

# 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. It is not immediately clear how the likelihood of an image of binary (discrete) values would be calculated under the continuous distribution of a multivariate Gaussian. 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:

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

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, 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 to solve for  $\hat{p}_d$ :

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

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:

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

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

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

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:

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

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

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

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

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

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:

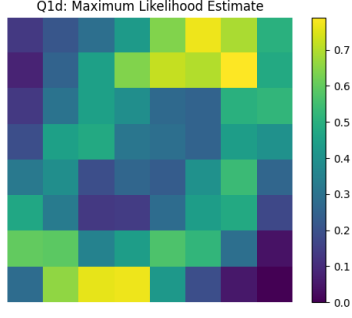


Figure 1: ML parameters

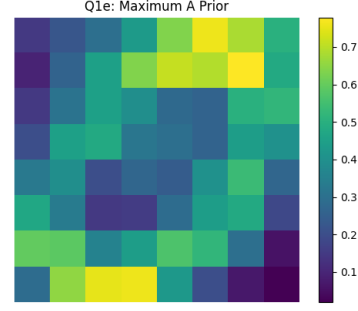


Figure 2: MAP parameters

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  $\frac{1}{N} \sum_{n=1}^N \mathbf{x}^{(n)}$ , the  $\hat{\mathbf{p}}^{MLE}$ . This makes sense because as our data set gets bigger, the effect of the prior diminishes. 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.

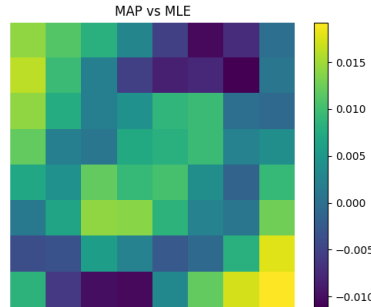


Figure 3:  $\hat{\mathbf{p}}^{MAP} - \hat{\mathbf{p}}^{MLE}$

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-likelihood 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 in size.

## Question 2

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

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

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:

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

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

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

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

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

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

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

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

Given that the prior of any unknown probabilities is uniform, i.e.  $P(\mathbf{p}^{(i)} | M_i) = 1$ . We can simplify:

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

For  $M_1$ :

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

We can remove the integrals:

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

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:

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

Rewriting:

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

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:

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

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  $i \in \{1, 2, 3\}$ :

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



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.

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

Table 1: Posterior Probabilities

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 just one possible model 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).astype(int)
24    return betaln(k + 1, n * d - 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_d = np.sum(x, axis=0).astype(int)
35    return logsumexp(betaln(k_d + 1, n - k_d + 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:

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

where  $\{\mathbf{x}^{(n)}\}_{n=1}^N$  is our data set with  $\mathbf{x}^{(n)} \in \mathbb{R}^{D \times 1}$  and  $\theta = \{\pi, \mathbf{P}\}$  are our parameters,  $\pi = [\pi_1, \dots, \pi_K] \in \mathbb{R}^{K \times 1}$  our  $K$  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 given 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  $P(\mathbf{x}^{(n)}|\theta)$ :

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

The log-likelihood  $\mathcal{L}(\mathbf{x}^{(n)}|\theta)$  can be expressed in vector 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  $P(s^{(n)} = k|\pi, \mathbf{P}) = P(s^{(n)} = k|\pi)$  as  $s^{(n)}$  isn't dependent on  $\mathbf{P}$ .

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

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

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,

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

and

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

which can be vectorised as  $\log \mathbf{r}$  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  $E$ . For each term of  $E$ :

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

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  $\pi$ :

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

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  $\mathbf{P}$ :

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

Simplifying:

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

Setting the derivative to zero:

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

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

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

Thus,

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

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  $\mathbf{P}$ , by solving for  $\hat{\mathbf{P}}_{dk}^{MAP}$  with:

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

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:

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

Simplifying:

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

Setting  $\frac{\partial E'}{\partial \mathbf{P}_{dk}} = 0$  we can calculate  $\hat{\mathbf{P}}_{dk}^{MAP}$ :

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

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

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. For the following parts, a very weak Beta(1+1e-5, 1+1e-5) prior was used.

- (d) Plotting the unnormalised posterior likelihood as a function of the iteration number for different  $k$  values:

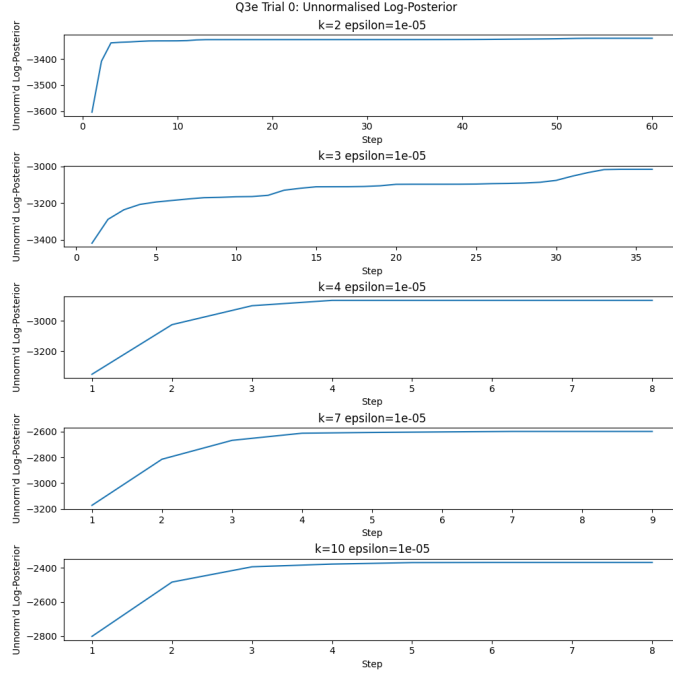


Figure 4: Unnormalised Log Posterior vs Iteration Number

where *epsilon* is the stopping condition for when the unnormalised log posterior converges sufficiently. Note that the normalisation constant for the log posterior  $\log P(\mathbf{x}^{(n)}, s^{(n)}|\pi)$  is intractable and so only the unnormalised portion  $\log P(\mathbf{x}^{(n)}, s^{(n)}|\pi, \mathbf{P}) + \log P(\mathbf{P})$  was computed and reported.

