94     n, _ = x.shape

```

```

94     log_component_probabilities = _compute_log_component_p_x_i_given_theta(
95         x, theta
96     ) # (n, k)
97     return np.array(
98         logsumexp(
99             log_component_probabilities
100             + theta.log_pi_repeated(n), # scale each component by component probability
101             axis=1,
102         )
103     )
104
105
106 def _compute_log_likelihood(x: np.ndarray, theta: Theta) -> float:
107     """
108     Computes the log likelihood of all images in the dataset x
109     :param x: the image data (n, d)
110     :param theta: the parameters of the model
111     :return: log_p_x_given_theta: the log likelihood array across all images
112     """
113     return np.sum(_compute_log_p_x_i_given_theta(x, theta)).item()
114
115
116 def _compute_log_e_step(x: np.ndarray, theta: Theta) -> np.ndarray:
117     """
118     Compute the e step of expectation maximisation
119     :param x: the image data (n, d)
120     :param theta: the parameters of the model
121     :return: an array of the log responsibilities of k mixture components for each image (n, k)
122     """
123     log_r_unnormalised = _compute_log_component_p_x_i_given_theta(x, theta)
124     log_r_normaliser = logsumexp(log_r_unnormalised, axis=1)
125     log_responsibility = log_r_unnormalised - log_r_normaliser[:, np.newaxis]
126     return log_responsibility
127
128
129 def _compute_log_pi_hat(log_responsibility: np.ndarray) -> np.ndarray:
130     """
131     Compute the log of the maximised mixing proportions
132     :param log_responsibility: an array of the log responsibilities of k mixture components for each image
133     (n, k)
134     :return: an array of the maximised log mixing proportions (1, k)
135     """
136     n, _ = log_responsibility.shape
137     return (logsumexp(log_responsibility, axis=0) - np.log(n)).reshape(1, -1)
138
139
140 def _compute_log_p_matrix_hat(
141     x: np.ndarray, log_responsibility: np.ndarray
142 ) -> np.ndarray:
143     """
144     Compute the log of the maximised pixel probabilities
145     :param x: the image data (n, d)
146     :param log_responsibility: an array of the log responsibilities of k mixture components for each image
147     (n, k)
148     :return: an array of the maximised pixel probabilities for each component (d, k)
149     """
150     n, d = x.shape
151     _, k = log_responsibility.shape
152
153     x_repeated = np.repeat(x[:, :, np.newaxis], k, axis=2) # (n, d, k)
154     log_responsibility_repeated = np.repeat(
155         log_responsibility[:, np.newaxis, :], d, axis=1
156     ) # (n, d, k)
157
158     alpha = 2
159     beta = 2
160
161     log_p_matrix_unnormalised_posterior = logsumexp(
162         log_responsibility_repeated, b=(x_repeated + alpha - 1), axis=0
163     ) # (d, k)
164
165     log_p_matrix_normaliser_posterior = logsumexp(
166         log_responsibility_repeated, b=(alpha + beta - 1), axis=0
167     ) # (d, k)
168
169     log_p_matrix_normalised_posterior = (
170         log_p_matrix_unnormalised_posterior - log_p_matrix_normaliser_posterior
171     )
172     return log_p_matrix_normalised_posterior
173
174
175 def _compute_log_m_step(x: np.ndarray, log_responsibility: np.ndarray) -> Theta:
176     """
177     Compute the m step of expectation maximisation
178     :param x: the image data (n, d)
179     :param log_responsibility: an array of the log responsibilities of k mixture components for each image
180     (n, k)
181     :return: thetas optimised after maximisation step
182     """
183     return Theta(
184         log_pi=_compute_log_pi_hat(log_responsibility),
185         log_p_matrix=_compute_log_p_matrix_hat(x, log_responsibility),
186     )
187
188 def _run_expectation_maximisation(

```

```

187 x: np.ndarray, theta: Theta, max_number_of_steps: int, epsilon: float
188 ) -> Tuple[Theta, np.ndarray, List[float]]:
189     """
190     Run the expectation maximisation algorithm
191     :param x: the image data (n, d)
192     :param theta: initial theta parameters
193     :param max_number_of_steps: the maximum number of steps to run the algorithm
194     :param epsilon: the minimum required change in log likelihood, otherwise the algorithm stops early
195     :return: a tuple containing the optimised thetas, the log responsibilities,
196             and the log likelihood at each step of the algorithm
197     """
198     log_responsibility = None
199     log_likelihoods = []
200     for _ in range(max_number_of_steps):
201         log_responsibility = _compute_log_e_step(x, theta)
202         theta = _compute_log_m_step(x, log_responsibility)
203
204         log_likelihoods.append(_compute_log_likelihood(x, theta))
205
206         # check for early stopping
207         if len(log_likelihoods) > 1:
208             if (log_likelihoods[-1] - log_likelihoods[-2]) < epsilon:
209                 break
210     return theta, log_responsibility, log_likelihoods
211
212
213 def _visualise_p_matrix(
214     thetas: List[Theta], ks: List[int], figure_title: str, figure_path: str
215 ) -> None:
216     """
217     Visualises the P matrix for different thetas and ks
218     :param thetas: list of Theta instances
219     :param ks: list of k values used for each Theta
220     :param figure_title: name of figure
221     :param figure_path: path to store figure
222     :return:
223     """
224     n = len(ks)
225     m = np.max(ks)
226     fig = plt.figure()
227     fig.set_figwidth(15)
228     fig.set_figheight(10)
229     for i, k in enumerate(ks):
230         for j in range(k):
231             ax = plt.subplot(n, m, m * i + j + 1)
232             ax.imshow(
233                 thetas[i].p_matrix[:, :, j],
234                 interpolation="None",
235             )
236             ax.tick_params(
237                 axis="x",
238                 which="both",
239                 bottom=False,
240                 top=False,
241             )
242             ax.tick_params(
243                 axis="y",
244                 which="both",
245                 left=False,
246                 right=False,
247             )
248             ax.xaxis.set_ticklabels([])
249             ax.yaxis.set_ticklabels([])
250             ax.set_title(f"pi-{j}: {np.round(thetas[i].pi[0, j], 2)}")
251             if j == 0:
252                 ax.set_ylabel(f"k={k}")
253     fig.suptitle(figure_title)
254     plt.savefig(figure_path)
255
256
257 def _visualise_responsibility_clusters(
258     log_responsibilities: List[np.ndarray],
259     ks: List[int],
260     figure_title: str,
261     figure_path: str,
262 ) -> None:
263     """
264     Visualise responsibility vectors of images using TSNE for different k values
265     :param log_responsibilities: list of log responsibilities for different ks
266     :param ks: list of k values used for each Theta
267     :param figure_title: name of figure
268     :param figure_path: path to store figure
269     :return:
270     """
271     n = len(ks)
272     fig = plt.figure()
273     fig.set_figwidth(5 * n)
274     fig.set_figheight(5)
275     for i, k in enumerate(ks):
276         if k > 2:
277             embedding = TSNE(
278                 n_components=2,
279                 learning_rate="auto",
280                 init="random",
281                 perplexity=10,
282                 random_state=DEFAULT_SEED,

```

