Probabilistic and Unsupervised Learning

Victor Fizesan

November 13, 2024

1 Models for binary vectors

Consider a data set of binary (black and white) images. Each image is arranged into a vector of pixels by concatenating the columns of pixels in the image. The data set has N images $\{\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(N)}\}$ and each image has D pixels, where D is (number of rows $\times$ number of columns) in the image. For example, image $\mathbf{x}^{(n)}$ is a vector $(x_1^{(n)}, \ldots, x_D^{(n)})$ where $x_d^{(n)} \in \{0, 1\}$ for all $n \in \{1, \ldots, N\}$ and $d \in \{1, \ldots, D\}$.

(a) Explain why a multivariate Gaussian would not be an appropriate model for this data set of images. [5 marks]

A multivariate Gaussian distribution is inappropriate for this binary image dataset because it assumes continuous values, while each pixel is binary, taking values in $\{0, 1\}$. The Gaussian's unbounded support allows real values beyond $[0, 1]$, resulting in probabilities assigned to undefined values for binary data. Additionally, each pixel's true marginal distribution is Bernoulli, not Gaussian, and even a Gaussian approximation (shown in Fig. 1) misrepresents the discrete nature of the data. A Gaussian distribution might be more appropriate for modeling pixel intensities, where there is a large set of possible outcomes per feature, approximating continuity.

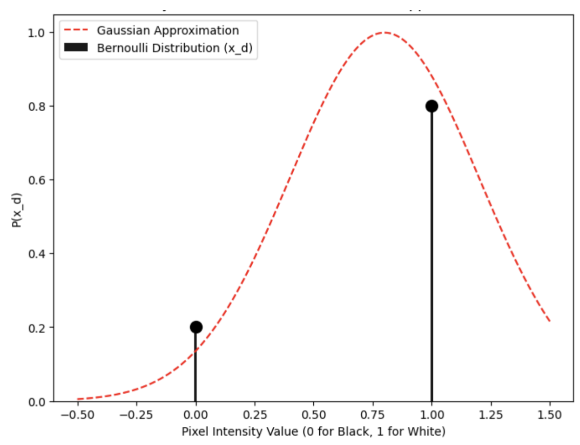

Figure 1: Binary Pixel Distribution with Gaussian Approximation

(b) Assume that the images were modelled as independently and identically distributed samples from a D-dimensional **multivariate Bernoulli distribution** with parameter vector $\mathbf{p} = (p_1, \ldots, p_D)$, which has the form

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

where both $\mathbf{x}$ and $\mathbf{p}$ are D-dimensional vectors.

What is the equation for the maximum likelihood (ML) estimate of $\mathbf{p}$? Note that you can solve for $\mathbf{p}$ directly. [5 marks]

To find the maximum likelihood (ML) estimate of $\mathbf{p} = (p_1, \ldots, p_D)$, we maximise the likelihood of observing our dataset of binary images. Given N images, each with D pixels represented by binary vectors $\mathbf{x}^{(n)} = (x_1^{(n)}, \ldots, x_D^{(n)})$, the likelihood of the dataset is

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

The maximum likelihood estimate $\hat{\mathbf{p}}$ can be obtained by solving

$$\hat{\mathbf{p}} = \arg\max_{\mathbf{p}} P(\{\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(N)}\}|\mathbf{p}) \tag{2}$$

Taking the logarithm to simplify maximisation,

$$\log P(\{\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(N)}\}|\mathbf{p}) = \sum_{n=1}^{N} \sum_{d=1}^{D} \left(x_d^{(n)} \log p_d + (1 - x_d^{(n)}) \log(1 - p_d) \right) \tag{3}$$

To find the maximum likelihood estimate, differentiate the log-likelihood with respect to each p_d and set it to zero:

$$\frac{\partial}{\partial p_d} \log P(\{\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(N)}\}|\mathbf{p}) = \sum_{n=1}^{N} \left(\frac{x_d^{(n)}}{p_d} - \frac{1 - x_d^{(n)}}{1 - p_d} \right) = 0 \tag{4}$$

Let $N_d = \sum_{n=1}^{N} x_d^{(n)}$, the count of "1"s in the d-th position across all images. Substitute N_d to simplify the summation:

$$\frac{N_d}{p_d} - \frac{N - N_d}{1 - p_d} = 0 \tag{5}$$

Multiply both sides by $p_d(1 - p_d)$ to clear the fractions:

$$N_d(1 - p_d) - (N - N_d)p_d = 0 \tag{6}$$

Expanding this, we get:

$$N_d - N_d p_d = N p_d - N_d p_d \tag{7}$$

Rearrange terms to isolate p_d on one side:

$$N_d = N p_d \tag{8}$$

Thus, the maximum likelihood estimate for each component p_d is

$$\boxed{\hat{p}_d = \frac{N_d}{N} = \frac{\sum_{n=1}^{N} x_d^{(n)}}{N}} \tag{9}$$

This is the equation for the ML estimate of $\mathbf{p}$, representing the probability that each pixel d is a "1" in the dataset.

(c) Assuming independent Beta priors on the parameters p_d

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

and $P(\mathbf{p}) = \prod_d P(p_d)$. Find the maximum a posteriori (MAP) estimator for $\mathbf{p}$. [5 marks]

To find the maximum a posteriori (MAP) estimate of $\mathbf{p} = (p_1, \ldots, p_D)$ with independent Beta priors on each p_d, we maximise the posterior distribution $P(\mathbf{p}|\{\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(N)}\})$, which is proportional to the product of the likelihood $P(\{\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(N)}\}|\mathbf{p})$ and the prior $P(\mathbf{p})$.

Given the prior on each p_d is a Beta distribution

$$P(p_d) = \frac{1}{B(\alpha, \beta)} p_d^{\alpha-1}(1-p_d)^{\beta-1}, \tag{10}$$

the full prior distribution for $\mathbf{p}$ is

$$P(\mathbf{p}) = \prod_{d=1}^{D} P(p_d) = \prod_{d=1}^{D} \frac{1}{B(\alpha, \beta)} p_d^{\alpha-1}(1-p_d)^{\beta-1} \tag{11}$$

The posterior distribution is proportional to the product of the likelihood and the prior:

$$P(\mathbf{p}|\{\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(N)}\}) \propto P(\{\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(N)}\}|\mathbf{p})P(\mathbf{p}) \tag{12}$$

Taking the logarithm, the log-posterior is

$$\log P(\mathbf{p}|\{\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(N)}\}) = \log P(\{\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(N)}\}|\mathbf{p}) + \log P(\mathbf{p}) \tag{13}$$

The log-likelihood term is

$$\log P(\{\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(N)}\}|\mathbf{p}) = \sum_{n=1}^{N}\sum_{d=1}^{D} \left(x_d^{(n)} \log p_d + (1 - x_d^{(n)}) \log(1-p_d) \right) \tag{14}$$

and the log-prior term, based on the Beta prior, is

$$\log P(\mathbf{p}) = \sum_{d=1}^{D} ((\alpha - 1)\log p_d + (\beta - 1)\log(1-p_d)) \tag{15}$$

Combining these, we get the log-posterior as:

$$\sum_{d=1}^{D} \left(\sum_{n=1}^{N} x_d^{(n)} \log p_d + \sum_{n=1}^{N} (1 - x_d^{(n)}) \log(1-p_d) + (\alpha-1)\log p_d + (\beta-1)\log(1-p_d) \right) \tag{16}$$

To find the MAP estimate, differentiate the log-posterior with respect to each p_d and set to zero:

$$\frac{\partial}{\partial p_d} \log P(\mathbf{p}|\{\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(N)}\}) = \frac{N_d + \alpha - 1}{p_d} - \frac{N - N_d + \beta - 1}{1 - p_d} = 0 \tag{17}$$

where $N_d = \sum_{n=1}^{N} x_d^{(n)}$, the count of "1"s at position d across all images.

Rearrange to solve for p_d:

$$(N_d + \alpha - 1)(1 - p_d) = (N - N_d + \beta - 1)p_d \tag{18}$$

Expanding and isolating p_d,

$$\boxed{p_d = \frac{N_d + \alpha - 1}{N + \alpha + \beta - 2} = \frac{\sum_{n=1}^{N} x_d^{(n)} + \alpha - 1}{N + \alpha + \beta - 2}} \tag{19}$$

Download the data set `binarydigits.txt` from the course website, which contains N = 100 images with D = 64 pixels each, in an N × D matrix. These pixels can be displayed as 8 × 8 images by rearranging them. View them in Matlab by running `bindigit.m` or in Python by running `bindigit.py`.

