# Unsupervised and Probabilistic Learning Coursework

October 2024

## 1 Models for binary vectors

**a)** In the question $x^{(n)}$ is a $D$ dimensional random vector that represents a binary pixel value of an image. Hence, for example, a square image the dimensions are $\sqrt{D} \times \sqrt{D}$. A multivariative Gaussian would follow:

$$p(\mathbf{x}) \propto \exp\left(-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^T\Sigma^{-1}(\mathbf{x}-\boldsymbol{\mu})\right)$$

where: $\boldsymbol{\mu}$ is the mean vector of length $D$, $\Sigma$ is the $D \times D$ covariance matrix, This is inappropriate for the dataset as it has a different sample space that takes binary values of 0 or 1 and, therefore, is bound in this range which does not match the $\mathbb{R}^D$ state space of the Gaussian. This means the Gaussian would assign non-zero probabilities to extreme values that are well beyond this range and are not achievable. The Gaussian model also assumes a continuous distribution that is maximised at the mean. The Gaussian model would therefore not be able to capture the binary distribution of data and would give high weighting to values around 0.5 that are not possible. The shapes are also fairly different as a Gaussian would be unimodal. Lastly, for binary variables, the covariance is bounded and depends on the joint probabilities of pixel occurrences, limiting the covariance matrix to a constrained range that the Gaussian distribution cannot adequately represent.

**b)** We now model the images as i.i.d from a multivariate Bernoulli distribution with parameter $p = (p_1, .., p_D)$:

the likelihood function for the dataset $\{\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(N)}\}$ is:

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

Taking the log of the likelihood we obtain:

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

To find the ML estimates, we would maximise the likelihood wrt. $p_d$ however, as log is increasing with positive x values, the maximum stays the same. Hence, we take the partial derivative of the log-likelihood wrt. $p_d$ and set this to 0:

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

Note that the partial wrt. a given $p_d$ goes to 0 for all other ds hence the sum over d disappears. Solving:

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

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

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

So the ML estimate for $p_d$ is:

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

We can interpret this maximum likelihood estimate of $p_d$ as the average value of the $d$-th pixel across all $N$ images.

**c)** Assuming Beta priors on $p_d$:

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

for $P(p) = \prod_d P(p_d)$. To find the MAP estimator want to, effectively, find the most probable parameter given the data. $\hat{p} =$ the maximisation of $P(p|\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(N)})$.

The posterior distribution is proportional to the product of the likelihood and the prior under Bayes' Theorem:

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

$$= \prod_{d=1}^{D} p_d^{\sum_{n=1}^{N} x_d^{(n)} + \alpha - 1}(1 - p_d)^{N - \sum_{n=1}^{N} x_d^{(n)} + \beta - 1}$$

The denominator is independent of $\mathbf{p}$ so that term can be ignored as it will go to 0 upon maximising. To find the MAP estimator, we maximise the posterior wrt. **p**. Since the priors are independent across dimensions, the MAP estimate for each $p_d$ can be found separately.

Using a similar technique as in part **b**:

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

Taking the derivative with respect to $p_d$ and setting to zero:

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

Solving and rearranging:

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

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

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

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

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

**d) and e)** please find the code below for parts d and e (including the given functions for displaying the data)

```python

import numpy as np
from matplotlib import pyplot as plt

def ml_parameters(Y):
    """
    Compute the ML estimates for the Bernoulli parameters.
    parameters:
        Y
    returns:
        params_ml: Array of ML estimates for each pixel
    """
    N, D = Y.shape
    S_d = np.sum(Y, axis=0)
    params_ml = S_d / N
    return params_ml

def map_parameters(Y, alpha=3, beta=3):
    """
    Compute the MAP estimates for the Bernoulli parameters
        with Beta priors.
    parameters:
        Y: N x D binary matrix.
        alpha: beta distribution param
        beta: beta distribution param
    returns:
        params_map: Array of MAP estimates
    """
    N, D = Y.shape
    S_d = np.sum(Y, axis=0)  #sum of each column
    params_map = (S_d + alpha - 1) / (N + alpha + beta - 2)
    return params_map

def display_image(params, title):
    """
    Display a D-dimensional parameter vector as an 8x8 image
         with a colour scale
    parameters:
        params: Parameter vector
        title: image title
    """
    plt.figure(figsize=(4,4))
    img=np.reshape(params, (8,8))
    plt.imshow(img, interpolation="none", cmap='winter')
    plt.title(title)
    plt.colorbar()
    plt.show()

def plot_difference(params_ml, params_map):
    """
    Plot the difference between ML and MAP parameters
    parameters:
        params_ml: ML estimates
        params_map: MAP estimates
    """
    difference = params_ml - params_map
    plt.figure(figsize=(4,4))
    plt.imshow(np.reshape(difference, (8,8)), interpolation=
        "none", cmap='bwr')
    plt.title('Difference')

    plt.colorbar()
    plt.show()

def main():

    Y = load_data('/Users/baidn/Downloads/binarydigits.txt')
    N, D = Y.shape

    #compute ml and map
    params_ml = ml_parameters(Y)
    alpha, beta_val = 3, 3
    params_map = map_parameters(Y, alpha, beta_val)

    # plot the images and difference
    display_image(params_ml, title='ML Parameters')
    display_image(params_map, title='MAP Parameters (alpha
        =3, beta=3)')

    plot_difference(params_ml, params_map)

if __name__ == "__main__":
    main()
```

Listing 1: Python Code for Learning ML and MAP Parameters

## Visualisation of Learned Parameters

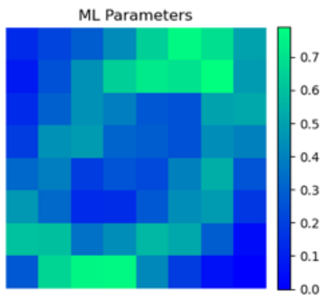


Figure 1: Learned ML Parameters

The MAP estimator integrates prior beliefs about the parameters with the observed data, balancing them based on the chosen prior parameters. Specifically, with $\alpha = \beta = 3$, the prior suggests a belief that each pixel has an inherent probability of 0.5 of being active, providing a regularisation effect that can prevent extreme estimates in scenarios with sparse or noisy data. For example, if a pixel was black in all the images the ML would be 1, hence the prior allows for some noise as you will not have a probability that is exactly 1 or 0.

As the size of the data increases the MAP and ML become increasingly similar as the prior becomes less significant. The Difference is very small as

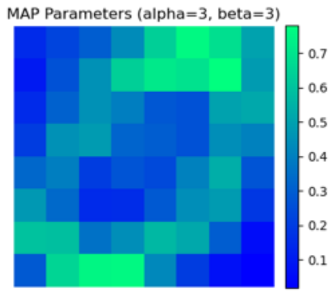


Figure 2: Learned MAP Parameters

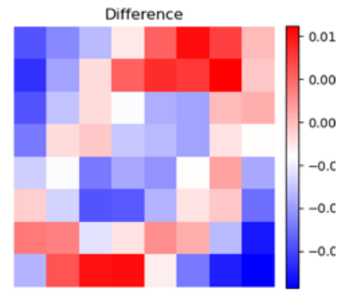


Figure 3: Difference: ML-MAP

shown in Fig 3.

The introduction of prior information can bias the estimates away from the true ML estimates, particularly if the prior is not well-aligned with the underlying data distribution so in this case a ML estimate may be better.

In our case our MAP parameters are biased to be closer to 0.5 which is shown by the difference plot in Fig 3. as the difference is positive for ML greater than 0.5 and negative for below.

## 2 Model selection

We aim to find the normalised posterior probability for each model given the data. This is given by

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

We are told that each distribution has equal priors $P(p)$ (i.e=1/3) and $P(\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(N)}\})$ is the same for each model hence for the relative/normalised probabilities we only need to find $P(\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(N)} \mid p^a)$ **a)** $p_d = 0.5$

In this model, each component $x_d$ of the data is generated from a Bernoulli distribution with $p_d = 0.5$. Thus, the probability of observing a specific dataset $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(N)}\}$ with $N$ binary vectors, each with $D$ components, is given by

$$P(\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(N)} \mid p^a) = \prod_{n=1}^{N} \prod_{d=1}^{D} P(x_d^{(n)} \mid p_d = 0.5) == \prod_{n=1}^{N} \prod_{d=1}^{D} (0.5)^{x_d^{(n)}} (0.5)^{1-x_d^{(n)}} = \prod_{n=1}^{N} \prod_{d=1}^{D} 0.5$$

Since there are $N \times D$ entries:

$$P(\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(N)} \mid p^a) = 0.5^{N \times D}.$$

Taking the log

$$\log P(\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(N)} \mid p^a) = (N \times D) \cdot \log(0.5).$$

**b)** Identical with unknown $p_d$

Here we assume that all components share the same unknown probability $p_d = p$, which is inferred from the data.

Let $S$ be the total count of 1s and $Y$ be the total count of 0s across all data):

$$S = \sum_{n=1}^{N} \sum_{d=1}^{D} x_d^{(n)}, Y = (N \times D) - S.$$

The likelihood for $S$ successes and $Y$ failures given $p$ is $p^S(1-p)^Y$. The posterior probability with a uniform prior is given by the Beta function from the following integral:

$$P(\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(N)} \mid p^b) = \int_0^1 p^S (1-p)^Y \, dp = \frac{\Gamma(S+1)\Gamma(Y+1)}{\Gamma(S+Y+2)}.$$

Taking the log-probability, we have

$$\log P(\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(N)} \mid p^b) = \log \left( \frac{\Gamma(S+1)\Gamma(Y+1)}{\Gamma(S+Y+2)} \right) = \text{betaln}(S+1, Y+1).$$

**c)** In this model, each component $x_d$ has its own independent Bernoulli distribution with separate unknown probability $p_d$, which follows a uniform prior distribution. We assume a Beta prior for each $p_d$.