```

283         ).fit_transform(log_responsibilities[i])
284     else:
285         embedding = np.exp(log_responsibilities[i])
286         ax = plt.subplot(1, n, i + 1)
287         ax.scatter(embedding[:, 0], embedding[:, 1])
288         ax.set_title(f"{k=}")
289     fig.suptitle(figure_title)
290     plt.savefig(figure_path, bbox_inches="tight")
291
292
293 def _plot_log_posteriors(
294     log_posteriors: List[List[float]],
295     ks: List[int],
296     epsilon: float,
297     figure_title: str,
298     figure_path: str,
299 ) -> None:
300     """
301     Plot log posteriors as a function of EM iteration for different ks
302     :param log_posteriors: list of vectors, each representing the log posterior during EM for a specific k
303     :param ks: list of k values used for each Theta
304     :param epsilon: value used for early stopping of EM
305     :param figure_title: name of figure
306     :param figure_path: path to store figure
307     :return:
308     """
309     fig, ax = plt.subplots(len(ks), 1, constrained_layout=True)
310     fig.set_figwidth(10)
311     fig.set_figheight(10)
312     for i, k in enumerate(ks):
313         ax[i].plot(np.arange(1, len(log_posteriors[i]) + 1), log_posteriors[i])
314         ax[i].set_xlabel("Step")
315         ax[i].set_ylabel(f"Log-Posterior")
316         ax[i].set_title(f"{k=} {epsilon=}")
317     plt.suptitle(figure_title)
318
319     plt.savefig(figure_path)
320
321
322 def e(
323     x: np.ndarray,
324     number_of_trials: int,
325     ks: List[int],
326     epsilon: float,
327     max_number_of_steps: int,
328     figure_path: str,
329     figure_title: str,
330 ) -> None:
331     """
332     Produces answers for question 3e
333     :param x: numpy array of shape (N, D)
334     :param number_of_trials: number of trials to run EM
335     :param ks: k values to use for each trial
336     :param epsilon: value used for early stopping of EM
337     :param max_number_of_steps: maximum number of steps during EM
338     :param figure_title: base name of figures
339     :param figure_path: base paths to store figure
340     :return:
341     """
342     n, d = x.shape
343     np.random.seed(DEFAULT_SEED)
344     for i in range(number_of_trials):
345         init_thetas: List[Theta] = []
346         em_thetas: List[Theta] = []
347         log_posteriors: List[List[float]] = []
348         log_responsibilities: List[np.ndarray] = []
349         for j, k in enumerate(ks):
350             init_theta = _init_params(k, d)
351             em_theta, log_responsibility, log_posterior = _run_expectation_maximisation(
352                 x,
353                 theta=init_theta,
354                 epsilon=epsilon,
355                 max_number_of_steps=max_number_of_steps,
356             )
357             init_thetas.append(init_theta)
358             em_thetas.append(em_theta)
359             log_responsibilities.append(log_responsibility)
360             log_posteriors.append(log_posterior)
361
362         _visualise_p_matrix(
363             init_thetas,
364             ks,
365             figure_title=f"{figure_title} Trial {i}: Initialised P",
366             figure_path=f"{figure_path}-{i}-initialised-p.png",
367         )
368         _visualise_p_matrix(
369             em_thetas,
370             ks,
371             figure_title=f"{figure_title} Trial {i}: EM Optimised P",
372             figure_path=f"{figure_path}-{i}-optimised-p.png",
373         )
374         _visualise_responsibility_clusters(
375             log_responsibilities,
376             ks,
377             figure_title=f"{figure_title} Trial {i}: TSNE Responsibility Visualisation",
378             figure_path=f"{figure_path}-{i}-tsne.png",

```