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

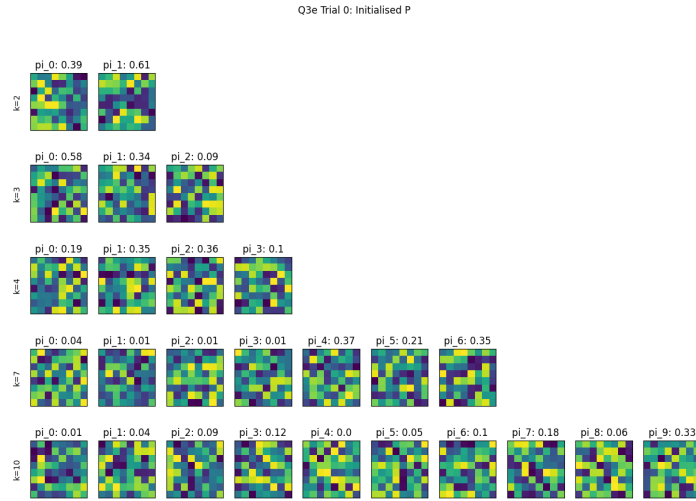


Figure 5: Randomly initialised parameters

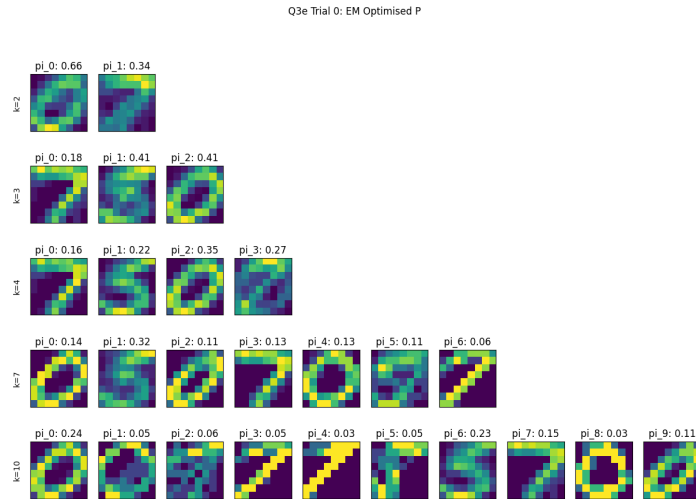


Figure 6: EM optimised parameters



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

```

```

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

```

```

188     _, k = log_responsibility.shape
189
190     x_repeated = np.repeat(x[:, :, np.newaxis], k, axis=2) # (n, d, k)
191     log_responsibility_repeated = np.repeat(
192         log_responsibility[:, np.newaxis, :], d, axis=1
193     ) # (n, d, k)
194
195     log_p_matrix_unnormalised_posterior = logsumexp(
196         log_responsibility_repeated, b=(x_repeated + alpha_parameter - 1), axis=0
197     ) # (d, k)
198
199     log_p_matrix_normaliser_posterior = logsumexp(
200         log_responsibility_repeated, b=(alpha_parameter + beta_parameter - 1), axis=0
201     ) # (d, k)
202
203     log_p_matrix_normalised_posterior = (
204         log_p_matrix_unnormalised_posterior - log_p_matrix_normaliser_posterior
205     ) # (d, k)
206     return log_p_matrix_normalised_posterior
207
208
209 def _compute_log_m_step(
210     x: np.ndarray,
211     log_responsibility: np.ndarray,
212     alpha_parameter: float,
213     beta_parameter: float,
214 ) -> Theta:
215     """
216     Compute the m step of expectation maximisation
217     :param x: the image data (n, d)
218     :param log_responsibility: an array of the log responsibilities of k mixture components for each image
219         (n, k)
220     :param alpha_parameter: alpha parameter of the beta prior
221     :param beta_parameter: beta parameter of the beta prior
222     :return: thetas optimised after maximisation step
223     """
224     return Theta(
225         log_pi=_compute_log_pi_hat(log_responsibility),
226         log_p_matrix=_compute_log_p_matrix_hat(
227             x, log_responsibility, alpha_parameter, beta_parameter
228         ),
229     )
230
231 def _run_expectation_maximisation(
232     x: np.ndarray,
233     theta: Theta,
234     alpha_parameter: float,
235     beta_parameter: float,
236     max_number_of_steps: int,
237     epsilon: float,
238 ) -> Tuple[Theta, np.ndarray, List[float]]:
239     """
240     Run the expectation maximisation algorithm
241     :param x: the image data (n, d)
242     :param theta: initial theta parameters
243     :param alpha_parameter: alpha parameter of the beta prior
244     :param beta_parameter: beta parameter of the beta prior
245     :param max_number_of_steps: the maximum number of steps to run the algorithm
246     :param epsilon: the minimum required change in log posterior, otherwise the algorithm stops early
247     :return: a tuple containing the optimised thetas, the log responsibilities,
248         and the log log-posteriors at each step of the algorithm
249     """
250     log_responsibility = None
251     log_posteriors = []
252     for _ in range(max_number_of_steps):
253         log_responsibility = _compute_log_e_step(x, theta)
254         theta = _compute_log_m_step(
255             x, log_responsibility, alpha_parameter, beta_parameter
256         )
257
258         log_posteriors.append(
259             _compute_unnormalised_log_posterior_likelihood(
260                 x, theta, alpha_parameter, beta_parameter
261             )
262         )
263
264         # check for early stopping
265         if len(log_posteriors) > 1:
266             if (log_posteriors[-1] - log_posteriors[-2]) < epsilon:
267                 break
268     return theta, log_responsibility, log_posteriors
269
270
271 def _visualise_p_matrix(
272     thetas: List[Theta], ks: List[int], figure_title: str, figure_path: str
273 ) -> None:
274     """
275     Visualises the P matrix for different thetas and ks
276     :param thetas: list of Theta instances
277     :param ks: list of k values used for each Theta
278     :param figure_title: name of figure
279     :param figure_path: path to store figure
280     :return:
281     """
282     n = len(ks)

```

```

283 m = np.max(ks)
284 fig = plt.figure()
285 fig.set_figwidth(15)
286 fig.set_figheight(10)
287 for i, k in enumerate(ks):
288     for j in range(k):
289         ax = plt.subplot(n, m, m * i + j + 1)
290         ax.imshow(
291             thetas[i].p_matrix[:, :, j],
292             interpolation="None",
293         )
294         ax.tick_params(
295             axis="x",
296             which="both",
297             bottom=False,
298             top=False,
299         )
300         ax.tick_params(
301             axis="y",
302             which="both",
303             left=False,
304             right=False,
305         )
306         ax.xaxis.set_ticklabels([])
307         ax.yaxis.set_ticklabels([])
308         ax.set_title(f"pi-{j}: {np.round(thetas[i].pi[0, j], 2)}")
309         if j == 0:
310             ax.set_ylabel(f"{k}")
311 fig.suptitle(figure_title)
312 plt.savefig(figure_path)
313
314
315 def _visualise_responsibility_clusters(
316     log_responsibilities: List[np.ndarray],
317     ks: List[int],
318     figure_title: str,
319     figure_path: str,
320 ) -> None:
321     """
322     Visualise responsibility vectors of images using TSNE for different k values
323     :param log_responsibilities: list of log responsibilities for different ks
324     :param ks: list of k values used for each Theta
325     :param figure_title: name of figure
326     :param figure_path: path to store figure
327     :return:
328     """
329     n = len(ks)
330     fig = plt.figure()
331     fig.set_figwidth(5 * n)
332     fig.set_figheight(5)
333     for i, k in enumerate(ks):
334         if k > 2:
335             # use TSNE when we have more than 2 dimensions
336             embedding = TSNE(
337                 n_components=2,
338                 learning_rate="auto",
339                 init="random",
340                 perplexity=10,
341                 random_state=DEFAULT_SEED,
342             ).fit_transform(log_responsibilities[i])
343         else:
344             # otherwise we can visualise responsibility vectors without dimensionality reduction
345             embedding = np.exp(log_responsibilities[i])
346             ax = plt.subplot(1, n, i + 1)
347             ax.scatter(embedding[:, 0], embedding[:, 1])
348             ax.set_title(f"{k}")
349     fig.suptitle(figure_title)
350     plt.savefig(figure_path, bbox_inches="tight")
351
352
353 def _plot_log_posteriors(
354     log_posteriors: List[List[float]],
355     ks: List[int],
356     epsilon: float,
357     figure_title: str,
358     figure_path: str,
359 ) -> None:
360     """
361     Plot log posteriors as a function of EM iteration for different ks
362     :param log_posteriors: list of vectors, each representing the log posterior during EM for a specific k
363     :param ks: list of k values used for each Theta
364     :param epsilon: value used for early stopping of EM
365     :param figure_title: name of figure
366     :param figure_path: path to store figure
367     :return:
368     """
369     fig, ax = plt.subplots(len(ks), 1, constrained_layout=True)
370     fig.set_figwidth(10)
371     fig.set_figheight(10)
372     for i, k in enumerate(ks):
373         ax[i].plot(np.arange(1, len(log_posteriors[i]) + 1), log_posteriors[i])
374         ax[i].set_xlabel("Step")
375         ax[i].set_ylabel(f"Unnorm'd Log-Posterior")
376         ax[i].set_title(f"{k} {epsilon}")
377     plt.suptitle(figure_title)
378

```