For each component $d$, let $S_d$ represent the total number of 1s observed in the $d$-th column of the data set and $Y_d = N - S_d$ be the number of 0s. The posterior probability for each $p_d$ is therefore

$$P(\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(N)} \mid p^c) = \prod_{d=1}^{D} \int_0^1 p_d^{S_d}(1-p_d)^{Y_d}\, dp_d = \prod_{d=1}^{D} \frac{\Gamma(S_d+1)\Gamma(Y_d+1)}{\Gamma(N+2)}.$$

Taking the log-probability for the whole dataset, we get

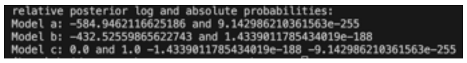
$$\log P(\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(N)} \mid p^c) = \sum_{d=1}^{D} \operatorname{betaln}(S_d+1, Y_d+1).$$

To determine the relative posterior probabilities for each model, we combine the log-probabilities calculated above. Assuming equal prior probabilities for each model, the posterior probabilities are proportional to the likelihoods, so we normalise by summing the exponentiated log-probabilities across all models:

$$P(p^i \mid \mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(N)}) = \frac{\exp(\log P(\{\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(N)}\} \mid p_i))}{\sum_{j=a,b,c} \exp(\log P(\{\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(N)}\} \mid p^j))}.$$

Due to small probabilities we use logs and after trying multiple methods we use the logsumexp python function to find these absolute probabilities.

```python

Y = np.loadtxt('/Users/baidn/Downloads/binarydigits.txt'
    )
N, D = Y.shape

# Model a
m1 = np.log(0.5) * (N * D)

# Model b
S = np.sum(Y)  # Total 1s
Y = N * D - S  # Total 0s
m2 = beta_func(S + 1, Y + 1)  # Log probability with
    Beta prior

# Model c
S_d = np.sum(Y, axis=0)  # Count of 1s for each
    component
m3 = np.sum(beta_func(S_d + 1, N - S_d + 1))  # Sum log
    probabilities for each p_d

# Normalise
log_probs = [m1, m2, m3]
total_log_prob = logsumexp(log_probs)

print("relative posterior log relative and absolute
    probabilities :")
print(f"Model a: {m1 - total_log_prob} and {np.exp(m1 -
    total_log_prob)}")
```



```
relative posterior log and absolute probabilities:
Model a: -584.9462116625186 and 9.142986210361563e-255
Model b: -432.52559865622743 and 1.4339011785434019e-188
Model c: 0.0 and 1.0 -1.4339011785434019e-188 -9.142986210361563e-255
```

Figure 4: relative log and absolute probabilities

```
    print(f"Model b: {m2 - total_log_prob} and {np.exp(m2 -
        total_log_prob)}")
    print(f"Model c: {m3 - total_log_prob} and {np.exp(m3 -
        total_log_prob)}")
```

Listing 2: Model Selection with Bernoulli Distributions

## 3 EM for Binary Data

**a)** The likelihood for a mixture model consisting of K multivariate Bernoulli distributions where $\pi_i$ represents the mixing proportions such that $\sum_i \pi_i = 1$ can be found by considering an image $x^{(n)}$.

Each image $\mathbf{x}^{(n)} \in \{0,1\}^D$ (for $n = 1, \ldots, N$) is i.i.d under the mixture model. The probability of an image $\mathbf{x}^{(n)}$ given component $k$ is defined as a product of Bernoulli distributions for each pixel, assuming pixel independence within each component.

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

where $\mathbf{P}$ matrix of Bernoulli parameters with elements where $p_{kd}$ is the probability that pixel $d$ takes the value 1 under component $k$ and $\mathbf{p_k}$ is the kth row in the $\mathbf{P}$ matrix.

The likelihood of the entire dataset is given by

$$P(\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \ldots, \mathbf{x}^{(N)} \mid \pi, P) = \prod_{n=1}^{N} \sum_{k=1}^{K} \pi_k \prod_{d=1}^{D} p_{kd}^{x_d^{(n)}} (1 - p_{kd})^{1 - x_d^{(n)}}.$$

Taking the log:

$$\log P(\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \ldots, \mathbf{x}^{(N)} \mid \pi, P) = \sum_{n=1}^{N} \log \left( \sum_{k=1}^{K} \pi_k \prod_{d=1}^{D} p_{kd}^{x_d^{(n)}} (1 - p_{kd})^{1 - x_d^{(n)}} \right).$$

**b)** The E-step involves computing the responsibility of each mixture component $k$ for a given vector $\mathbf{x}^{(n)}$. Using Bayes' theorem

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

where this mixture can be written as a latent variable model with $s^{(n)} \in 1, ...K$ and $P(s^{(n)} = k \mid \pi) = \pi_k$

$$r_{nk} = \frac{P(\mathbf{x}^{(n)} \mid s^{(n)} = k, \mathbf{P})P(s^{(n)} = k \mid \pi)}{\sum_{j=1}^{K} P(s^{(n)} = j \mid \pi)P(\mathbf{x}^{(n)} \mid s^{(n)} = j, \mathbf{P})}.$$

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

**c)** In this part, we aim to determine the parameters $\pi$ and $\mathbf{P}$ that maximise the expected log-joint probability of the observed data and the latent variables within the EM algorithm framework. The latent variables are denoted as $s^{(n)}$.

$$\arg\max_{\pi,P} \sum_{n=1}^{N} \langle \log P(\mathbf{x}^{(n)}, \mathbf{s}^{(n)} \mid \pi, P)\rangle = \arg\max_{\pi,P} \sum_{n=1}^{N} \langle \log P(\mathbf{x}^{(n)} \mid \mathbf{s}^{(n)}, \pi, P)P(s^{(n)} \mid \pi, \mathbf{P})\rangle$$

this will help obtain the iterative update for the parameters $\pi$ and $P$ in the M-step of EM.

In the EM algorithm, the M-step involves maximising the expected complete data log-likelihood with respect to the parameters $\pi$ and $P$, given the current estimate of the responsibilities $r_{nk}$ computed in the E-step.

Using the above equation, the expected complete log-likelihood is given by:

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

To maximise $\mathcal{L}$ with respect to $\pi_k$, we set up the Lagrangian with a multiplier $\lambda$ with the constraint $\sum_{k=1}^{K} \pi_k = 1$:

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

Taking the partial derivative with respect to $\pi_k$ and setting it to zero:

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

Solving for $\pi_k$:

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

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

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

Since $\sum_{k=1}^{K}\pi_k = 1$, we have:

$$-\frac{N}{\lambda} = 1 \implies \lambda = -N.$$

The updated mixing proportions are:

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

To maximise $\mathcal{L}$ with respect to $p_{kd}$, we take the derivative of $\mathcal{L}$ with respect to $p_{kd}$, incorporating the constraint $0 \leq p_{kd} \leq 1$.

The relevant part of $\mathcal{L}$ with $p_{kd}$ (for which derivativ would not go to 0) is:

$$\mathcal{L}_{p_{kd}} = \sum_{n=1}^{N} r_{nk}\left(x_d^{(n)}\log p_{kd} + (1 - x_d^{(n)})\log(1 - p_{kd})\right).$$

Taking the derivative with respect to $p_{kd}$ and setting it to zero:

$$\frac{\partial \mathcal{L}_{p_{kd}}}{\partial p_{kd}} = \sum_{n=1}^{N} r_{nk}\left(\frac{x_d^{(n)}}{p_{kd}} - \frac{1 - x_d^{(n)}}{1 - p_{kd}}\right) = 0.$$

Simplify the derivative:

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

Multiply both sides by $p_{kd}(1 - p_{kd})$:

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

Simplifying:

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

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

Therefore:

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

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

**d)** the code implements the EM algorithm using the above equations using logs due to small likelihoods. When running the algorithm multiple times the convergence for different Ks seemed to vary quite a lot but in general higher Ks took longer to converge to the threshold. **d)** Please find the code for the EM algorithm below. I used log-likelihoods and initialised $\pi$ uniformly and $\mathbf{P}$ was intialised between 0.1 and 0.9 to avoid extreme probabilities to start with.

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

# load data
X = np.loadtxt('/Users/baidn/Downloads/binarydigits.txt')
N, D = X.shape  # N = number of data points, D =
    dimensionality
K_val = [2, 3, 4, 7, 10]  # different Ks

def EM(K, X, max_iter):
    """
    Performs the EM algorithm

    parameters:
    K: Number of mixture components
    X: Data matrix of shape (N, D)
    max_iter: Maximum number of iterations

    returns:
    pi: Learned mixing proportions of shape (K,)
    P: Learned Bernoulli parameters of shape (K, D)
    log_likelihoods: List of log-likelihood values at each
        iteration
    """
    thrs = 1e-6  # threshold for convergence

    # unifrom mixing
    pi = np.full(K, 1.0 / K)

    # Bernoulli parameters random between 0.1 and 0.9 -
        avoid log 1
    P = np.random.rand(K, D) * 0.8 + 0.1

    def E_step(X, P, pi):
        """
        E-step - responsibility

        returns:
        R: Responsibility
        """
        #log probabilities to prevent underflow
        log_P = np.log(P)
        log_1_minus_P = np.log(1 - P)

        # log-likelihood for each component and data point
        log_likelihood = np.zeros((N, K))
        for k in range(K):
            # log-likelihood for component k
            log_prob = X @ log_P[k] + (1 - X) @
                log_1_minus_P[k]  #
            log_likelihood[:, k] = log_prob + np.log(pi[k])

        #responsibilities in log space to prevent underflow
        log_sum = logsumexp(log_likelihood, axis=1, keepdims
            =True)  #
        log_R = log_likelihood - log_sum
        R = np.exp(log_R)  #convert back

        return R

    def M_step(X, R):
        """
        M - Update params pi and P.