```
379     )
380     -plot_log_posteriors (
381         log_posteriors ,
382         ks,
383         epsilon ,
384         figure_title=f"{figure_title} Trial {i}: Log-Posterior",
385         figure_path=f"{figure_path}-{i}-log-pos.png",
386     )
```

src/solutions/q3.py

- (e) Running the algorithm a few times starting from randomly chosen initial conditions and visualising the parameters:

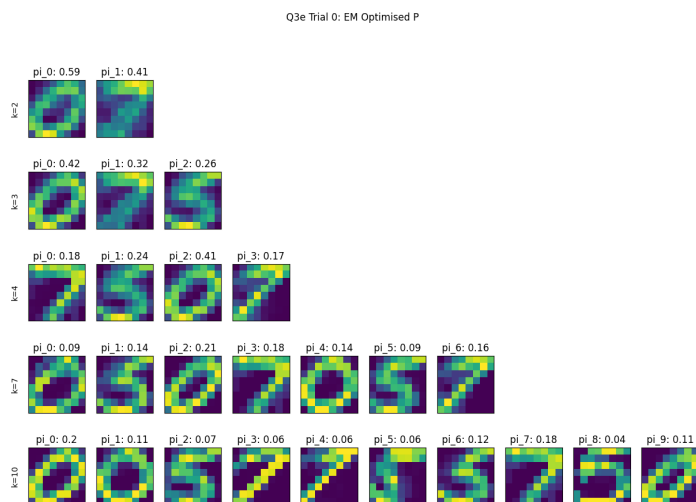


Figure 7: EM optimised parameters: Trial 0

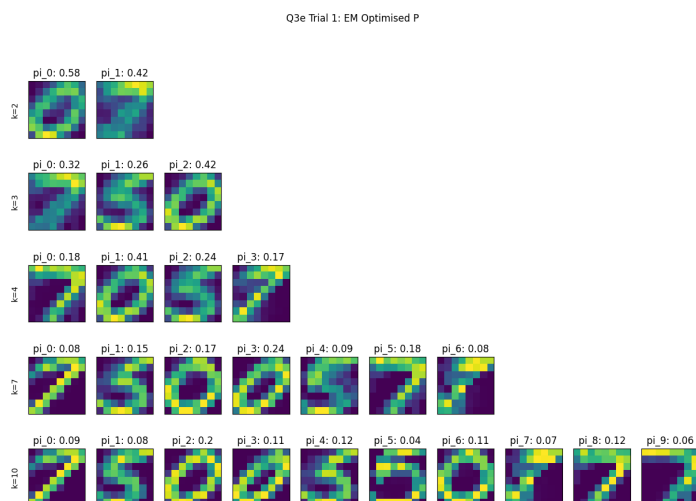


Figure 8: EM optimised parameters: Trial 1

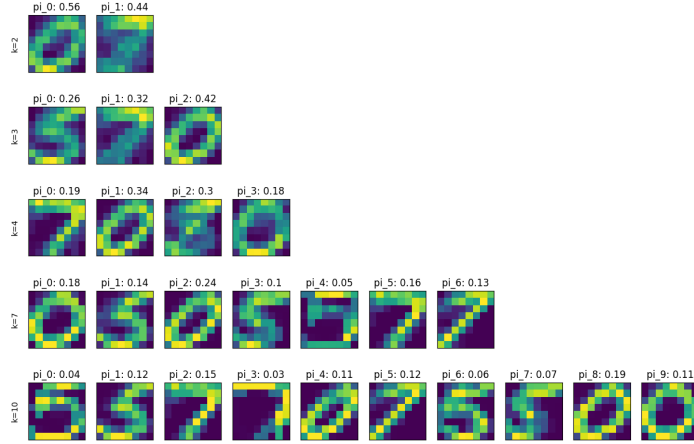


Figure 9: EM optimised parameters: Trial 2

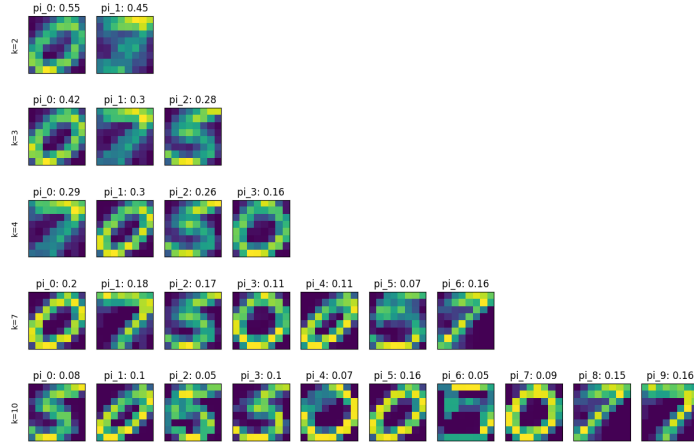


Figure 10: EM optimised parameters: Trial 3

For smaller  $k$ , we can visually see that we obtain very similar solutions (a 7 and a 0 for  $k = 2$ ). However for higher  $K$ , we see that this may not always be the case. For Trial 2 of  $k = 10$ , we have three 5's whereas in Trial 4 we have two 5's. Interestingly, different clusters of the same digits can be different, representing different variants of the written digit (i.e. a slanted zero, a slightly slanted zero, and a symmetric zero).



Moreover, looking at the responsibilities of each mixture component, we can see that when  $k$  is relatively small they are relatively evenly distributed. However for  $k = 7$  and especially  $k = 10$ , we can see some components have very small or zero probability (i.e.  $\pi_2$  of trial 2). It will be unlikely for those components to represent very distinct clusters (i.e. the parameters for  $\pi_2$  and  $\pi_9$  are very similar in trial 2) This can be verified when we perform a TSNE visualisation of the responsibility vector for each of the images (Note that for  $k = 2$ , the responsibility vector is displayed). We can see that for large  $k$ , qualitatively the number of clusters no longer matches the  $k$  value, indicating that some clusters are redundant. For example for  $k = 7$  and  $k = 10$  we can only qualitatively see four or five clusters with TSNE.

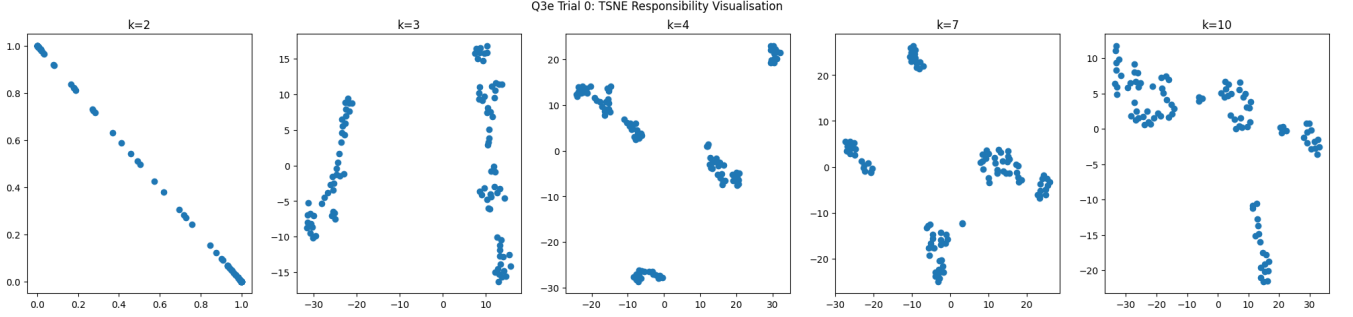


Figure 11: TSNE Visualisation of Image responsibilities: Trial 0

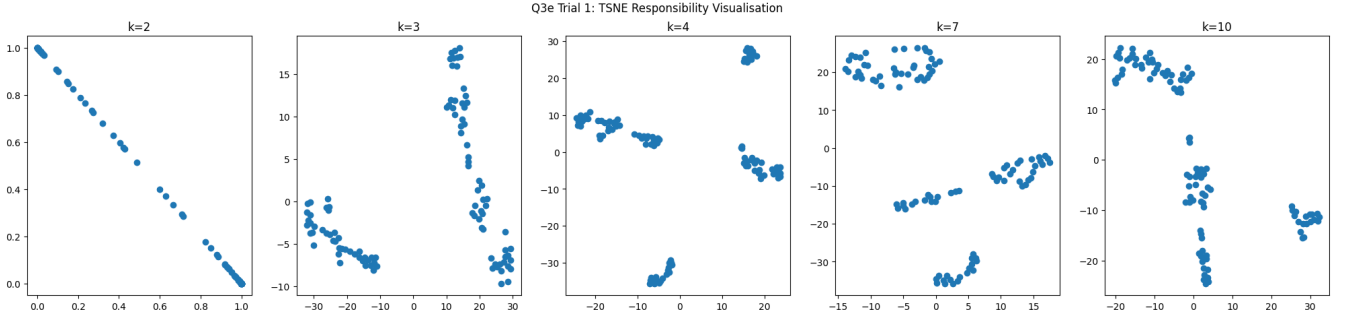


Figure 12: TSNE Visualisation of Image responsibilities: Trial 1

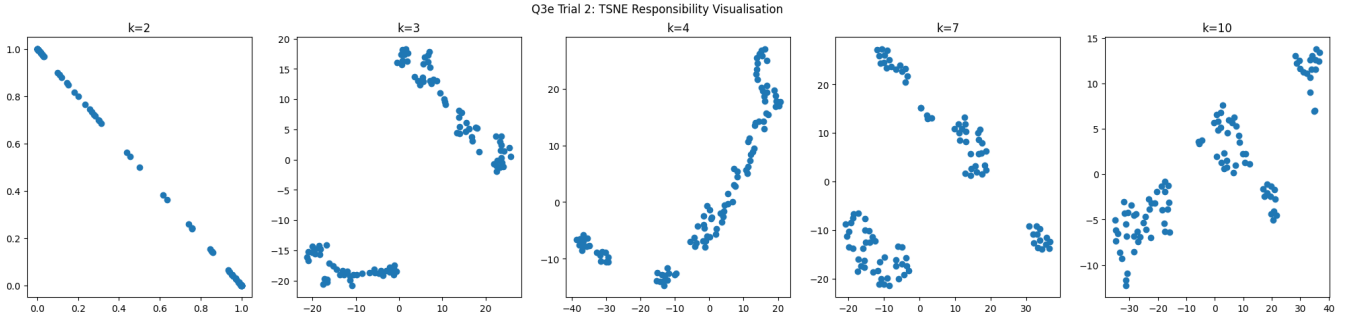


Figure 13: TSNE Visualisation of Image responsibilities: Trial 2

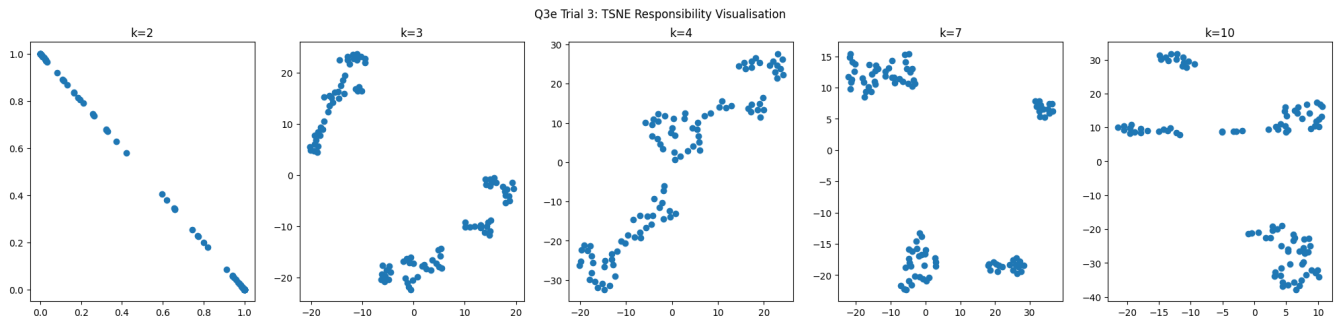


Figure 14: TSNE Visualisation of Image responsibilities: Trial 3

Improvements to the model could include searching for an optimal  $k$  by maximising the log posterior with regularisation on the magnitude of  $k$  to balancing maximising log posterior with minimising model complexity. Additionally, adding a prior on the responsibility components can be helpful to ensure non-zero mixing components unlike the components visualised here. This could help promote more meaningful clusters as  $k$  increases.

[BONUS] Express the log-likelihoods obtained in bits and relate these numbers to the length of the naive encoding of these binary data. How does your number compare to gzip (or another compression algorithm)? Why the difference? [5 marks]

[BONUS] Consider the total cost of encoding both the model parameters and the data given the model. How does this total cost compare to gzip (or similar)? How does it depend on  $K$ ? What might this tell you? [5 marks]

### Question 5

- (a) The formulae for the ML estimates of  $P(s_i = \alpha | s_{i-1} = \beta) = \Psi(\alpha, \beta)$ :

$$\Psi(\alpha, \beta) = \frac{N_{s_i, s_{i-1}}}{N_{s_{i-1}}}$$

where  $N_{s_i, s_{i-1}}$  is the count of the number of occurrences of the pair  $(s_i, s_{i-1})$ , where  $s_{i-1}$  is followed by  $s_i$  and  $N_{s_{i-1}}$  is the number of occurrences of  $s_{i-1}$ .

Moreover, the stationary distribution  $\phi$  can be calculated using the power method:

- (i) Initialise any  $\phi_0 \in \mathbb{R}^{53 \times 1}$
- (ii) Repeat  $\phi_{i+1} = \Psi \phi_i$
- (iii) Terminate when  $\phi_{i+1} - \phi_i < \epsilon$

where  $\Psi \in \mathbf{R}^{53 \times 53}$  containing the transition probabilities,  $\Psi_{i,j} = P(\alpha_j | \alpha_i)$  where  $\alpha_i$  is the  $i^{th}$  symbol and  $\alpha_j$  is the  $j^{th}$  symbol, and  $\epsilon$  is some small number indicating sufficient convergence of the distribution to be considered stationary. The function  $\phi(\gamma)$  is simply the index of  $\gamma$  in the vector  $\phi$ .

The transition matrix  $\Psi$ :

[illegible]

(Apologies for the tiny font, latex was being difficult)

The invariant distribution  $\phi$ :

<i>Symbol</i>	<i>Probability</i>
=	1.7e-05
space	1.7e-01
-	6.1e-04
,	1.2e-02
;	3.9e-04
:	2.9e-04
!	6.0e-04
?	4.7e-04
/	1.9e-05
.	7.7e-03
,	1.9e-05
double quotes	2.4e-05
(	2.3e-04
)	2.2e-04
[	1.7e-05
]	1.7e-05
*	1.1e-04
0	6.9e-05
1	1.4e-04
2	6.0e-05
3	3.4e-05
4	2.3e-05
5	3.2e-05
6	3.2e-05
7	2.8e-05
8	7.6e-05
9	2.6e-05
a	6.6e-02
b	1.1e-02
c	2.0e-02
d	3.8e-02
e	1.0e-01
f	1.8e-02
g	1.6e-02
h	5.4e-02
i	5.6e-02
j	8.5e-04
k	6.4e-03
l	3.1e-02
m	2.0e-02
n	5.9e-02
o	6.2e-02
p	1.5e-02
q	7.7e-04
r	4.7e-02
s	5.2e-02
t	7.2e-02
u	2.1e-02
v	8.5e-03
w	1.9e-02
x	1.4e-03
y	1.5e-02
z	7.4e-04

- (b) The latent variables  $\sigma(s)$  for different symbols  $s$  are not independent. This is because by choosing an encoding for one symbol  $e = \sigma(s)$ , the encoding for a second symbol  $\sigma(s')$  cannot be  $e$ . We have 53 symbols but only 52 degrees of freedom, because once we have defined the encoding for 52 symbols, the encoding for the 53<sup>rd</sup> symbol cannot be chosen. Thus, there exists a dependence between the symbols for a given  $\sigma$ .

The joint probability of the encrypted text  $e_1 e_2 \dots e_n$  given  $\sigma$ :

$$P(e_1, e_2, \dots, e_n | \sigma) = \phi(\gamma = \sigma^{-1}(e_1)) \prod_{i=2}^n \psi(\alpha = \sigma^{-1}(e_i), \beta = \sigma^{-1}(e_{i-1}))$$

because  $\sigma$  is the encoding function, mapping a symbol  $s$  into the encoded symbol  $e$ , we require  $\sigma^{-1}$  the decoding function mapping the encoded symbol  $e$  back to  $s$ .

- (c) The proposal probability  $S(\sigma \rightarrow \sigma')$  depends on the permutations of  $\sigma$  and  $\sigma'$ . Our proposal generating process restricts us to choose a proposal  $\sigma'$  that differs from  $\sigma$  only at *two* spots:

$$\sigma'(s^i) = \sigma(s^j)$$

$$\sigma'(s^j) = \sigma(s^i)$$

for any two symbols  $s^i$  and  $s^j$  of the 53 possible symbols ( $s^i \neq s^j$ ).

Therefore, if the above doesn't hold for  $\sigma'$ ,  $S(\sigma \rightarrow \sigma') = 0$ . From  $\sigma$  there are  $\binom{53}{2}$  possible proposal  $\sigma'$ 's with the above property. Because we are assuming a uniform prior distribution over  $\sigma$ 's, the transition probability of a  $\sigma'$  that satisfies the above property is  $S(\sigma \rightarrow \sigma') = \frac{1}{\binom{53}{2}}$ .

The MH acceptance probability is given as:

$$A(\sigma \rightarrow \sigma' | \mathcal{D}) = \min\left\{1, \frac{S(\sigma' \rightarrow \sigma) P(\sigma' | \mathcal{D})}{S(\sigma \rightarrow \sigma') P(\sigma | \mathcal{D})}\right\}$$

because  $S(\sigma \rightarrow \sigma')$  is the conditional transition probability of  $\sigma'$  given  $\sigma$  and  $\mathcal{D}$  is our encrypted text  $e_1, e_2, \dots, e_n$ .

$S(\sigma \rightarrow \sigma') = S(\sigma' \rightarrow \sigma)$  for all  $\sigma$  and  $\sigma'$  that differ only at two spots because the probability in this case will always be  $\frac{1}{\binom{53}{2}}$ , we can simplify:

$$A(\sigma \rightarrow \sigma' | \mathcal{D}) = \min\left\{1, \frac{P(\sigma' | \mathcal{D})}{P(\sigma | \mathcal{D})}\right\}$$

From Bayes' Theorem:

$$P(\sigma | \mathcal{D}) = \frac{P(\mathcal{D} | \sigma) P(\sigma)}{\sum_{\sigma'} P(\mathcal{D} | \sigma') P(\sigma')}$$