(d) Write code to learn the ML parameters of a multivariate Bernoulli from this data set and display these parameters as an 8 × 8 image. Include your code with your submission, and a visualisation of the learned parameter vector as an image. (You may use Matlab, Octave or Python) [5 marks]

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

Y = np.loadtxt('binarydigits.txt')

def calc_MLE(Y):
    MLE = np.mean(Y, axis=0)
    return MLE

def plot_estimate(Y, calc_method):
    plt.figure()
    plt.imshow(np.reshape(calc_method(Y), (8,8)), interpolation="
        None", cmap='gray')
    plt.axis('off')
    plt.show()

plot_estimate(Y, calc_MLE)
```

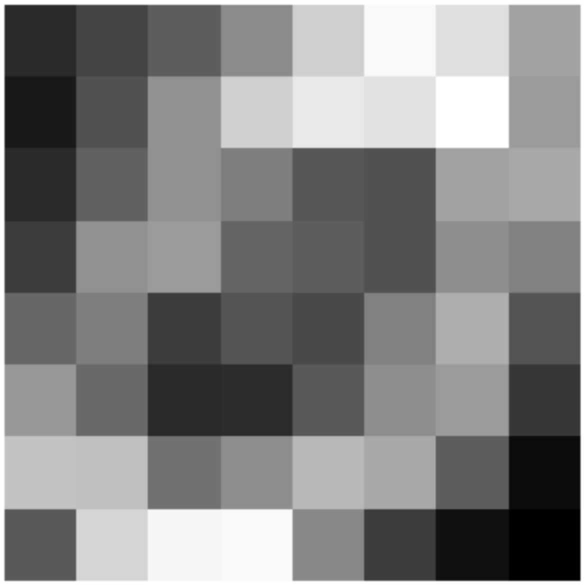

Figure 2: Maximum likelihood parameters modeled under a multivariate Bernoulli distribution

(e) Modify your code to learn MAP parameters with $\alpha = \beta = 3$. Show the new learned parameter vector for this data set as an image. Explain why this might be better or worse than the ML estimate. [5 marks]

```python
    def calc_MAP(Y, alpha=3, beta=3):
        n, _ = Y.shape
        MAP = (alpha - 1 + np.sum(Y, axis=0)) / (n + alpha + beta -
            2)
        return MAP

    plot_estimate(Y, calc_MAP)
```

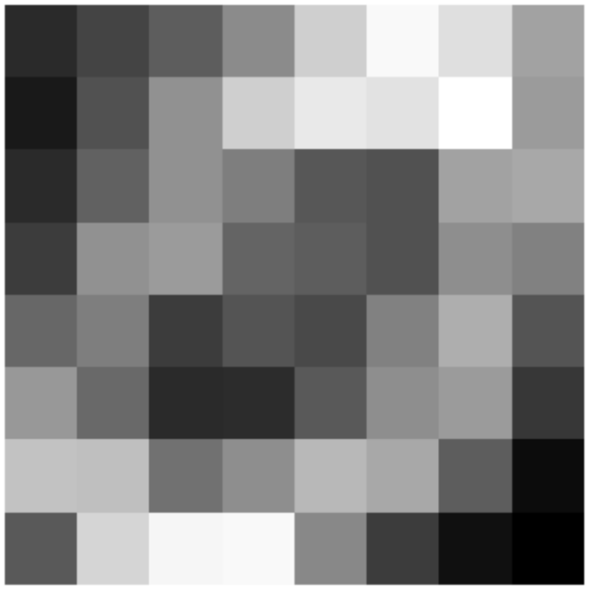

Figure 3: MAP parameters modeled with a Beta prior, $\alpha = \beta = 3$

When comparing Maximum Likelihood (ML) and Maximum A Posteriori (MAP) estimation, the key difference is that MAP incorporates prior information, while ML relies solely on the data. ML provides unbiased estimates with sufficient data but may overfit when data is limited, especially for binary cases with infrequent events. MAP estimation uses a prior, such as Beta(α, β), to regularise estimates, stabilising results in sparse datasets by avoiding extreme values. As the dataset, N, grows, MAP should approach ML. We can see this in their derived equations: (9) and (19), where as $N \to \infty$, the prior Beta parameters become overshadowed. Practically, however, MAP depends on the quality of the prior – an appropriate prior enhances estimation, but a poor prior can introduce bias, especially with small datasets. In cases where no reliable prior exists, ML is safer; with good prior knowledge, MAP can improve results.

2 Model selection

In the binary data model above, find the expressions needed to calculate the (relative) probability of the following three different models:

(a) all D components are generated from a Bernoulli distribution with $p_d = 0.5$

(b) all D components are generated from Bernoulli distributions with unknown, but identical, p_d

(c) each component is Bernoulli distributed with separate, unknown p_d

Assume that all three models are equally likely *a priori*, and take the prior distributions for any unknown probabilities to be uniform. Calculate the posterior probabilities of each of the three models having generated the data in `binarydigits.txt`.

To identify the most probable model for a dataset of binary vectors, we calculate the posterior probability $P(M_i|\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(N)})$ for each model M_i, where $i \in \{1, 2, 3\}$.

By Bayes' theorem,

$$P(M_i|\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(N)}) = \frac{P(\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(N)}|M_i)P(M_i)}{P(\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(N)})} \tag{20}$$

Assuming equal prior probabilities, $P(M_i) = \frac{1}{3}$ for all i, the posterior simplifies to

$$P(M_i|\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(N)}) = \frac{P(\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(N)}|M_i)}{\sum_{j=1}^{3} P(\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(N)}|M_j)} \tag{21}$$

We thus focus on calculating the likelihood $P(\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(N)}|M_i)$ for each model and normalizing these likelihoods.

—

In **Model (a)**, each element $x_d^{(n)}$ in the binary vectors $\mathbf{x}^{(n)}$ is generated independently from a Bernoulli distribution with parameter $p_d = 0.5$. Thus, each pixel is equally likely to be 0 or 1.

The likelihood of observing a single binary vector $\mathbf{x}^{(n)} = (x_1^{(n)}, \ldots, x_D^{(n)})$ under this model is

$$P(\mathbf{x}^{(n)}|M_1) = \prod_{d=1}^{D} (0.5)^{x_d^{(n)}} (0.5)^{1-x_d^{(n)}} = (0.5)^D \tag{22}$$

because $(0.5)^{x_d^{(n)}} (0.5)^{1-x_d^{(n)}} = 0.5$ for each d. Assuming independence across all N observations, the likelihood for the entire dataset is

$$P(\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(N)}|M_1) = (0.5)^{ND} \tag{23}$$

—

In **Model (b)**, each element $x_d^{(n)}$ in each binary vector $\mathbf{x}^{(n)}$ is generated independently from a Bernoulli distribution with an unknown but identical parameter p for all dimensions. The likelihood of observing the entire dataset given p is

$$P(\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(N)}|p, M_2) = \prod_{n=1}^{N} \prod_{d=1}^{D} p^{x_d^{(n)}} (1-p)^{1-x_d^{(n)}} \tag{24}$$

Let $\sum_{n=1}^{N} \sum_{d=1}^{D} x_d^{(n)}$ represent the total count of 1's across all elements, and let $\sum_{n=1}^{N} \sum_{d=1}^{D} (1 - x_d^{(n)})$ represent the total count of 0's. Then, we can rewrite the likelihood as

$$P(\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(N)}|p, M_2) = p^{\sum_{n=1}^{N} \sum_{d=1}^{D} x_d^{(n)}} (1-p)^{\sum_{n=1}^{N} \sum_{d=1}^{D} (1-x_d^{(n)})} \tag{25}$$

To account for the unknown p, we integrate over all possible values of p with a uniform prior:

$$P(\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(N)}|M_2) = \int_0^1 p^{\sum_{n=1}^{N} \sum_{d=1}^{D} x_d^{(n)}} (1-p)^{\sum_{n=1}^{N} \sum_{d=1}^{D} (1-x_d^{(n)})} \, dp \tag{26}$$

This integral is a Beta function $B(\alpha, \beta)$ where

$$\alpha = 1 + \sum_{n=1}^{N} \sum_{d=1}^{D} x_d^{(n)}, \quad \beta = 1 + \sum_{n=1}^{N} \sum_{d=1}^{D} (1 - x_d^{(n)}) \tag{27}$$

Thus,

$$P(\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(N)}|M_2) = B(\alpha, \beta) \tag{28}$$

—

In **Model (c)**, each dimension d has its own independent Bernoulli parameter p_d, unknown and uniformly distributed over $[0, 1]$. The likelihood for the entire dataset is

$$P(\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(N)} | \mathbf{p}, M_3) = \prod_{d=1}^{D} \prod_{n=1}^{N} p_d^{x_d^{(n)}} (1 - p_d)^{1 - x_d^{(n)}} \tag{29}$$

Let $\sum_{n=1}^{N} x_d^{(n)}$ denote the count of 1's in dimension d across all N observations, and $\sum_{n=1}^{N} (1 - x_d^{(n)})$ denote the count of 0's. The likelihood simplifies to

$$P(\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(N)} | \mathbf{p}, M_3) = \prod_{d=1}^{D} p_d^{\sum_{n=1}^{N} x_d^{(n)}} (1 - p_d)^{\sum_{n=1}^{N} (1 - x_d^{(n)})} \tag{30}$$

To marginalise over each p_d, integrate each with a uniform prior:

$$P(\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(N)} | M_3) = \prod_{d=1}^{D} \int_0^1 p_d^{\sum_{n=1}^{N} x_d^{(n)}} (1 - p_d)^{\sum_{n=1}^{N} (1 - x_d^{(n)})} \, dp_d \tag{31}$$

Each integral is a Beta function $B(\alpha_d, \beta_d)$ where

$$\alpha_d = 1 + \sum_{n=1}^{N} x_d^{(n)}, \quad \beta_d = 1 + \sum_{n=1}^{N} (1 - x_d^{(n)}) \tag{32}$$

Thus,

$$P(\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(N)} | M_3) = \prod_{d=1}^{D} B(\alpha_d, \beta_d) \tag{33}$$

—

To determine which model best explains the binary data Y, we calculate the posterior probability for each model given the observed data. Assuming equal prior probability for each model, the posterior for each model M_i is given by equation (21).

We make this calculation numerically stable by computing log-likelihoods, normalizing with the '`logsumexp`' trick, and then exponentiating to get posterior probabilities.
The log-likelihoods for each model are:

(a) fixed $p_d = 0.5$:

$$\log P(Y | M_a) = ND \log(0.5) \tag{34}$$

(b) identical unknown p across dimensions:

$$\log P(Y | M_b) = \log \Gamma(\alpha) + \log \Gamma(\beta) - \log \Gamma(\alpha + \beta) \tag{35}$$

where $\alpha = 1 +$ total count of 1's and $\beta = 1 +$ total count of 0's.

(c) independent unknown p_d for each dimension:

$$\log P(Y | M_c) = \sum_{d=1}^{D} (\log \Gamma(\alpha_d) + \log \Gamma(\beta_d) - \log \Gamma(\alpha_d + \beta_d)) \tag{36}$$

where $\alpha_d = 1 +$ count of 1's in dimension d and $\beta_d = 1 +$ count of 0's in dimension d.

This approach enables stable calculation of posterior probabilities by utilising log transformations and the '`logsumexp`' trick to prevent underflow.