        returns:
        pi_new: Updated mixing proportions
        P_new: Updated Bernoulli params
        """
        epsilon = 1e-7  #prevetn div by 0

        # Update mixing proportions as in c)
        N_k = np.sum(R, axis=0)
        pi_new = N_k / N

        # Update Bernoulli parameters as in c)
        P_new = np.zeros((K, D))
        for k in range(K):
            # weighted sum of data points for each feature
            numerator = R[:, k] @ X
            denominator = N_k[k] + epsilon  # no div by 0
            P_new[k] = numerator / denominator

        # Ensure P_new values are within (0, 1)
        P_new = np.clip(P_new, epsilon, 1 - epsilon)

        return pi_new, P_new

    def compute_log_likelihood(X, pi, P):
        """
        computes the log-likelihood

        params:
        uses current parameters for pi and P

        returns:
        total_log_likelihood: Scalar value of the total log-
            likelihood
        """
        # log probabilities
        log_P = np.log(P)
        log_1_minus_P = np.log(1 - P)

        #log-likelihood for components and data points
        log_likelihood = np.zeros((N, K))
        for k in range(K):
            log_prob = X @ log_P[k] + (1 - X) @
                log_1_minus_P[k]
            log_likelihood[:, k] = log_prob + np.log(pi[k])

        #total log-likelihood using logsumexp
        log_sum = logsumexp(log_likelihood, axis=1)
        total_log_likelihood = np.sum(log_sum)

        return total_log_likelihood

    log_likelihoods = []

    for iteration in range(max_iter):
        #E
        R = E_step(X, P, pi)

        #M
        pi, P = M_step(X, R)

        #log-likelihood
        ll = compute_log_likelihood(X, pi, P)
        log_likelihoods.append(ll)

        # convergence check
        if iteration > 0:
            ll_change = np.abs(log_likelihoods[-1] -
                log_likelihoods[-2])
            if ll_change < thrs:
                print(f"Converged at iteration {iteration}."
                    )
                break

        print(f"Iteration {iteration + 1}: Log-Likelihood =
            {ll:.4f}")

    return pi, P, log_likelihoods

# Dictionaries to store results for different K values
log_likelihood_histories = {}
learned_parameters = {}

#loop over each K value
for K in K_val:
    print(f"Run {K}")


    pi_learned, P_learned, ll_history = EM(K, X, max_iter
        =100)

    # store history
    log_likelihood_histories[K] = ll_history
    learned_parameters[K] = {
        'pi': pi_learned,
        'P': P_learned
    }

    #display mixing
    print(f"pi for K={K}")
    print(pi_learned)

# plot log-likelihoods for each K
plt.figure(figsize=(10, 6))
for K in K_val:
    ll_history = log_likelihood_histories[K]
    iterations = range(1, len(ll_history) + 1)
    plt.plot(iterations, ll_history, label=f'K = {K}')

plt.title('Log-likelihood vs iteration for Different K')
plt.xlabel('Iteration')
plt.ylabel('Log-likelihood')
plt.legend()
plt.show()

# plot the learned Bernoulli parameters P for each K
for K in K_val:
    P_learned = learned_parameters[K]['P']
    num_components = P_learned.shape[0]


    #subplot
    cols = min(num_components, 5)
    rows = (num_components + cols - 1) // cols
    #looping through each if tge mixings to plot the
        paramter distribution
```

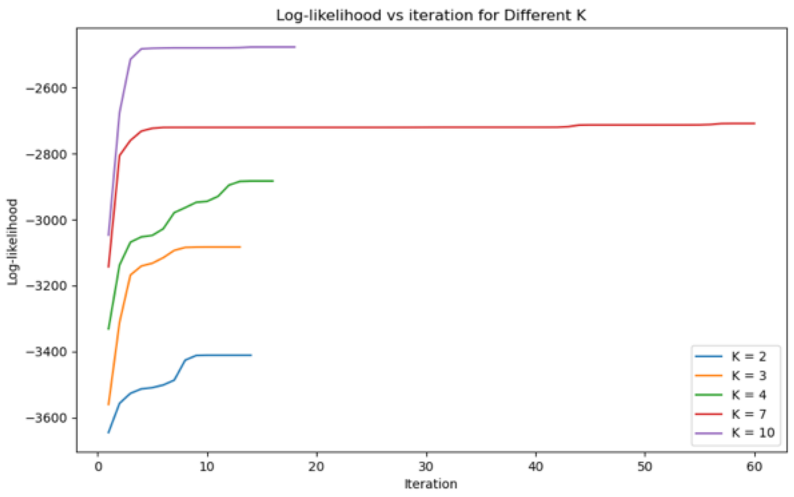


Figure 5: Log-likelihood vs Iteration for varying K

```
    plt.figure(figsize=(cols * 2, rows * 2))
    for k in range(num_components):
        plt.subplot(rows, cols, k + 1)
        plt.imshow(P_learned[k].reshape(8, 8), interpolation
            ='none', cmap='bwr')
        plt.title(f"K={K}, p = {round(learned_parameters[K
            ]['pi'][k],2)}")