We are assuming a uniform prior for  $\sigma$ , so  $P(\sigma)$  is a constant and we can simplify further:

$$A(\sigma \rightarrow \sigma' | \mathcal{D}) = \min\left\{1, \frac{P(\mathcal{D} | \sigma')}{P(\mathcal{D} | \sigma)}\right\}$$

This is the acceptance probability for a given proposal  $\sigma'$ . The expression for  $P(\mathcal{D} | \sigma)$  is  $P(e_1, e_2, \dots, e_n | \sigma)$  described in the previous part.

(d) Reporting the current decryption of the first 60 symbols after every 100 iterations:

MH Iteration	Current Decryption
0	6m p2n uamr= jmk pu= batm =j3t' 2'=eq p2 s)9'=rjb' p' q'n
100	er pl losrua= drk po=a bstra=dita lad=n pl dfa= udba pa no
200	e r l loiruh dnr nola bitraadla ladhlp nl xdyah udaba na po
300	e r l loirnua srw nola bitravsdia lasvp nl xsynay usha na po
400	er vd dsir,an orw vsna bitranolta daony vd upoab ,oba va ys
500	er c, sirdan or. esna bitranolta .aony c, uophan doba ca ys
600	en ck kyindar on. cyra bitnarolta kairs ck uophar doba ca
700	en pk klindar on. pla bitnaroyta kairs pk uochar doba pa sl
800	en p, londar in. pla botariyta aoirs p, fighar diba pa sl
900	en pu ulondar in. pura botariyeta uairis pu fighar diba pa sl
1000	en pl houndar in. pura botariyeta lairs pl fighar diba pa su
1100	en pl houdar in. pura cotnarixta lairs pl fighar dica pa su
1200	en pk kuondar ind pura commarixna kairs pk fighar dica pa su
1300	en ck kuondar ind cura pommarixna kairs ck fighar dipa ca so
1400	en ck komdar ind cora punmarixna kairs ck fighar dipa ca so
1500	en ck komdar ind cora vummarixna kairs ck fitnar diva ca so
1600	en ck komdar ind cora vumarixna kairs ck fitnar diva ca so
1700	an ck kounder ind core vumerixme keirs ck fither dive ce so
1800	an ck koumlr ind core vumerixme keirs ck fither live ce so
1900	an ck koumlr ind core vumerixme keirs ck fither live ce so
2000	an ck koumlr ind core vumerixme keirs ck fither live ce so
2100	an ck koumlr ind core vumerixme keirs ck fither live ce so
2200	an ck koumlr ind core vumerixme keirs ck fither live ce so
2300	an ck kougner ind core vumerixme keirs ck fither give ce so
2400	an ck kougner ind core vulherixe keirs ck fither give ce so
2500	an mk kougner ind more vulherixe keirs mk fithet give me so
2600	an mk kougner ind more vulperike keirs mk fithet give me so
2700	an mk kougner ind more vulperike keirs mk fithet give me so
2800	an mk kougner ind more vulperike keirs mk fithet give me so
2900	an mk kougner ind more vulperike keirs mk fithet give me so
3000	an mk kougner ind more vulperike keirs mk fithet give me so
3100	an mf founger ind more vulperife feirs mf kitther give me so
3200	an mf founger ind more vulperife feirs mf kitther give me so
3300	an mf founger ind more vulperife feirs mf kitther give me so
3400	an mf founger ind more vulperife feirs mf kitther give me so
3500	an mf founger ind more vulperife feirs mf kitther give me so
3600	an mf founger ind more vulperife feirs mf kitther give me so
3700	an mf founger ind more vulperife feirs mf kitther give me so
3800	an mf founger ind more vulperife feirs mf kitther give me so
3900	in mf founger and more vulnerable fears mf kather gave me so
4000	in mf founger and more vulnerable fears mf kather gave me so
4100	in mf founger and more vulnerable fears mf kather gave me so
4200	in mf founger and more vulnerable fears mf kather gave me so
4300	in mf founger and more vulnerable fears mf kather gave me so
4400	in mf founger and more vulnerable fears mf yather gave me so
4500	in mf founger and more vulnerable fears mf yather gave me so
4600	in mf founger and more vulnerable fears mf yather gave me so
4700	in mf founger and more vulnerable fears mf yather gave me so
4800	in mf founger and more vulnerable fears mf yather gave me so
4900	in mf founger and more vulnerable fears mf yather gave me so
5000	in mf founger and more vulnerable fears mf yather gave me so
5100	in mf founger and more vulnerable fears mf yather gave me so
5200	in mf founger and more vulnerable fears mf yather gave me so
5300	in my younger and more vulnerable years my father gave me so
5400	in my younger and more vulnerable years my father gave me so
5500	in my younger and more vulnerable years my father gave me so
5600	in my younger and more vulnerable years my father gave me so
5700	in my younger and more vulnerable years my father gave me so
5800	in my younger and more vulnerable years my father gave me so
5900	in my younger and more vulnerable years my father gave me so
6000	in my younger and more vulnerable years my father gave me so
6100	in my younger and more vulnerable years my father gave me so
6200	in my younger and more vulnerable years my father gave me so
6300	in my younger and more vulnerable years my father gave me so
6400	in my younger and more vulnerable years my father gave me so
6500	in my younger and more vulnerable years my father gave me so
6600	in my younger and more vulnerable years my father gave me so
6700	in my younger and more vulnerable years my father gave me so
6800	in my younger and more vulnerable years my father gave me so
6900	in my younger and more vulnerable years my father gave me so
7000	in my younger and more vulnerable years my father gave me so
7100	in my younger and more vulnerable years my father gave me so
7200	in my younger and more vulnerable years my father gave me so
7300	in my younger and more vulnerable years my father gave me so
7400	in my younger and more vulnerable years my father gave me so
7500	in my younger and more vulnerable years my father gave me so
7600	in my younger and more vulnerable years my father gave me so
7700	in my younger and more vulnerable years my father gave me so
7800	in my younger and more vulnerable years my father gave me so
7900	in my younger and more vulnerable years my father gave me so
8000	in my younger and more vulnerable years my father gave me so
8100	in my younger and more vulnerable years my father gave me so
8200	in my younger and more vulnerable years my father gave me so
8300	in my younger and more vulnerable years my father gave me so
8400	in my younger and more vulnerable years my father gave me so
8500	in my younger and more vulnerable years my father gave me so
8600	in my younger and more vulnerable years my father gave me so
8700	in my younger and more vulnerable years my father gave me so
8800	in my younger and more vulnerable years my father gave me so
8900	in my younger and more vulnerable years my father gave me so
9000	in my younger and more vulnerable years my father gave me so
9100	in my younger and more vulnerable years my father gave me so
9200	in my younger and more vulnerable years my father gave me so
9300	in my younger and more vulnerable years my father gave me so
9400	in my younger and more vulnerable years my father gave me so
9500	in my younger and more vulnerable years my father gave me so
9600	in my younger and more vulnerable years my father gave me so
9700	in my younger and more vulnerable years my father gave me so
9800	in my younger and more vulnerable years my father gave me so
9900	in my younger and more vulnerable years my father gave me so
10000	in my younger and more vulnerable years my father gave me so

The corresponding  $\sigma$ :

<b>s</b>	<b><math>\sigma(s)</math></b>
=	{
space	x
-	h
,	,
;	l
:	n
!	r
?	e
/	f
.	b
'	3
double quotes	5
(	4
)	9
[	i
]	o
*	l
0	z
1	m
2	c
3	/
4	;
5	.
6	*
7	k
8	:
9	q
a	)
b	2
c	-
d	7
e	'
f	0
g	s
h	!
i	]
j	(
k	8
l	y
m	v
n	d
o	=
p	space
q	6
r	g
s	t
t	double quotes
u	p
v	j
w	a
x	u
y	?
z	w

To help with chain initialisation, 10000 different  $\sigma$ 's were randomly and independently sampled. The  $\sigma$  providing the best log-likelihood was chosen as the starting point for the MH chain and algorithm was then run for 10000 iterations. Moreover, ten different trials were performed, where the trial with the best log-likelihood is displayed.

## The Python code for the MH sampler:

```

1 from typing import Dict, List, Tuple
2
3 import numpy as np
4 import pandas as pd
5 from sklearn.preprocessing import normalize
6
7 from src.constants import DEFAULT_SEED
8
9
10 def _convert_to_scientific_notation(x: float) -> str:
11     """
12     Convert value to string in scientific notation
13     :param x: value to convert
14     :return: string of x in scientific notation
15     """
16     return "{:.1e}".format(float(x))
17
18
19 class Decrypter:
20     def __init__(self, decryption_dict: Dict[str, str]) -> None:
21         """
22         Decrypter containing the mapping a symbol to its encrypted symbol
23         :param decryption_dict:
24         """
25         self.decryption_dict = decryption_dict
26
27     def decrypt(self, encrypted_message: str) -> str:
28         """
29         Decrypts an encrypted message using the decryption dictionary
30         :param encrypted_message: the encrypted message to decrypt
31         :return: decrypted message
32         """
33         return "".join([self.decryption_dict[x] for x in encrypted_message])
34
35     @property
36     def table(self) -> pd.DataFrame:
37         """
38         Generate table containing symbol decryptions
39         :return: pandas table of decryptions
40         """
41         decrypter_table = pd.DataFrame(
42             self.decryption_dict.items(), columns=["s", "sigma(s)"]
43         )
44         decrypter_table[decrypter_table == " "] = "space"
45         decrypter_table[decrypter_table == "' '"] = "double quotes"
46         return decrypter_table.set_index("s")
47
48
49 class Statistics:
50     def __init__(
51         self,
52         training_text: str,
53         symbols: List[str],
54         invariant_stopping_epsilon: float = 5e-20,
55     ) -> None:
56         """
57         Statistics for text
58         :param training_text: training text for calculating transition and invariant probability
59         :param symbols: symbols in the training text
60         :param invariant_stopping_epsilon: stopping condition for constructing the invariant distribution
61         """
62         self.training_text = training_text
63         self.symbols = symbols
64         self.num_symbols = len(symbols)
65         self.symbols_dict = self._construct_symbols_dictionary(symbols)
66         self.transition_matrix = self._construct_transition_matrix(
67             training_text, self.symbols_dict
68         )
69         self.invariant_distribution = self._approximate_invariant_distribution(
70             invariant_stopping_epsilon
71         )
72         self.log_transition_matrix = np.log(self.transition_matrix)
73         self.log_invariant_distribution = np.log(self.invariant_distribution)
74
75     @property
76     def list_of_symbols_for_df(self) -> List[str]:
77         """
78         Replace certain symbols to prepare for dataframe
79         :return: list of symbols with some replacements
80         """
81         x = self.symbols.copy()
82         x[x.index(" ")] = "space"
83         x[x.index("' '")] = "double quotes"
84         return x
85
86     @property
87     def transition_table(self) -> pd.DataFrame:
88         """
89         Generate a table containing transition probabilities
90         :return: transition probabilities
91         """
92         df_transitions = pd.DataFrame(
93             data=self.transition_matrix,
94             columns=self.list_of_symbols_for_df,

```



```

95         )
96         df_transitions.index = self.list_of_symbols_for_df
97         return df_transitions.applymap(_convert_to_scientific_notation)
98
99     @property
100     def invariant_distribution_table(self) -> pd.DataFrame:
101         """
102         Generate a table containing invariant distribution probabilities
103         :return: invariant distribution probabilities
104         """
105         df = (
106             pd.DataFrame(
107                 data=self.invariant_distribution.reshape(1, -1),
108                 columns=self.list_of_symbols_for_df,
109             )
110             .applymap(_convert_to_scientific_notation)
111             .transpose()
112             .reset_index()
113         )
114         df.columns = ["Symbol", "Probability"]
115         return df.set_index("Symbol")
116
117     @staticmethod
118     def _construct_symbols_dictionary(symbols: List[str]) -> Dict[str, int]:
119         """
120         Construct a dictionary mapping each symbol to an integer
121         :param symbols: list of symbols to map
122         :return: symbol to integer mapping
123         """
124         return {k: v for v, k in enumerate(symbols)}
125
126     def _construct_transition_matrix(
127         self, text: str, symbols_dict: Dict[str, int]
128     ) -> np.ndarray:
129         """
130         Constructs the transition matrix for a given text
131         :param text: string to calculate transition matrix with
132         :param symbols_dict: dictionary mapping symbol to a dictionary
133         :return:
134         """
135         # initialise with ones to ensure ergodicity
136         transition_matrix = np.ones((self.num_symbols, self.num_symbols))
137         for i in range(1, len(text)):
138             # check symbols are valid
139             if text[i] in symbols_dict and text[i - 1] in symbols_dict:
140                 transition_matrix[symbols_dict[text[i - 1]], symbols_dict[text[i]]] += 1
141         # normalise to get transition probabilities
142         transition_matrix = normalize(transition_matrix, axis=0, norm="l1")
143         return transition_matrix
144
145     def _approximate_invariant_distribution(
146         self, invariant_stopping_epsilon: float
147     ) -> np.ndarray:
148         """
149         Approximate the invariant distribution with the power method
150         :param invariant_stopping_epsilon: stopping condition for constructing the invariant distribution
151         :return: the invariant distribution as a vector (number of symbols, 1)
152         """
153         invariant_distribution = np.zeros((self.num_symbols, 1))
154         previous_invariant_distribution = invariant_distribution.copy()
155
156         # make sure it's a proper distribution that sums to one
157         invariant_distribution[0] = 1
158
159         while (
160             np.linalg.norm(invariant_distribution - previous_invariant_distribution)
161             > invariant_stopping_epsilon
162         ):
163             previous_invariant_distribution = invariant_distribution.copy()
164             invariant_distribution = self.transition_matrix @ invariant_distribution
165         return invariant_distribution
166
167     def log_transition_probability(self, alpha: str, beta: str) -> float:
168         """
169         Look up the log probability of the transition from symbol alpha to beta
170         :param alpha: symbol that is being transitioned from
171         :param beta: symbol that is being transitioned to
172         :return: probability of transition
173         """
174         return self.log_transition_matrix[
175             self.symbols_dict[beta], self.symbols_dict[alpha]
176         ]
177
178     def log_invariant_probability(self, gamma: str) -> float:
179         """
180         Look up the log probability of a symbol with respect to the invariant distribution
181         :param gamma: symbol to query
182         :return: log probability of the symbol
183         """
184         return self.log_invariant_distribution[self.symbols_dict[gamma]].item()
185
186     def compute_log_probability(self, text: str) -> float:
187         """
188         Compute the log probability of a given text containing symbols
189         :param text: text to compute log probability for
190         :return: log probability of the text

```