```python
import numpy as np
from scipy.special import gammaln, logsumexp
import matplotlib.pyplot as plt

Y = np.loadtxt('binarydigits.txt')

def model_a_log_likelihood(Y):
    N, D = Y.shape
    return N * D * np.log(0.5)

def model_b_log_likelihood(Y):
    N, D = Y.shape
    total_ones = np.sum(Y)
    total_zeros = N * D - total_ones
    alpha = 1 + total_ones
    beta = 1 + total_zeros
    return gammaln(alpha) + gammaln(beta) - gammaln(alpha + beta)

def model_c_log_likelihood(Y):
    N, D = Y.shape
    log_likelihood = 0.0
    for d in range(D):
        count_ones = np.sum(Y[:, d])
        count_zeros = N - count_ones
        alpha_d = 1 + count_ones
        beta_d = 1 + count_zeros
        log_likelihood += gammaln(alpha_d) + gammaln(beta_d) -
            gammaln(alpha_d + beta_d)
    return log_likelihood

def log_posterior(Y):
    log_likelihoods = {
        'Model A': model_a_log_likelihood(Y),
        'Model B': model_b_log_likelihood(Y),
        'Model C': model_c_log_likelihood(Y)
    }
    total_log_likelihood = logsumexp(list(log_likelihoods.values()))
    log_posteriors = {model: ll - total_log_likelihood for model, ll
        in log_likelihoods.items()}
    return log_posteriors

# Calculate posterior probabilities based on log posteriors
log_posteriors = log_posterior(Y)
unscaled_posteriors = {model: np.exp(lp) for model, lp in
    log_posteriors.items()}
total_unscaled = sum(unscaled_posteriors.values())
posterior_probabilities = {model: unscaled / total_unscaled for model
    , unscaled in unscaled_posteriors.items()}

for model, posterior in posterior_probabilities.items():
    print(f"{model}: Posterior Probability = {posterior:.4e}")
```

```
Model A: Posterior Probability = 9.14e-255
Model B: Posterior Probability = 1.43e-188
Model C: Posterior Probability = 1.00e+00
```

3 EM for Binary Data

Consider the data set of binary (black and white) images used in the previous question.

(a) Write down the likelihood for a model consisting of a mixture of K multivariate Bernoulli distributions. Use the parameters $\pi_1, \dots, \pi_K$ to denote the mixing proportions ($0 \le \pi_k \le 1; \sum_k \pi_k = 1$) and arrange the K Bernoulli parameter vectors into a matrix P with elements p_{kd} denoting the probability that pixel d takes value 1 under mixture component k. Assume the images are iid under the model, and that the pixels are independent of each other within each component distribution. [5 marks]

Let $x^{(n)} = \{x_d^{(n)}\}_{d=1}^D$ represent a binary image, where each $x_d^{(n)} \in \{0,1\}$.

The likelihood for a single image $x^{(n)}$ is:

$$p(x^{(n)}|\pi, P) = \sum_{k=1}^K \pi_k \prod_{d=1}^D p_{kd}^{x_d^{(n)}} (1 - p_{kd})^{1-x_d^{(n)}} \tag{37}$$

Thus the total likelihood for the dataset $\{x^{(n)}\}_{n=1}^N$ is:

$$\boxed{\mathcal{L}(\pi, P) = P(x^{(1)}, \dots, x^{(N)}|\pi, P) = \prod_{n=1}^N \left(\sum_{k=1}^K \pi_k \prod_{d=1}^D p_{kd}^{x_d^{(n)}} (1 - p_{kd})^{1-x_d^{(n)}} \right)} \tag{38}$$

Just as we can for a mixture of Gaussians, we can formulate this mixture as a latent variable model, introducing a discrete hidden variable $s^{(n)} \in \{1, \dots, K\}$ where $P(s^{(n)} = k|\pi) = \pi_k$.

(b) Write down the expression for the responsibility of mixture component k for data vector $\mathbf{x}^{(n)}$, i.e., $r_{nk} \equiv P(s^{(n)} = k|\mathbf{x}^{(n)}, \pi, P)$. This computation provides the E-step for an EM algorithm. [5 marks]

The responsibility r_{nk}, representing the probability that mixture component k is responsible for observation $x^{(n)}$, is given by:

$$r_{nk} \equiv P(s^{(n)} = k|x^{(n)}, \pi, P) = \frac{\pi_k P(x^{(n)}|s^{(n)} = k, P)}{\sum_{j=1}^K \pi_j P(x^{(n)}|s^{(n)} = j, P)} \tag{39}$$

Expanding the terms, we get:

$$\boxed{r_{nk} = \frac{\pi_k \prod_{d=1}^D p_{kd}^{x_d^{(n)}} (1 - p_{kd})^{1-x_d^{(n)}}}{\sum_{j=1}^K \pi_j \prod_{d=1}^D p_{jd}^{x_d^{(n)}} (1 - p_{jd})^{1-x_d^{(n)}}}} \tag{40}$$

(c) Find the maximizing parameters for the expected log-joint

$$\arg\max_{\pi, P} \left\langle \sum_n \log P(\mathbf{x}^{(n)}, s^{(n)}|\pi, P) \right\rangle_{q(\{s^{(n)}\})} \tag{41}$$

thus obtaining an iterative update for the parameters π and P in the M-step of EM. [10 marks]

From Equation (37), we can derive the log-likelihood of a single image $x^{(n)}$ under a Bernoulli mixture model as:

$$\log P(\mathbf{x}^{(n)}, s^{(n)}|\pi, P) = \log \pi_{s^{(n)}} + \sum_{d=1}^{D} \left(x_d^{(n)} \log p_{s^{(n)}d} + (1 - x_d^{(n)}) \log(1 - p_{s^{(n)}d}) \right) \tag{42}$$

We now declare the variational distribution, $q(s^{(n)}) = P(s^{(n)}|\mathbf{x}^{(n)}, \pi, P)$, as the set of responsibilities r_{nk}, denoted by Equation (40). This allows us to equate the expected log-joint as:

$$\sum_{n=1}^{N} \sum_{k=1}^{K} r_{nk} \left(\log \pi_k + \sum_{d=1}^{D} \left(x_d^{(n)} \log p_{kd} + (1 - x_d^{(n)}) \log(1 - p_{kd}) \right) \right) \tag{43}$$

We can now maximise this expectation by separately optimising with respect to each element in π and P.

To maximise with respect to π_k, we extract the differentiable terms from Equation (43), and introduce a Lagrange multiplier λ that ensures $\sum_{k=1}^{K} \pi_k = 1$:

$$\mathcal{L} = \sum_{n=1}^{N} \sum_{k=1}^{K} r_{nk} \log \pi_k + \lambda \left(1 - \sum_{k=1}^{K} \pi_k \right) \tag{44}$$

Taking the derivative with respect to π_k and setting it to zero, we obtain:

$$\frac{\partial \mathcal{L}}{\partial \pi_k} = \frac{\sum_{n=1}^{N} r_{nk}}{\pi_k} - \lambda = 0, \tag{45}$$

where λ ensures the mixing coefficients sum to 1. Thus we find our first maximised parameter:

$$\boxed{\hat{\pi}_k = \frac{1}{N} \sum_{n=1}^{N} r_{nk}} \tag{46}$$

To maximise with respect to p_{kd}, we again separate the differentiable terms from the expected log-joint and set the derivative to zero:

$$\frac{\partial}{\partial p_{kd}} \sum_{n=1}^{N} r_{nk} \left(x_d^{(n)} \log p_{kd} + (1 - x_d^{(n)}) \log(1 - p_{kd}) \right) = 0 \tag{47}$$

Expanding the derivative, we have:

$$\sum_{n=1}^{N} r_{nk} \left(\frac{x_d^{(n)}}{p_{kd}} - \frac{1 - x_d^{(n)}}{1 - p_{kd}} \right) = 0 \tag{48}$$

Multiplying through by $p_{kd}(1 - p_{kd})$ to clear the denominators:

$$\sum_{n=1}^{N} r_{nk} x_d^{(n)} (1 - p_{kd}) = \sum_{n=1}^{N} r_{nk} (1 - x_d^{(n)}) p_{kd} \tag{49}$$

Cancelling out $\sum_{n=1}^{N} r_{nk} x_d^{(n)} p_{kd}$ and solving for p_{kd}, we obtain:

$$\boxed{\hat{p}_{kd} = \frac{\sum_{n=1}^{N} r_{nk} x_d^{(n)}}{\sum_{n=1}^{N} r_{nk}}} \tag{50}$$

(d) Implement the EM algorithm for a mixture of K multivariate Bernoullis.

Your code should take as input the number K, a matrix X containing the data set, and a maximum number of iterations to run. The algorithm should terminate after that number of iterations, or earlier if the log likelihood converges (does not increase by more than a very small amount).

Hand in clearly commented code.

Run your algorithm on the data set for values of K in $\{2, 3, 4, 7, 10\}$. Plot the log likelihood as a function of the iteration number, and display the parameters found. [30 marks]