    plt.tight_layout()
    plt.show()
```

**e)** I ran the EM a few times with random initialisation. For smaller values of K it seems there is somewhat similar solutions with the parameters looking like 0,5 for 2 of the trials and 0,7 for the other for K=2. Even when running for more than 3 trials the 0 always seems to be one of the parameter images for K=2. These common features become less true as the value of K increases with a considerable amount for variance for K=10 among the trials. We can also observe that for certain digits their is different representations as the zero seems to have a very slanted structure or a centred one. Also, as K increases the EM algorithm is more susceptible to variability across runs due to multiple local optima.

However, a very small K would not allow us to train multivariative distributions for each number as we have 0,5,7 and various variants of these. Hence, maybe with more trials we could find a value of K that is greater than this but less than 10. Hence, we could improve the model by choosing K according to

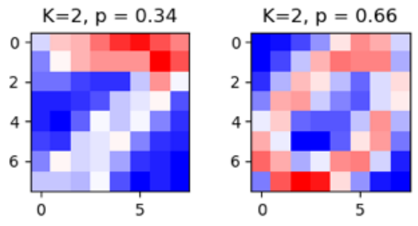


Figure 6: K=2, trial 1

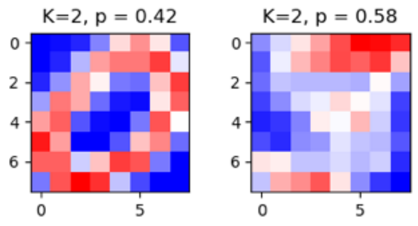


Figure 7: K=2, trial 2

the number of different variants of numbers in the dataset, probably a K just higher than 3.
We could consider a larger/more diverse dataset to improve our model and make it more flexible to new numbers.
Also, for high values of K it is interesting to see that there are some mixtures that have very low probabilities and their Parameter distribution is relatively binary implying they may only represent a single image. Hence, the model could also be improved by adding intelligent priors to allow for a more representative distribution.

Parameter plots for different values of K:

**f)** The log likelihoods are converted to bits by $L_b = \frac{\log L}{\ln(2)}$ and the number of bits from the naive encoding is $N \times D$. We can find the average code length per bit by dividing the compressed size and $L_b$ by the encoding. I am unsure

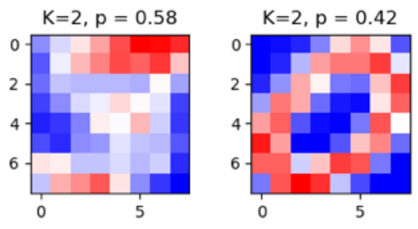


Figure 8: K=2, trial 3

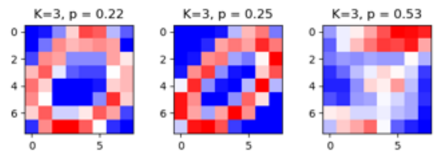


Figure 9: K=3, trial 1

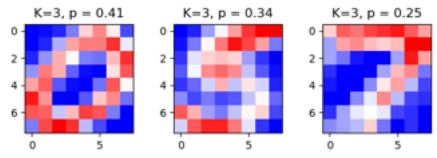


Figure 10: K=3, trial 2

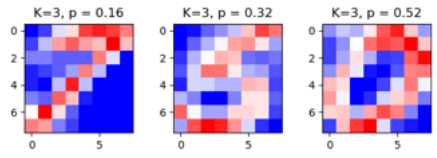


Figure 11: K=3, trial 3

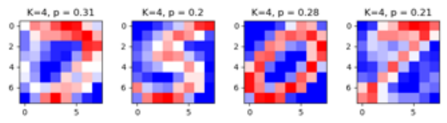


Figure 12: K=4, trial 1

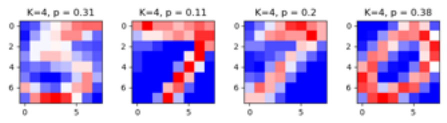


Figure 13: K=4, trial 2

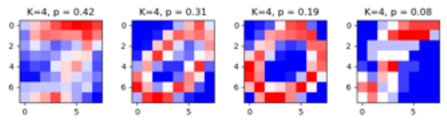


Figure 14: K=4, trial 3

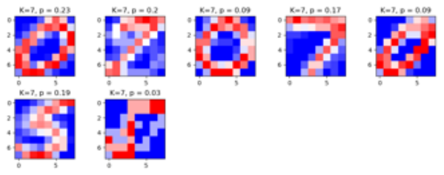


Figure 15: K=7, trial 1

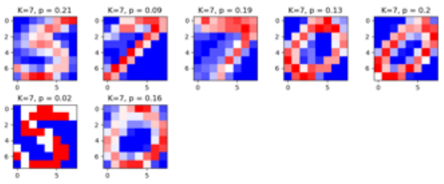


Figure 16: K=7, trial 2

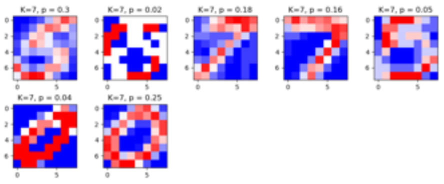


Figure 17: K=7, trial 3

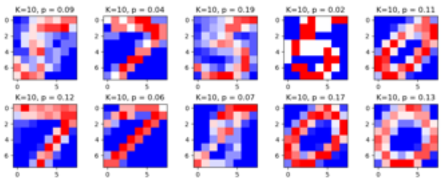


Figure 18: K=10, trial 1

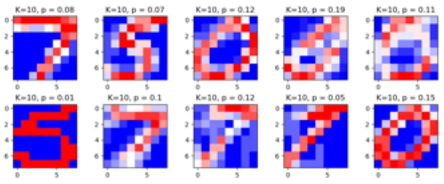


Figure 19: K=10, trial 2

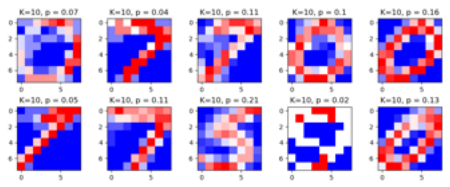


Figure 20: K=10, trial 3

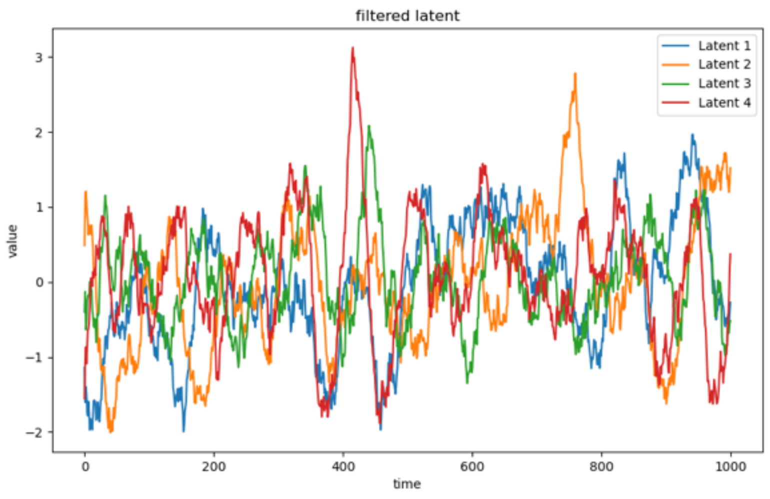


Figure 21: Filtered plot

as to how to find the exact gzip encoding but the average code length per bit increases with K as seem from the log-likelihood graph.

## 4 LGSSMs, EM and SSID

**a)** To run the functions on the provided training data, I first had to transpose the loaded matrices. The code defines the parameters as in the questions and plots the data. k=4 from matrix A. Following the pseudocode in the question I find the following plots:

**b)** We try to to estimate the parameters $\mathbf{A}, \mathbf{Q}, \mathbf{C}$ and $\mathbf{R}$ given the data we have. Note:

$$\mathbf{y_t} \sim N(A\mathbf{y_{t-1}}, Q)$$

$$\mathbf{x_t} \sim N(C\mathbf{y_t}, R)$$

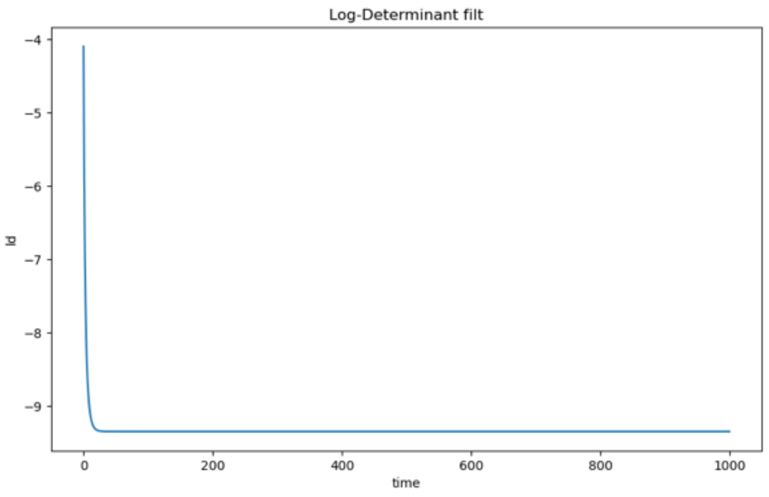


Figure 22: Filtered log determinant

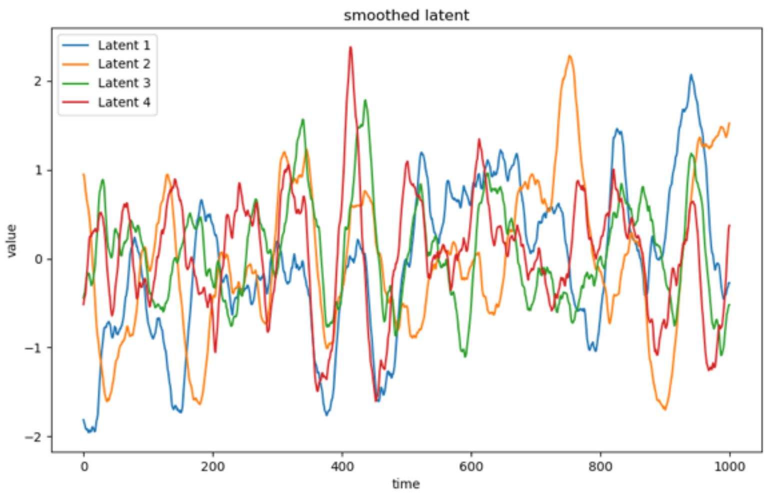


Figure 23: Smoothed plot

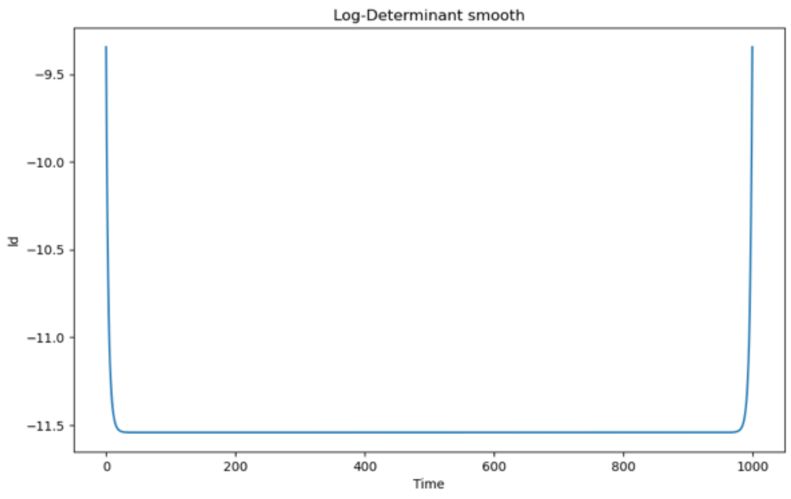


Figure 24: Smoothed log determinent

Showing the M-Step for R:

$$p(x_t|y_t) \propto \exp(-1/2(x_t - Cy_t)^T R^{-1}(x_t = Cy_t))$$

given this the log-likelihood

$$L = \left\langle \sum_{t=1}^{T} \ln p(x_t|y_t) \right\rangle = -\frac{1}{2}\sum_{t=1}^{T} \langle (x_t - Cy_t)^T R^{-1}(x_t - Cy_t)\rangle - \frac{T}{2}\ln|R|$$

$$\langle (x_t - Cy_t)(x_t - Cy_t)^T \rangle = x_t x_t^T - x_t\langle y_t^T\rangle C^T - C\langle y_t\rangle x_t^T + C\langle y_t y_t^T\rangle C^T$$

$$L = -\frac{1}{2}Tr(R^{-1}(x_t x_t^T - x_t\langle y_t^T\rangle C^T - C\langle y_t\rangle x_t^T + C\langle y_t y_t^T\rangle C^T)) - \frac{T}{2}\ln|R| + \text{const}$$

minimising $R^{-1}$ maximises R

$$\frac{\partial L}{\partial R^{-1}} = \frac{1}{2}\sum_t (x_t x_t^T - x_t\langle y_t^T\rangle C^T - C\langle y_t\rangle x_t^T + C\langle y_t y_t^T\rangle C^T) - \frac{T}{2}R = 0$$

$$R = \frac{1}{T}\sum_t (x_t x_t^T - x_t\langle y_t^T\rangle C^T - C\langle y_t\rangle x_t^T + C\langle y_t y_t^T\rangle C^T)^T$$

as $\frac{\partial Tr(AB)}{\partial B} = B^T$ Note that the update for C (from lectures) is,

$$C_{\text{new}} = \sum_t x_t\langle y_t\rangle^T (\sum_t \langle y_t y_t^T\rangle)^{-1}$$

given

$\mathrm{C}_{\text{new}} \sum_t \langle y_t y_t^T \rangle = \sum_t x_t \langle y_t^T \rangle$

this gives

$$R_{\text{new}} = \frac{1}{T}(\sum_{t=1}^{T} x_t x_t^T - \sum_{t=1}^{T} x_t \langle y_t^T \rangle C_{\text{new}}^T)$$

For Q,

$$p(y_t|y_{t-1}) \propto \exp\left(-\frac{1}{2}(y_t - Ay_{t-1})^T Q^{-1}(y_t - Ay_{t-1})\right)$$

$$L = \left\langle \sum_{t=2}^{T} \ln p(y_t|y_{t-1}) \right\rangle = -\frac{1}{2}\sum_{t=2}^{T} \left\langle (y_t - Ay_{t-1})^T Q^{-1}(y_t - Ay_{t-1}) \right\rangle - \frac{T-1}{2} \ln |Q| + \text{const}$$

$$\left\langle (y_t - Ay_{t-1})(y_t - Ay_{t-1})^T \right\rangle = \langle y_t y_t^T \rangle - A\langle y_{t-1} y_t^T \rangle - \langle y_t y_{t-1}^T \rangle A^T + A\langle y_{t-1} y_{t-1}^T \rangle A^T$$

Maximising with Respect to $Q$ (same as before)

$$\frac{\partial L}{\partial Q^{-1}} = \frac{1}{2}(\langle y_t y_t^T \rangle - A\langle y_{t-1} y_t^T \rangle - \langle y_t y_{t-1}^T \rangle A^T + A\langle y_{t-1} y_{t-1}^T \rangle A^T) - \frac{T-1}{2} Q = 0$$

$$Q = \frac{1}{T-1}(\langle y_t y_t^T \rangle - A\langle y_{t-1} y_t^T \rangle - \langle y_t y_{t-1}^T \rangle A^T + A\langle y_{t-1} y_{t-1}^T \rangle A^T)$$

Using update for A in lecture notes and using a similar method as above to cancel terms Final update for $Q$:

$$Q_{\text{new}} = \frac{1}{T-1}(\sum_{t=2}^{T} \langle y_t y_t^T \rangle - A_{\text{new}}(\sum_{t=2}^{T} \langle y_t y_{t-1}^T \rangle)^T)$$

The code includes the plotting of the figures in a), the Kalman function given to us and the EM

When run, I obtain a plot that shows the when starting with the generating initialisations there is much faster convergence compared to random - which makes sense as these are generated parameters. Also, the generating starts off with a much higher log-likelihood.

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


X = np.loadtxt('/Users/baidn/Downloads/ssm_spins.txt').T
k = 4  # dimension of latent
```

