# Unsupervised Learning - Coursework 1

Student Number: 24056681

## 1 Question 1: Models for binary vectors.

(a)
- Each pixel is either 0 (black) or 1 (white), making the data discrete. A Gaussian distribution models continuous data, which is unsuitable for this discrete format. Binary values are better modeled by discrete probability distributions like the Bernoulli distribution.
- A Gaussian distribution assigns non-zero probability to all real numbers, whereas pixel values can only be 0 or 1. The Gaussian model might assign probabilities to invalid pixel values such as 0.5, which are not possible in this data set.
- The covariance matrix of a Gaussian distribution represents linear relationships between continuous variables. However, relationships between binary pixels, such as co-occurrence of black or white pixels, are not necessarily linear.

(b) Given the data are modeled as i.i.d. samples from a multivariate Bernoulli distribution with parameter vector $\mathbf{p} = (p_1, \ldots, p_D)$, the pmf for a single image $\mathbf{x}$ is:

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

Given $N$ independent images $\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \ldots, \mathbf{x}^{(N)}$, the likelihood function is:

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

Taking the log, we get:

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

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

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

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

Let $S_d = \sum_{n=1}^{N} x_d^{(n)}$ represent the total number of times the pixel $d$ is "on" (i.e., 1) across all images. This simplifies to:

$$\frac{S_d}{p_d} = \frac{N-S_d}{1-p_d}$$

Solving for $p_d$:

$$S_d(1-p_d) = (N-S_d)p_d$$

$$S_d - S_d p_d = (N-S_d)p_d$$

$$S_d = Np_d$$

$$p_d = \frac{S_d}{N}$$

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

We have found that the MLE of the parameter vector is the mean of all the images.

(c) Given that the prior for each $p_d$ is a Beta distribution:

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

and assuming independence across the dimensions, the prior for the full vector $\mathbf{p}$ is:

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

Using the Bayes' theorem:

$$P(\mathbf{p}|\mathbf{x}) \propto P(\mathbf{x}|\mathbf{p})P(\mathbf{p})$$

Using the likelihood from (b), the posterior becomes:

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

Taking the log of the posterior:

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

We maximize this log-posterior by differentiating with respect to $p_d$ and setting it to zero:

$$\frac{\partial}{\partial p_d}\log P(\mathbf{p}|\mathbf{x}) = \frac{S_d}{p_d} - \frac{N-S_d}{1-p_d} + \frac{\alpha-1}{p_d} - \frac{\beta-1}{1-p_d} = 0$$

Simplifying:

$$\frac{S_d+\alpha-1}{p_d} = \frac{N-S_d+\beta-1}{1-p_d}$$

$$p_d = \frac{S_d+\alpha-1}{N+\alpha+\beta-2}$$

where $S_d = \sum_{n=1}^{N} x_d^{(n)}$ is the total number of times pixel $d$ is "on" across all images. The MAP estimate for each $p_d$ is:

$$\hat{p}_d = \frac{S_d+\alpha-1}{N+\alpha+\beta-2}$$

(d) Please find the code in Appendix 1d.

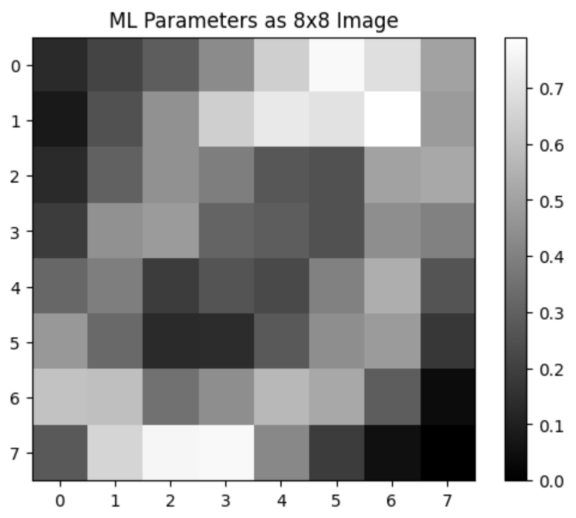


Figure 1: ML Parameters as an 8x8 Image

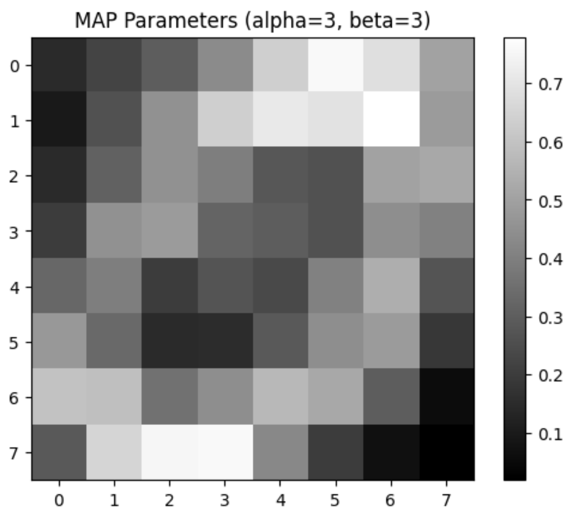


Figure 2: MAP Parameters (with $\alpha = 3$, $\beta = 3$) as an 8x8 Image

(e) Please find the code in Appendix 1e. The MAP estimate incorporates prior knowledge, giving a a slight smoothing effect compared to the ML estimate, which is purely data-driven. This smoothing effect is useful when the data is limited or noisy, as it regularizes the parameter estimates, preventing extreme values (very close to 0 or 1) and potentially improving generalization.

However, the MAP estimate can introduce bias if the prior doesn't align with the true distribution of the data, making it less optimal than ML when we have a large, clean dataset, where ML provides a more accurate reflection of the empirical distribution. Therefore, MAP is preferable when there is limited data or potential noise, while ML can be more effective with large datasets that are free from noise.