```python

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

Y = np.loadtxt('binarydigits.txt')
N, D = Y.shape
parameters = {}

def init_params(K, D):
    """
    Initialise parameters for Bernoulli mixture model.
    """
    log_pi = np.log(np.random.dirichlet(np.ones(K), size=1)).flatten()
        # Mixing proportions
    log_p_matrix = np.log(np.random.uniform(low=0.1, high=0.9, size=(K,
        D)))  # Bernoulli probabilities
    return log_pi, log_p_matrix

def compute_log_component_likelihood(Y, log_p_matrix):
    """
    Compute log-likelihood of each pixel under each mixture component.
    """
    return Y @ log_p_matrix.T + (1 - Y) @ (np.log(1 - np.exp(
        log_p_matrix))).T

def e_step(Y, log_pi, log_p_matrix):
    """
    E-step: compute log responsibilities.
    """
    log_component_likelihood = compute_log_component_likelihood(Y,
        log_p_matrix)
    log_r_nk = log_pi + log_component_likelihood  # Log responsibility
        numerator
    log_r_nk -= logsumexp(log_r_nk, axis=1, keepdims=True)  # Normalise
        across components
    return log_r_nk

def m_step(Y, log_r_nk, alpha_prior, beta_prior):
    """
    M-step: update log mixing proportions and log Bernoulli parameters.
    """
    N_k = np.exp(logsumexp(log_r_nk, axis=0))
    log_pi = np.log(N_k / len(Y))

    # Update log_p_matrix using MAP with a Beta prior
    p_matrix_hat = (np.exp(log_r_nk).T @ Y + alpha_prior - 1) / (N_k[:,
        None] + alpha_prior + beta_prior - 2)
    log_p_matrix = np.log(p_matrix_hat)
```

```python
        return log_pi, log_p_matrix

    def compute_log_posterior(Y, log_pi, log_p_matrix, alpha_prior, beta_prior):
        """
        Compute total log posterior for dataset, across components.
        """
        log_component_likelihood = compute_log_component_likelihood(Y, log_p_matrix)
        log_likelihood = np.sum(logsumexp(log_pi + log_component_likelihood, axis=1))

        log_prior_pi = np.sum((alpha_prior - 1) * log_pi)
        log_prior_p_matrix = np.sum(
            (alpha_prior - 1) * log_p_matrix + (beta_prior - 1) * np.log(1 - np.exp(log_p_matrix)))

        return log_likelihood + log_prior_pi + log_prior_p_matrix

    def em_algorithm(Y, K, max_iter=100, tol=1e-6, alpha_prior=1.1, beta_prior=1.1):
        """
        Run EM algorithm with specified parameters.
        """
        log_pi, log_p_matrix = init_params(K, D)
        log_posteriors = []

        for iteration in range(max_iter):
            log_r_nk = e_step(Y, log_pi, log_p_matrix)
            log_pi, log_p_matrix = m_step(Y, log_r_nk, alpha_prior, beta_prior)
            log_posterior = compute_log_posterior(Y, log_pi, log_p_matrix, alpha_prior, beta_prior)
            log_posteriors.append(log_posterior)

            if iteration > 0 and abs(log_posteriors[-1] - log_posteriors[-2]) < tol:
                break

        return log_pi, log_p_matrix, log_posteriors

    def plot_log_posterior(log_posteriors, K):
        """
        Plot the log posterior as a function of iteration number.
        """
        plt.plot(log_posteriors, label=f'K={K}', color='black', linewidth=1.5)

    plt.figure(figsize=(10, 6))

    for K in [2, 3, 4, 7, 10]:
        log_pi, log_p_matrix, log_posteriors = em_algorithm(Y, K)
        plot_log_posterior(log_posteriors, K)
        parameters[K] = (log_pi, log_p_matrix)