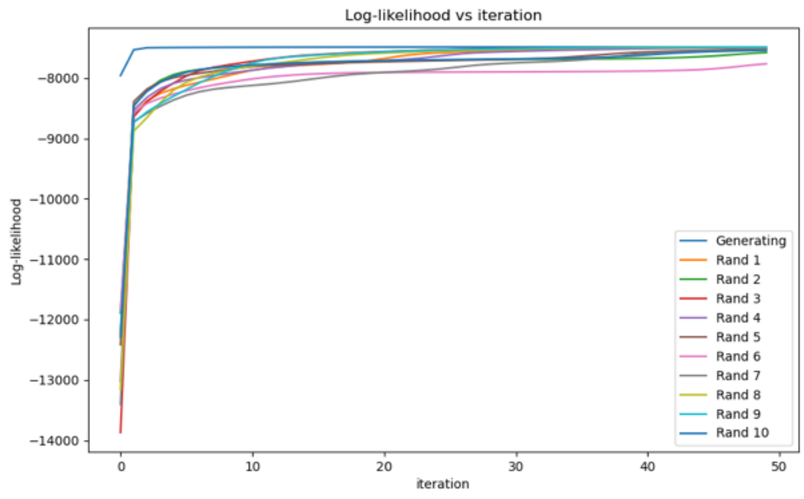


Figure 25: Log-likelihood vs iteration for generating and random initialisations over 50 iterations - and increase in likelihood and convergence is seen

```
d, T = X.shape  #dimension data and number of time steps

#defining A
t1 = 2 * np.pi / 180
t2 = 2 * np.pi / 90
A = 0.99 * np.array([
    [np.cos(t1), -np.sin(t1), 0, 0],
    [np.sin(t1), np.cos(t1), 0, 0],
    [0, 0, np.cos(t2), -np.sin(t2)],
    [0, 0, np.sin(t2), np.cos(t2)]
])
#Q
Q = np.eye(k) - A @ A.T
#C
C = np.array([
    [1, 0, 1, 0],
    [0, 1, 0, 1],
    [1, 0, 0, 1],
    [0, 0, 1, 1],
    [0.5, 0.5, 0.5, 0.5]])
R = np.eye(d)

#initial states taken as mean of normal - 0 and Identity -
    was not sure if I should instead sample from the normal
    dist ect. but ended up converging to similar results
y_init = np.zeros(k)
```

```python
Q_init = np.eye(k)

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

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

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

    I_k = np.eye(k)

    # forward pass

    V_pred = Q_init
    y_pred = y_init

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

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

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

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

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

    # backward pass

    if mode == 'filt' or mode == 'forw':
        # skip if filtering/forward pass only
        y_hat = y_filt
        V_hat = V_filt
        V_joint = None
    else:
        V_joint = np.zeros_like(V_filt)
        y_hat[:, -1] = y_filt[:, -1]
        V_hat[-1] = V_filt[-1]
        for t in range(t_max - 2, -1, -1):
            J[t] = V_filt[t].dot(A.T).dot(np.linalg.inv(A.
                dot(V_filt[t]).dot(A.T) + Q))
            y_hat[:, t] = y_filt[:, t] + J[t].dot((y_hat[:,
                t + 1] - A.dot(y_filt[:, t])))
            V_hat[t] = V_filt[t] + J[t].dot(V_hat[t + 1] - A
                .dot(V_filt[t]).dot(A.T) - Q).dot(J[t].T)

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

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

    return y_hat, V_hat, V_joint, likelihood


#running kalman for filt
y_filt, V_filt, _, L_filt = run_ssm_kalman(X, y_init, Q_init
    , A, Q, C, R, mode='filt')

#plotting filtered means
plt.figure(figsize=(10, 6))
for i in range(k):
    plt.plot(y_filt[i, :], label=f'Latent {i+1}')
plt.title('filtered latent')
plt.xlabel('time')
plt.ylabel('value')
plt.legend()
plt.show()
#running kalman for smooth
y_smooth, V_smooth, _, L_smooth = run_ssm_kalman(X, y_init,
    Q_init, A, Q, C, R, mode='smooth')

# plotting smooth mean
plt.figure(figsize=(10, 6))
for i in range(k):
    plt.plot(y_smooth[i, :], label=f'Latent {i+1}')
plt.title('smoothed latent')
plt.xlabel('time')
plt.ylabel('value')
plt.legend()
plt.show()
# plot the log-determinants
plt.figure(figsize=(10, 6))
plt.plot(np.linalg.slogdet(V_filt)[1])
plt.title('Log-Determinant filt')
```

```python
plt.xlabel('time')
plt.ylabel('ld')
plt.show()

plt.figure(figsize=(10, 6))
plt.plot(np.linalg.slogdet(V_smooth)[1])
plt.title('Log-Determinant smooth')
plt.xlabel('Time')
plt.ylabel('ld')
plt.show()

def em_lgssm(X, y_init, Q_init, A_init, Q_init_param, C_init
    , R_init, num_iter=50):
    """
    Run EM algorithm to estimate parameters of LGSSM using E
         and M updates pre calculated
    params
    X: data [d,t]
    y_init: initial latent state
    Q_init: initial variance
    A_init:initial latent dynamics matrix [k, k]
    Q_init_param: initial innovariations covariance matrix [
        k, k]
    C_init: initial output loading matrix [d, k]
    R_init:output noise matrix [d, d]
    num_iter:number of iterations
    output
    A: updated A param
    Q: updated q param
    C: updated C param
    R: updated R param
    llh: log likelihoods stored each iteration
    """
    # Initialize parameters
    A = A_init
    Q = Q_init_param
    C = C_init
    R = R_init

    # Store log-likelihoods
    llh = []

    for iteration in range(num_iter):
        # E - running kalman smoother and log likelihoods
        y_hat, V_hat, V_joint, likelihoods = run_ssm_kalman(
            X, y_init, Q_init, A, Q, C, R, mode='smooth')

        total_llh = np.sum(likelihoods)
        llh.append(total_llh)

        #M
        T = X.shape[1]
        k = y_hat.shape[0]
        d = X.shape[0]

        xx = np.zeros((d, d))
        xy = np.zeros((d, k))
        yy = np.zeros((k, k))
        y_y_prev = np.zeros((k, k))
        yy_prev = np.zeros((k, k))
        yy_t = np.zeros((k, k))

        for t in range(T):
            x_t = X[:, t][:, np.newaxis]#d
            y_t = y_hat[:, t][:, np.newaxis] #k
            V_t = V_hat[t] #kx k

            xx += x_t @ x_t.T
            xy += x_t @ y_t.T

            yy += y_t @ y_t.T + V_t

        for t in range(1, T):
            y_t = y_hat[:, t][:, np.newaxis] # k
            y_prev = y_hat[:, t-1][:, np.newaxis] # k
            V_t = V_hat[t] # k xk
            V_prev = V_hat[t-1] # k x k
            V_joint_t = V_joint[t-1] # kxk

            y_y_prev += y_t @ y_prev.T + V_joint_t
            yy_prev += y_prev @ y_prev.T + V_prev
            yy_t += y_t @ y_t.T + V_t

        # Update all parameters as in M step
        C = xy @ np.linalg.inv(yy)
        A = y_y_prev @ np.linalg.inv(yy_prev)
        R = (1/T) * (xx - xy @ C.T)
        Q = (1/(T-1)) * (yy_t - y_y_prev @ A.T)
    return A, Q, C, R, llh
#use generating parameters
A_gen, Q_gen, C_gen, R_gen, llh_gen = em_lgssm(X, y_init, Q_init, A, Q, C, R, num_iter=50)

#random initialisations - i randomised the A and C matrices and let Q and R be identities - was unsure if I shuld randomise everything?
llh_random_runs = []

for i in range(10):
    A_rand = np.random.randn(k, k)
    Q_rand = np.eye(k)
    C_rand = np.random.randn(d, k)
    R_rand = np.eye(d)

    A_rand_n, Q_rand_n, C_rand_n, R_rand_n, llh_rand =
        em_lgssm(X, y_init, Q_init, A_rand, Q_rand, C_rand,
        R_rand, num_iter=50)

    llh_random_runs.append(llh_rand)

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

# plotting log likelihoods
plt.plot(llh_gen, label='Generating')
for idx, llh_rand in enumerate(llh_random_runs):
    plt.plot(llh_rand, label=f'Rand {idx+1}')

plt.title('Log-likelihood vs iteration')
plt.xlabel('iteration')
plt.ylabel('Log-likelihood')
plt.legend()
plt.show()
```

Listing 4: LGSSM python code and plots

## 5 Decrypting Messages with MCMC

**a)** We model the English text $s_1 s_2 ...$ as a first order Markov chain such that

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

We let the stationary distribution be represented as $\phi(\gamma) = \lim_{i->\infty} p(s_i = \gamma)$ and transition probabilities as $p(s_i = \alpha | s_{i-1} = \beta) = \psi(\alpha, \beta)$.

