COMP0086 Summative Assignment

Nov 14, 2022

Question 1

(a) Our sample space for images is $\{0, 1\}^D$, a binary space with D dimensions, the number of pixels in the image. Thus, picking the exponential family best suited on this sample space is the D-dimensional multivariate Bernoulli distribution which shares the same sample space. On the other hand, a D-dimensional multivariate Gaussian has the sample space $\mathbb{R}^D$, which does not match the sample space of our data. To match our data sample space, we would have to define additional mapping between our data and model spaces which adds unnecessary complexity. Thus it would be inappropriate to model this dataset of images with a multivariate Gaussian.

(b) For $\mathcal{D} := \{x^{(n)}\}_{n=1}^N$ a data set of N images, the joint likelihood (assuming images are independently and identically distributed) is the product of N D-dimensional multivariate Bernoulli distributions, one for each image:

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

Substituting the D-dimensional multivariate Bernoulli:

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

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

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

Note that since the logarithm of the mean is a monotonic increasing function on $\mathbb{R}_+$, the maximisers and minimisers of the likelihood do not change.

To solve for the maximum likelihood estimate, $\hat{p}_d$, we can take the derivative of $\mathcal{L}(\mathcal{D}|\mathbf{p})$ with respect to p_d, the d^{th} element of $\mathbf{p}$:

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

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

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

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

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

$$\hat{p}_d = \frac{1}{N} \sum_{n=1}^{N} x_d^{(n)}$$

Moreover, the second derivative with respect to p_d:

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

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

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

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

(c) From Bayes' Theorem:

$$P(\mathbf{p}|\mathcal{D}) = \frac{P(\mathcal{D}|\mathbf{p})P(\mathbf{p})}{P(\mathcal{D})}$$

Taking the logarithm:

$$\mathcal{L}(\mathbf{p}|\mathcal{D}) = \mathcal{L}(\mathcal{D}|\mathbf{p}) + \mathcal{L}(\mathbf{p}) - \mathcal{L}(\mathcal{D})$$

Taking the derivative with respect to p_d:

$$\frac{\partial \mathcal{L}(\mathbf{p}|\mathcal{D})}{\partial p_d} = \frac{\partial \mathcal{L}(\mathcal{D}|\mathbf{p})}{\partial p_d} + \frac{\partial \mathcal{L}(\mathbf{p})}{\partial p_d}$$

where $\frac{\partial \mathcal{L}(\mathcal{D})}{\partial p_d}$=0 because it doesn't depend on p_d.

We know (b):

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

For the second term $\frac{\partial \mathcal{L}(\mathbf{p})}{\partial p_d}$, we start with $P(\mathbf{p})$):

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

Assuming independent Beta priors on the parameters 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 to have an expression for the log posterior derivative $\frac{\partial \mathcal{L}(\mathbf{p}|\mathcal{D})}{\partial p_d}$:

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

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

To find the maximum a posteriori (MAP) estimate $\hat{p}_d$ with $\frac{\partial \mathcal{L}(\mathbf{p}|\mathcal{D})}{\partial p_d} = 0$:

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

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

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

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

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

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

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

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

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

.

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

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

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

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

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


def _compute_maximum_likelihood_estimate(x: np.ndarray) -> np.ndarray:
    """
    X: numpy array of shape (N, D)
    """
    return np.mean(x, axis=0)


def _compute_maximum_a_priori_estimate(
    x: np.ndarray, alpha: float, beta: float
) -> np.ndarray:
    """
    X: numpy array of shape (N, D)
    alpha: param of prior distribution
    beta: param of prior distribution
    """

    n, _ = x.shape
    return (alpha - 1 + np.sum(x, axis=0)) / (n + alpha + beta - 2)


def d(x, figure_path, figure_title):
    maximum_likelihood = _compute_maximum_likelihood_estimate(x)
    plt.figure()
    plt.imshow(
        np.reshape(maximum_likelihood, (8, 8)),
        interpolation="None",
    )
    plt.colorbar()
    plt.axis("off")
    plt.title(figure_title)
    plt.savefig(figure_path)


def e(x, alpha, beta, figure_path, figure_title):
    maximum_a_priori = _compute_maximum_a_priori_estimate(x, alpha, beta)
    plt.figure()
    plt.imshow(
        np.reshape(maximum_a_priori, (8, 8)),
        interpolation="None",
    )
    plt.colorbar()
    plt.axis("off")
    plt.title(figure_title)
    plt.savefig(f"{figure_path}.png")