```

379 plt.savefig(figure_path)
380
381
382 def _compute_compression_rate(
383     ks: List[int], log_posteriors: List[List[float]], i: int, n: int, d: int
384 ) -> pd.DataFrame:
385     """
386     Compute the compress rate, not taking into account the cost of storing model parameters
387     :param ks: k values to use for each trial
388     :param log_posteriors: list of vectors, each representing the log posterior during EM for a specific k
389     :param i: trial number
390     :param n: number of data points
391     :param d: number of dimensions per data point
392     :return: dataframe containing the compression rate for this trial
393     """
394     df = pd.DataFrame(
395         data=[
396             [
397                 np.round(1 - (-log_posterior[-1] / (np.log(2) * n * d)), 2)
398                 for log_posterior in log_posteriors
399             ],
400             columns=ks,
401         ).T
402     df = df.reset_index()
403     df.columns = ["k value", f"Trial {i}"]
404     return df.set_index("k value")
405
406
407 def _compute_total_compression_ratio(
408     ks: List[int], log_posteriors: List[List[float]], i: int, n: int, d: int
409 ) -> pd.DataFrame:
410     """
411     Compute the total compress ratio, taking into account the cost of storing model parameters (assuming
412     float64)
413     :param ks: k values to use for each trial
414     :param log_posteriors: list of vectors, each representing the log posterior during EM for a specific k
415     :param i: trial number
416     :param n: number of data points
417     :param d: number of dimensions per data point
418     :return: dataframe containing the total compression ratios for this trial
419     """
420     df = pd.DataFrame(
421         data=[
422             [
423                 np.round(
424                     (-log_posterior[-1] + (64 * ks[j] * (d + 1))) / (np.log(2) * n * d),
425                     2,
426                 )
427                 for j, log_posterior in enumerate(log_posteriors)
428             ],
429             columns=ks,
430         ).T
431     df = df.reset_index()
432     df.columns = ["k value", f"Trial {i}"]
433     return df.set_index("k value")
434
435
436 def e(
437     x: np.ndarray,
438     alpha_parameter: float,
439     beta_parameter: float,
440     number_of_trials: int,
441     ks: List[int],
442     epsilon: float,
443     max_number_of_steps: int,
444     figure_path: str,
445     figure_title: str,
446     compression_csv_path: str,
447 ) -> None:
448     """
449     Produces answers for question 3e
450     :param x: numpy array of shape (N, D)
451     :param alpha_parameter: alpha parameter of the beta prior
452     :param beta_parameter: beta parameter of the beta prior
453     :param number_of_trials: number of trails to run EM
454     :param ks: k values to use for each trial
455     :param epsilon: value used for early stopping of EM
456     :param max_number_of_steps: maximum number of steps during EM
457     :param figure_title: base name of figures
458     :param figure_path: base paths to store figure
459     :param compression_csv_path: path to store bits data
460     :return:
461     """
462     n, d = x.shape
463     np.random.seed(DEFAULT_SEED)
464     df_compression_list: List[pd.DataFrame] = []
465     df_total_compression_list: List[pd.DataFrame] = []
466     for i in range(number_of_trials):
467         init_thetas: List[Theta] = []
468         em_thetas: List[Theta] = []
469         log_posteriors: List[List[float]] = []
470         log_responsibilities: List[np.ndarray] = []
471         for j, k in enumerate(ks):
472             init_theta = _init_params(k, d)

```