## 2 Model Selection

For each Bernoulli model $M_i$ (where $i = a, b, c$) with equal prior, we compute the probabilities $P(M_i|X)$ relative to other models by using Bayes' Theorem:

$$P(M_i|X) = \frac{P(X|M_i)P(M_i)}{P(X)} \propto P(X|M_i)$$

where $P(M_i) = \frac{1}{3}$. Thus, the posterior probabilities are proportional to the likelihoods $P(X|M_i)$. To find the relative probabilities, we can marginalize over the parameters of the model:

$$P(X|M_i) = \int dp^{(i)}\, P(X|p^{(i)}, M_i)P(p^{(i)}|M_i)$$

### Model (a): Bernoulli with $p_d = 0.5$

Model (a) assumes that each component parameter $p_d^{(a)} = 0.5$ for all $d \in \{1, \ldots, D\}$. This assumption is represented by a Dirac delta distribution $P(p_d^{(a)}) = \delta(p_d^{(a)} - 0.5)$, which indicates that each $p_d^{(a)}$ is fixed at 0.5 with no variation.

The likelihood for a dataset $X$, where each image has $D$ pixels, $N$ is the number of images, can then be computed as:

$$P(X|M_a) = \prod_{d=1}^{D} \int_0^1 dp_d^{(a)} \left(\frac{1}{2}\right)^N \delta\left(p_d^{(a)} - \frac{1}{2}\right) = \left(\frac{1}{2}\right)^{ND}$$

### Model (b): Bernoulli with Unknown but Identical $p_d$

Model (b) assumes that all $D$ components are generated from a Bernoulli distribution with an identical but unknown parameter $p_d$. To incorporate our uncertainty about this parameter, we assign a uniform prior distribution to $p_d$.

The probability of observing the data $X$ given the model can be expressed by marginalizing over the unknown parameter $p_d$:

$$P(X|M_b) = \int_0^1 P(X|p_d)P(p_d)\, dp_d$$

Substituting $p_d$ into the Bernoulli likelihood leads us to the following expression, which corresponds to a Beta-Binomial distribution:

$$P(X|M_b) = \int_0^1 p^{\sum_{n=1}^{N}\sum_{d=1}^{D} x_d^{(n)}} (1-p)^{ND - \sum_{n=1}^{N}\sum_{d=1}^{D} x_d^{(n)}}\, dp$$

This simplifies to the form:

$$P(X|M_b) = B(\alpha^{(b)}, \beta^{(b)})$$

where $\alpha^{(b)} = 1 + \sum_{n=1}^{N}\sum_{d=1}^{D} x_d^{(n)}$ and $\beta^{(b)} = 1 + ND - \sum_{n=1}^{N}\sum_{d=1}^{D} x_d^{(n)}$, and $B$ denotes the Beta function. The resulting expression quantifies the likelihood of the observed data under Model (b).

### Model (c): Bernoulli with Separate, Unknown $p_d$ for Each Component

Model (c) assumes that each pixel $d$ is generated from a Bernoulli distribution with its own, separate unknown parameter $p_d$, which is independently sampled for each pixel. We place a uniform prior on each $p_d$ over the interval $[0, 1]$. By the Bernoulli distribution, the probability of the data given $p = \{p_d\}_{d=1}^{D}$ is:

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

To compute the marginal likelihood of the data under Model (c), we integrate over each $p_d$ independently:

$$P(X|M_c) = \int dp\, P(X|p)P(p) = \prod_{d=1}^{D} \int_0^1 p_d^{\sum_{n=1}^{N} x_d^{(n)}} (1 - p_d)^{N-\sum_{n=1}^{N} x_d^{(n)}}\, dp_d$$

This integral is in the form of the Beta function:

$$P(X|M_c) = \prod_{d=1}^{D} B(\alpha_d, \beta_d)$$

where $\alpha_d = 1 + \sum_{n=1}^{N} x_d^{(n)}$ and $\beta_d = 1 + N - \sum_{n=1}^{N} x_d^{(n)}$, and $B(\alpha_d, \beta_d)$ denotes the Beta function.
Here's the posterior probabilities of the models (Code: Appendix 2).:

| Model | Log-Probability | Relative Probability | Normalized Posterior Probability |
|---|---|---|---|
| Model (a) | -4436.14 | $3.85 \times 10^{-190}$ | $9.14 \times 10^{-255}$ |
| Model (b) | -4283.72 | $6.04 \times 10^{-124}$ | $1.43 \times 10^{-188}$ |
| Model (c) | -3851.20 | $4.22 \times 10^{64}$ | 1.0 |

Table 1: Posterior probabilities of the three different models.

## 3 Question 3: EM for Binary Data.

(a) Let the parameters $\pi_1, \pi_2, \ldots, \pi_K$ denote the mixing proportions, where $0 \le \pi_k \le 1$ and $\sum_{k=1}^{K} \pi_k = 1$. Let $P_{kd}$ represent the probability that pixel $d$ takes the value 1 under mixture component $k$.

#### Likelihood for a Single Image Under One Component

For an individual image $x_n$ and a single Bernoulli component $k$, the likelihood is:

$$P(x_n|z_n = k, P_k) = \prod_{d=1}^{D} P_{kd}^{x_{nd}} (1 - P_{kd})^{1-x_{nd}}$$

where:

- $x_{nd}$ is the value of pixel $d$ in image $n$,
- $P_{kd}$ is the probability that pixel $d$ is 1 under component $k$.

#### Likelihood for a Mixture Model

For an individual image $x_n$, the likelihood under the mixture model is:

$$P(x_n|\pi, P) = \sum_{k=1}^{K} \pi_k \prod_{d=1}^{D} P_{kd}^{x_{nd}} (1 - P_{kd})^{1-x_{nd}}$$

**Likelihood for the Entire Dataset**

Given $N$ images, the likelihood for the entire dataset $X = \{x_1, x_2, \ldots, x_N\}$ is the product of the individual image likelihoods:

$$P(X|\pi, P) = \prod_{n=1}^{N} P(x_n|\pi, P)$$

Substituting the likelihood for each image to get the final answer:

$$P(X|\pi, P) = \prod_{n=1}^{N} \sum_{k=1}^{K} \pi_k \prod_{d=1}^{D} P_{kd}^{x_{nd}} (1 - P_{kd})^{1-x_{nd}}$$

(b) The responsibility of mixture component $k$ for data vector $x^{(n)}$ is denoted as:

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

This is the E-step in the EM algorithm, where we compute the probability that the data point $x^{(n)}$ was generated by component $k$, given the current model parameters $\pi$ and $P$.