    maximum_likelihood = _compute_maximum_likelihood_estimate(x)
    plt.figure()
    plt.imshow(
        np.reshape(maximum_a_priori - maximum_likelihood, (8, 8)),
        interpolation="None",
    )
    plt.colorbar()
    plt.axis("off")
    plt.title(f"MAP vs MLE")
    plt.savefig(f"{figure_path}-mle-vs-map.png")
```

src/solutions/q1.py

Displaying the learned parameters:

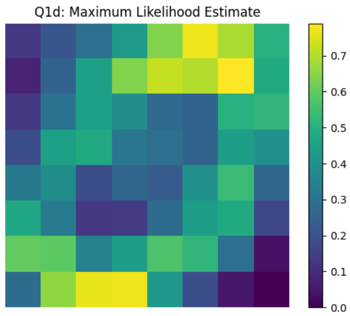

Figure 1: ML parameters

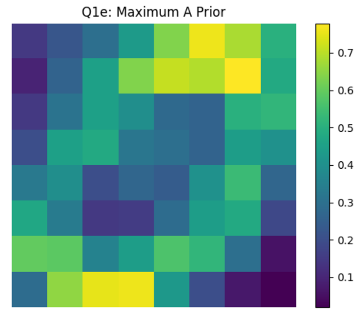

Figure 2: MAP parameters

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, the MAP estimate approaches $\frac{1}{N}\sum_{n=1}^{N}\mathbf{x}^{(n)}$, the MLE. This makes sense because as our data set gets bigger, we are less reliant on our prior. Moreover, if a specific pixel in all of the images of our data set are white or all black, the MLE for that pixel will be binary. This may not be representative of our intuitions about image pixels, 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. Figure 3 visualises $\hat{\mathbf{p}}^{MAP} - \hat{\mathbf{p}}^{MLE}$ and we can see that for likelihoods greater than 0.5 in the MLE, the MAP has a lower value and for likelihoods less than 0.5, the MAP has a higher value, confirming our intuitions.

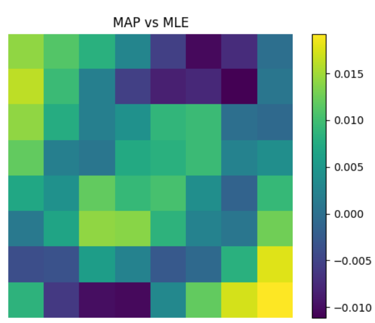

Figure 3: $\hat{\mathbf{p}}^{MAP} - \hat{\mathbf{p}}^{MLE}$

Priors can also help ensure numerical stability during calculations. The logarithm of zero is negative infinity, so having if the MLE is zero it can be problematic for log-likelihoods calculations whereas MAP can ensure non-zero probabilities. We can see in Figure 1 and Figure 2 that the MLE has parameter values of zero whereas the MAP does not. Interestingly, when $\alpha = 1$ and $\beta = 1$, $\hat{\mathbf{p}}^{MLE} = \hat{\mathbf{p}}^{MAP}$. Intuitively this is when the prior is a uniform distribution and so there aren't any biases 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.

Question 2

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

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

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

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

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

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

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

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

where $d = 1, ..., D$ and $\{\mathbf{x}^{(n)}\}_{n=1}^N$ is our data set.
Given that the prior of any unknown probabilities is uniform, i.e. $P(p_d|M_i) = 1$. We can simplify:

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

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

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

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

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

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

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

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

Changing the products to sums:

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

Rewriting:

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

where $k = \sum_{n=1}^N \sum_{d'=1}^D x_{d'}^{(n)}$.
This integral is the beta function:

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

For M_3, we need an integral for each p_d:

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

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

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

Changing the products to sums:

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

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

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

where $k_d = \sum_{n=1}^N x_d^{(in}$.
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 below:

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

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

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

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


def _log_p_d_given_m1(x):
    n, d = x.shape
    return n * d * np.log(0.5)


def _log_p_d_given_m2(x):
    n, d = x.shape
    k = np.sum(x, axis=0).astype(int)
    return betaln(np.sum(k) + 1, n * d - np.sum(k) + 1)


def _log_p_d_given_m3(x):
    n, _ = x.shape
    k = np.sum(x, axis=0).astype(int)
    return logsumexp(betaln(k + 1, n - k + 1))


def c(x, table_path):
    log_p_d_given_m = np.array(
        [
            _log_p_d_given_m1(x),
            _log_p_d_given_m2(x),
            _log_p_d_given_m3(x),
        ]
    )
    log_p_m_given_d = log_p_d_given_m - logsumexp(log_p_d_given_m)
    df = pd.DataFrame(
        data=np.array(
            [
                np.arange(len(log_p_m_given_d)).astype(int) + 1,
                [f"1E{int(x/np.log(10))}" for x in log_p_m_given_d[:-1]]
                + [
                    f"1-{'-'.join([f'(1E{int(x/np.log(10))})' for x in log_p_m_given_d[:-1]])}"
                ],
            ]
        ).T,
        columns=["Model", "P(M_i|D)"],
    )
    df.set_index("Model", inplace=True)
    df.to_csv(table_path)
```

src/solutions/q2.py

Question 3

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

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

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

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

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

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

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

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:

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

(b) The expression for the responsibility of mixture component k for data vector $x^{(n)}$, i.e. $r_{nk} \equiv P(s^{(n)} = k|x^{(n)}, \pi, \mathbf{P})$. This computation provides the E-step for an EM algorithm.

We know that:

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

Thus,

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

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

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

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

and r_{nk}, we have calculated $P(s^{(n)} = k|x^{(n)}, \pi, \mathbf{P})$ for the E step.

Moreover,

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

and

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

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

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

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

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

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

and

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

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

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

Combining:

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

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

For the k^{th} element of π:

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

The second derivative:

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

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

We can calculate the maximiser:

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

where λ 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:

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

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

Setting the derivative to zero:

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

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

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

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

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

thus obtaining an iterative update for the parameters π and $\mathbf{P}$ in the M-step of EM.

For numerical stability, we can compute the maximisation step for the MAP of $\mathbf{P}$,$\hat{\mathbf{P}}_{dk}^{MAP}$:

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

where

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

and

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

Assuming the same 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:

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

Substituting the appropriate expressions:

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

Simplifying:

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

This form is very similar to Question 2 (c). By a similar argument, we can see that the second derivative will always be negative (additionally $r_{nk} \geq 0$ and $\sum_n r_{nk} = 1$), ensuring a maximum in the calculation of the next step.