```

191     """
192     log_probability = self.log_invariant_probability(text[0])
193     for i in range(1, len(text)):
194         log_probability += self.log_transition_probability(text[i], text[i - 1])
195     return log_probability
196
197
198 class MetropolisHastingsDecryption:
199     def __init__(self, symbols: List[str]):
200         """
201         Metropolis Hastings MCMC for Decryption
202         :param symbols: set of symbols to decrypt
203         """
204         self.symbols = symbols
205
206     def generate_random_decrypter(self) -> Decrypter:
207         """
208         Generates a random decrypter
209         :return: a Decrypter instantiation
210         """
211         return Decrypter(
212             {
213                 self.symbols[i]: self.symbols[x]
214                 for i, x in enumerate(
215                     np.random.permutation(np.arange(len(self.symbols)))
216                 )
217             }
218         )
219
220     @staticmethod
221     def generate_proposal_decryption(decrypter: Decrypter) -> Decrypter:
222         """
223         Generate a proposal decrypter by randomly swapping two of the decryption mappings
224         :param decrypter: the decrypter used to generate the proposal
225         :return: a proposal decrypter
226         """
227         x1 = np.random.choice(list(decrypter.decryption_dict.keys()))
228         x2 = np.random.choice(list(decrypter.decryption_dict.keys()))
229         proposal_decryption = decrypter.decryption_dict.copy()
230         proposal_decryption[x2], proposal_decryption[x1] = (
231             decrypter.decryption_dict[x1],
232             decrypter.decryption_dict[x2],
233         )
234         return Decrypter(proposal_decryption)
235
236     @staticmethod
237     def _choose_decrypter(
238         statistics: Statistics,
239         encrypted_message: str,
240         current_decrypter: Decrypter,
241         proposal_decrypter: Decrypter,
242     ) -> Decrypter:
243         """
244         Choose between the current and proposal decrypter
245         :param statistics: Statistics instantiation for calculating log probabilities
246         :param encrypted_message: the encrypted message
247         :param current_decrypter: the current decrypter
248         :param proposal_decrypter: the proposal decrypter
249         :return:
250         """
251         # calculate log probabilities
252         current_log_probability = statistics.compute_log_probability(
253             text=current_decrypter.decrypt(encrypted_message),
254         )
255         proposal_log_probability = statistics.compute_log_probability(
256             text=proposal_decrypter.decrypt(encrypted_message),
257         )
258
259         # calculate acceptance probability
260         acceptance_probability = np.min(
261             [1, np.exp(proposal_log_probability - current_log_probability)]
262         )
263         # choose decrypter using the acceptance probability
264         return np.random.choice(
265             [current_decrypter, proposal_decrypter],
266             p=[1 - acceptance_probability, acceptance_probability],
267         )
268
269     def _find_good_starting_decrypter(
270         self,
271         statistics: Statistics,
272         encrypted_message,
273         number_start_attempts,
274     ) -> Decrypter:
275         """
276         Find a good starting decrypter for the sampler by choosing the one with the best log likelihood
277         :param statistics: Statistics instantiation for calculating log probabilities
278         :param encrypted_message: the encrypted message
279         :param number_start_attempts: number of possible starting decrypters to check
280         :return: the best starting decrypter for the sampler
281         """
282         best_log_likelihood = -np.float("inf")
283         best_decrypter = None
284         for _ in range(number_start_attempts):
285             decrypter = self.generate_random_decrypter()
286             if (

```

```

287         statistics.compute_log_probability(
288             text=decrypter.decrypt(encrypted_message)
289         )
290         > best_log_likelihood
291     ):
292         best_decrypter = decrypter
293     return best_decrypter
294
295 def run(
296     self,
297     encrypted_message: str,
298     statistics: Statistics,
299     number_of_mh_loops: int,
300     number_start_attempts: int,
301     log_decryption_interval: int,
302     log_decryption_size: int,
303 ) -> Tuple[Decrypter, List[str]]:
304     """
305     Run the sampler with two steps:
306     1. find a good starting decrypter for the sampler
307     2. run the sampler
308     :param encrypted_message: the encrypted message
309     :param statistics: Statistics instantiation for calculating log probabilities
310     :param number_of_mh_loops: number of loops to run the metropolis hasting sampler
311     :param number_start_attempts: number of possible starting decrypters to check
312     :param log_decryption_interval: number of samples between logging the decrypted message
313     :param log_decryption_size: number of symbols to decrypt when logging the decrypted message
314     :return: a tuple containing the decrypter found from the sampler and the logged decryption message
315     """
316     decrypter = self._find_good_starting_decrypter(
317         statistics, encrypted_message, number_start_attempts
318     )
319     logged_decryption_message = [
320         decrypter.decrypt(encrypted_message)[:log_decryption_size]
321     ]
322     for i in range(1, number_of_mh_loops + 1):
323         if (i + 1) % log_decryption_interval == 0:
324             logged_decryption_message.append(
325                 decrypter.decrypt(encrypted_message)[:log_decryption_size]
326             )
327         proposal_decrypter = self.generate_proposal_decryption(decrypter)
328         decrypter = self._choose_decrypter(
329             statistics, encrypted_message, decrypter, proposal_decrypter
330         )
331     return decrypter, logged_decryption_message
332
333 def _construct_logged_decryptions_table(
334     logged_decryption_message, log_decryption_interval
335 ) -> pd.DataFrame:
336     decrypted_message_iterations_table = pd.DataFrame(
337         [
338             np.arange(0, len(logged_decryption_message)) * log_decryption_interval,
339             logged_decryption_message,
340         ]
341     ).transpose()
342     decrypted_message_iterations_table.columns = ["MH Iteration", "Current Decryption"]
343     return decrypted_message_iterations_table.set_index("MH Iteration")
344
345 def a(
346     symbols: List[str],
347     training_text: str,
348     transition_matrix_path: str,
349     invariant_distribution_path: str,
350 ) -> None:
351     """
352     Produces answers for question 5a
353     :param symbols: symbols in the training text
354     :param training_text: training text for calculating transition and invariant probability
355     :param transition_matrix_path: path to store transition matrix
356     :param invariant_distribution_path: path to store invariant distribution
357     :return:
358     """
359     statistics = Statistics(
360         training_text,
361         symbols,
362     )
363     statistics.transition_table.to_csv(transition_matrix_path)
364     statistics.invariant_distribution_table.to_csv(invariant_distribution_path, sep="|")
365
366 def d(
367     encrypted_message: str,
368     symbols: List[str],
369     training_text: str,
370     number_trials: int,
371     number_of_mh_loops: int,
372     number_start_attempts: int,
373     log_decryption_interval: int,
374     log_decryption_size: int,
375     decryptor_table_path: str,
376     decrypted_message_iterations_table_path: str,
377 ) -> None:
378     """
379     Produces answers for question 5d

```