```

474         em_theta, log_responsibility, log_posterior = _run_expectation_maximisation(
475             x,
476             theta=init_theta,
477             alpha_parameter=alpha_parameter,
478             beta_parameter=beta_parameter,
479             epsilon=epsilon,
480             max_number_of_steps=max_number_of_steps,
481         )
482         init_thetas.append(init_theta)
483         em_thetas.append(em_theta)
484         log_responsibilities.append(log_responsibility)
485         log_posteriors.append(log_posterior)
486
487     _visualise_p_matrix(
488         init_thetas,
489         ks,
490         figure_title=f"{figure_title} Trial {i}: Initialised P",
491         figure_path=f"{figure_path}-{i}-initialised-p.png",
492     )
493     _visualise_p_matrix(
494         em_thetas,
495         ks,
496         figure_title=f"{figure_title} Trial {i}: EM Optimised P",
497         figure_path=f"{figure_path}-{i}-optimised-p.png",
498     )
499     _visualise_responsibility_clusters(
500         log_responsibilities,
501         ks,
502         figure_title=f"{figure_title} Trial {i}: TSNE Responsibility Visualisation",
503         figure_path=f"{figure_path}-{i}-tsne.png",
504     )
505     _plot_log_posteriors(
506         log_posteriors,
507         ks,
508         epsilon,
509         figure_title=f"{figure_title} Trial {i}: Unnormalised Log-Posterior",
510         figure_path=f"{figure_path}-{i}-log-pos.png",
511     )
512     df_compression_list.append(
513         _compute_compression_rate(ks, log_posteriors, i, n, d)
514     )
515     df_total_compression_list.append(
516         _compute_total_compression_ratio(ks, log_posteriors, i, n, d)
517     )
518     pd.concat(df_compression_list, axis=1).to_csv(f"{compression_csv_path}.csv")
519     pd.concat(df_total_compression_list, axis=1).to_csv(
520         f"{compression_csv_path}-total.csv"
521     )

```

src/solutions/q3.py

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

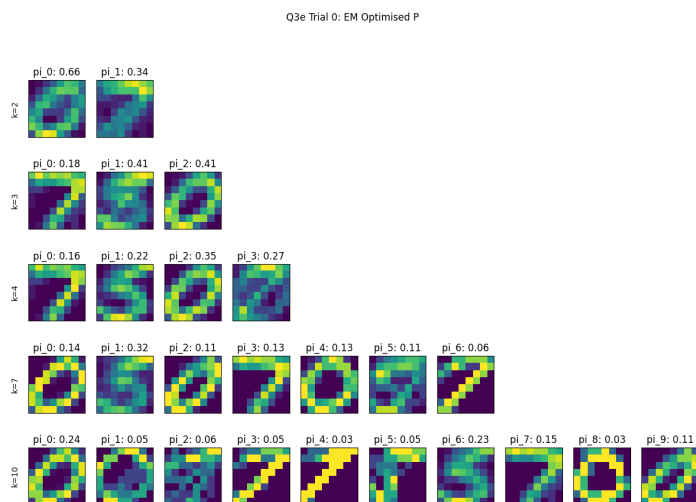


Figure 7: EM optimised parameters: Trial 0

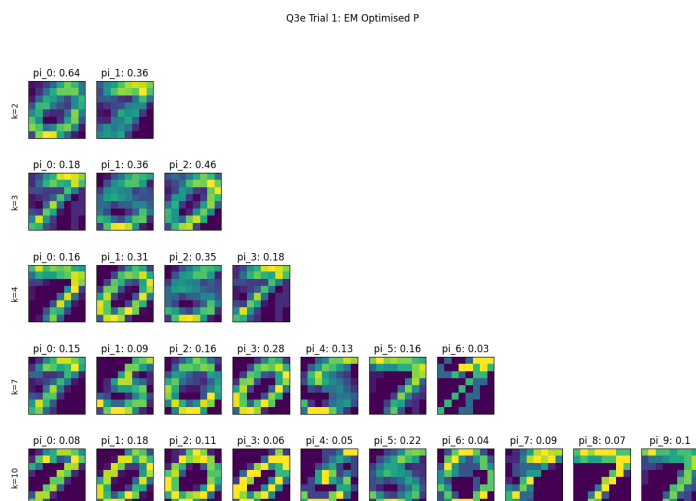


Figure 8: EM optimised parameters: Trial 1

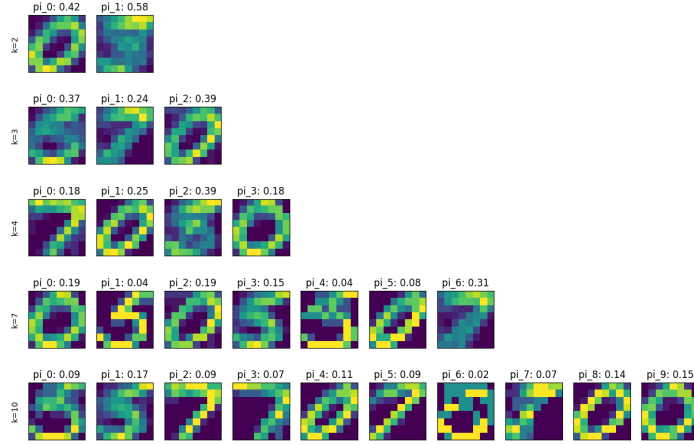


Figure 9: EM optimised parameters: Trial 2

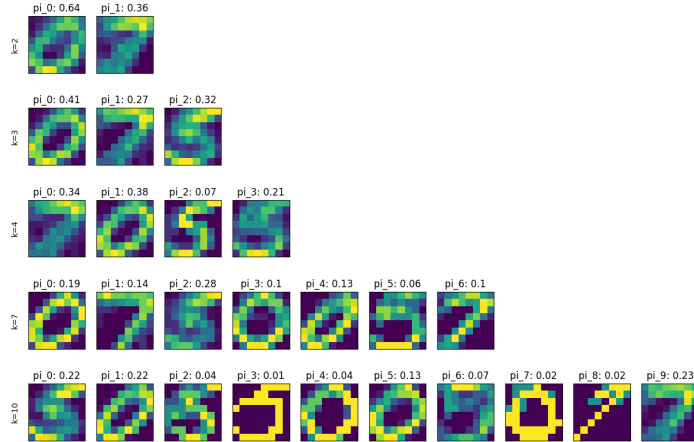


Figure 10: EM optimised parameters: Trial 3

For smaller  $k$ , we can visually see that we obtain very similar solutions (a seven and a zero for  $k = 2$ , although for trial 2, it looks more like zero and five mixed with seven). For  $k = 3$ , we get one each of zero, seven, and five, but in different permutations for different trials. However for higher  $K$ , we see that this may not always be the case. For Trial 1 of  $k = 10$ , we have two 5's whereas in Trial 3 we have four 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.  $\pi_3$  of trial 0 and  $k = 10$ ). It will be unlikely for those components to represent very distinct clusters (i.e. the parameters for  $\pi_3$  and  $\pi_4$  are very similar in trial 0 and  $k = 10$ ). Moreover, for these small probabilities, we see that the parameters are almost like binary images. This shows that there are not many images represented in these components. This can be verified when we perform a TSNE visualisation of the responsibility vector for each of the images (Note that for  $k = 2$ , just the responsibility vector is plotted because it is two dimensional). We can see that for large  $k$ , qualitatively the number of clusters no longer matches the  $k$  value, indicating that some mixtures are redundant. For example for  $k = 7$  and  $k = 10$  we can only qualitatively see three or four clusters with TSNE.

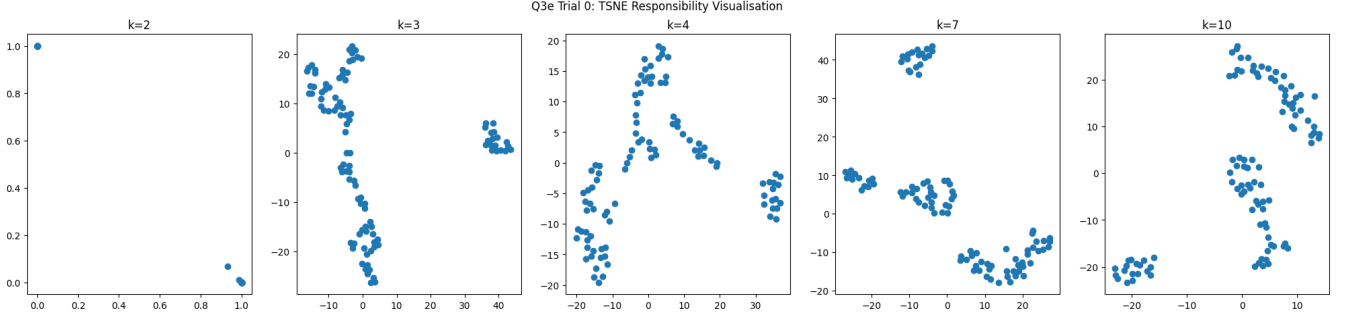


Figure 11: TSNE Visualisation of Image responsibilities: Trial 0

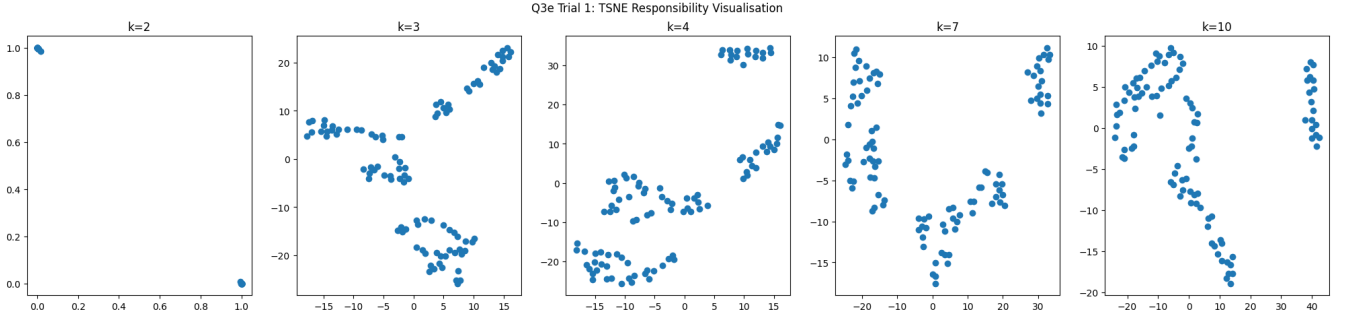


Figure 12: TSNE Visualisation of Image responsibilities: Trial 1

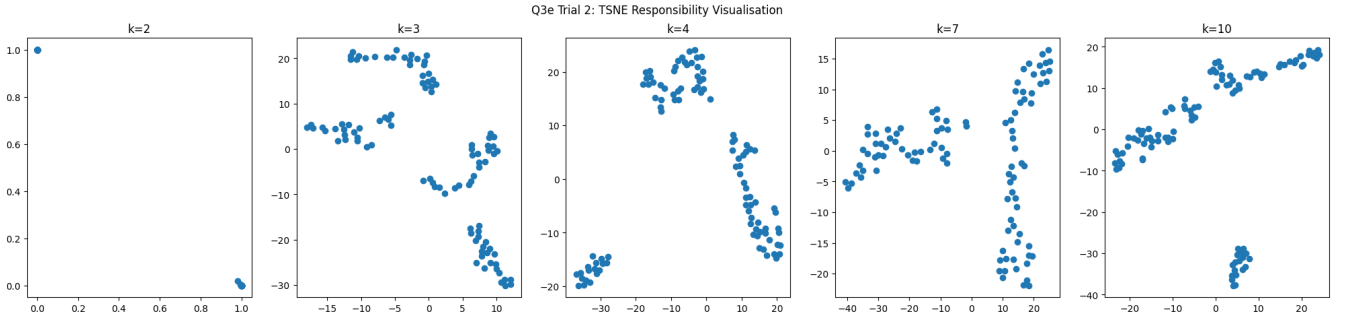


Figure 13: TSNE Visualisation of Image responsibilities: Trial 2

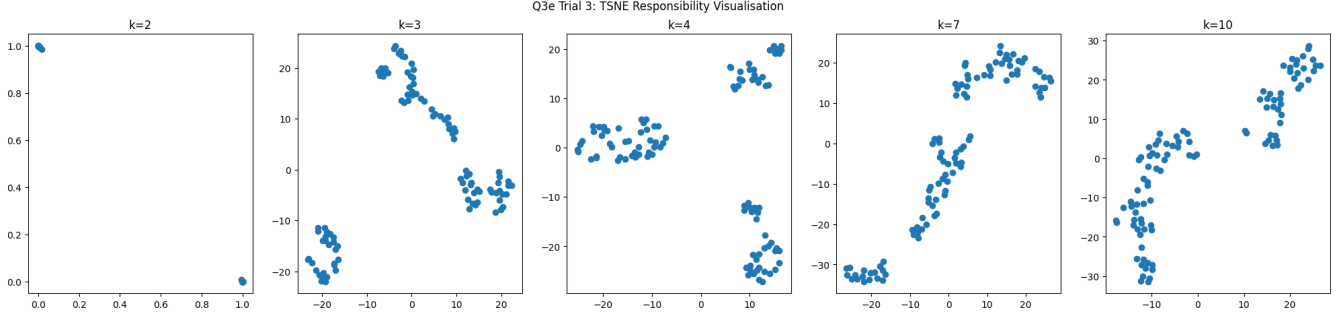


Figure 14: TSNE Visualisation of Image responsibilities: Trial 3

Improvements to the model could include searching for an optimal  $k$  by maximising the log posterior with regularisation on the magnitude of  $k$  to balance maximising log posterior with minimising model complexity. Additionally, adding a prior on the responsibility components can be helpful to ensure a more even distribution across mixture components unlike the components visualised here. This could help promote more meaningful clusters as  $k$  increases. Moreover, more experimentation for choosing better priors can be helpful to find better separation between mixtures. Finally, increasing the size of our data set (i.e. more images) and resolution of our images (i.e. more pixels) can help the model better understand the distinguishing nuances of different mixtures and provide better clustering, although the number of images and the resolution should scale together to ensure that the model doesn't learn the noise in the higher resolution images. This is assuming we are able to scale our computing resources. Finally, given that we know that there are ten digits and our current data set only includes a subset of these digits, we can also expand our data set to include all ten digits. Hopefully, for  $k = 10$ , we will then be able to achieve a unique digit for each mixing component, rather than variations of repeated digits as we see now.

(f) The log-likelihood in bits can be expressed as:

$$\log_2(P(\{\mathbf{x}^{(n)}\}_{n=1}^N|\theta))$$

The length of the naive encoding of these binary data is  $N \cdot D$ , the number of pixels in  $\{\mathbf{x}^{(n)}\}_{n=1}^N$ . This is because the images are binary so each pixel can be represented with a single bit. We can compute the ratio of log-likelihood bits to the length of the naive encoding for each  $k$ :

$$rate = 1 - \frac{\log_2(P(\{\mathbf{x}^{(n)}\}_{n=1}^N|\theta))}{N \cdot D}$$

Presenting the compression rates for different trials and  $k$  values:

<b>k value</b>	Trial 0	Trial 1	Trial 2	Trial 3
2	0.25	0.25	0.26	0.25