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

$$\hat{\mathbf{P}}_{dk}^{MAP} = \frac{\alpha - 1 + \sum_n r_{nk} x_d^{(n)}}{(N + \alpha + \beta - 2)}$$

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

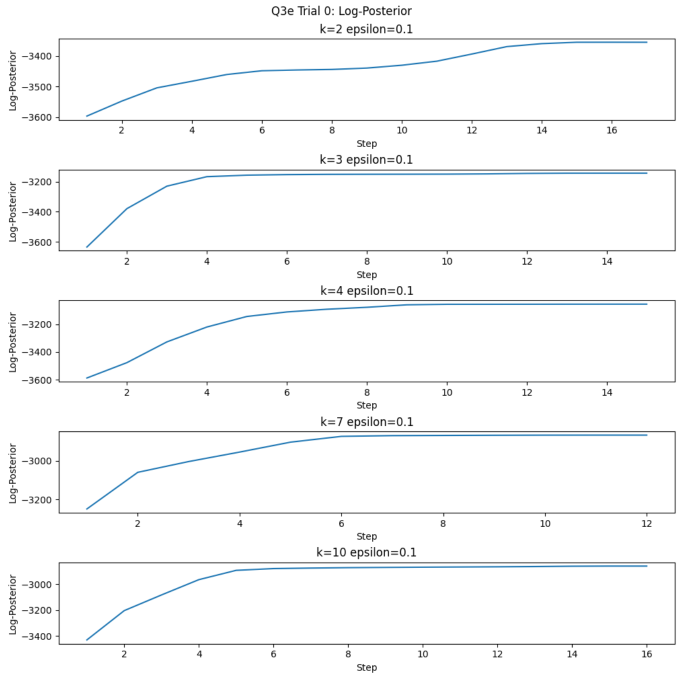

Figure 4: Log Likelihood vs Iteration Number

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

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

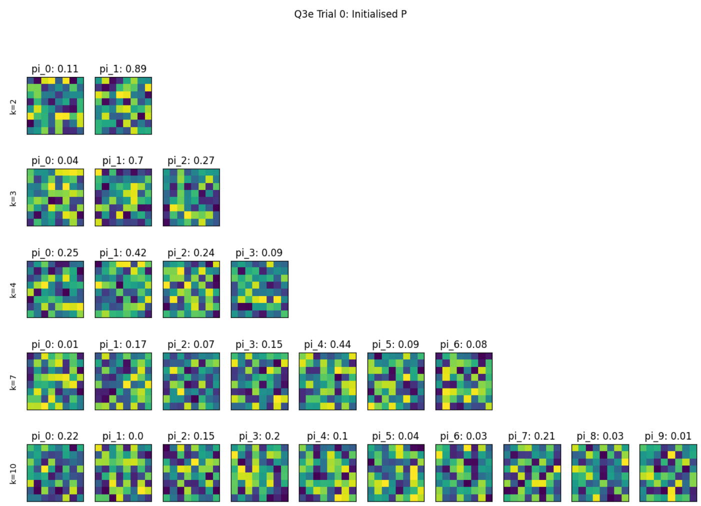

Figure 5: Randomly initialised parameters

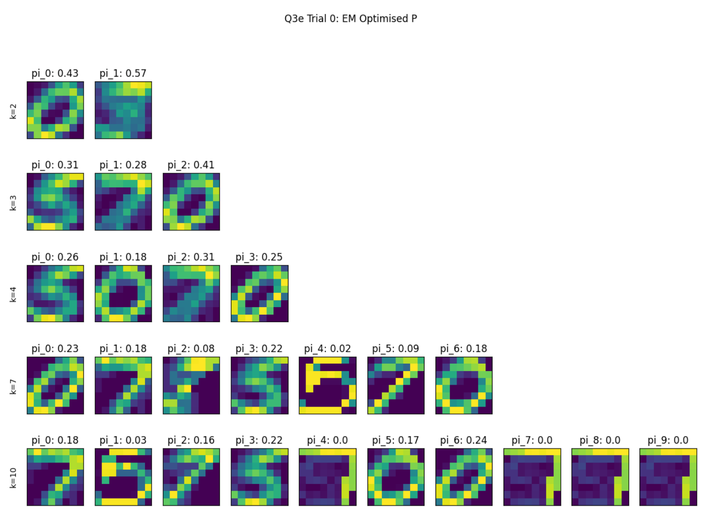

Figure 6: EM optimised parameters

The Python code for the EM algorithm:

```python
from dataclasses import dataclass
from typing import List, Tuple

import matplotlib.pyplot as plt
import numpy as np
from scipy.special import logsumexp
from sklearn.manifold import TSNE

from src.constants import DEFAULT_SEED


@dataclass
class Theta:
    """
    log_pi: the logarithm of the mixing proportions (1, k)
    log_p_matrix: the logarithm of the probability where the (i,j)th element is the probability that
                  pixel j takes value 1 under mixture component i (d, k)
    """

    log_pi: np.ndarray
    log_p_matrix: np.ndarray

    @property
    def pi(self):
        return np.exp(self.log_pi)

    @property
    def p_matrix(self):
        d, k = self.log_p_matrix.shape
        image_dimension = int(np.sqrt(d))
        return np.exp(self.log_p_matrix).reshape(image_dimension, image_dimension, -1)

    @property
    def log_one_minus_p_matrix(self) -> np.ndarray:
        """
        Compute log(1-P) where P=exp(log_p_matrix)
        :return: an array of the same shape as log_p_matrix (d, k)
        """
        log_of_one = np.zeros(self.log_p_matrix.shape)
        stacked_sum = np.stack((log_of_one, self.log_p_matrix))
        weights = np.ones(stacked_sum.shape)
        weights[1] = -1  # scale p matrix by -1 for subtraction
        return np.array(logsumexp(stacked_sum, b=weights, axis=0))

    def log_pi_repeated(self, n: int):
        """
        Repeats the log_pi vector n times along axis 0
        :param n: number of repetitions
        :return: an array of shape (n, k)
        """
        return np.repeat(self.log_pi, n, axis=0)


def _init_params(k: int, d: int, seed: int = DEFAULT_SEED) -> Theta:
    """
    Random initialisation of theta parameters (log_pi and log_p_matrix)
    :param k: Number of components
    :param d: Image dimension (number of pixels in a single image)
    :param seed: seed initialisation for random methods
    :return: theta: the parameters of the model
    """
    np.random.seed(seed)
    return Theta(
        log_pi=np.log(np.random.dirichlet(np.ones(k), size=1)),
        log_p_matrix=np.log(np.random.uniform(low=0, high=1, size=(d, k))),
    )


def _compute_log_component_p_x_i_given_theta(x: np.ndarray, theta: Theta) -> np.ndarray:
    """
    Compute the unweighted probability of each mixing component for each image
    :param x: the image data (n, d)
    :param theta: the parameters of the model
    :return: an array of the unweighted probabilities (n, k)
    """
    return x @ theta.log_p_matrix + (1 - x) @ theta.log_one_minus_p_matrix


def _compute_log_p_x_i_given_theta(x: np.ndarray, theta: Theta) -> np.ndarray:
    """
    Computes the log likelihood of each image in the dataset x
    :param x: the image data (n, d)
    :param theta: the parameters of the model
    :return: log_p_x_i_given_theta: a log likelihood array containing the log likelihood of each image (n,1)
    """
    n, _ = x.shape
    log_component_probabilities = _compute_log_component_p_x_i_given_theta(
        x, theta
    )  # (n, k)
    return np.array(
        logsumexp(
            log_component_probabilities
            + theta.log_pi_repeated(n),  # scale each component by component probability
```

```python
            axis=1,
        )
    )


def _compute_log_likelihood(x: np.ndarray, theta: Theta) -> float:
    """
    Computes the log likelihood of all images in the dataset x
    :param x: the image data (n, d)
    :param theta: the parameters of the model
    :return: log_p_x_given_theta: the log likelihood array across all images
    """
    return np.sum(_compute_log_p_x_i_given_theta(x, theta)).item()