**Bayes' Theorem for Responsibility**

Using Bayes' theorem, the responsibility $r_{nk}$ is given by:

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

where:

- $\pi_k$ is the mixing proportion for component $k$,
- $P(x^{(n)}|s^{(n)} = k, P_k)$ is the likelihood of the data vector $x^{(n)}$ given component $k$.

**Bernoulli Likelihood**

The likelihood of $x^{(n)}$ under component $k$, where each pixel is Bernoulli-distributed, is given by:

$$P(x^{(n)}|s^{(n)} = k, P_k) = \prod_{d=1}^{D} P_{kd}^{x_{nd}} (1 - P_{kd})^{1-x_{nd}}$$

Therefore, the responsibility $r_{nk}$ is:

$$r_{nk} = \frac{\pi_k \prod_{d=1}^{D} P_{kd}^{x_{nd}} (1 - P_{kd})^{1-x_{nd}}}{\sum_{j=1}^{K} \pi_j \prod_{d=1}^{D} P_{jd}^{x_{nd}} (1 - P_{jd})^{1-x_{nd}}}$$

This expression provides the responsibility of component $k$ for data point $x^{(n)}$.

(c) To find the maximizing parameters for the expected log-joint of the observed data and the latent variables, it corresponds to the M-step in the EM algorithm. The expected log-joint is:

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

where $q(s^{(n)}) = P(s^{(n)} = k|x^{(n)}, \pi, P) = r_{nk}$, the responsibility computed in part (b).

**The Log-Joint Probability**

The joint probability $P(x^{(n)}, s^{(n)}|\pi, P)$ is given by:

$$P(x^{(n)}, s^{(n)}|\pi, P) = \pi_k \prod_{d=1}^{D} P_{kd}^{x_{nd}}(1 - P_{kd})^{1-x_{nd}}$$

Taking the log:

$$\log P(x^{(n)}, s^{(n)}|\pi, P) = \log \pi_k + \sum_{d=1}^{D} [x_{nd} \log P_{kd} + (1 - x_{nd}) \log(1 - P_{kd})]$$

**Expected Log-Joint**

The expected log-joint is computed as:

$$\mathbb{E}\left[\sum_n \log P(x^{(n)}, s^{(n)}|\pi, P)\right] = \sum_{n=1}^{N}\sum_{k=1}^{K} r_{nk} \left(\log \pi_k + \sum_{d=1}^{D} [x_{nd} \log P_{kd} + (1 - x_{nd}) \log(1 - P_{kd})]\right)$$

**Maximizing With Respect to $\pi$**

To maximize with respect to $\pi$, we maximize:

$$\sum_{n=1}^{N}\sum_{k=1}^{K} r_{nk} \log \pi_k$$

subject to $\sum_{k=1}^{K} \pi_k = 1$.
We introduce a Lagrange multiplier $\lambda$ and define the Lagrangian function as:

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

where $\lambda \left(1 - \sum_{k=1}^{K} \pi_k\right)$ enforces the constraint by penalizing any deviation from $\sum_{k=1}^{K} \pi_k = 1$.
To find the maximum, we take the partial derivative of $\mathcal{L}$ with respect to each $\pi_k$ and set it to zero:

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

Rearranging, we get:

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

Next, we use the constraint $\sum_{k=1}^{K} \pi_k = 1$ to solve for $\lambda$. Substituting $\pi_k = \frac{1}{\lambda} \sum_{n=1}^{N} r_{nk}$ into the constraint, we have:

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

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

$$\lambda = \sum_{n=1}^{N}\sum_{k=1}^{K} r_{nk} = N$$

Substituting $\lambda$ back into the expression for $\pi_k$, we obtain:

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

**Maximizing With Respect to $P$**

To maximize with respect to $P_{kd}$, we maximize:

$$\sum_{n=1}^{N} r_{nk}\sum_{d=1}^{D}\left[x_{nd}\log P_{kd} + (1-x_{nd})\log(1-P_{kd})\right]$$

Taking the partial derivative of the objective function with respect to $P_{kd}$ and setting it to zero:

$$\sum_{n=1}^{N} r_{nk}\frac{x_{nd}}{P_{kd}} = \sum_{n=1}^{N} r_{nk}\frac{1-x_{nd}}{1-P_{kd}}$$

Rearranging terms and solving for $P_{kd}$:

$$P_{kd} = \frac{\sum_{n=1}^{N} r_{nk}x_{nd}}{\sum_{n=1}^{N} r_{nk}}$$

Thus, the optimal value of $P_{kd}$ is the weighted average of $x_{nd}$ across all data points $n$, where the weights are given by $r_{nk}$, the responsibility of the $k$-th component for each data point.

Hence, the updates for the parameters in the M-step are:

1. **Mixing Proportions**:

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

2. **Bernoulli Parameters**:

$$P_{kd} = \frac{\sum_{n=1}^{N} r_{nk}x_{nd}}{\sum_{n=1}^{N} r_{nk}}$$

(d) Please find the code in Appendix 3d.

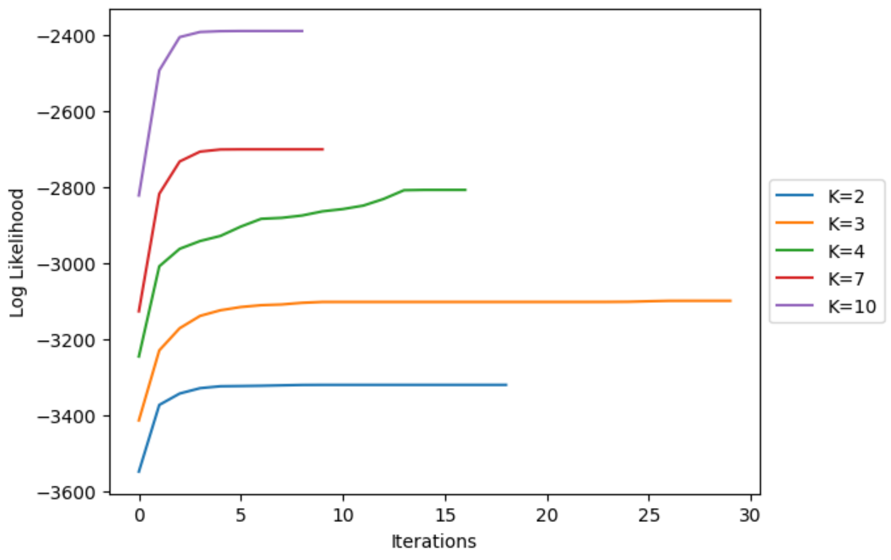


Figure 3: Log-likelihood by numbers of Bernoulli models v.s Iterations of the EM Algorithm