```

383 :param encrypted_message: the encrypted message
384 :param symbols: symbols in the training text
385 :param training_text: training text for calculating transition and invariant probability
386 :param number_trials: number of times to restart and run the sampler
387 :param number_of_mh_loops: number of loops to run the metropolis hasting sampler
388 :param number_start_attempts: number of possible starting decrypters to check
389 :param log_decryption_interval: number of samples between logging the decrypted message
390 :param log_decryption_size: number of symbols to decrypt when logging the decrypted message
391 :param decryptor_table_path: path to store decrypter mapping table
392 :param decrypted_message_iterations_table_path: path to store logged decryption messages
393 :return:
394 """
395 statistics = Statistics(
396     training_text,
397     symbols,
398 )
399 np.random.seed(DEFAULT_SEED)
400 metropolis_hastings_decryption = MetropolisHastingsDecryption(symbols)
401 decrypters: List[Decrypter] = []
402 log_likelihoods: List[float] = []
403 logged_decryption_messages: List[List[str]] = []
404 decryption_messages = []
405 for i in range(number_trials):
406     (decrypter, logged_decryption_message,) = metropolis_hastings_decryption.run(
407         encrypted_message,
408         statistics,
409         number_of_mh_loops,
410         number_start_attempts,
411         log_decryption_interval,
412         log_decryption_size,
413     )
414     decrypters.append(decrypter)
415     log_likelihoods.append(
416         statistics.compute_log_probability(decrypter.decrypt(encrypted_message))
417     )
418     logged_decryption_messages.append(logged_decryption_message)
419     decryption_messages.append(
420         decrypter.decrypt(encrypted_message)[:log_decryption_size]
421     )
422
423 # sort trials by log likelihood
424 best_trial = np.argmax(log_likelihoods)
425 decrypters[best_trial].table.to_csv(decryptor_table_path, sep="|")
426 df_logged_decryptions = _construct_logged_decryptions_table(
427     logged_decryption_messages[best_trial], log_decryption_interval
428 )
429 df_logged_decryptions.to_csv(decrypted_message_iterations_table_path, sep="|")

```

src/solutions/q5.py

- (e) When some values of  $\Psi(\alpha, \beta) = 0$ , this affects the ergodicity of the chain. An ergodic chain is one that is irreducible (i.e. all possible transitions between symbols have probability greater than zero). If  $\Psi(\alpha, \beta) = 0$ , this means that there is zero probability that  $\beta$  will transition to  $\alpha$ , breaking our definition. To restore ergodicity, we can add a small transition probability between all symbols of the chain. This essentially acts as a prior, stating that the probability of a symbol to transition to any other symbol (including itself) should never be zero.
- (f) Analyse this approach to decoding. For instance, would symbol probabilities alone (rather than transitions) be sufficient? If we used a second order Markov chain for English text, what problems might we encounter? Will it work if the encryption scheme allows two symbols to be mapped to the same encrypted value? Would it work for Chinese with  $> 10000$  symbols? [13 marks]

If we were to use symbol probabilities alone for decoding, the joint probability would be:

$$P(e_1, e_2, \dots, e_n | \sigma) = \prod_{i=1}^n P(\sigma^{-1}(e_i))$$

the product of the likelihoods of the decoded letters. In this case, the optimal decoding would simply replace the most frequent symbols in the encrypted message with the most frequent symbols in the training text. This is much more difficult because each letter is assumed to be independent of its neighbours. For a first order Markov chain, we exploit the structure of language by considering pairs of letters. Assuming that as the training text size approaches infinity and the size of the encrypted message also approaches infinity, that the two will have the same symbol frequency and that the probability of each symbol is unique, (i.e. two different decodings can't have the same likelihood), then using symbol probabilities alone should theoretically work. However, in practise we would unlikely to be able to make these assumptions about symbol frequencies from the size of our training set and encrypted message.

A second-order chain should also work in theory. However, this approach is probably practically more difficult for finding a suitable decoding. This is because our transition matrix would contain  $N^3$ , where  $N$  is the number of symbols, to account for all possible second order transitions. Our training text would need to increase quadratically to maintain the same ratio of possible transitions to example transitions (number of second order transitions in a text of length  $N$  is  $N - 2$  and third order its  $N - 3$ ).

For an encryption scheme where two symbols map to the same encrypted value:

$$\exists \alpha, \beta, \sigma(\alpha) = \sigma(\beta), \alpha \neq \beta$$

this approach can become much more complicated. Our  $\sigma^{-1}(e)$  is ill-defined, and therefore how we computing the joint probability of the encrypted text is no longer immediately clear. Moreover, generating proposal encodings is not as simple as swapping the encryption for two symbols. This is because we do not know which two symbols map to the same encrypted symbol and simply swapping would preserve the same collision mapping of the current encoding. Overall, many changes would need to be made to the approach to accommodate for these complications. It is not immediately obvious how current approach could work for this case.

If we used this approach for Chinese with  $\geq 10000$  symbols, we would be attempting to solve the same problem but with  $N \geq 10000$  instead of  $N = 53$ . Similar to the second order Markov chain, although this is theoretically possible, it would require a transition matrix of size  $\geq 10000^2$  which is quite impractical. An alternative set up could be with using Chinese phonetics, for which there are likely much fewer than 10000, however this would require a mapping from a phonetic to an encrypted phonetic.

## Question 7

- (a) To find the local extrema of the function  $f(x, y) = x + 2y$  subject to the constraint  $y^2 + xy = 1$ , first we define  $g(x, y)$ :

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

where  $g(x, y) = 0$  is an equivalent representation of the given constraint.

We can therefore construct the optimisation problem:

$$\min_{\mathbf{x}} f(\mathbf{x})$$

such that  $g(\mathbf{x}) = 0$  and  $\mathbf{x} := [x, y]^T$ .

We can calculate  $\nabla f(\mathbf{x})$ :

$$\nabla f(\mathbf{x}) = \left[ \frac{\partial}{\partial x}(x + 2y), \frac{\partial}{\partial y}(x + 2y) \right]^T$$

$$\nabla f(\mathbf{x}) = [1, 2]^T$$

and calculating  $\nabla g(\mathbf{x})$ :

$$\nabla g(\mathbf{x}) = \left[ \frac{\partial}{\partial x}(y^2 + xy - 1), \frac{\partial}{\partial y}(y^2 + xy - 1) \right]^T$$

$$\nabla g(\mathbf{x}) = [y, 2y + x]^T$$

Solving the constraint optimisation problem with Lagrange multipliers, we set up the equations:

$$\nabla f(\mathbf{x}) + \lambda \nabla g(\mathbf{x}) = \mathbf{0}$$

and

$$g(\mathbf{x}) = 0$$

Giving us the three equations:

$$1 + \lambda y = 0$$

$$2 + \lambda(2y + x) = 0$$

$$y^2 + xy - 1 = 0$$

Substituting  $y = \frac{-1}{\lambda}$  from the first equation into the second equation:

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

$$\frac{-x}{y} = 0$$

We see that  $x = 0$ . Solving for  $y$  in our third equation with  $x = 0$ :

$$y^2 - 1 = 0$$

We see that  $y = \pm 1$  and from the first equation  $\lambda \mp 1$ .

The local extrema are  $(x = 0, y = 1)$  when our  $\lambda = -1$  and  $(x = 0, y = -1)$  when our  $\lambda = 1$ .

(b)

(i) Given that  $g(a) = \ln(a)$ , we want to transform this to the form  $f(x, a) = 0$ :

$$x = \ln(a)$$

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

Thus,

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

(ii) We know that for Newton's method's

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

where  $f(x_n) = \exp(x_n) - a$

We can calculate:

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

Assuming we can evaluate  $\exp(x)$ , our update equation:

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

Simplifying:

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



## Question 8

(a) For:

$$\sup_{\{\mathbf{x} \in \mathbb{R}^n\}} R_A(\mathbf{x})$$

where  $R_A(\mathbf{x}) = \frac{\mathbf{x}^T \mathbf{A} \mathbf{x}}{\|\mathbf{x}\|^2}$ , we want to show that a maximum is attained.

To do this, we will show the above optimisation can be equivalently formulated:

$$\sup_{\{\mathbf{x} \in \mathbb{R}^n \mid \|\mathbf{x}\|=1\}} R_A(\mathbf{x})$$

We begin by considering any  $\mathbf{w} \in \mathbb{R}^n$  and let  $\mathbf{x} = \frac{\mathbf{w}}{\|\mathbf{w}\|}$ . Because  $\|\mathbf{x}\| = 1$  we can substitute:

$$\sup_{\{\frac{\mathbf{w}}{\|\mathbf{w}\|} \in \mathbb{R}^n \mid \|\frac{\mathbf{w}}{\|\mathbf{w}\|}\|=1\}} R_A(\mathbf{x}) = \sup_{\mathbf{x} \in \mathbb{R}^n \mid \|\mathbf{x}\|=1} \frac{\mathbf{w}^T \mathbf{A} \mathbf{w} \|\mathbf{w}\|^2}{\|\mathbf{w}\|^2 \mathbf{w}^T \mathbf{w}}$$

where  $\|\mathbf{x}\|^2 = \mathbf{x}^T \mathbf{x}$ .

The set  $\{\frac{\mathbf{w}}{\|\mathbf{w}\|} \in \mathbb{R}^n \mid \|\frac{\mathbf{w}}{\|\mathbf{w}\|}\| = 1\}$  holds for all  $\mathbf{w} \in \mathbb{R}^n$  so we can rewrite:

$$\sup_{\{\mathbf{w} \in \mathbb{R}^n\}} \frac{\mathbf{w}^T \mathbf{A} \mathbf{w} \|\mathbf{w}\|^2}{\|\mathbf{w}\|^2 \mathbf{w}^T \mathbf{w}}$$

We can simplify the expression:

$$\begin{aligned} & \sup_{\{\mathbf{w} \in \mathbb{R}^n\}} \frac{\mathbf{w}^T \mathbf{A} \mathbf{w}}{\mathbf{w}^T \mathbf{w}} \\ & \sup_{\{\mathbf{w} \in \mathbb{R}^n\}} R_A(\mathbf{w}) \end{aligned}$$

and recover our original optimisation problem by letting  $\mathbf{x} = \mathbf{w}$ , showing that it is equivalent to the supremum over the unit sphere. Assuming the set containing the unit sphere is compact, the extreme value theory of calculus states that  $\sup_{\{\mathbf{x} \in \mathbb{R}^n \mid \|\mathbf{x}\|=1\}} R_A(\mathbf{x})$  is attained so equivalently  $\sup_{\{\mathbf{x} \in \mathbb{R}^n\}} R_A(\mathbf{x})$  is attained as required.

(b) We can now reformulate the optimisation as:

$$\sup_{\{\mathbf{x} \in \mathbb{R}^n \mid \|\mathbf{x}\|=1\}} \frac{\mathbf{x}^T \mathbf{A} \mathbf{x}}{\|\mathbf{x}\|^2}$$

But because  $\|\mathbf{x}\| = 1$ , we can equivalently write:

$$\sup_{\{\mathbf{x} \in \mathbb{R}^n \mid \|\mathbf{x}\|=1\}} \mathbf{x}^T \mathbf{A} \mathbf{x}$$

Thus, showing  $\mathbf{x}^T \mathbf{A} \mathbf{x} \leq \lambda_1$  will be equivalent to showing  $R_A(\mathbf{x}) \leq \lambda_1$  for  $\|\mathbf{x}\| = 1$ . We know that for all  $\mathbf{x} \in \mathbb{R}^n$ :

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

so we can write:

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

Given that  $\xi_i$  are eigenvectors of  $\mathbf{A}$  corresponding to eigenvalues  $\lambda_i$ :

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

Given that the eigenvectors  $\xi_i$  form an orthonormal basis, we know that  $\xi_i^T \xi_j = 0$  when  $i \neq j$  and  $\xi_i^T \xi_i = 1$  when  $i = j$ , so:

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

We know that  $\|\mathbf{x}\|^2 = 1$  so  $\|\mathbf{x}\|^2 = \sum_{j=1}^n x_j^2 = \sum_{j=1}^n (\xi_j^T \mathbf{x})^2 = 1$ . Thus the quantity  $\sum_{i=1}^n \lambda_i (\xi_i^T \mathbf{x})^2$  is simply a weighted average of  $\lambda_i$ 's and we can write:

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

where  $\lambda_1$  is the largest eigenvalue of eigenvalues  $\lambda_i$ . Therefore,  $R_A(\mathbf{x}) \leq \lambda_1$  as required.

(c) Given that  $\lambda_j < \lambda_1 \forall j > k$  and  $\mathbf{x} \in \text{span}\{\xi_{k+1}, \dots, \xi_n\}$ , we can rewrite  $\mathbf{x}$ :

$$\mathbf{x} = \sum_{i=k+1}^n (\xi_i^T \mathbf{x}) \xi_i$$

From the same argument as (b) we can bound  $\mathbf{x}^T \mathbf{A} \mathbf{x}$ :

$$\mathbf{x}^T \mathbf{A} \mathbf{x} = \sum_{i=k+1}^n \lambda_i (\xi_i^T \mathbf{x})^2 \leq \max\{\lambda_{k+1}, \dots, \lambda_n\}$$

But given that the maximum eigenvalue  $\lambda_1$  is not contained in  $\{\lambda_{k+1}, \dots, \lambda_n\}$ :

$$\max\{\lambda_{k+1}, \dots, \lambda_n\} < \lambda_1$$

and therefore,

$$R_A(\mathbf{x}) < \lambda_1$$

as required.

## Appendix 1: constants.py