def _compute_log_e_step(x: np.ndarray, theta: Theta) -> np.ndarray:
    """
    Compute the e step of expectation maximisation
    :param x: the image data (n, d)
    :param theta: the parameters of the model
    :return: an array of the log responsibilities of k mixture components for each image (n, k)
    """
    log_r_unnormalised = _compute_log_component_p_x_i_given_theta(x, theta)
    log_r_normaliser = logsumexp(log_r_unnormalised, axis=1)
    log_responsibility = log_r_unnormalised - log_r_normaliser[:, np.newaxis]
    return log_responsibility


def _compute_log_pi_hat(log_responsibility: np.ndarray) -> np.ndarray:
    """
    Compute the log of the maximised mixing proportions
    :param log_responsibility: an array of the log responsibilities of k mixture components for each image (n, k)
    :return: an array of the maximised log mixing proportions (1, k)
    """
    n, _ = log_responsibility.shape
    return (logsumexp(log_responsibility, axis=0) - np.log(n)).reshape(1, -1)


def _compute_log_p_matrix_hat(
    x: np.ndarray, log_responsibility: np.ndarray
) -> np.ndarray:
    """
    Compute the log of the maximised pixel probabilities
    :param x: the image data (n, d)
    :param log_responsibility: an array of the log responsibilities of k mixture components for each image (n, k)
    :return: an array of the maximised pixel probabilities for each component (d, k)
    """
    n, d = x.shape
    _, k = log_responsibility.shape

    x_repeated = np.repeat(x[:, :, np.newaxis], k, axis=2)  # (n, d, k)
    log_responsibility_repeated = np.repeat(
        log_responsibility[:, np.newaxis, :], d, axis=1
    )  # (n, d, k)

    log_p_matrix_unnormalised_likelihood = logsumexp(
        log_responsibility_repeated, b=x_repeated, axis=0
    )  # (d, k)
    log_p_matrix_normaliser_likelihood = np.array(
        logsumexp(log_responsibility_repeated, axis=0)
    )  # (d, k)

    alpha = 2
    beta = 2
    log_p_matrix_unnormalised_posterior = logsumexp(
        np.stack(
            (
                (alpha - 1) * np.ones(log_p_matrix_unnormalised_likelihood.shape),
                log_p_matrix_unnormalised_likelihood,
            ),
            axis=0,
        ),
        axis=0,
    )
    log_p_matrix_normaliser_posterior = logsumexp(
        np.stack(
            (
                (alpha + beta - 2) * np.ones(log_p_matrix_normaliser_likelihood.shape),
                log_p_matrix_normaliser_likelihood,
            ),
            axis=0,
        ),
        axis=0,
    )
    log_p_matrix_normalised_posterior = log_p_matrix_unnormalised_posterior - log_p_matrix_normaliser_posterior
    return log_p_matrix_normalised_posterior


def _compute_log_m_step(x: np.ndarray, log_responsibility: np.ndarray) -> Theta:
    """
    Compute the m step of expectation maximisation
    :param x: the image data (n, d)
    :param log_responsibility: an array of the log responsibilities of k mixture components for each image
```

```
    (n, k)
    :return: thetas optimised after maximisation step
    """
    return Theta(
        log_pi=_compute_log_pi_hat(log_responsibility),
        log_p_matrix=_compute_log_p_matrix_hat(x, log_responsibility),
    )


def _run_expectation_maximisation(
    x: np.ndarray, theta: Theta, max_number_of_steps: int, epsilon: float
) -> Tuple[Theta, np.ndarray, List[float]]:
    """
    Run the expectation maximisation algorithm
    :param x: the image data (n, d)
    :param theta: initial theta parameters
    :param max_number_of_steps: the maximum number of steps to run the algorithm
    :param epsilon: the minimum required change in log likelihood, otherwise the algorithm stops early
    :return: a tuple containing the optimised thetas, the log responsibilities,
             and the log likelihood at each step of the algorithm
    """
    log_responsibility = None
    log_likelihoods = []
    for _ in range(max_number_of_steps):
        log_responsibility = _compute_log_e_step(x, theta)
        theta = _compute_log_m_step(x, log_responsibility)

        log_likelihoods.append(_compute_log_likelihood(x, theta))

        # check for early stopping
        if len(log_likelihoods) > 1:
            if (log_likelihoods[-1] - log_likelihoods[-2]) < epsilon:
                break
    return theta, log_responsibility, log_likelihoods


def _plot_p_matrix(
    thetas: List[Theta], ks: List[int], figure_title: str, figure_path: str
):
    n = len(ks)
    m = np.max(ks)
    fig = plt.figure()
    fig.set_figwidth(15)
    fig.set_figheight(10)
    for i, k in enumerate(ks):
        for j in range(k):
            ax = plt.subplot(n, m, m * i + j + 1)
            ax.imshow(
                thetas[i].p_matrix[:, :, j],
                interpolation="None",
            )
            ax.tick_params(
                axis="x",
                which="both",
                bottom=False,
                top=False,
            )
            ax.tick_params(
                axis="y",
                which="both",
                left=False,
                right=False,
            )
            ax.xaxis.set_ticklabels([])
            ax.yaxis.set_ticklabels([])
            ax.set_title(f"pi_{j}: {np.round(thetas[i].pi[0, j], 2)}")
            if j == 0:
                ax.set_ylabel(f"{k=}")
    fig.suptitle(figure_title)
    plt.savefig(figure_path)


def _plot_tsne_responsibility_clusters(
    log_responsibilities: List[np.ndarray], ks: List[int], figure_title: str, figure_path: str
):
    n = len(ks)
    fig = plt.figure()
    fig.set_figwidth(5*n)
    fig.set_figheight(5)
    for i, k in enumerate(ks):
        embedding = TSNE(n_components=2, learning_rate='auto', init='random', perplexity=10).fit_transform(
            log_responsibilities[i])
        ax = plt.subplot(1, n, i+1)
        ax.scatter(embedding[:, 0], embedding[:, 1])
        ax.set_title(f"{k=}")
    fig.suptitle(figure_title)
    plt.savefig(figure_path,bbox_inches='tight')


def _plot_log_posteriors(
    log_posteriors: List[List[float]],
    ks: List[int],
    epsilon: float,
    figure_title: str,
    figure_path: str,
) -> None:
```

```python
    fig, ax = plt.subplots(len(ks), 1, constrained_layout=True)
    fig.set_figwidth(10)
    fig.set_figheight(10)
    for i, k in enumerate(ks):
        ax[i].plot(np.arange(1, len(log_posteriors[i]) + 1), log_posteriors[i])
        ax[i].set_xlabel("Step")
        ax[i].set_ylabel(f"Log-Posterior")
        ax[i].set_title(f"{k=} {epsilon=}")
    plt.suptitle(figure_title)

    plt.savefig(figure_path)


def e(
    x: np.ndarray,
    number_of_trials: int,
    ks: List[int],
    epsilon: float,
    max_number_of_steps: int,
    figure_path: str,
    figure_title: str,
) -> None:
    n, d = x.shape
    seeds = np.random.randint(
        low=number_of_trials * len(ks), size=(number_of_trials, len(ks))
    )
    for i in range(number_of_trials):
        init_thetas = []
        em_thetas = []
        log_posteriors = []
        log_responsibilities = []
        for j, k in enumerate(ks):
            init_theta = _init_params(k, d, seed=seeds[i, j])
            em_theta, log_responsibility, log_posterior = _run_expectation_maximisation(
                x,
                theta=init_theta,
                epsilon=epsilon,
                max_number_of_steps=max_number_of_steps,
            )
            init_thetas.append(init_theta)
            em_thetas.append(em_theta)
            log_responsibilities.append(log_responsibility)
            log_posteriors.append(log_posterior)

        _plot_p_matrix(
            init_thetas,
            ks,
            figure_title=f"{figure_title} Trial {i}: Initialised P",
            figure_path=f"{figure_path}-{i}-initialised-p.png",
        )
        _plot_p_matrix(
            em_thetas,
            ks,
            figure_title=f"{figure_title} Trial {i}: EM Optimised P",
            figure_path=f"{figure_path}-{i}-optimised-p.png",
        )
        _plot_tsne_responsibility_clusters(
            log_responsibilities,
            ks,
            figure_title=f"{figure_title} Trial {i}: TSNE Responsibility Visualisation",
            figure_path=f"{figure_path}-{i}-tsne.png",
        )
        _plot_log_posteriors(
            log_posteriors,
            ks,
            epsilon,
            figure_title=f"{figure_title} Trial {i}: Log-Posterior",
            figure_path=f"{figure_path}-{i}-log-pos.png",
        )
```