We implemented the EM algorithm to fit a mixture of Bernoulli distributions with varying numbers of components $K$. For each choice of $K$, we initialized the parameters $P$ and $\pi$ randomly. The log-likelihood plot presented here shows the result from one random initialization. The algorithm was run for 30 iterations for each value of $K$, and we recorded the log-likelihood at each step to assess the convergence behavior.

A few observations from the plot:

- **Higher $K$ Generally Increases the Log-Likelihood**: As $K$ increases, the model achieves a higher final log-likelihood. This behavior is expected because a larger number of components allows the model to better capture the complexity of the data, thereby improving its fit.
- **Plateauing Behavior for Each $K$**: The log-likelihood for each $K$ quickly plateaus within the first few iterations, indicating stable convergence. This suggests that the EM algorithm finds an optimal or near-optimal solution for each $K$ within the 30 iterations.
- **Distinct Gaps in Log-Likelihood Across $K$ Values**: There are noticeable gaps in the final log-likelihoods for different $K$ values. This distinction helps identify which values of $K$ might be more appropriate, as the differences in log-likelihood reflect how well each model fits the data.

However, it is essential to consider the potential for **overfitting with larger $K$**. Although higher $K$ values lead to a better fit, they also increase the model's complexity, which may lead to overfitting if $K$ becomes too large. Thus, an optimal $K$ should balance model fit and simplicity.

(e) We ran the algorithm 7 times with random initializations for each value of $K$. The output learned probability vectors and update parameters $\pi$ is shown below. Each row of the $P$ matrix is visualized as a grayscale image, with the mixing proportions, $\pi$, shown as a torch tensor. Please find the code in Appendix 3e.