    plt.xlabel('Iteration', fontsize=14)
    plt.ylabel('Log Posterior', fontsize=14)
    plt.title('Log Posterior Convergence for Different K Values (Grayscale)', fontsize=16)
    plt.tight_layout()
    plt.show()
```

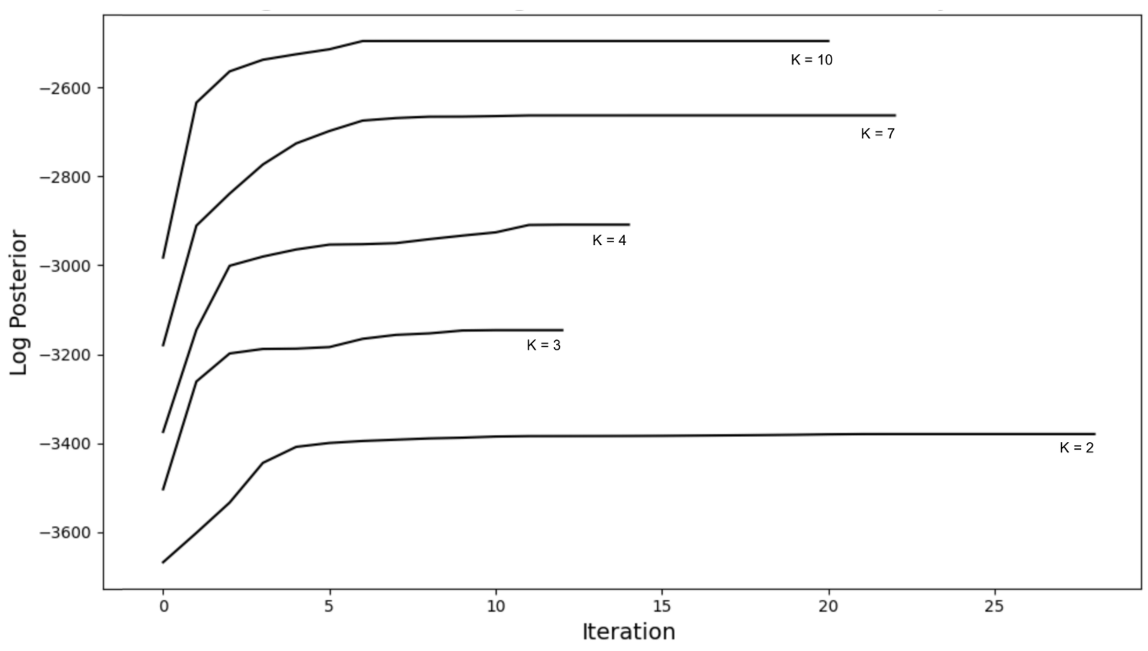

Figure 4: As the number of mixture components increases, the model tends to explain the data with a greater likelihood.

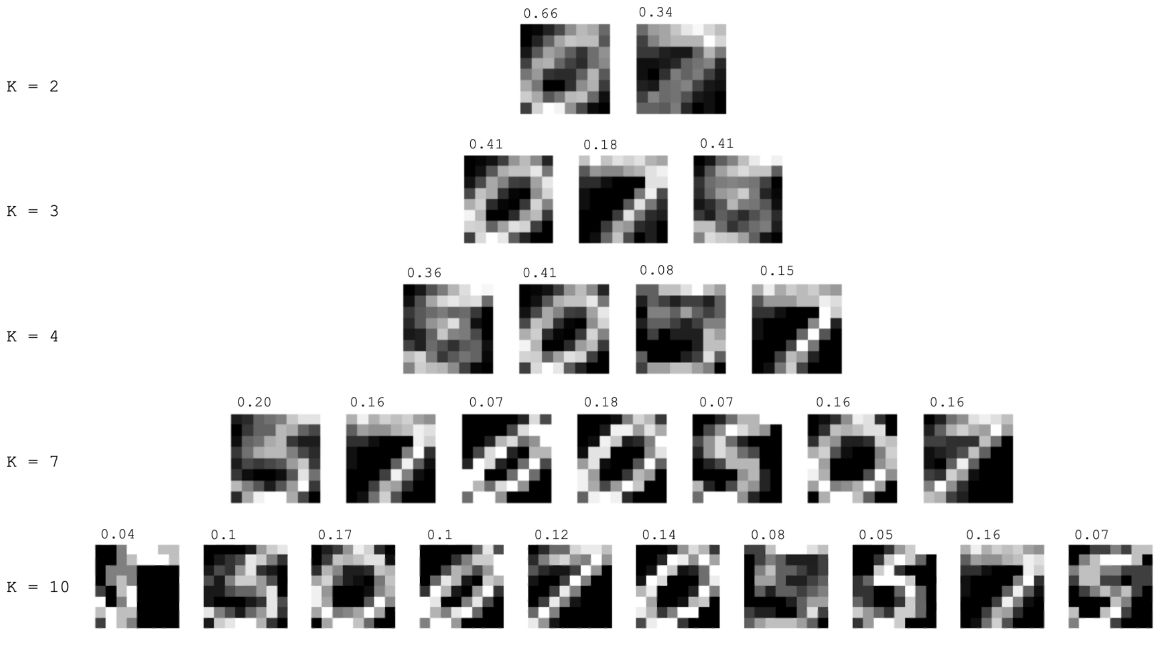

Figure 5: EM optimised pixel intensities and mixing coefficients for various K.

The methodology begins by initialising the mixing proportions ($\log \pi$) and the Bernoulli parameters ($\log p\_\text{matrix}$) for each component. We initialise $\log p\_\text{matrix}$ values between 0.1 and 0.9 to prevent extreme values. We also use a Dirichlet distribution on $\log \pi$ to ensure non-negative, summing-to-one proportions for valid mixing weights. The choice of log-space helps to mitigate numerical instability, which is critical as small likelihood values are common in high-dimensional data.

For the M-step, we use a Beta prior on the Bernoulli parameters to compute maximum a posteriori (MAP) estimates rather than maximum likelihood (ML) estimates. This choice further stabilises the model by providing regularisation, particularly beneficial when the data is sparse or the model has many components. We chose a symmetric Beta prior with parameters slightly greater than 1.0 to weakly guide the parameter updates. With greater parameter values on the Beta prior, we noticed a significant increase in "switched-off" components for high K values. We interpret this as the prior over-regularising the model and the high K count giving the model excessive flexibility to overfit the dataset.

(e) Run the algorithm a few times starting from randomly chosen initial conditions. Do you obtain the same solutions (up to permutation)? Does this depend on K? Show the learned probability vectors as images.

Comment on how well the algorithm works, whether it finds good clusters (look at the cluster means and responsibilities and try to interpret them), and how you might improve the model. [10 marks]

For different values of K, we observed notable clustering patterns, as seen in Figure 5. We note that in the original dataset three distinct digits appear: 0, 5, and 7. As such, we expect that drawing from three different latent distributions would effectively represent our data. Indeed, under $K = 3$, we see three distinct digits. In contrast, higher values of K, such as $K = 7$ and $K = 10$, introduce redundancy, with multiple components capturing subtle variations of the same digit (e.g., different ways of writing 7). This leads to some components having very low probabilities, indicating over-partitioning of similar patterns rather than capturing truly distinct clusters. While larger K values allow for finer detail – as can be seen for $K = 10$, where numbers are very granular – they risk overfitting by fitting minor variations rather than broader digit forms, underscoring the importance of balancing model complexity with data diversity.

In Figure 6 we display the mean and inter-quartile log-likelihoods over 50 runs. We notice a well defined pattern where increasing K leads to higher log-likelihoods. We also notice that models with lower complexity tend to converge with fewer iterations.

To improve the model, we could apply selection criteria such as the Bayesian Information Criterion (BIC) or Akaike Information Criterion (AIC) to determine an optimal K value that balances model complexity with data fit. Additionally, refining the initialisation step by using a soft clustering method or setting more informative priors could yield more distinct clusters without relying heavily on higher K values to increase our model likelihood. Finally, enforcing a minimum probability threshold during updates could prevent clusters from becoming inactive, ensuring each component contributes meaningfully to the data representation.

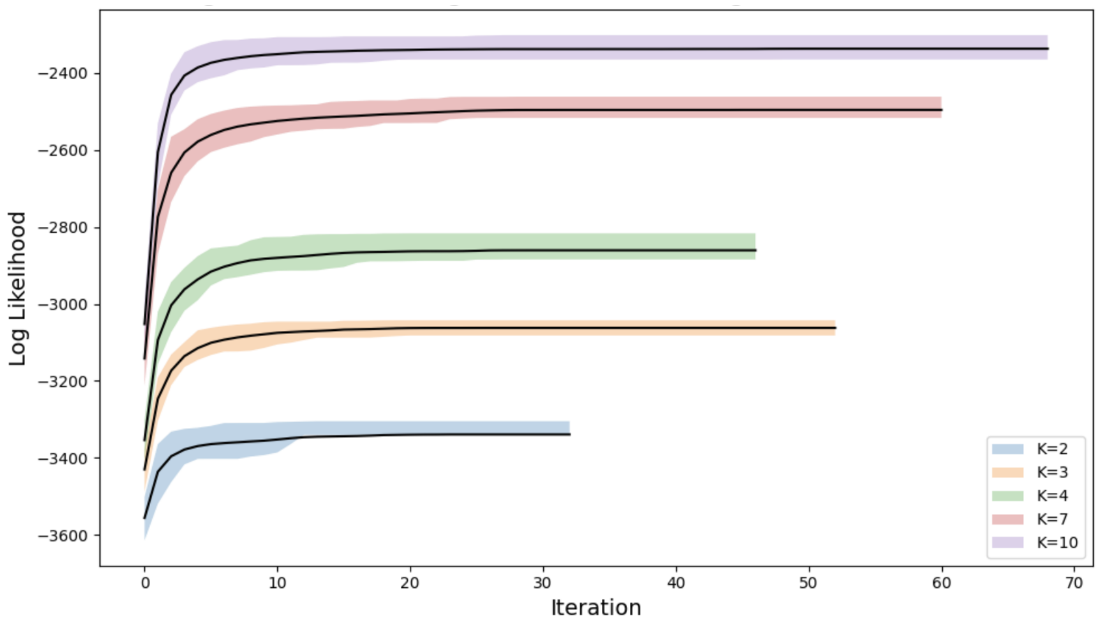

Figure 6: Inter-quartile ranges for 50 runs of the EM algorithm on different K.

5. Decrypting Messages with MCMC

You are given a passage of English text that has been encrypted by remapping each symbol to a (usually) different one. For example,

$$\begin{array}{ccc} a & \to & s \\ b & \to & ! \\ \text{(space)} & \to & v \\ \vdots & & \vdots \end{array}$$

Thus a text like 'a boy...' might be encrypted by 'sv!op...'. Assume that the mapping between symbols is one-to-one. The file `symbols.txt` gives the list of symbols, one per line (the second line is (space)). The file `message.txt` gives the encrypted message.

Decoding the message by brute force is impossible, since there are 53 symbols and thus 53! possible permutations to try. Instead we will set up a Markov chain Monte Carlo sampler to find modes in the space of permutations.

We model English text, say $s_1 s_2 \cdots s_n$ where s_i are symbols, as a Markov chain, so that each symbol is independent of the preceding text given only the symbol before:

$$p(s_1 s_2 \cdots s_n) = p(s_1) \prod_{i=2}^{n} p(s_i | s_{i-1})$$