3	0.32	0.28	0.29	0.31
4	0.35	0.36	0.36	0.32
7	0.41	0.42	0.38	0.43
10	0.47	0.47	0.49	0.43

Table 2: Compression Rates

As  $k$  increases, we can see that our compression rate gets better. This is intuitive because with higher  $k$  we are specifying a more complex and expressive model (i.e. with more parameters) and thus we are able to capture more of the structure of the data in the model. Thus, the bit rate, or information provided by a sample decreases with respect to the complexity of the model and thus our compression rate increases. From the source coding theorem, lossless compression algorithms are lower bounded by the entropy of the underlying data generating distribution  $P(\mathbf{x})$ . This is  $-\sum_{\mathbf{x} \in \mathcal{X}} P(\mathbf{x}) \log P(\mathbf{x})$ . On the other hand, with EM we are maximising  $\langle \log P(Z, X|\theta) \rangle_{q(X)} + H[q]$  or minimising the negative of this. When our proposal distribution  $P$  matches  $q$ , we recover  $H[P]$  and we get the optimal model, the data generating model for encoding our data, which makes sense because we would be able to compress our data with the best possible distribution. This matches the lower bound of the source coding theorem. However, because it is very unlikely that our proposal  $q$  will actually match  $P$ , our compression rate will always be worse than the optimal compression rate of  $H[P]$ . On the other hand, a compression algorithm would compressions on a per image basis, independent of the other images. And thus, it is able to attain a better compression rate for that image and is much closer to the source coding theorem lower bound. Depending on the data (i.e.  $H[P]$  of the data), the compression rate of gzip can range from 60% to 88% (<https://web.dev/optimizing-content-efficiency-optimize-encoding-and-transfer/text-compression-with-gzip>), much higher than that of our models, as we expected.

(g) The total cost of encoding with model parameters and data:

$$\log_2(P(\{\mathbf{x}^{(n)}\}_{n=1}^N|\theta)) + M \cdot K \cdot D + M \cdot K$$

Where  $M$  is the cost of storing a single float value, in our case we used *float64* so 64 bits. The first term is the log-likelihood as expressed in (f), the second term is the cost of storing  $\mathbf{P}$ , and the last term is the cost of storing  $\pi$ . The latter two terms scale with the value of  $k$ . This means that as  $k$  increases, our compression rate deteriorates. Looking at the total compression ratio  $\frac{\log_2(P(\{\mathbf{x}^{(n)}\}_{n=1}^N|\theta)) + M \cdot K \cdot D + M \cdot K}{N \cdot D}$  in a table:

<b>k value</b>	Trial 0	Trial 1	Trial 2	Trial 3
2	2.62	2.63	2.62	2.63
3	3.49	3.54	3.52	3.5
4	4.4	4.39	4.39	4.43
7	7.15	7.14	7.18	7.14
10	9.91	9.91	9.89	9.95

Table 3: Total Compression Ratios

We can see that the ratio is greater than one, meaning that this is actually worse than the naive encoding. This is due to the high cost of each value in the model parameters being 64 bits. However, because this remains constant with respect to our data set size, so as  $N$  increases, these ratios will approach  $\frac{\log_2(P(\{\mathbf{x}^{(n)}\}_{n=1}^N|\theta))}{N \cdot D}$  and we'll recover our compression rates from part (f). By increasing  $k$ , we see that the ratio increases, and thus our compression rate is worse. This makes sense because we are essentially slowly storing the data into the model. In the extreme example of  $k = N$  we can have the parameters for each mixture model as an image in the data. Although in this case, we could store our parameters as binary values instead of 64 bit floats. Our mixture component is uniform because each mixture is equally likely (all images are equally likely) so there is no need to store any values for  $\pi$ . Thus, we would recover the cost of naive encoding our data. Taking into account the model parameters further verifies that the cost of encoding with this approach is much higher than the cost of gzip.

## Question 5

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

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

where  $N_{\alpha,\beta}$  is the count of the number of occurrences of the pair  $(\alpha, \beta)$ , where  $\beta$  is followed by  $s\alpha$  in the text and  $N_\beta$  is the number of occurrences of  $\beta$ . Moreover to ensure ergodicity, a one was added to each  $N_{\alpha,\beta}$ . This was also taken into account for the normaliser  $N_\beta$ .

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

- (i) Initialise any  $\phi^{(0)} \in \mathbb{R}^{53 \times 1}$  and  $\sum_i \phi_i^{(0)} = 1$
- (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,  $\Psi_{i,j} = P(s_j | s_i)$  where  $s_i$  is the  $i^{th}$  symbol and  $s_j$  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 symbol  $\gamma$  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  $\sigma(s)$  for different symbols  $s$ .

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 p2 2uanmr = jmk pa= batn ~j3t' 2')~q p2 8)'9'= y/b' p' qn
100	er pl losura= drk po= a bstra=dita lad= n~pl dca 8) uduha pa so
200	er nl koiruah srrw nova hitraadta ladhlp nl xdyman udha na po
300	er nl koiruah srrw nova bitrawedta ladhlp nl xyayuw ndba na po
400	er vd idit an orw ysua bitranolta daonv ud uphan obaa ya ys
500	er c_ ,siridan cr_ esna bitrauolt_ aony c_ uophan doba ca sy
600	en ck kyindar on cytra bitnarolta kaors pk uochar doba pa sl
700	en pk klindar on cytra bitnaroyta kaors pk uochar doba pa sy
800	= en p_ londar in pura botaniariyta aiirs p_ fighar diba pa su
900	= en p_ slondar in pura botaniariyta aiirs p_ fighar diba pa sl
1000	en pl huondar in pura botaniariyta lairns pl fighar diba pa su
1100	en pl huondar in pura cotanixatika lairns pl fighar dica pa su
1200	en pk kuounger ind pura commatnixka kairs pk fighar dica pa su
1300	en ck kuounger ind cura pommarnixka kairs ck fighar dica pa su
1400	en ck kuounder ind cura pommarnixka kairs ck fighar dica pa su
1500	en ck kuounder ind cura vummarnixka kairs ck fithar diya ca so
1600	en ck kuouder ind cura vummarnixka kairs ck fithar diya ca so