```
1 import os
2
3 DATA_FOLDER = "data"
4
5 BINARY_DIGITS_FILE_PATH = os.path.join(DATA_FOLDER, "binarydigits.txt")
6 MESSAGE_FILE_PATH = os.path.join(DATA_FOLDER, "message.txt")
7 SYMBOLS_FILE_PATH = os.path.join(DATA_FOLDER, "symbols.txt")
8 TRAINING_TEXT_FILE_PATH = os.path.join(DATA_FOLDER, "war_and_peace.txt")
9
10 OUTPUTS_FOLDER = "outputs"
11
12 DEFAULT_SEED = 0
```

src/constants.py

## Appendix 2: main.py

```
1 import os
2
3 import numpy as np
4
5 from src.constants import (
6     BINARY_DIGITS_FILE_PATH,
7     MESSAGE_FILE_PATH,
8     OUTPUTS_FOLDER,
9     SYMBOLS_FILE_PATH,
10    TRAINING_TEXT_FILE_PATH,
11 )
12 from src.solutions import q1, q2, q3, q5
13
14 if __name__ == "__main__":
15     if not os.path.exists(OUTPUTS_FOLDER):
16         os.makedirs(OUTPUTS_FOLDER)
17     x = np.loadtxt(BINARY_DIGITS_FILE_PATH)
18
19     # Question 1
20     Q1.OUTPUT_FOLDER = os.path.join(OUTPUTS_FOLDER, "q1")
21     if not os.path.exists(Q1.OUTPUT_FOLDER):
22         os.makedirs(Q1.OUTPUT_FOLDER)
23     q1.d(
24         x,
25         figure_path=os.path.join(Q1.OUTPUT_FOLDER, "q1d.png"),
26         figure_title="Q1d: Maximum Likelihood Estimate",
27     )
28     q1.e(
29         x,
30         alpha=3,
31         beta=3,
32         figure_path=os.path.join(Q1.OUTPUT_FOLDER, "q1e"),
33         figure_title="Q1e: Maximum A Prior",
34     )
35
36     # Question 2
37     Q2.OUTPUT_FOLDER = os.path.join(OUTPUTS_FOLDER, "q2")
38     if not os.path.exists(Q2.OUTPUT_FOLDER):
39         os.makedirs(Q2.OUTPUT_FOLDER)
40     q2.c(x, table_path=os.path.join(Q2.OUTPUT_FOLDER, "q2c.csv"))
41
42     # Question 3
43     Q3.OUTPUT_FOLDER = os.path.join(OUTPUTS_FOLDER, "q3")
44     if not os.path.exists(Q3.OUTPUT_FOLDER):
45         os.makedirs(Q3.OUTPUT_FOLDER)
46     q3.e(
47         x,
48         number_of_trials=4,
49         ks=[2, 3, 4, 7, 10],
50         epsilon=1e-1,
51         max_number_of_steps=int(1e2),
52         figure_path=os.path.join(Q3.OUTPUT_FOLDER, "q3e"),
53         figure_title="Q3e",
54     )
55
56     # Question 5
57     Q5.OUTPUT_FOLDER = os.path.join(OUTPUTS_FOLDER, "q5")
58     if not os.path.exists(Q5.OUTPUT_FOLDER):
59         os.makedirs(Q5.OUTPUT_FOLDER)
60     with open(TRAINING_TEXT_FILE_PATH) as fp:
61         training_text = fp.read().replace("\n", "").lower()
62     with open(SYMBOLS_FILE_PATH) as fp:
63         symbols = fp.read().split("\n")
64     with open(MESSAGE_FILE_PATH) as fp:
65         encrypted_message = fp.read()
66     q5.a(
67         symbols,
68         training_text,
69         transition_matrix_path=os.path.join(Q5.OUTPUT_FOLDER, "q5a-transition.csv"),
70         invariant_distribution_path=os.path.join(Q5.OUTPUT_FOLDER, "q5a-invariant.csv"),
71     )
72     q5.d(
73         encrypted_message,
74         symbols,
75         training_text,
76         number_trials=10,
77         number_of_mh_loops=int(1e4),
78         number_start_attempts=int(1e4),
79         log_decryption_interval=100,
80         log_decryption_size=60,
81         decryptor_table_path=os.path.join(Q5.OUTPUT_FOLDER, "q5d-decrypter.csv"),
82         encrypted_message_iterations_table_path=os.path.join(
83             Q5.OUTPUT_FOLDER, "q5d-iterations.csv"
84         ),
85     )
```

main.py