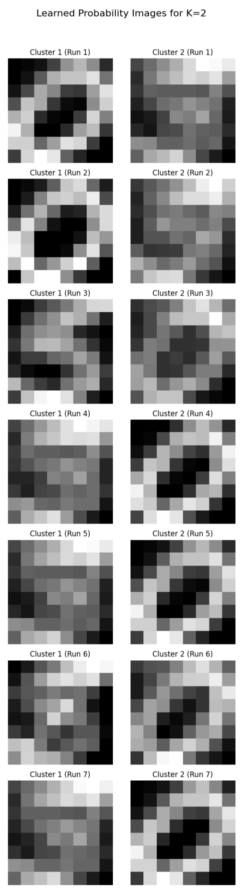


Figure 4: Learned Probability Vectors for $K = 2$

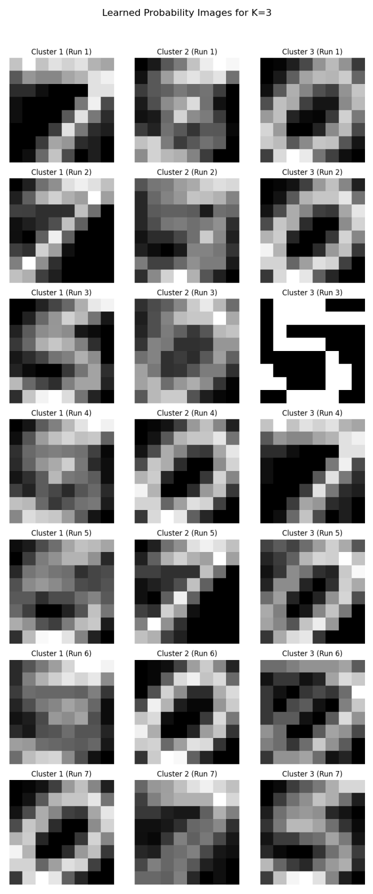


Figure 5: Learned Probability Vectors for $K = 3$

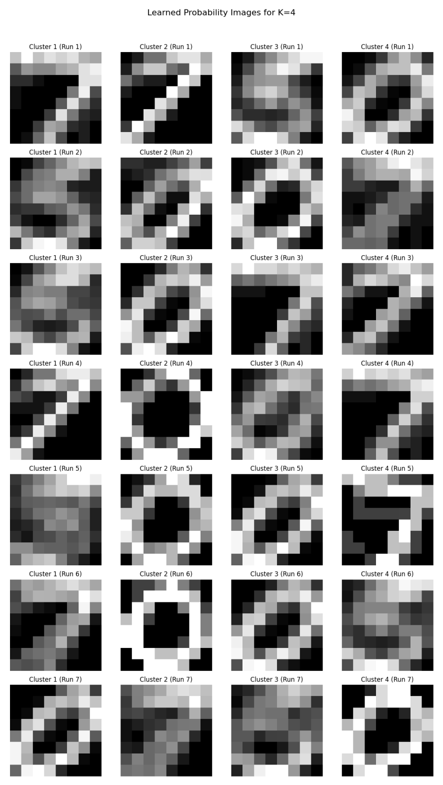


Figure 6: Learned Probability Vectors for $K = 4$

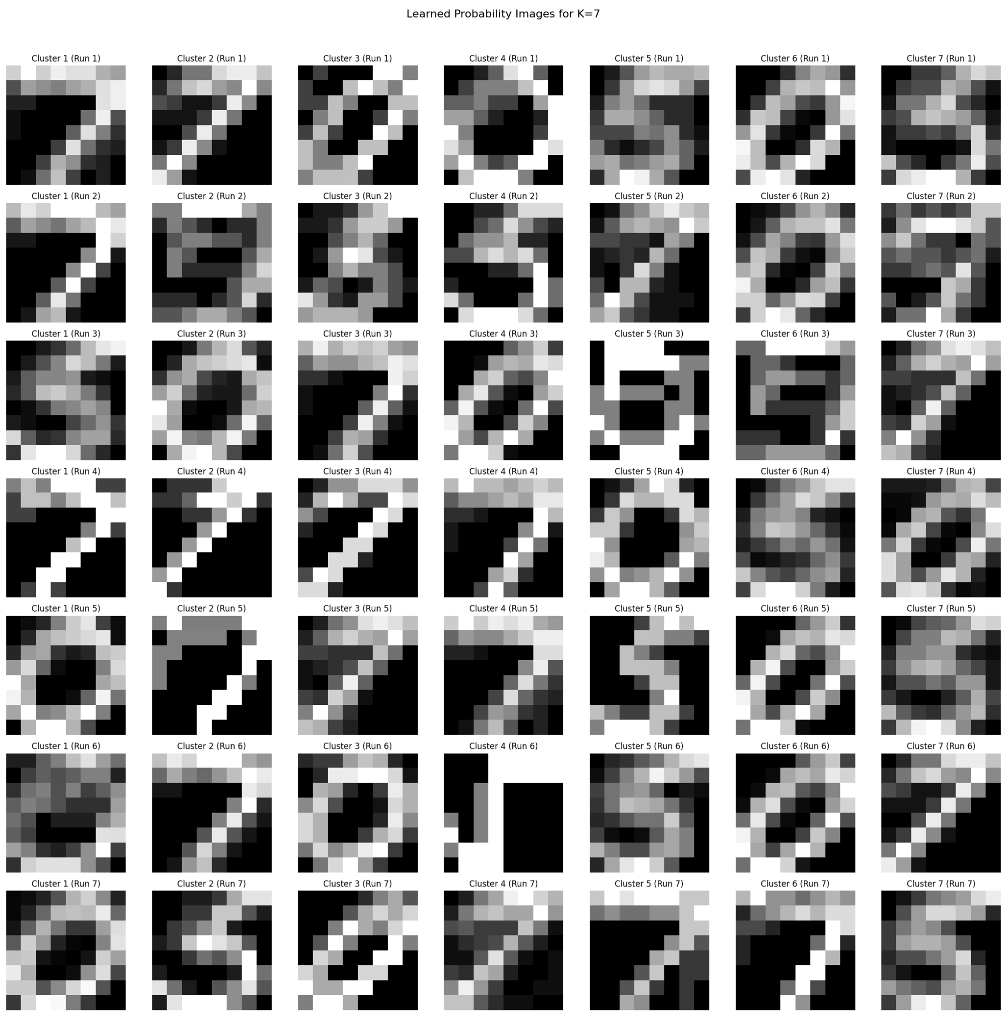


Figure 7: Learned Probability Vectors for $K = 7$

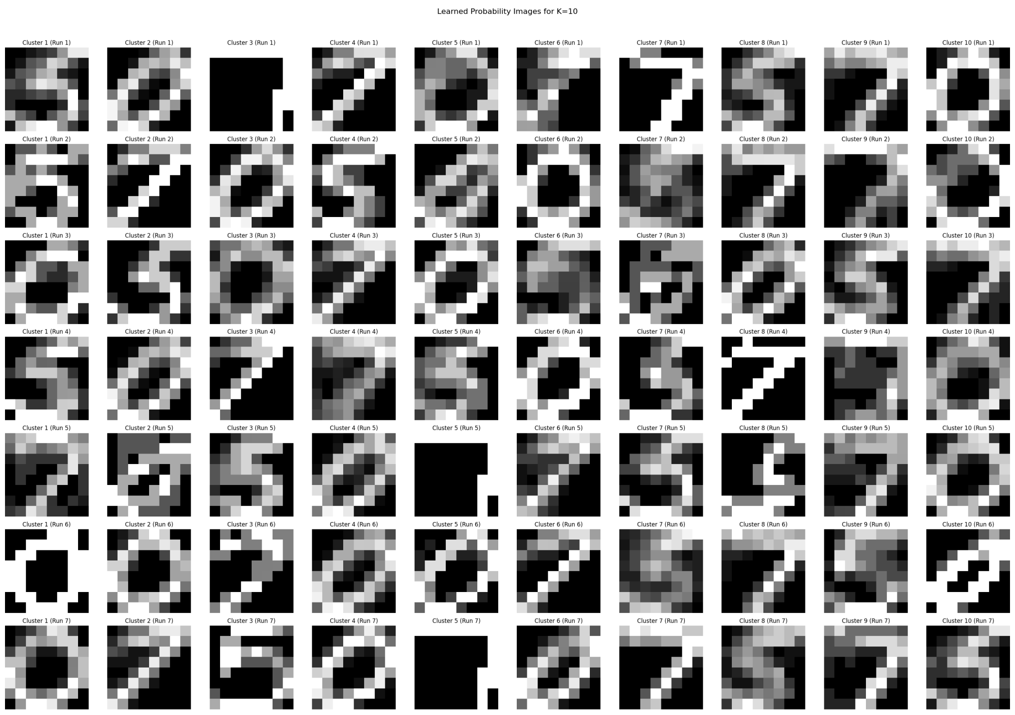


Figure 8: Learned Probability Vectors for $K = 10$

**Consistency Across Runs and Dependence on $K$:** Running the algorithm multiple times with different random initializations generally does not yield exactly identical solutions, though the clusters are often similar up to a permutation. For smaller values of $K$ (e.g., $K = 2$ and $K = 3$), the algorithm tends to converge to similar clusters across different runs, demonstrating greater consistency. However, as $K$ increases (e.g., $K = 7$ and $K = 10$), the solutions exhibit greater variability due to the model's increased complexity and flexibility, resulting in potential differences in local optima reached by the EM algorithm. This observation suggests that the consistency of clustering is dependent on $K$, with larger values introducing more variance in the final cluster assignments due to the model's added flexibility.