The stationary distribution $\phi(\gamma)$ represents the long-run proportion of times symbol $\gamma$ appears in the text. In the limit where you have a very large dataset the probability tends to:

$$\phi(\gamma) = \frac{N_\gamma}{N}$$

where $N$ is the total number of symbols in the text:

$$N = \sum_{\gamma} N_\gamma$$

The ML estimate for $\psi(\alpha, \beta)$ is given by:

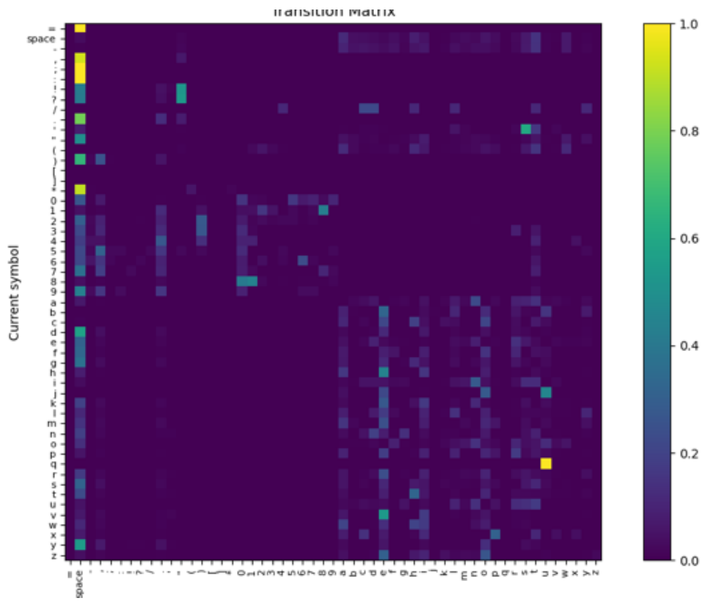


Figure 26: Transition Matrix

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

when calculating the transition matrix I ensured that for each symbol $\beta$ the transition probabilities sum to 1: **b)** The likelihood $P(s_1 s_2 \ldots s_n)$ is:

$$P(s_1 s_2 \ldots s_n) = \phi(s_1) \prod_{i=2}^{n} \psi(s_i \mid s_{i-1})$$

Taking the natural logarithm, the log-likelihood becomes:

$$\log P(s_1 s_2 \ldots s_n) = \log \phi(s_1) + \sum_{i=2}^{n} \log \psi(s_i \mid s_{i-1})$$

The latent variables are not independent due to uniqueness as, for example, if $\sigma(s) = e$ then $\sigma(s) \neq e$.

Given a first order Markov chain for the encrypted symbols

$$p(s_1 s_2..s_n) = p(e_1 e_2...|\sigma) = p(s_1) \prod_{i=2}^{n} p(s_i|s_{i-1})$$

from the definitions this equals

$$\implies \phi(s_i) \prod_{i=2}^{n} \psi(s_i|s_{i-1})$$

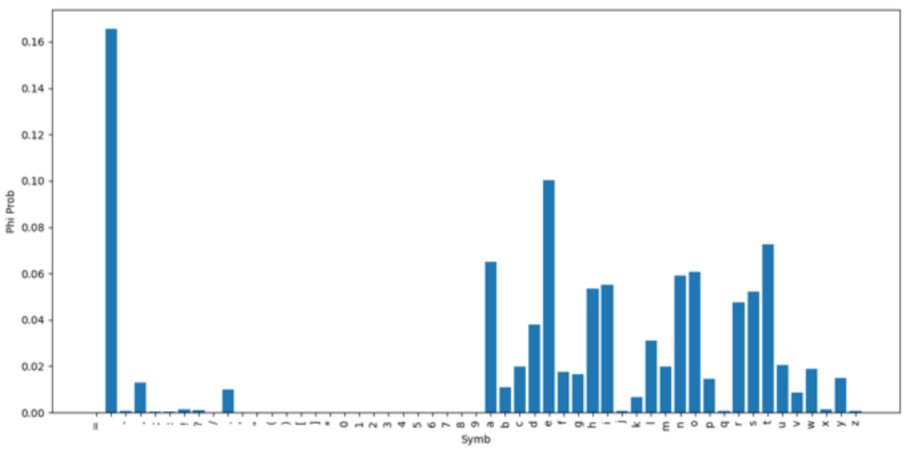


Figure 27: Phi probabilities - CDF

Hence, for $s_i = \sigma^{-1}(e_1)$ i.e. the decryption

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

**c)** We define a function $S(\sigma \to \sigma')$ that represents a swap of 2 of the symbols at random from 2 permutations $\sigma$ and $\sigma'$. There is $53C2$ different pairings of symbols among the total of 53 where C represents a combination. Given that we are restricted by this action we have that $S(\sigma \to \sigma') = \frac{1}{53C2}$ for $\sigma'(s_i) = \sigma(s_j)$ and $\sigma'(s_j) = \sigma(s_i)$ and otherwise $S(\sigma \to \sigma') = 0$.

The acceptance in lectures is defined as

$$A = \min\{1, \frac{S(\sigma' \to \sigma) P(\sigma' | e_1 ... e_n)}{S(\sigma \to \sigma') P(\sigma | e_1 ... e_n)}\}$$

However we know that S is a symmetric distribution so these cancel out. Using Bayes theorem we also have that

$$P(\sigma' | e_1 ... e_n) = \frac{P(e_1 ... e_n | \sigma') P(\sigma')}{\sum_i P(e_1 ... e_n | \sigma_i) P(\sigma_i)}$$

$P(\sigma | e_1 ... e_n)$ produces something very similar and given the denominator is a constant and we have a uniform prior, $P(\sigma) = P(\sigma')$, this means we can reduce the acceptance to:

$$A = \min\{1, \frac{P(e_1 ... e_n | \sigma')}{P(e_1 ... e_n | \sigma)}\}$$

**d)** The proposal distributions were sampled from the probabilities calculated from the war and peace text as this seemed to be a large dataset in comparison