(a) Learn the transition statistics of letters and punctuation in English: Download a large text [say the English translation of *War and Peace*] and estimate the transition probabilities $p(s_i = \alpha | s_{i-1} = \beta) \equiv \psi(\alpha, \beta)$, as well as the stationary distribution $\lim_{i\to\infty} p(s_i = \gamma) \equiv \phi(\gamma)$. Assume that the first letter of your text (and also that of the encrypted text provided) is itself sampled from the stationary distribution.

Give formulae for the ML estimates of these probabilities as functions of the counts of numbers of occurrences of symbols and pairs of symbols.

Compute the estimated probabilities. Report the values as a table. [6 marks]

The Maximum Likelihood Estimate (MLE) for the transition probability $\psi(\alpha, \beta)$ is given by

$$\psi^{\mathrm{ML}}(\alpha, \beta) = \frac{N(\alpha, \beta)}{N_\beta} \tag{51}$$

where $N_\beta = \sum_\alpha N(\alpha, \beta)$ is the total count of occurrences of β as a preceding symbol. To ensure ergodicity and avoid zero probabilities, we add a smoothing factor of 1 to each $N(\alpha, \beta)$, updating N_β accordingly by adding the number of unique symbols.

The stationary distribution $\varphi(\gamma)$, representing the long-term probability of each symbol γ, is calculated as

$$\varphi(\gamma) = \frac{N(\gamma)}{\sum_{\gamma'} N(\gamma')} \tag{52}$$

We approximate $\varphi(\gamma)$ through the power method to find the steady-state vector of the transition matrix Ψ, where $\Psi_{i,j} = p(s^j | s^i)$ and s^i is the i-th symbol.

The estimated values for $\psi(\alpha, \beta)$ and $\varphi(\gamma)$ are presented below.

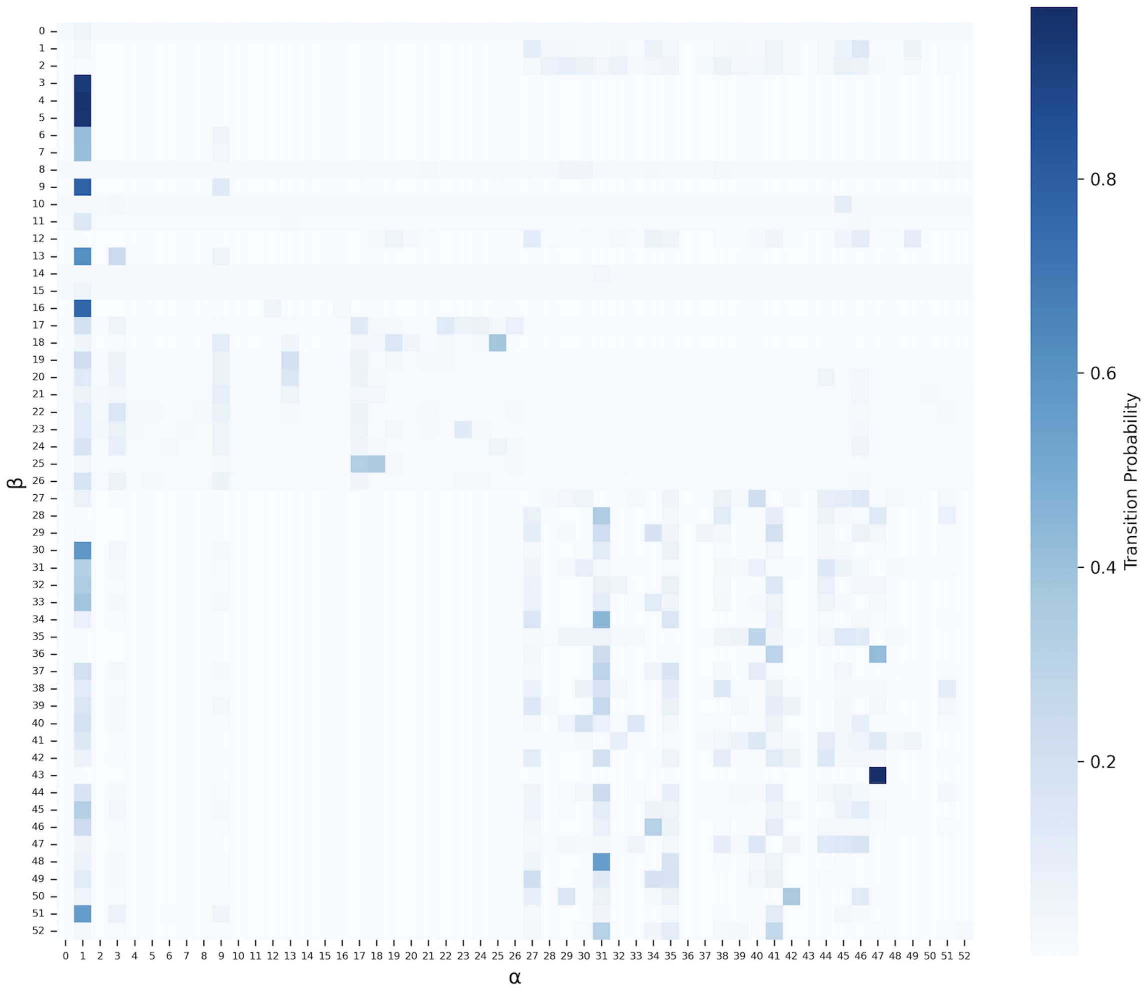

Figure 7: Heatmap of bigram transition probabilities under the War and Peace distribution.

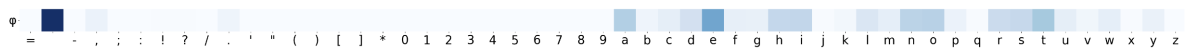

Figure 8: Heatmap of stationary distribution probabilities for each symbol.

(b) The state variable for our MCMC sampler will be the symbol permutation. Let $\sigma(s)$ be the symbol that stands for symbol s in the encrypted text, e.g., $\sigma(a) = s$ and $\sigma(b) =!$ above.

Assume a uniform prior distribution over permutations.

Are the latent variables $\sigma(s)$ for different symbols s independent? Let $e_1 e_2 \cdots e_n$ be an encrypted English text. Write down the joint probability of $e_1 e_2 \cdots e_n$ given σ. [6 marks]

The latent variables for different symbols s exhibit interdependence due to the one-to-one mapping of the permutation σ. Once a symbol in the plaintext is assigned a particular encoding, other symbols are restricted from being mapped to the same encoding.

Assuming a first-order Markov chain for the original text, the same transition matrix applies to the encrypted text, allowing us to compute the joint probability of observing the sequence $e_1, e_2, \ldots, e_n$ in the encrypted text under a specific permutation σ. This joint probability is given by

$$\boxed{p(e_1, e_2, \ldots, e_n \mid \sigma) = \prod_{i=1}^{n} p(e_i \mid e_{i-1}, \sigma) = \prod_{i=1}^{n} \psi(\sigma^{-1}(e_i), \sigma^{-1}(e_{i-1}))} \tag{53}$$

where e_i denotes the i-th symbol in the encrypted sequence, e_{i-1} is the preceding symbol, and $\psi(\sigma^{-1}(e_i), \sigma^{-1}(e_{i-1}))$ represents the transition probability from e_{i-1} to e_i based on the established mapping σ. Here, σ^{-1} reverses the mapping to yield the decrypted original symbols.

(c) We use a Metropolis-Hastings (MH) chain, with the proposal given by choosing two symbols s and s' at random and swapping the corresponding encrypted symbols $\sigma(s)$ and $\sigma(s')$.

How does the proposal probability $S(\sigma \to \sigma')$ depend on the permutations σ and σ'? What is the MH acceptance probability for a given proposal? [10 marks]

We implement a Metropolis-Hastings (MH) chain for decoding, where each proposal swaps two randomly chosen symbols s and s', modifying the encrypted mapping $\sigma(s)$ and $\sigma(s')$.

$$S(\sigma \to \sigma') = \frac{1}{\binom{53}{2}} = \frac{2}{53 \times 52} \tag{54}$$

For the MH acceptance probability $A(\sigma \to \sigma')$, given the symmetric proposal, we have

$$A(\sigma \to \sigma') = \min\left(1, \frac{\pi(\sigma')}{\pi(\sigma)}\right) \tag{55}$$

Using Bayes' Theorem, we express the posterior $\pi(\sigma) = P(\sigma \mid \mathcal{D})$ as

$$P(\sigma \mid \mathcal{D}) = \frac{P(\mathcal{D} \mid \sigma) P(\sigma)}{\sum_{\sigma'} P(\mathcal{D} \mid \sigma') P(\sigma')} \tag{56}$$

Assuming a uniform prior $P(\sigma)$, we treat $P(\sigma)$ as a constant, which allows us to simplify the acceptance probability further:

$$A(\sigma \to \sigma') = \min\left(1, \frac{P(\mathcal{D} \mid \sigma')}{P(\mathcal{D} \mid \sigma)}\right) \tag{57}$$

The target distribution $\pi(\sigma)$ for the encrypted sequence $e_1, e_2, \ldots, e_n$ under a permutation σ is

$$\pi(\sigma) = p(e_1, e_2, \ldots, e_n \mid \sigma) = P(\sigma^{-1}(e_1)) \prod_{i=2}^{n} \psi(\sigma^{-1}(e_i), \sigma^{-1}(e_{i-1})) \tag{58}$$

Thus, the acceptance probability becomes

$$\boxed{A(\sigma \to \sigma') = \min\left(1, \frac{P((\sigma')^{-1}(e_1)) \prod_{i=2}^{n} \psi((\sigma')^{-1}(e_i), (\sigma')^{-1}(e_{i-1}))}{P(\sigma^{-1}(e_1)) \prod_{i=2}^{n} \psi(\sigma^{-1}(e_i), \sigma^{-1}(e_{i-1}))}\right)} \tag{59}$$

This approach favours permutations that maximise the likelihood of observed transitions.

(d) 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]