**Performance and Interpretation of Clusters:** For lower values of $K$, the learned clusters generally reveal clear, interpretable patterns that capture coarse features of the data, and the algorithm converges quickly to these patterns. For instance, with $K = 2$, we observe that clusters broadly categorize the data, while at $K = 3$ and $K = 4$, clusters represent more distinct binary patterns. As $K$ increases to higher values, such as $K = 7$ and $K = 10$, the clusters become more specialized, potentially capturing finer details in the data. However, this increased specificity may lead to overfitting, where some clusters capture noise rather than meaningful structures. The learned probability vectors for larger $K$ values may sometimes represent overly specific patterns, which may not generalize well to new data.

**Suggestions for Improvement:**

- **Regularization:** Adding a prior or regularization term could help mitigate overfitting for larger values of $K$. For example, applying a Dirichlet prior to the mixing proportions $\pi$ could encourage more balanced cluster sizes.
- **Parameter Initialization:** Employing more effective initialization strategies, such as K-means++ or spectral clustering-based initialization, might improve the consistency of results across runs, particularly for higher values of $K$.
- **Model Selection:** Determining the optimal number of clusters $K$ could be approached through model selection techniques such as cross-validation or the Bayesian Information Criterion (BIC). This could help in preventing over-clustering, thereby reducing the risk of overfitting.

Table 2: Mixing Proportions ($\pi$) for $K = 2$

| Run | Cluster 1 | Cluster 2 |
|---|---|---|
| Run 1 | 0.4196 | 0.5804 |
| Run 2 | 0.2700 | 0.7300 |
| Run 3 | 0.2481 | 0.7519 |
| Run 4 | 0.6200 | 0.3800 |
| Run 5 | 0.5804 | 0.4196 |
| Run 6 | 0.3394 | 0.6606 |
| Run 7 | 0.6001 | 0.3999 |

Table 3: Mixing Proportions ($\pi$) for $K = 3$

| Run | Cluster 1 | Cluster 2 | Cluster 3 |
|---|---|---|---|
| Run 1 | 0.1800 | 0.3397 | 0.4803 |
| Run 2 | 0.1600 | 0.4403 | 0.3997 |
| Run 3 | 0.2192 | 0.7708 | 0.0100 |
| Run 4 | 0.4200 | 0.4000 | 0.1800 |
| Run 5 | 0.3901 | 0.1600 | 0.4499 |
| Run 6 | 0.5203 | 0.3396 | 0.1402 |
| Run 7 | 0.4000 | 0.3500 | 0.2500 |

Table 4: Mixing Proportions ($\pi$) for $K = 4$

| Run | Cluster 1 | Cluster 2 | Cluster 3 | Cluster 4 |
|---|---|---|---|---|
| Run 1 | 0.1800 | 0.0900 | 0.3302 | 0.3998 |
| Run 2 | 0.2800 | 0.1602 | 0.2498 | 0.3100 |
| Run 3 | 0.4202 | 0.3098 | 0.1300 | 0.1400 |
| Run 4 | 0.1300 | 0.0500 | 0.6600 | 0.1600 |
| Run 5 | 0.5600 | 0.1400 | 0.2600 | 0.0400 |
| Run 6 | 0.2394 | 0.0400 | 0.3197 | 0.4009 |
| Run 7 | 0.2490 | 0.3395 | 0.3415 | 0.0700 |

Table 5: Mixing Proportions ($\pi$) for $K = 7$

| Run | Cl. 1 | Cl. 2 | Cl. 3 | Cl. 4 | Cl. 5 | Cl. 6 | Cl. 7 |
|---|---|---|---|---|---|---|---|
| Run 1 | 0.1700 | 0.1300 | 0.0398 | 0.0800 | 0.1800 | 0.2702 | 0.1300 |
| Run 2 | 0.1100 | 0.0600 | 0.1000 | 0.0700 | 0.1400 | 0.4300 | 0.0900 |
| Run 3 | 0.1900 | 0.2100 | 0.1600 | 0.2100 | 0.0200 | 0.0500 | 0.1600 |
| Run 4 | 0.0400 | 0.0500 | 0.0700 | 0.1100 | 0.1400 | 0.2900 | 0.3000 |
| Run 5 | 0.2000 | 0.0200 | 0.1600 | 0.1603 | 0.0400 | 0.2000 | 0.2200 |
| Run 6 | 0.1601 | 0.1200 | 0.1300 | 0.0200 | 0.2100 | 0.2300 | 0.1300 |
| Run 7 | 0.3800 | 0.0900 | 0.0600 | 0.1800 | 0.0900 | 0.0700 | 0.1300 |

Table 6: Mixing Proportions ($\pi$) for $K = 10$

| Run | Cl. 1 | Cl. 2 | Cl. 3 | Cl. 4 | Cl. 5 | Cl. 6 | Cl. 7 | Cl. 8 | Cl. 9 | Cl. 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| Run 1 | 0.1195 | 0.2400 | 0.0200 | 0.0800 | 0.1000 | 0.0800 | 0.0200 | 0.1105 | 0.1400 | 0.0900 |
| Run 2 | 0.0300 | 0.0600 | 0.2102 | 0.0400 | 0.1297 | 0.0500 | 0.2301 | 0.1002 | 0.0798 | 0.0700 |
| Run 3 | 0.0599 | 0.0500 | 0.1100 | 0.1300 | 0.0905 | 0.0801 | 0.0300 | 0.1995 | 0.1100 | 0.1400 |
| Run 4 | 0.0497 | 0.2300 | 0.0500 | 0.2500 | 0.0903 | 0.0700 | 0.0500 | 0.0100 | 0.0500 | 0.1500 |
| Run 5 | 0.0999 | 0.0300 | 0.0600 | 0.2300 | 0.0200 | 0.1600 | 0.0800 | 0.0200 | 0.1101 | 0.1900 |
| Run 6 | 0.0100 | 0.1100 | 0.0200 | 0.1810 | 0.0790 | 0.0800 | 0.2801 | 0.1400 | 0.0699 | 0.0300 |
| Run 7 | 0.1202 | 0.1300 | 0.0300 | 0.1598 | 0.0200 | 0.1200 | 0.0900 | 0.1500 | 0.0700 | 0.1100 |

(f) To compare the log-likelihoods obtained in bits and compare it with length of the naive encoding of these binary data, we got following result (code in Appendix 3f):

| $K$ | Log-Likelihood (Bits) | Naive Encoding (Bits) | gzip Compression (Bits) |
|---|---|---|---|
| 2 | -4816.65 | 6400 | 8376 |
| 3 | -4550.13 | 6400 | 8376 |
| 4 | -4020.76 | 6400 | 8376 |
| 7 | -3755.49 | 6400 | 8376 |
| 10 | -3519.69 | 6400 | 8376 |

Table 7: Comparison of Log-Likelihood, Naive Encoding, and gzip Compression in Bits

- **Log-Likelihood in Bits:** The log-likelihood values are consistently lower than both the naive encoding and gzip compression lengths, indicating the EM model's ability to exploit the patterns in the
- **Naive Encoding Length:** The naive encoding length remains constant at 6400 bits, calculated as $N \times D$, where each bit is stored individually without modeling any structure.
- **Gzip Compression Length:** Gzip compression is also constant at 8376 bits, reflecting its general-purpose dictionary-based compression, which is not optimized for binary data modeled by Bernoulli distributions.