src/solutions/q3.py

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

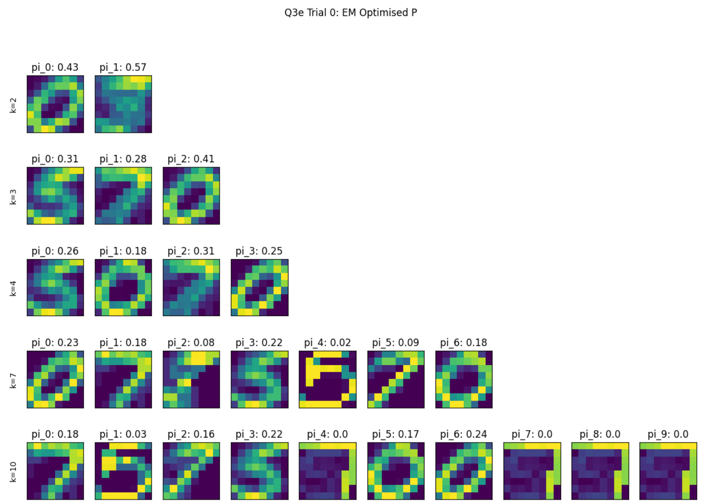

Figure 7: EM optimised parameters: Trial 0

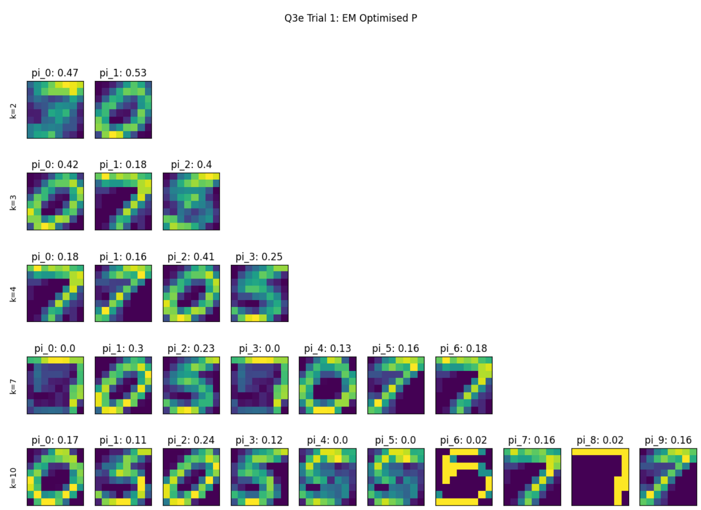

Figure 8: EM optimised parameters: Trial 1

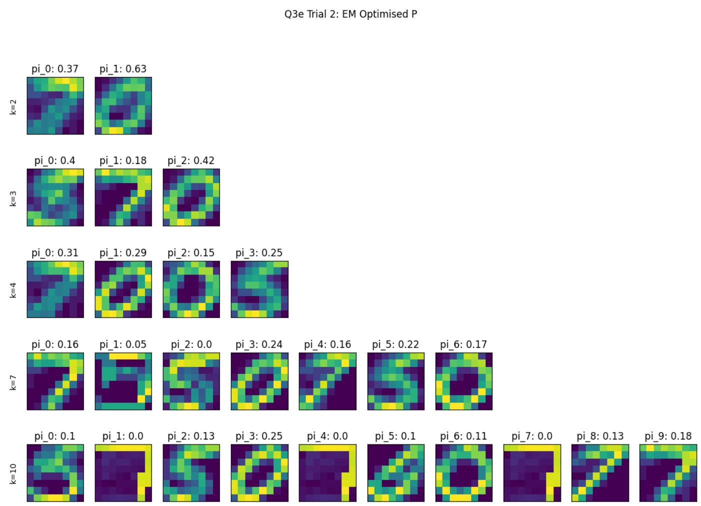

Figure 9: EM optimised parameters: Trial 2

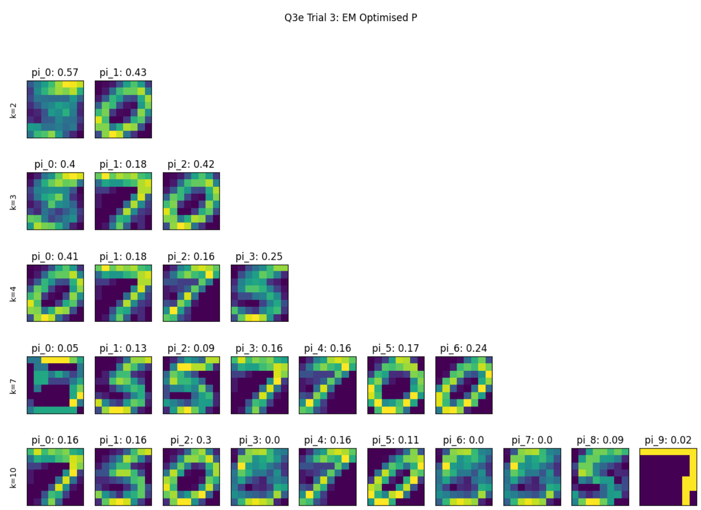

Figure 10: EM optimised parameters: Trial 3

For smaller k, we can visually see that we obtain very similar solutions (a 7 and a 0 for $k = 2$). However for higher K, we see that this may not always be the case. For Trial 0 of $k = 10$, we have two 0's whereas in Trial 1 we have three 0's. Interestingly, the zeros are all 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. π_1 and π_2 of trial 2). The corresponding parameter visualisations will essentially never be utilised because no probability is assigned to the component. This can be verified when we perform a TSNE visualisation of the responsibility vector for each of the images. We can see that for large k, qualitatively the number of clusters no longer matches the k value, indicating that some clusters are redundant. For example for $k = 7$ and $k = 10$ we can only qualitatively see four or five clusters with TSNE.