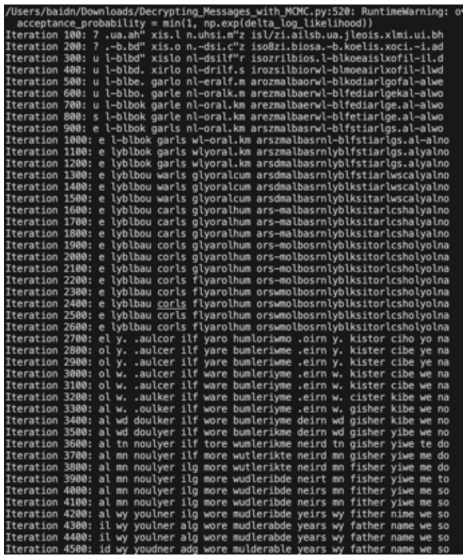
to our encrypted text and one that was reasonable to use as a proposal. Initially, I had random initialisation which sort of worked giving a somewhat accurate answer after approximately 8-10 tries.
I then tried initialising such that each of the characters in the $\phi$ table matched to the encrypted text from most common to least common in the encrypted matching up with the highest to lowest probabilities from the large text. This however produced some non-sensical results as I think the model was falling into an incorrect local stationary point or because there were symbols that did not appear in the encrypted text or had 0 probabilities for $\phi$.
In order to avoid this I added 1 to all of the frequencies meaning that no symbol had an exactly 0 probability and I mapped the space to the most common symbol and then after testing a few different combinations and varying the number of symbols mapped I picked the next 5 to the top 5 most common symbols. This allowed for some initialisation as well as randomness and seemed to work much better - getting a reasonable answer around every 4 runs.

```python
import numpy as np
import random
import pandas as pd
from matplotlib import pyplot as plt
# reading the file
with open('/Users/baidn/Downloads/symbols.txt', 'r') as f:
    symbols = [line.rstrip('\n') for line in f]
    symbols = [' ' if symbol == '' else symbol for symbol in
        symbols]
#print("Symbols:", symbols)

# symbol to index and index to symbol mappings
symbol_to_i = {s: i for i, s in enumerate(symbols)}
i_to_symbol = {i: s for i, s in enumerate(symbols)}
K = len(symbols)

# read war and peace for calculating proposals
with open('/Users/baidn/Downloads/WarAndPeace.txt', 'r',
    encoding='utf-8') as f:
    text = f.read()
#ignoring the intro text - only consdering the book
text = text[834:].lower()
#count symbols and pairs in loop and calc probability
symbol_c = np.zeros(K, dtype=np.int64)
pair_c = np.zeros((K, K), dtype=np.int64)
prev_idx = None
for s in text:
    if s in symbol_to_i:
        idx = symbol_to_i[s]
        symbol_c[idx] += 1
        if prev_idx is not None:
            pair_c[prev_idx, idx] += 1
        prev_idx = idx
    else:
        continue  # skip symbols not in our set
symbol_c[symbol_c == 0] = 1 #make probabilities slightly
    more than 0 for symb that dont occur such as not to leave
     these out - leads to better results
N = symbol_c.sum()
phi_arr = symbol_c / N

symbol_counts_no_zeros = np.where(symbol_c == 0, 1, symbol_c
    )
psi_arr = pair_c / symbol_counts_no_zeros[:, None]
psi_arr = np.nan_to_num(psi_arr)
print("Psi array shape:", psi_arr.shape)
print("Psi array:", psi_arr)
print(phi_arr)
#heatmap for the transition matrix and a graph forphi
char = ['space' if s == ' ' else s for s in symbols]

plt.figure(figsize=(12, 10))
plt.imshow(psi_arr, cmap='viridis', interpolation='nearest')
plt.colorbar()
plt.xticks(ticks=np.arange(K), labels=char, rotation=90,
    fontsize=8)
plt.yticks(ticks=np.arange(K), labels=char, fontsize=8)
plt.title('Transition Matrix')
plt.xlabel('Next symbol')
plt.ylabel('Current symbol')
plt.tight_layout()
plt.show()

plt.figure(figsize=(12, 6))
plt.bar(char, phi_arr)
plt.xlabel('Symb')
plt.ylabel('Phi Prob')

plt.xticks(rotation=90)
plt.tight_layout()
plt.show()

epsilon = 1e-10  #to prevent log(0)
#log probabilities
log_phi = np.log(phi_arr + epsilon)
log_psi = np.log(psi_arr + epsilon)

# read encrypted text
with open('/Users/baidn/Downloads/message.txt', 'r') as f:
    encrypt_txt = f.read().strip()

# encrypted symbols to indices
encrypt_i = [symbol_to_i[s] for s in encrypt_txt if s in
```

```
    symbol_to_i]
sigma = -np.ones(K, dtype=int)  # initializing sigma
# keep track of assigned characters
cipher_i = set()
plaintext_i = set()

com_char = [' ', 'e', 'a', 't', 'i', 'n']

# get indices of common characters
com_char_i = [symbol_to_i[c] for c in com_char if c in symbol_to_i]

# Sort encrypted symbols by frequency in descending order
sort_encrypt_i = sorted(range(K), key=lambda x: -symbol_c[x])

# Select the top frequent encrypted symbols and map to common characters
top_encrypt = sort_encrypt_i[:len(com_char_i)]
for enc_idx, plain_idx in zip(top_encrypt, com_char_i):
    sigma[enc_idx] = plain_idx
    cipher_i.add(enc_idx)
    plaintext_i.add(plain_idx)

# remaining cipher and plaintext indices and random mapping of these
r_cipher_i = set(range(K)) - cipher_i
r_plaintext_i = set(range(K)) - plaintext_i
r_cipher_i = list(r_cipher_i)
r_plaintext_i = list(r_plaintext_i)
random.shuffle(r_plaintext_i)

for enc_idx, avail_idx in zip(r_cipher_i, r_plaintext_i):
    sigma[enc_idx] = avail_idx
    cipher_i.add(enc_idx)
    plaintext_i.add(avail_idx)

#inverse
sigma_inv = np.zeros(K, dtype=int)
sigma_inv[sigma.astype(int)] = np.arange(K)

#log likelihodd
def compute_log_likelihood(decrypt_i):
    log_ml = log_phi[decrypt_i[0]]
    for i in range(1, len(decrypt_i)):
        prev_idx = decrypt_i[i - 1]
        curr_idx = decrypt_i[i]
        log_ml += log_psi[prev_idx, curr_idx]
    return log_ml

```

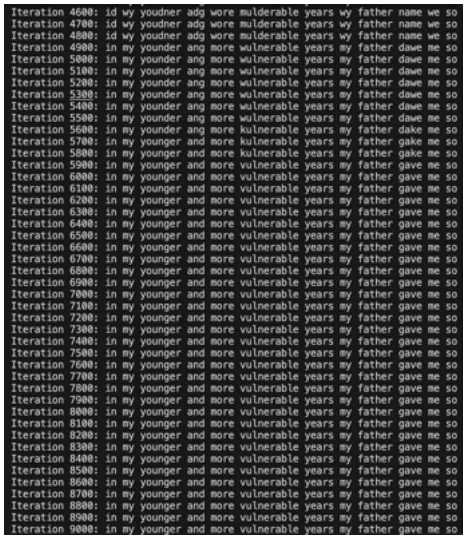
```python

decrypt_i = [sigma_inv[idx] for idx in encrypt_i]
log_p_e_g_sigma = compute_log_likelihood(decrypt_i)

# MH sampler and tracking acceptance rate
iter = 10000
acceptance = 0
for iteration in range(1, iter + 1):
    #new sigma by swapping two symbols
    i, j = random.sample(range(K), 2)
    sigma_prop = sigma.copy()
    sigma_prop[i], sigma_prop[j] = sigma_prop[j], sigma_prop[i]

    #update sigma_inverse accordingly
    sigma_i_prop = np.zeros(K, dtype=int)
    sigma_i_prop[sigma_prop.astype(int)] = np.arange(K)

    #decrypt the message with proposed sigma
    decrypted_i_prop = [sigma_i_prop[idx] for idx in encrypt_i]

    #new log-likelihood, use logs to prevent some small values being treated as zeros
    log_p_e_g_sigma_prop = compute_log_likelihood(decrypted_i_prop)

    #acceptance probability
    del_log_ml = log_p_e_g_sigma_prop - log_p_e_g_sigma
    acceptance_prob = min(1, np.exp(del_log_ml))

    # accpet/reject step
    if random.random() < acceptance_prob:
        sigma = sigma_prop
        sigma_inv = sigma_i_prop
        decrypt_i = decrypted_i_prop
        log_p_e_g_sigma = log_p_e_g_sigma_prop
        acceptance += 1


    # 100 iterations print
    if iteration % 100 == 0:
        decrypted_text_sample = ''.join([i_to_symbol[idx] for idx in decrypt_i[:60]])
        print(f"Iteration {iteration}: {decrypted_text_sample}")
print(f"overall Acceptance Rate: {acceptance / iter}")
```

Listing 5: MCMC Decryption python code and transition matrix and phi plots



```
/Users/baidn/Downloads/Decrypting_Messages_with_MCMC.py:520: RuntimeWarning: o
  acceptance_probability = min(1, np.exp(delta_log_likelihood))
Iteration 100: 7 .ua.ah" xis.l n.uhsi.m"z isl/zi.ailsb.ua.jleois.xlmi.ui.bh
Iteration 200: ? .-b.bd" xis.o n.-dsi.c"z iso8zi.biosa.-b.koelis.xoci.-i.ad
Iteration 300: u l-blbd" xislo nl-dsilf"r isozrilbios.l-blkoeaislxofil-il.d
Iteration 400: u l-blbd. xirlo nl-drilf.s irozsilbiorwl-blmoeairlxofil-ilwd
Iteration 500: u l-blbe. garlo nl-eralf.m arozmalbaorwl-blkodiarlgofal-alwe
Iteration 600: u l-blbo. garle nl-oralk.m arezmalbaerwl-blfediarlgekal-alwo
Iteration 700: u l-blbok garle nl-oral.km arezmalbaerwl-blfediarlge.al-alwo
Iteration 800: s l-blbok garle nl-oral.km arezmalbaerwl-blfetiarlge.al-alwo
Iteration 900: e l-blbok garls nl-oral.km arszmalbasrwl-blfstiarlgs.al-alwo
Iteration 1000: e l-blbok garls wl-oral.km arszmalbasrnl-blfstiarlgs.al-alno
Iteration 1100: e lyblbok garls wlyoral.km arszmalbasrnlyblfstiarlgs.alyalno
Iteration 1200: e lyblbok garls wlyoral.km arsdmalbasrnlyblfstiarlgs.alyalno
Iteration 1300: e lyblbou warls glyoralcum arsdmalbasrnlyblfstiarlwscalyalno
Iteration 1400: e lyblbou warls glyoralcum arsdmalbasrnlyblkstiarlwscalyalno
Iteration 1500: e lyblbou warls glyoralcum arsdmalbasrnlyblksitarlwscalyalno
Iteration 1600: e lyblbou carls glyoralhum ars-malbasrnlyblksitarlcshalyalno
Iteration 1700: e lyblbou carls glyoralhum ars-malbasrnlyblksitarlcshalyalno
Iteration 1800: e lyblbou carls glyoralhum ars-malbasrnlyblksitarlcshalyalno
Iteration 1900: e lyblbau corls glyarolhum ors-molbosrnlyblksitorlcsholyolna
Iteration 2000: e lyblbau corls glyarolhum ors-molbosrnlyblksitorlcsholyolna
Iteration 2100: e lyblbau corls glyarolhum ors-molbosrnlyblksitorlcsholyolna
Iteration 2200: e lyblbau corls flyarolhum ors-molbosrnlyblksitorlcsholyolna
Iteration 2300: e lyblbau corls flyarolhum orswmolbosrnlyblksitorlcsholyolna
Iteration 2400: e lyblbau corls flyarolhum orswmolbosrnlyblksitorlcsholyolna
Iteration 2500: e lyblbau corls flyarolhum orswmolbosrnlyblksitorlcsholyolna
Iteration 2600: e lyblbau corls flyarolhum orswmolbosrnlyblksitorlcsholyolna
Iteration 2700: el y. .aulcor ilf yaro humloriwmo .oirn y. kistor ciho yo na
Iteration 2800: ol y. .aulcer ilf yare bumleriwme .eirn y. kister cibe ye na
Iteration 2900: ol y. .aulcer ilf yare bumleriwme .eirn y. kister cibe ye na
Iteration 3000: ol w. .aulcer ilf ware bumleriyme .eirn w. kister cibe we na
Iteration 3100: ol w. .aulcer ilf ware bumleriyme .eirn w. kister cibe we na
Iteration 3200: ol w. .aulker ilf ware bumleriyme .eirn w. cister kibe we na
Iteration 3300: al w. .oulker ilf wore bumleriyme .eirn w. gisher kibe we no
Iteration 3400: al wd doulker ilf wore bumleriyme deirn wd gisher kibe we no
Iteration 3500: al wd doulyer ilf wore bumlerikme deirn wd gisher yibe we no
Iteration 3600: al tn noulyer ilf tore wumlerikme neird tn gisher yiwe te do
Iteration 3700: al mn noulyer ilf more wutlerikte neird mn gisher yiwe me do
Iteration 3800: al mn noulyer ilg more wutlerikte neird mn fisher yiwe me do
Iteration 3900: al mn noulyer ilg more wudleribde neirt mn fisher yiwe me to
Iteration 4000: al mn noulyer ilg more wudleribde neirs mn fither yiwe me so
Iteration 4100: al mn noulyer ilg more wudleribde neirs mn fither yiwe me so
Iteration 4200: al wy youlner ilg wore mudleribde yeirs wy fither nime we so
Iteration 4300: il wy youlner alg wore mudlerabde years wy father name we so
Iteration 4400: il wy youlner alg wore mudlerabde years wy father name we so
Iteration 4500: id wy youdner adg wore mulderable years wy father name we so
```