The differences between the methods arise because:

- **Naive Encoding:** Assumes no structure in the data, storing each binary value individually.
- **Gzip Compression:** Uses dictionary-based methods to compress sequences but cannot effectively capture statistical dependencies in binary data.
- **EM Model (Log-Likelihood):** Tailored to the dataset, explicitly modeling its structure as a mixture of multivariate Bernoulli distributions, resulting in more efficient representation.

(g) The total encoding cost consists of two components:

- **Parameter cost**: This reflects the cost of encoding the model parameters, including the mixing coefficients ($\pi$) and Bernoulli parameters ($P$).
- **Data encoding cost**: This is proportional to the negative log-likelihood of the data under the model.

The results for different values of $K$ are as follows (code in Appendix 3g):

- $K = 2$: Total cost = 5806.07 bits, Parameter cost = 1040 bits.
- $K = 3$: Total cost = 6001.20 bits, Parameter cost = 1560 bits.
- $K = 4$: Total cost = 6194.12 bits, Parameter cost = 2080 bits.
- $K = 7$: Total cost = 7282.54 bits, Parameter cost = 3640 bits.
- $K = 10$: Total cost = 8735.99 bits, Parameter cost = 5200 bits.

The total encoding cost increases as $K$ increases. This is because the *parameter cost* grows linearly with $K$, as more components require additional mixing coefficients ($\pi$) and Bernoulli parameters ($P$). Also, the improvement in log-likelihood diminishes as $K$ increases, leading to higher total costs due to overfitting.

The results indicate that the total cost is minimized at $K = 2$, where the total cost is 5806.07 bits. This suggests that a simpler model ($K = 2$) is sufficient to capture the structure of the data without overfitting. Increasing $K$ results in diminishing returns for the log-likelihood, while the parameter cost grows significantly.

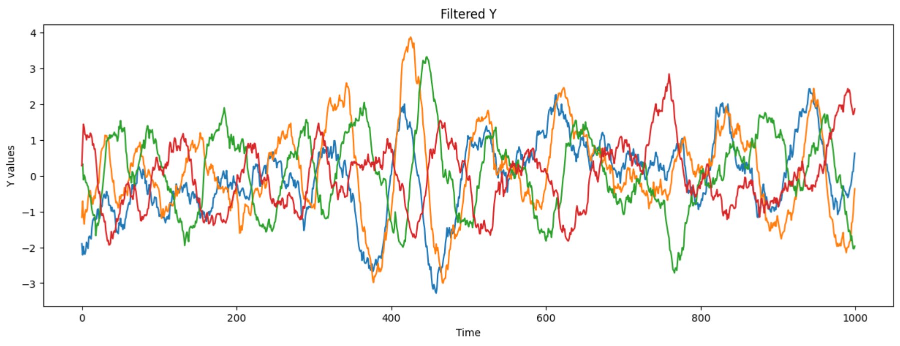
**Comparison with gzip:** Compression algorithms like gzip are designed to identify patterns in the data and encode them efficiently, without considering model complexity. The EM algorithm, however, explicitly models the data using a mixture of Bernoulli distributions. As a result:

- For smaller $K$, the EM model can achieve a lower total cost compared to gzip, as it balances parameter cost and data fit effectively.
- For larger $K$, the parameter cost outweighs the gains in log-likelihood, making gzip potentially more efficient in terms of overall compression.

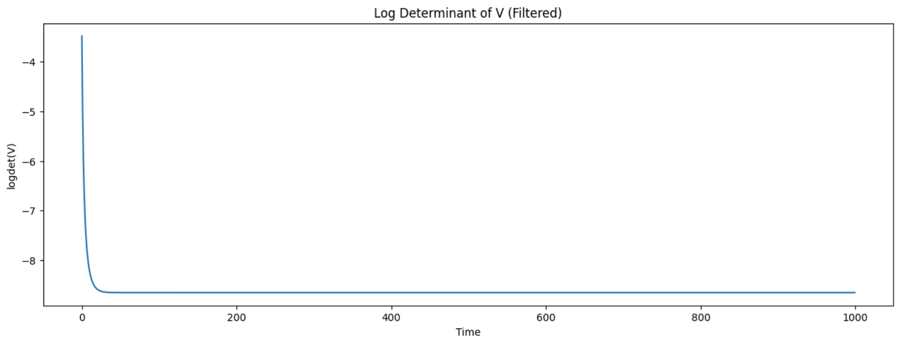
## 4 Question 4: LGSSMs, EM and SSID

### Filtered Results

(a)
- **Filtered $Y$:**
  The filtered $Y$ plot shows a time series that reflects the hidden states as estimated by the Kalman filter after observing each data point up to the current time. This time series is noisier than the smoothed result, as filtering relies only on past observations up to the current time step, resulting in estimates that are more sensitive to immediate fluctuations in the data. The filtered $Y$ trajectory can be thought of as a real-time estimate, focusing on adjusting to each new observation without considering future information.
- **Log Determinant of Filtered $V$:**
  The log determinant of $V$ decreases rapidly and then stabilizes over time. $V$ represents the variance (uncertainty) of the state estimates; hence, a smaller log determinant indicates reduced uncertainty as the Kalman filter gains more observations. The rapid decline at the beginning suggests that the filter quickly gains confidence in its state estimates as it integrates initial data, after which the uncertainty stabilizes.