1700	an ck kounder ind core vummerixe keirs ck fither dive ce so
1800	an ck kouler ind core vummerixe keirs ck fither live ce so
1900	an ck koumler ind core vummerixe keirs ck fither live ce so
2000	an ck koumler ind core vummerixe keirs ck fither live ce so
2100	an ck koumler ind core vummerixe keirs ck fither live ce so
2200	an ck koumler ind core vummerixe keirs ck fither live ce so
2300	an ck kouinger ind core vummerixe keirs ck fither give ce so
2400	an ck kounger ind core vuluerike keirs ck fither givc ce so
2500	an mk kounger ind more vuluerike keirs mk fither give me so
2600	an mk kounger ind more vuluerike keirs mk fither give me so
2700	an mk kounger ind more vuluerike keirs mk fither give me so
2800	an mk kounger ind more vuluerike keirs mk fither give me so
2900	an mk kounger ind more vuluerike keirs mk fither give me so
3000	an mk kounger ind more vuluerike keirs mk fither give me so
3100	an mf founger ind more vulneripfe feirs mf kither give me so
3200	an mf founger ind more vulneripfe feirs mf kither give me so
3300	an mf founger ind more vulneripfe feirs mf kither give me so
3400	an mf founger ind more vulneripfe feirs mf kither give me so
3500	an mf founger ind more vulneripfe feirs mf kither give me so
3600	an mf founger ind more vulneripfe feirs mf kither give me so
3700	an mf founger ind more vulneripfe feirs mf kither give me so
3800	an mf founger ind more vulneripfe feirs mf kither give me so
3900	in mf founger and more vulnerape fears mf kather gave me so
4000	in mf founger and more vulneraple fears mf kather gave me so
4100	in mf founger and more vulnerape 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 first randomly and independently sampled. The  $\sigma$  with the best log-likelihood was chosen as the starting point for the MH chain and the algorithm was then run for 10000 iterations. Moreover, ten different trials of this was performed, where the trial with the best log-likelihood was displayed. The decrypted message for each of the ten trials:

Trial	Decryption
0	litedeecont! featpedof eyunt fa.n ee afredcevas, felay ed ero
1	in my younger and more vulnerable years my father gave me so
2	in cy younker and core vulnerable years cy father kave ee so
3	is hy ytower asd ltre volseraule yearn hy fanger wave he nt
4	in my younger and more vulnerable years my father gave me so
5	$^{''5407^{''}0^{''}}[4]81004307180((^{''4819^{''}80^{''}}891207^{''}0.96-810)9/807802)^{''}$
6	$^{''542)(2[(94^{''}18234-2)]812.9^{}4183^{''}12(13862)(2/307182^{''}3.12)126)^{''}$
7	ioadcaelyon earowadle agyo erk. ac retadcafsu canrg ad atl
8	in my younker and more vulnerable years my father kave me so
9	in my younker and more vulnerable years my father kave me so

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 to index the transition matrix
121         and the invariant distribution
122         :param symbols: list of symbols to map
123         :return: symbol to integer mapping
124         """
125         return {k: v for v, k in enumerate(symbols)}
126
127     def _construct_transition_matrix(
128         self, text: str, symbols_dict: Dict[str, int]
129     ) -> np.ndarray:
130         """
131         Constructs the transition matrix for a given text
132         :param text: string to calculate transition matrix with
133         :param symbols_dict: dictionary mapping symbol to a dictionary
134         :return:
135         """
136         # initialise with ones to ensure ergodicity
137         transition_matrix = np.ones((self.num_symbols, self.num_symbols))
138         for i in range(1, len(text)):
139             # check symbols are valid
140             if text[i] in symbols_dict and text[i - 1] in symbols_dict:
141                 transition_matrix[symbols_dict[text[i - 1]], symbols_dict[text[i]]] += 1
142         # normalise to get transition probabilities
143         transition_matrix = normalize(transition_matrix, axis=0, norm="l1")
144         return transition_matrix
145
146     def _approximate_invariant_distribution(
147         self, invariant_stopping_epsilon: float
148     ) -> np.ndarray:
149         """
150         Approximate the invariant distribution with the power method
151         :param invariant_stopping_epsilon: stopping condition for constructing the invariant distribution
152         :return: the invariant distribution as a vector (number of symbols, 1)
153         """
154         invariant_distribution = np.zeros((self.num_symbols, 1))
155         previous_invariant_distribution = invariant_distribution.copy()
156
157         # make sure it's a proper distribution that sums to one
158         invariant_distribution[0] = 1
159
160         while (
161             np.linalg.norm(invariant_distribution - previous_invariant_distribution)
162             > invariant_stopping_epsilon
163         ):
164             previous_invariant_distribution = invariant_distribution.copy()
165             invariant_distribution = self.transition_matrix @ invariant_distribution
166         return invariant_distribution
167
168     def log_transition_probability(self, alpha: str, beta: str) -> float:
169         """
170         Look up the log probability of the transition from symbol alpha to beta
171         :param alpha: symbol that is being transitioned from
172         :param beta: symbol that is being transitioned to
173         :return: probability of transition
174         """
175         return self.log_transition_matrix[
176             self.symbols_dict[beta], self.symbols_dict[alpha]
177         ]
178
179     def log_invariant_probability(self, gamma: str) -> float:
180         """
181         Look up the log probability of a symbol with respect to the invariant distribution
182         :param gamma: symbol to query
183         :return: log probability of the symbol
184         """
185         return self.log_invariant_distribution[self.symbols_dict[gamma]].item()
186
187     def compute_log_probability(self, text: str) -> float:
188         """
189         Compute the log probability of a given text containing symbols
190         :param text: text to compute log probability for