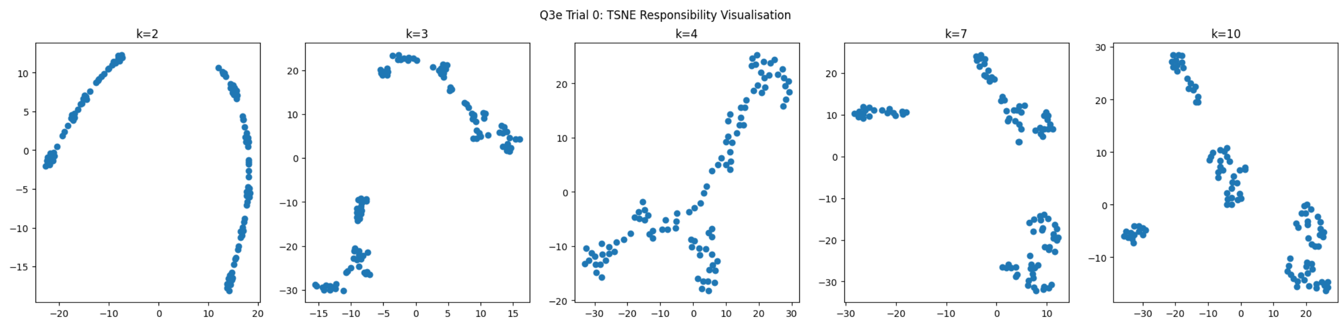

Figure 11: TSNE Visualisation of Image responsibilities: Trial 0

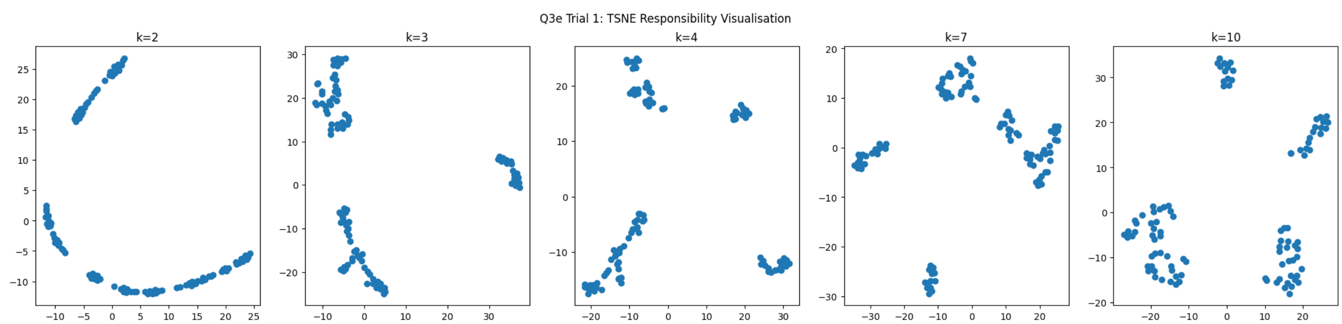

Figure 12: TSNE Visualisation of Image responsibilities: Trial 1

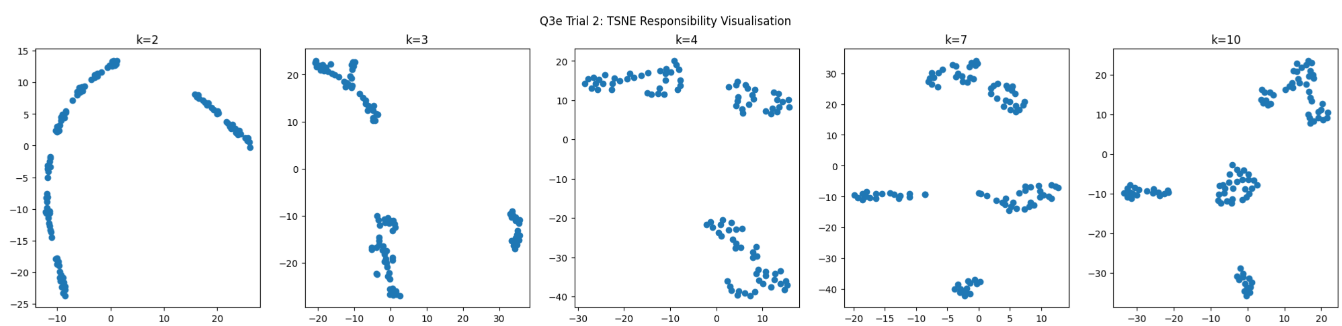

Figure 13: TSNE Visualisation of Image responsibilities: Trial 2

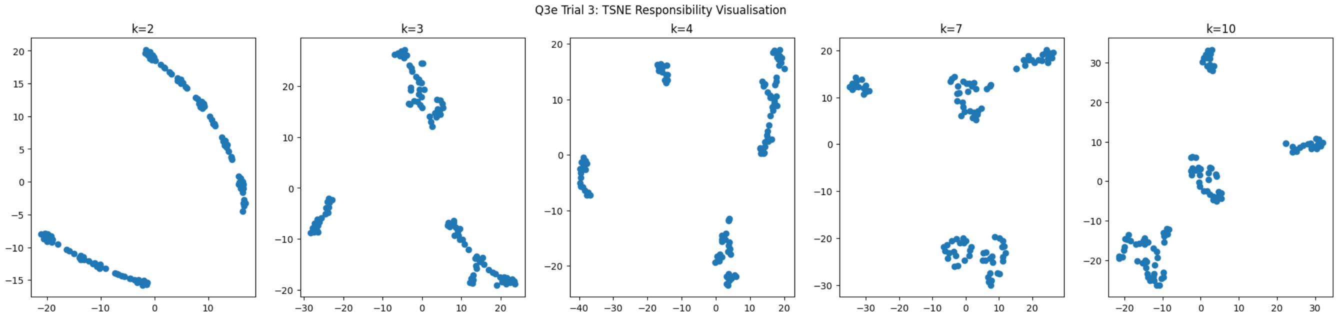

Figure 14: TSNE Visualisation of Image responsibilities: Trial 3

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

Question 5

(a) Let $p(\mathbf{s}_i, \mathbf{s}_{i-1})$ be the probability of the pair of symbols $\mathbf{s}_i$ and $\mathbf{s}_{i-1}$ occurring together in the text ($\mathbf{s}_{i-1}$ followed by $\mathbf{s}_i$ where order matters). We can model $p(\mathbf{s}_i, \mathbf{s}_{i-1})$ as a multinomial distribution with $N = 1$ and $D = 53^2$:

$$p(\mathbf{s}_i, \mathbf{s}_{i-1}) = \frac{1}{s^1!s^2!\cdots s^{53^2}} \prod_{j\in\{1,...53\},k\in\{1,...53\}} p_{s_i,s_{i-1}}^{t^{s_i,s_{i-1}}}$$

where $t^{s_i,s_{i-1}}$ is an indicator of transition s_{i-1} to s_i. For convenience we will denote $t^{(\alpha,\beta)})$ as the transition from A multinomial distribution is appropriate because there can only be only one of 53 symbols chosen as s (i.e. a 53 dimensional dice). Thus, $p(s = s_i)$ represents the probability of the symbol s_i in the text.

We can convert p(s) into exponential family form:

$$p(\mathbf{s}) = \frac{1}{s^1!s^2!\cdots s^D} \exp\left(\mathbf{s}^T \log(\mathbf{p})\right)$$

where $\mathbf{p}$ is the vector of p_i's. Thus the sufficient statistic is $T(\mathbf{s}) = \mathbf{s}^T$.

The ML estimate

(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^{rd} symbol cannot be chosen. Thus, there exists a dependence between the symbols for a given σ.

The joint probability of the encrypted text $e_1e_2\cdots e_n$ given σ:

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

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

(c) The proposal probability $S(\sigma \to \sigma')$ depends on the permutations of σ and σ' because we only choose a proposal σ' that differs 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 σ', $S(\sigma \to \sigma') = 0$, because with our method of choosing proposals, it is not possible to choose σ'. At σ there are $\binom{53}{2}$ possible proposal σ''s with the above property. Because we are assuming a uniform prior distribution over σ's, the transition probability of a σ' that satisfies the above property is $S(\sigma \to \sigma') = \frac{1}{\binom{53}{2}}$.

The MH acceptance probability is given as:

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

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

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

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

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 σ, so $P(\sigma)$ is a constant and we can simplify further:

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

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

(d) The Python code for the MH sampler:

```
```

src/solutions/q5.py

Implement the MH sampler, and run it on the provided encrypted text. Report the current decryption of the first 60 symbols after every 100 iterations. Your Markov chain should converge to give you a fairly sensible message. (Hint: it may help to initialize your chain intelligently and to try multiple times; in any case, please describe what you did). [30 marks]

TODO

(e) Note that some $\Psi(\alpha, \beta)$ values may be zero. Does this affect the ergodicity of the chain? If the chain remains ergodic, give a proof; if not, explain and describe how you can restore ergodicity. [5 marks]

TODO

(f) Analyse this approach to decoding. For instance, would symbol probabilities alone (rather than transitions) be sufficient? If we used a second order Markov chain for English text, what problems might we encounter? Will it work if the encryption scheme allows two symbols to be mapped to the same encrypted value? Would it work for Chinese with > 10000 symbols? [13 marks]

TODO

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

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

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

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

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

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 $\lambda y = -1$ from the first equation into the second equation:

$$2 + 2(-1) + x = 0$$

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

$$y^2 - 1 = 0$$

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

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

(b)

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

$$x = \ln(a)$$

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

Thus,

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

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

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

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

We can calculate:

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

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

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

Simplifying:

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

Appendix: main.py

```python
import os

import numpy as np

from src.constants import BINARY_DIGITS_FILE_PATH, OUTPUTS_FOLDER
from src.solutions import q1, q2, q3

if __name__ == "__main__":

    if not os.path.exists(OUTPUTS_FOLDER):
        os.makedirs(OUTPUTS_FOLDER)

    x = np.loadtxt(BINARY_DIGITS_FILE_PATH)
    # Question 1
    Q1_OUTPUT_FOLDER = os.path.join(OUTPUTS_FOLDER, "q1")
    if not os.path.exists(Q1_OUTPUT_FOLDER):
        os.makedirs(Q1_OUTPUT_FOLDER)

    q1.d(
        x,
        figure_path=os.path.join(Q1_OUTPUT_FOLDER, "q1d.png"),
        figure_title="Q1d: Maximum Likelihood Estimate",
    )
    q1.e(
        x,
        alpha=3,
        beta=3,
        figure_path=os.path.join(Q1_OUTPUT_FOLDER, "q1e"),
        figure_title="Q1e: Maximum A Prior",
    )

    # Question 2
    Q2_OUTPUT_FOLDER = os.path.join(OUTPUTS_FOLDER, "q2")
    if not os.path.exists(Q2_OUTPUT_FOLDER):
        os.makedirs(Q2_OUTPUT_FOLDER)
    q2.c(x, table_path=os.path.join(Q2_OUTPUT_FOLDER, "q2c.csv"))

    # Question 3
    Q3_OUTPUT_FOLDER = os.path.join(OUTPUTS_FOLDER, "q3")
    if not os.path.exists(Q3_OUTPUT_FOLDER):
        os.makedirs(Q3_OUTPUT_FOLDER)
    q3.e(
        x,
        number_of_trials=4,
        ks=[2, 3, 4, 7, 10],
        epsilon=1e-1,
        max_number_of_steps=int(1e2),
        figure_path=os.path.join(Q3_OUTPUT_FOLDER, "q3e"),
        figure_title="Q3e",
    )
```

main.py