```python

import numpy as np
import random
from collections import Counter

class Decoder:
    def __init__(self, symbol_file, message_file, training_file):
        # Initialise data and preprocess text
        self.symbol_list = self.load_symbols(symbol_file)
        self.symbol_index, self.index_symbol = self.
            create_symbol_mappings()
        self.num_symbols = len(self.symbol_list)
        self.encrypted_msg = self.load_file_as_list(message_file)
        self.training_text = self.load_file_as_list(training_file)
        self.processed_training_text = self.preprocess(self.
            training_text)

        # Initialise symbol counts and pair counts
        self.symbol_counts = np.zeros(self.num_symbols, dtype=int)
        self.pair_counts = np.zeros((self.num_symbols, self.num_symbols
            ), dtype=int)
        self.calculate_symbol_and_pair_counts()

        # Calculate stationary and transition probabilities
        self.stationary_dist, self.transitions = self.
            calculate_probabilities()
        self.log_stationary = np.log(self.stationary_dist + 1e-10)
        self.log_transitions = np.log(self.transitions + 1e-10)

        # Prepare encrypted message indices and initialise permutation
        self.encrypted_indices = [self.symbol_index[ch] for ch in self.
            encrypted_msg if ch in self.symbol_index]
        self.perm, self.perm_inverse = self.intelligent_initialisation
            ()
        self.current_log_likelihood = self.calculate_log_likelihood([
            self.perm_inverse[idx] for idx in self.encrypted_indices])

    @staticmethod
    def load_symbols(file_path):
        # Load symbols from file
        with open(file_path) as f:
            return [line.strip() if line.strip() else '␣' for line in f
                ]

    @staticmethod
    def load_file_as_list(file_path):
        # Load file as character list
        with open(file_path) as f:
            return [char for line in f for char in line]

    def create_symbol_mappings(self):
        # Create mappings between symbols and indices
        symbol_index = {s: i for i, s in enumerate(self.symbol_list)}
        index_symbol = {i: s for i, s in enumerate(self.symbol_list)}
        return symbol_index, index_symbol

    def preprocess(self, text):
        allowed = set(self.symbol_list)
        return ''.join([ch.lower() if ch in allowed else '' for ch in
            text]).replace('  ', "'").replace('  ', '"').replace('
            ', '"').replace('  ', '—').replace('\n', '␣')

    def calculate_symbol_and_pair_counts(self):
        # Count individual symbols and symbol pairs
        prev_idx = None
        for ch in self.processed_training_text:
            if ch in self.symbol_index:
                curr_idx = self.symbol_index[ch]
                self.symbol_counts[curr_idx] += 1
                if prev_idx is not None:
                    self.pair_counts[prev_idx, curr_idx] += 1
                prev_idx = curr_idx

    def calculate_probabilities(self):
        # Compute stationary distribution and transition matrix
        total_count = self.symbol_counts.sum()
        stationary_dist = self.symbol_counts / total_count
        non_zero_symbol_counts = np.where(self.symbol_counts == 0, 1,
            self.symbol_counts)
        transitions = self.pair_counts / non_zero_symbol_counts[:, None
            ]
        return stationary_dist, np.nan_to_num(transitions)

    def intelligent_initialisation(self):
        # Initialise permutation using stationary distribution
        top_symbols = np.argsort(self.stationary_dist)[-4:][::-1]
        encrypted_freq = Counter(self.encrypted_msg).most_common(4)
        initial_map = {self.symbol_index[enc_sym]: top_symbols[i]
                       for i, (enc_sym, _) in enumerate(encrypted_freq)
                           }

        # Randomly assign remaining symbols
        available_symbols = [i for i in range(self.num_symbols) if i
            not in initial_map]
        remaining_indices = [i for i in range(self.num_symbols) if i
            not in initial_map.values()]
        random.shuffle(remaining_indices)
        for idx, sym in zip(available_symbols, remaining_indices):
            initial_map[idx] = sym
```

```python

        # Convert to permutation arrays
        perm = np.array([initial_map[i] for i in range(self.num_symbols)])
        perm_inverse = np.zeros(self.num_symbols, dtype=int)
        perm_inverse[perm] = np.arange(self.num_symbols)
        return perm, perm_inverse

    def calculate_log_likelihood(self, decoded_indices):
        # Calculate log likelihood of decoded message
        log_likelihood = self.log_stationary[decoded_indices[0]]
        for i in range(1, len(decoded_indices)):
            prev_idx, curr_idx = decoded_indices[i - 1], decoded_indices[i]
            log_likelihood += self.log_transitions[prev_idx, curr_idx]
        return log_likelihood

    def metropolis_hastings(self, iterations=10000):
        # Perform Metropolis-Hastings sampling to optimise permutation
        accepted_moves = 0
        decoded_message = [self.perm_inverse[idx] for idx in self.encrypted_indices]

        for iter_num in range(1, iterations + 1):
            # Propose new permutation by swapping two symbols
            a, b = random.sample(range(self.num_symbols), 2)
            new_perm = self.perm.copy()
            new_perm[a], new_perm[b] = new_perm[b], new_perm[a]

            # Calculate new log likelihood
            new_perm_inverse = np.zeros(self.num_symbols, dtype=int)
            new_perm_inverse[new_perm] = np.arange(self.num_symbols)
            new_decoded_message = [new_perm_inverse[idx] for idx in self.encrypted_indices]
            new_log_likelihood = self.calculate_log_likelihood(new_decoded_message)
            log_likelihood_diff = new_log_likelihood - self.current_log_likelihood
            acceptance_prob = min(1, np.exp(log_likelihood_diff))

            # Accept or reject proposal
            if random.random() < acceptance_prob:
                self.perm, self.perm_inverse = new_perm, new_perm_inverse
                decoded_message = new_decoded_message
                self.current_log_likelihood = new_log_likelihood
                accepted_moves += 1

            # Report progress every 100 iterations
            if iter_num % 100 == 0:
                partial_decoded_text = ''.join(self.index_symbol[idx] for idx in decoded_message[:60])
                acceptance_rate = accepted_moves / iter_num
                print(f"Iteration {iter_num}: {partial_decoded_text}")
                print(f"Acceptance Rate: {acceptance_rate:.4f}")

        return {self.index_symbol[self.perm[i]]: self.symbol_list[i] for i in range(self.num_symbols)}