```

```

191         :return: log probability of the text
192         """
193         log_probability = self.log_invariant_probability(text[0])
194         for i in range(1, len(text)):
195             log_probability += self.log_transition_probability(text[i], text[i - 1])
196         return log_probability
197
198
199 class MetropolisHastingsDecryption:
200     def __init__(self, symbols: List[str]):
201         """
202         Metropolis Hastings MCMC for Decryption
203         :param symbols: set of symbols to decrypt
204         """
205         self.symbols = symbols
206
207     def generate_random_decrypter(self) -> Decrypter:
208         """
209         Generates a random decrypter
210         :return: a Decrypter instantiation
211         """
212         return Decrypter(
213             {
214                 self.symbols[i]: self.symbols[x]
215                 for i, x in enumerate(
216                     np.random.permutation(np.arange(len(self.symbols)))
217                 )
218             }
219         )
220
221     @staticmethod
222     def generate_proposal_decryption(decrypter: Decrypter) -> Decrypter:
223         """
224         Generate a proposal decrypter by randomly swapping two of the decryption mappings
225         :param decrypter: the decrypter used to generate the proposal
226         :return: a proposal decrypter
227         """
228         x1 = np.random.choice(list(decrypter.decryption_dict.keys()))
229         x2 = np.random.choice(list(decrypter.decryption_dict.keys()))
230         proposal_decryption = decrypter.decryption_dict.copy()
231         proposal_decryption[x2], proposal_decryption[x1] = (
232             decrypter.decryption_dict[x1],
233             decrypter.decryption_dict[x2],
234         )
235         return Decrypter(proposal_decryption)
236
237     @staticmethod
238     def _choose_decrypter(
239         statistics: Statistics,
240         encrypted_message: str,
241         current_decrypter: Decrypter,
242         proposal_decrypter: Decrypter,
243     ) -> Decrypter:
244         """
245         Choose between the current and proposal decrypter
246         :param statistics: Statistics instantiation for calculating log probabilities
247         :param encrypted_message: the encrypted message
248         :param current_decrypter: the current decrypter
249         :param proposal_decrypter: the proposal decrypter
250         :return:
251         """
252         # calculate log probabilities
253         current_log_probability = statistics.compute_log_probability(
254             text=current_decrypter.decrypt(encrypted_message),
255         )
256         proposal_log_probability = statistics.compute_log_probability(
257             text=proposal_decrypter.decrypt(encrypted_message),
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: str,
273         number_start_attempts: int,
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()