Figure 28: First 60 decrypted characters -part 1



```
Iteration 4600: id wy youdner adg wore mulderable years wy father name we so
Iteration 4700: id wy youdner adg wore mulderable years wy father name we so
Iteration 4800: id wy youdner adg wore mulderable years wy father name we so
Iteration 4900: in my younder ang more wulnerable years my father dawe me so
Iteration 5000: in my younder ang more wulnerable years my father dawe me so
Iteration 5100: in my younder ang more wulnerable years my father dawe me so
Iteration 5200: in my younder ang more wulnerable years my father dawe me so
Iteration 5300: in my younder ang more wulnerable years my father dawe me so
Iteration 5400: in my younder ang more wulnerable years my father dawe me so
Iteration 5500: in my younder ang more wulnerable years my father dawe me so
Iteration 5600: in my younder ang more kulnerable years my father dake me so
Iteration 5700: in my younger and more kulnerable years my father gake me so
Iteration 5800: in my younger and more kulnerable years my father gake me so
Iteration 5900: in my younger and more vulnerable years my father gave me so
Iteration 6000: in my younger and more vulnerable years my father gave me so
Iteration 6100: in my younger and more vulnerable years my father gave me so
Iteration 6200: in my younger and more vulnerable years my father gave me so
Iteration 6300: in my younger and more vulnerable years my father gave me so
Iteration 6400: in my younger and more vulnerable years my father gave me so
Iteration 6500: in my younger and more vulnerable years my father gave me so
Iteration 6600: in my younger and more vulnerable years my father gave me so
Iteration 6700: in my younger and more vulnerable years my father gave me so
Iteration 6800: in my younger and more vulnerable years my father gave me so
Iteration 6900: in my younger and more vulnerable years my father gave me so
Iteration 7000: in my younger and more vulnerable years my father gave me so
Iteration 7100: in my younger and more vulnerable years my father gave me so
Iteration 7200: in my younger and more vulnerable years my father gave me so
Iteration 7300: in my younger and more vulnerable years my father gave me so
Iteration 7400: in my younger and more vulnerable years my father gave me so
Iteration 7500: in my younger and more vulnerable years my father gave me so
Iteration 7600: in my younger and more vulnerable years my father gave me so
Iteration 7700: in my younger and more vulnerable years my father gave me so
Iteration 7800: in my younger and more vulnerable years my father gave me so
Iteration 7900: in my younger and more vulnerable years my father gave me so
Iteration 8000: in my younger and more vulnerable years my father gave me so
Iteration 8100: in my younger and more vulnerable years my father gave me so
Iteration 8200: in my younger and more vulnerable years my father gave me so
Iteration 8300: in my younger and more vulnerable years my father gave me so
Iteration 8400: in my younger and more vulnerable years my father gave me so
Iteration 8500: in my younger and more vulnerable years my father gave me so
Iteration 8600: in my younger and more vulnerable years my father gave me so
Iteration 8700: in my younger and more vulnerable years my father gave me so
Iteration 8800: in my younger and more vulnerable years my father gave me so
Iteration 8900: in my younger and more vulnerable years my father gave me so
Iteration 9000: in my younger and more vulnerable years my father gave me so
```

Figure 29: First 60 decrypted characters -part 2

**e)** The initial set up is not ergodic as this would imply you could reach any state from any other state through some path and the chain is aperiodic. Due to the zero values I would have though there is still potential to reach x from y even if $\psi(x, y) = 0$ through a series of swaps. However, I think there are 0 columns in the transition matrix which means we will not be able to reach certain states as there is a 0 chance that state x will go to y for example if $\psi(z, y) = 0$ for z being all characters. We can fix this by adding small probabilities to all of the 0 values such that the probability is never 0 i.e. start with $N_\gamma = 1$.

**f)** Symbol probabilities are not sufficient as it would converge to mapping the most probable symbols in each of the others. This would remove all contextual information which is crucial to language. Also, there would be ambiguity if frequency of symbols matched.
Using a second-order Markov chain introduces challenges, including data sparsity, as the there is an increase zero probabilities, and computational complexity due to the number of transition probabilities growing to $S^3$ (for S the number of symbols) which would also require a larger training dataset. Additionally, ergodicity concerns arise due to more zero transitions, contributing to irreducibility issues.

This approach fails if two symbols map to the same encrypted value as the model relies on a bijective mapping. This assumption of permutation fails when the mapping is not unique, leading to ambiguity in recovering the original symbols. With this approach we would have to have some probability associated with the possible mappings and it would be much harder to create a proposal.

The approach is impractical for languages with large symbol sets, such as Chinese, due to computational infeasibility and the large state space from over 10,000 symbols. The memory required for storing bigram probabilities is unmanageable, and there is insufficient data to accurately estimate frequencies and transitions. Also, teh sampler could take a very long time to converge as it needs to explore this space.

## 6

## 7 Optimisation

**a)** To find the local extrema of the function $f(x, y) = x + 2y$ subject to the constraint $g(x, y) = y^2 + xy - 1 = 0$, we use the method of Lagrange multipliers.

Let $\lambda$ be the Lagrange multiplier, and set up the Lagrangian function:

$$L(x, y, \lambda) = x + 2y - \lambda(y^2 + xy - 1)$$

The partial derivative with respect to $x$ is

$$\frac{\partial L}{\partial x} = 1 - \lambda y = 0, \quad \Rightarrow \quad \lambda y = 1$$

With respect to $y$,

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

The constraint equation is

$$g(x, y) = y^2 + xy - 1 = 0$$

**b)** From $\lambda = \frac{1}{y}$, we substitute into the second equation and solve for $x$ and $y$. The local extrema occur at $(0, 1)$ and $(0, -1)$, where the function $f(x, y)$ attains its local extrema under the given constraint.

To find $x = \ln(a)$ using Newton's method, we consider the equation $\exp(x) = a$, rewritten as a root-finding problem:

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

Newton's method updates $x_n$ using:

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

## 8 Eigenvalues as solutions of an optimisation problem

**a)** In this question $q_A = x^T Ax$ and $R_A = \frac{x^T Ax}{x^T x} = \frac{q_A}{|x|^2}$ and we aim to show that $\sup_{x\in\mathbb{R}^n} R_A(x)$ is attained. consider the unit sphere as $\mathbb{R}^n$ is not contained and every $\mathbb{R}^n$ can be converted to the unit sphere by $\frac{1}{|x|} = c$ for scalar c This is formally defined as:

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

Note that for any nonzero scalar c,

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

Noting that c is scalar its transpose is itself so $R_A(x)$ is homogeneous of degree zero and depends only on the direction of $x$.

Therefore,

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

as $q_A(x)$ is continuous and $|x| = 1$ on $S$, $R_A(x)$ is continuous on $S$. By the Extreme Value Theorem, $R_A(x)$ attains its maximum on the compact set $S$.

**b)** $A$ is symmetric with eigenvalues $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_n$ and coresponding orthonormal eigenvectors $\{\xi_1, \ldots, \xi_n\}$.

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

Given that $A\xi_i = \lambda_i$

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

Sothe Rayleigh quotient becomes

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

Since $\lambda_i \leq \lambda_1$ for all $i$ and given $\lambda_1$ is a constant we can take it out to give

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

dividing through

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

$$\implies R_A(x) \leq \lambda_1.$$

**(c)** Suppose $x \in \mathbb{R}^n$ is not in $\text{span}\{\xi_1, \ldots, \xi_k\}$, where $\lambda_1 = .. = \lambda_k$ are the largest eigenvalues with multiplicity $k$.

Then there exists $j > k$ such that $\xi_j^T x \neq 0$. Using

$$R_A(x) = \frac{\sum_{i=1}^{n} \lambda_i (\xi_i^T x)^2}{\sum_{i=1}^{n} (\xi_i^T x)^2} = \lambda_1 - \frac{\sum_{i=k+1}^{n} (\lambda_1 - \lambda_i)(\xi_i^T x)^2}{\sum_{i=1}^{n} (\xi_i^T x)^2}.$$

Since $\lambda_i < \lambda_1$ for $i > k$ and $(\xi_i^T x)^2 > 0$ for some $i > k$, it follows that

$$\sum_{i=k+1}^{n} (\lambda_1 - \lambda_i)(\xi_i^T x)^2 > 0,$$

and hence,

$$R_A(x) = \lambda_1 - \frac{\text{something +ve}}{\sum_{i=1}^{n} (\xi_i^T x)^2} < \lambda_1.$$

Therefore, $R_A(x) < \lambda_1$ whenever $x \notin \text{span}\{\xi_1, \ldots, \xi_k\}$.