decoder = Decoder("symbols.txt", "message.txt", "war_and_peace.txt")
final_map = decoder.metropolis_hastings()
```

To initialise the chain intelligently, we analysed symbol frequencies by estimating the stationary distribution $\psi(\alpha, \beta)$ from the training text. The most common were found to be: { `space: 0.169, e: 0.102, t: 0.072, a: 0.064, o: 0.062, n: 0.059, ...`}. We found that only using the four most common symbols and randomly initialising the rest was most effective. This is probably because excessively using the stationary distribution can trap the MCMC in a local minimum.

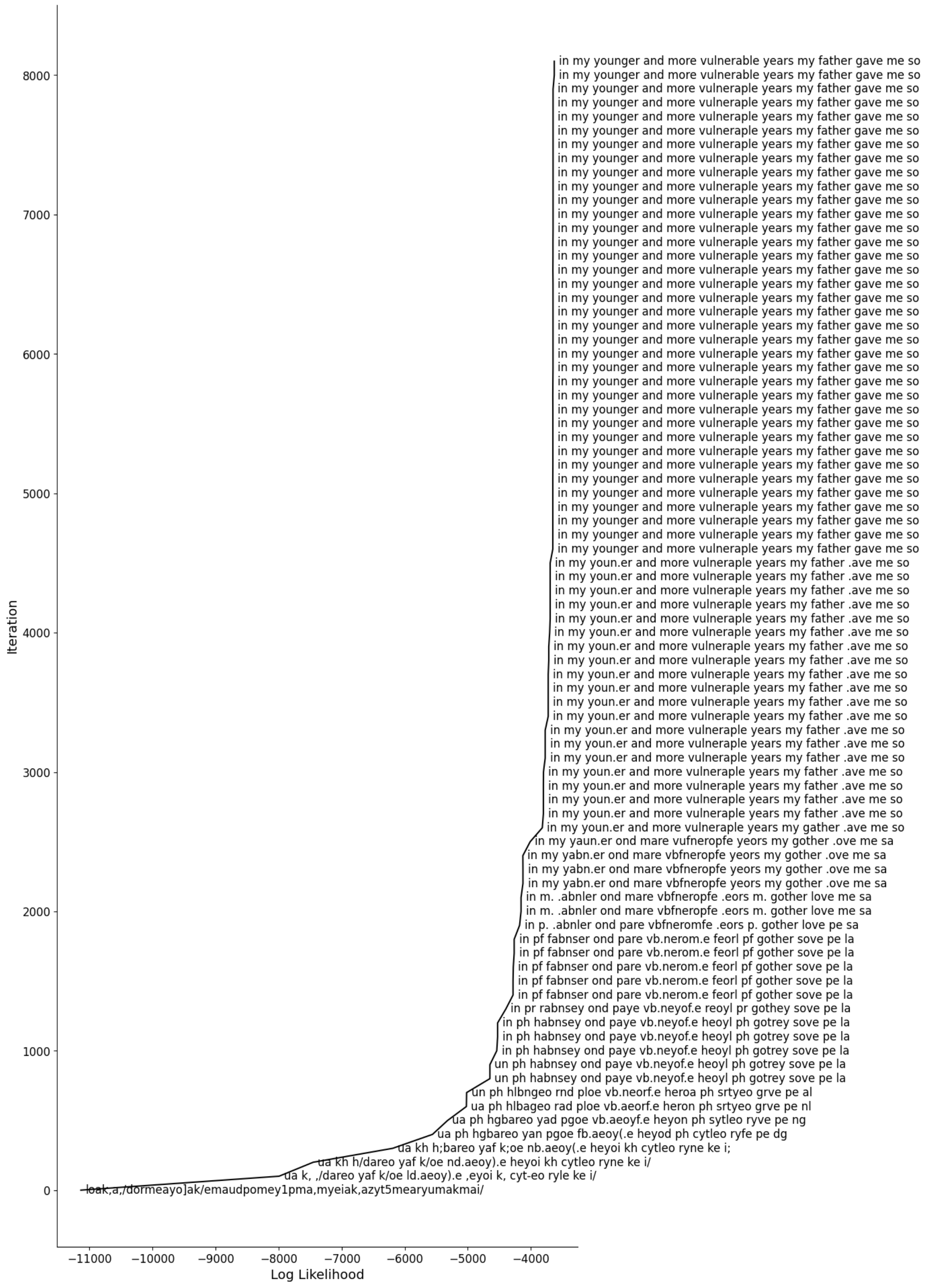

Figure 9: The log likelihood of the decrypted message across iterations converges.

(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]

Since some $\psi(\alpha, \beta)$ values are zero, the chain is not ergodic, as ergodicity requires irreducibility – each state must be reachable from every other state. Zero transition probabilities $\psi(\alpha, \beta) = 0$ prevent certain transitions.

To restore ergodicity, we apply smoothing by adding a small constant ϵ to each count:

$$\hat{\psi}(\alpha, \beta) = \frac{N(\alpha, \beta) + \epsilon}{N_\beta + \epsilon \cdot |\mathcal{S}|} \tag{60}$$

where $|\mathcal{S}|$ is the symbol count, and $\epsilon > 0$ is small (e.g., 1). Smoothing ensures each $\hat{\psi}(\alpha, \beta)$ is positive, making the chain irreducible and therefore ergodic.

(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]

Symbol probabilities alone are inadequate for decryption. While they capture frequency distributions, they do not account for syntactic structures, leading to outputs that lack coherence. For instance, it will not have any knowledge of the distribution of vowel-consonant pairing, which is very important. Without transition probabilities, different symbol arrangements with identical frequencies are treated as equally likely, reducing the chances of finding the true sentence structure.

A second-order Markov chain would improve accuracy by considering pairs of preceding symbols. n-gram models do exactly this – they are n-order Markov chains – but they experience significant computational challenges. Their complexity scales with $O(\mathcal{V}^n)$, where $\mathcal{V}$ is the vocabulary size and n is the n-gram order. In the case of $n = 2$, the transition matrix becomes three-dimensional, which increases the storage and processing requirements.

The method also assumes a bijective encryption scheme. If two symbols map to the same encrypted value, the decryption process fails because the one-to-one mapping required for the permutation structure is violated. This causes ambiguity in the likelihood function, disrupting the Metropolis-Hastings sampler's ability to converge.

In languages like Chinese, which have large symbol sets, the model faces scalability issues. A transition matrix for thousands of symbols would be sparse, leading to low or zero transition probabilities and making it difficult for the MCMC sampler to explore the state space effectively.

8. Eigenvalues as solutions of an optimization problem

Let A be a symmetric $n \times n$-matrix, and define

$$q_A(x) := x^\top A x \quad \text{and} \quad R_A(x) := \frac{x^\top A x}{x^\top x} = \frac{q_A(x)}{\|x\|^2} \quad \text{for } x \in \mathbb{R}^n.$$

We have already encountered the quadratic form q_A in class. The purpose of this problem is to verify the following fact:

If A is a symmetric $n \times n$-matrix, the optimization problem

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

has a solution, $R_A(x^*)$ is the largest eigenvalue of A, and x^* is a corresponding eigenvector.

This result is very useful in machine learning, where we are often interested in the largest eigenvalue specifically—it allows us to compute the largest eigenvalue without computing the entire spectrum, and it replaces an algebraic characterization (the eigenvalue equation) by an optimization problem. We will assume as known that the function q_A is continuous.

(a) Use the extreme value theorem of calculus (recall: a continuous function on a compact domain attains its maximum and minimum) to show that $\sup_{x \in \mathbb{R}^n} R_A(x)$ is attained. **Hint:** Since $\mathbb{R}^n$ is not compact, transform the supremum over $\mathbb{R}^n$ into an equivalent supremum over the unit sphere $S = \{x \in \mathbb{R}^n \mid \|x\| = 1\}$. The set S is compact (which you can assume as known). [6 marks]

We seek to show that the supremum of $R_A(x)$ is attained, where:

$$R_A(x) = \frac{x^\top A x}{\|x\|^2}. \tag{61}$$

Since $R_A(x)$ is homogeneous of degree zero, we can restrict the domain to the unit sphere $S = \{x \in \mathbb{R}^n : \|x\| = 1\}$, as the supremum over $\mathbb{R}^n$ is equivalent to the supremum over S.

On the unit sphere, $x^\top x = 1$, so:

$$R_A(x) = x^\top A x. \tag{62}$$

The unit sphere S is compact, and since $R_A(x)$ is continuous, the extreme value theorem guarantees that $R_A(x)$ attains its maximum on S. Therefore:

$$\sup_{x \in \mathbb{R}^n} R_A(x) = \sup_{x \in S} R_A(x), \tag{63}$$

and the supremum is attained at some $x^* \in S$, completing the proof.

(b) Let $\lambda_1 \geq \cdots \geq \lambda_n$ be the eigenvalues of A enumerated by decreasing size, and $\xi_1, \ldots, \xi_n$ corresponding eigenvectors that form an ONB. Recall from class that we can represent any vector $x \in \mathbb{R}^n$ as

$$x = \sum_{i=1}^n (\xi_i^\top x)\xi_i.$$

Show that $R_A(x) \leq \lambda_1$.

Since clearly $R_A(\xi_1) = \lambda_1$, we have in fact shown the existence of the maximum twice, using two different arguments! In summary, we now know the maximum exists, and that ξ_1 attains it. What we shall have to show is that any vector in S that is not an eigenvector for λ_1 does not maximize R_A. [9 marks]

Let $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_n$ be the eigenvalues of A, with corresponding eigenvectors $\xi_1, \ldots, \xi_n$. Any vector $x \in \mathbb{R}^n$ can be expressed as:

$$x = \sum_{i=1}^{n} (\xi_i^\top x)\xi_i. \tag{64}$$

We aim to show that $R_A(x) \le \lambda_1$. Using this expansion of x, we have:

$$R_A(x) = \frac{x^\top A x}{x^\top x} = \frac{\sum_{i=1}^{n} \lambda_i (\xi_i^\top x)^2}{\sum_{i=1}^{n} (\xi_i^\top x)^2}. \tag{65}$$

Since $\lambda_1 \ge \lambda_2 \ge \cdots \ge \lambda_n$, it follows that:

$$R_A(x) \le \lambda_1. \tag{66}$$

In particular, when $x = \xi_1$, we have $R_A(\xi_1) = \lambda_1$. Thus, the maximum value of $R_A(x)$ is attained at $x = \xi_1$.

If x is not an eigenvector for λ_1, then x contains components along $\xi_2, \dots, \xi_n$. Since $\lambda_2 < \lambda_1$, these components lead to:

$$R_A(x) < \lambda_1. \tag{67}$$

Thus, the maximum is attained only at ξ_1, the eigenvector corresponding to λ_1.

To show that any vector in S that is not an eigenvector corresponding to λ_1 does not maximize $R_A(x)$, let $x \in S$ be a unit vector not aligned with ξ_1. Then, the expansion of x as:

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

will include contributions from eigenvectors $\xi_2, \dots, \xi_n$, corresponding to eigenvalues $\lambda_2, \dots, \lambda_n$. Since $\lambda_2 < \lambda_1$, these non-zero components will result in a strictly smaller value for $R_A(x)$ compared to λ_1, hence $R_A(x) < \lambda_1$. This confirms that the maximum is attained only at ξ_1, which corresponds to the largest eigenvalue λ_1.

(c) Recall that there may be several linearly independent eigenvectors that all have eigenvalue λ_1. Let these be $\{\xi_1, \dots, \xi_k\}$, for some $k \le n$. Show that, if $x \in \mathbb{R}^n$ is not contained in span$\{\xi_1, \dots, \xi_k\}$, then $R_A(x) < \lambda_1$. [5 marks]

Suppose x is not in the span of $\{\xi_1, \dots, \xi_k\}$. Then, we can decompose x as:

$$x = \sum_{i=1}^{k} (\xi_i^\top x)\xi_i + \sum_{i=k+1}^{n} (\xi_i^\top x)\xi_i, \tag{68}$$

where $\sum_{i=1}^{k} (\xi_i^\top x)\xi_i$ lies in the span of $\xi_1, \dots, \xi_k$ and the second sum corresponds to the components of x orthogonal to this span.

Now, using this decomposition in the definition of $R_A(x)$, we get:

$$R_A(x) = \frac{\sum_{i=1}^{n} \lambda_i (\xi_i^\top x)^2}{\sum_{i=1}^{n} (\xi_i^\top x)^2}. \tag{69}$$

Since $\lambda_1 > \lambda_2 \ge \cdots \ge \lambda_k$ and $\lambda_{k+1}, \dots, \lambda_n < \lambda_1$, the terms for $\xi_{k+1}, \dots, \xi_n$ contribute less than the terms for $\xi_1, \dots, \xi_k$. Therefore, we have:

$$R_A(x) < \lambda_1.$$