```

```

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

```

```

383 """
384 Produces answers for question 5d
385 :param encrypted_message: the encrypted message
386 :param symbols: symbols in the training text
387 :param training_text: training text for calculating transition and invariant probability
388 :param number_trials: number of times to restart and run the sampler
389 :param number_of_mh_loops: number of loops to run the metropolis hasting sampler
390 :param number_start_attempts: number of possible starting decrypters to check
391 :param log_decryption_interval: number of samples between logging the decrypted message
392 :param log_decryption_size: number of symbols to decrypt when logging the decrypted message
393 :param trial_decryptions_table_path: path to store decryption messages for each trial
394 :param decryptor_table_path: path to store decrypter mapping table
395 :param decrypted_message_iterations_table_path: path to store logged decryption messages
396 :return:
397 """
398 statistics = Statistics(
399     training_text,
400     symbols,
401 )
402 np.random.seed(DEFAULT_SEED)
403 metropolis_hastings_decryption = MetropolisHastingsDecryption(symbols)
404 decrypters: List[Decrypter] = []
405 log_likelihoods: List[float] = []
406 logged_decryption_messages: List[List[str]] = []
407 decryption_messages = []
408 for i in range(number_trials):
409     (decrypter, logged_decryption_message) = metropolis_hastings_decryption.run(
410         encrypted_message,
411         statistics,
412         number_of_mh_loops,
413         number_start_attempts,
414         log_decryption_interval,
415         log_decryption_size,
416     )
417     decrypters.append(decrypter)
418     log_likelihoods.append(
419         statistics.compute_log_probability(decrypter.decrypt(encrypted_message))
420     )
421     logged_decryption_messages.append(logged_decryption_message)
422     decryption_messages.append(
423         decrypter.decrypt(encrypted_message)[:log_decryption_size]
424     )
425 df_trial_decryptions = _construct_decryptions_table(
426     decryption_messages=[x[:log_decryption_size] for x in decryption_messages],
427     decryption_interval=1,
428     columns=["Trial", "Decryption"],
429 )
430 df_trial_decryptions.to_csv(trial_decryptions_table_path, sep="|")
431
432 # sort trials by log likelihood
433 best_trial = np.argmax(log_likelihoods)
434 decrypters[best_trial].table.to_csv(decryptor_table_path, sep="|")
435 df_logged_decryptions = _construct_decryptions_table(
436     decryption_messages=logged_decryption_messages[best_trial],
437     decryption_interval=log_decryption_interval,
438     columns=["MH Iteration", "Current Decryption"],
439 )
440 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, including to itself, 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) 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 decoding approach 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 symbols can't have the same frequency), then using symbol probabilities alone should theoretically work by matching symbol probabilities. However, in practise it would be unlikely to be able to make these assumptions about symbol frequencies, especially with the finite size of our training set and encrypted message.

A second-order chain should also work in theory. However, with this approach it is probably practically more difficult for finding a suitable decoding. This is because our transition tensor would contain  $N^3$  elements, 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 first order transitions in a text of length  $N$  is  $N - 1$  and second order its  $N - 2$ ). This can also introduce sparsity (as in small non-zero probabilities in many entries) in our transition matrix despite our ability to maintain ergodicity with small probabilities to prevent non-zero entries. However, the log-likelihood of many areas of  $\sigma$  space might be very small or all just the same, when all the transition probabilities are just the offset probability (added to maintain ergodicity). Navigating this space will be much more difficult for the sampler as the space could be relatively flat.

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  $\sigma$  is no longer as easily inverted and therefore for each duplicate mapping, we would have to integrate out the probability for the two possible decrypted symbols when computing the log-likelihood. 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. Moreover, the number of proposal  $\sigma'$ 's will depend on how many duplicates exist in the current  $\sigma$ .



Thus  $S(\sigma \rightarrow \sigma')$  would no longer be symmetric, complicating the acceptance probability calculation as it would be dependent on the  $\sigma$  and  $\sigma'$ . Overall, this approach could work but would require many changes to accommodate for these complications. Integrating out collision mappings in the log-likelihood, non-symmetric proposal probabilities, and a much larger  $\sigma$  space because duplicates are allowed, means that it will take much longer for the sampler to find a reasonable  $\sigma$ .

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 and we'd run into similar problems as for second order Markov Chains. An alternative set up could be with using Chinese phonetics, for which there are 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:

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

$$x = 0$$

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  $\lambda = -1$  and  $(x = 0, y = -1)$  when  $\lambda = 1$ .

(b)

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

$$x = \ln(a)$$

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

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 is:

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

Simplifying:

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

we have our update equation in Newton's algorithm for this problem.

## 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 first show that the above optimisation can be equivalently formulated as:

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

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:

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

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

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

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

We can simplify the expression:

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

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:

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

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  $R_A(\mathbf{x}) \leq \lambda_1$  will be equivalent to showing  $\mathbf{x}^T \mathbf{A} \mathbf{x} \leq \lambda_1$  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:

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

From our above reformulation with the unit sphere, 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 a weighted average of  $\lambda_i$ 's, which is always less than or equal to the largest  $\lambda_i$  value so:

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

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

(c) Given that  $\mathbf{x} \in \text{span}\{\xi_{k+1}, \dots, \xi_n\}$ , we can rewrite  $\mathbf{x}$ :

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

Using the same argument as in (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         alpha_parameter=1 + 1e-5,
49         beta_parameter=1 + 1e-5,
50         number_of_trials=4,
51         ks=[2, 3, 4, 7, 10],
52         epsilon=1e-5,
53         max_number_of_steps=int(1e2),
54         figure_path=os.path.join(Q3_OUTPUT_FOLDER, "q3e"),
55         figure_title="Q3e",
56         compression_csv_path=os.path.join(Q3_OUTPUT_FOLDER, "q3e-compression"),
57     )
58
59     # Question 5
60     Q5_OUTPUT_FOLDER = os.path.join(OUTPUTS_FOLDER, "q5")
61     if not os.path.exists(Q5_OUTPUT_FOLDER):
62         os.makedirs(Q5_OUTPUT_FOLDER)
63     with open(TRAINING_TEXT_FILE_PATH) as fp:
64         training_text = fp.read().replace("\n", "").lower()
65     with open(SYMBOLS_FILE_PATH) as fp:
66         symbols = fp.read().split("\n")
67     with open(MESSAGE_FILE_PATH) as fp:
68         encrypted_message = fp.read()
69     q5.a(
70         symbols,
71         training_text,
72         transition_matrix_path=os.path.join(Q5_OUTPUT_FOLDER, "q5a-transition.csv"),
73         invariant_distribution_path=os.path.join(Q5_OUTPUT_FOLDER, "q5a-invariant.csv"),
74     )
75     q5.d(
76         encrypted_message,
77         symbols,
78         training_text,
79         number_trials=10,
80         number_of_mh_loops=int(1e4),
81         number_start_attempts=int(1e4),
82         log_decryption_interval=100,
83         log_decryption_size=60,
84         trial_decryptions_table_path=os.path.join(Q5_OUTPUT_FOLDER, "q5d-trials.csv"),
85         decryptor_table_path=os.path.join(Q5_OUTPUT_FOLDER, "q5d-decrypter.csv"),
86         decrypted_message_iterations_table_path=os.path.join(
87             Q5_OUTPUT_FOLDER, "q5d-iterations.csv"
88         ),
89     )
```

main.py