(a) Filtered $Y$

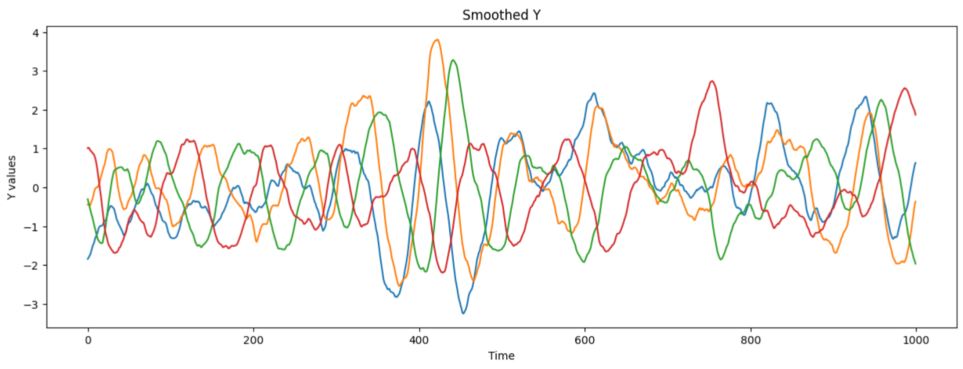


(b) Log Determinant of $V$ (Filtered)

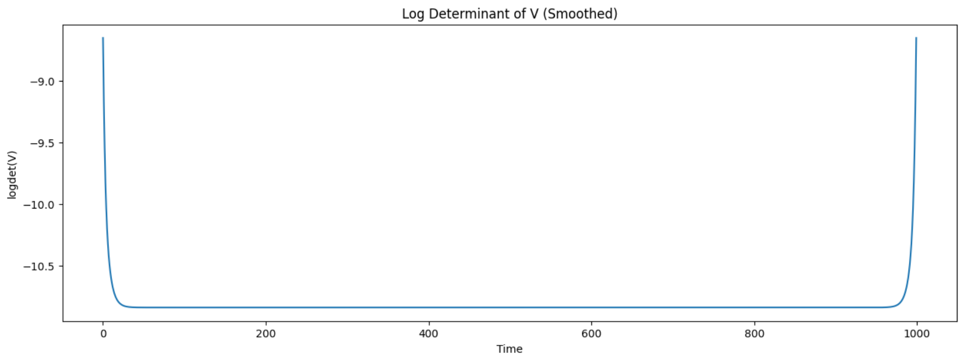
Figure 9: Filtered Results for $Y$ and Log Determinant of $V$

## Smoothed Results

- **Smoothed $Y$:**
  The smoothed $Y$ plot is generally smoother and less noisy compared to the filtered $Y$. Smoothing takes into account both past and future observations for each time step, resulting in more stable estimates that effectively "average out" noise from individual observations. The smoothed $Y$ trajectory captures the underlying trend more accurately by leveraging all data points, leading to a cleaner representation of the hidden states.
- **Log Determinant of Smoothed $V$:**
  The log determinant of $V$ for the smoothed case exhibits a similar initial drop in uncertainty, but it shows a more symmetric pattern that dips in the middle and increases slightly towards the end. This symmetric pattern arises because smoothing considers information from the past and the future. At the start, uncertainty is high because of limited availability of observations. Towards the end, there are fewer future observations available, which caused the increase in uncertainty at the boundaries.



(a) Filtered $Y$



(b) Log Determinant of $V$ (Filtered)

Figure 10: Filtered Results for $Y$ and Log Determinant of $V$

## Summary of Differences

- **Filtered vs. Smoothed $Y$:**
  The filtered $Y$ captures immediate fluctuations and reacts quickly to new data, while the smoothed $Y$ provides a cleaner, less noisy estimate by using information from the entire time series.
- **Filtered vs. Smoothed $V$:**
  The filtered $V$ shows a monotonic decrease in uncertainty, while the smoothed $V$ demonstrates a balanced pattern, reflecting lower uncertainty in the central region of the time series and slightly higher uncertainty at the edges.

These differences illustrate how smoothing provides a more refined estimate of the hidden states by considering future observations, in contrast to filtering, which operates in real-time and thus yields noisier estimates. Please find the code in Appendix 4a.

(b) To learn the parameters $A$, $Q$, $C$, and $R$ for a Linear Gaussian State-Space Model (LGSSM) using the Expectation-Maximization (EM) algorithm, we update each parameter iteratively based on the expected log-likelihood of the observed data. The updates for $R$ and $Q$ are derived from the M-step.

## M-step Update Equations

The update rules for these parameters are derived as follows:

For the matrix $C$, we maximize the expected log likelihood of the observed data conditioned on the latent state $z_t$:

$$C_{\text{new}} = \arg\max_C \left\langle \sum_t \ln p(x_t|z_t) \right\rangle_q .$$

This gives us the formula for $C_{\text{new}}$:

$$C_{\text{new}} = \left( \sum_t x_t \langle z_t \rangle^T \right) \left( \sum_t \langle z_t z_t^T \rangle \right)^{-1} .$$

For the matrix $A$, we maximize the expected log likelihood of the latent state at time $t+1$ given the latent state at time $t$:

$$A_{\text{new}} = \arg\max_A \left\langle \sum_t \ln p(z_{t+1}|z_t) \right\rangle_q .$$

The resulting update for $A$ is:

$$A_{\text{new}} = \left( \sum_t \langle z_{t+1} z_t^T \rangle \right) \left( \sum_t \langle z_t z_t^T \rangle \right)^{-1} .$$

Finally, the updates for $R$ and $Q$ are given by:

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

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

We use these update equations to iteratively refine the parameters $A$, $Q$, $C$, and $R$ until convergence.

## Log-Likelihood Plot and Analysis

The EM algorithm was implemented with the code in Appendix 4b. The code initializes parameters randomly for 10 different initializations and once with the generating parameters. The log-likelihood values are then plotted to observe convergence. The log-likelihood plot generated from the code is shown below. This plot shows how the log-likelihood changes over 50 iterations for different initializations, including the generating parameters and 10 random initializations.

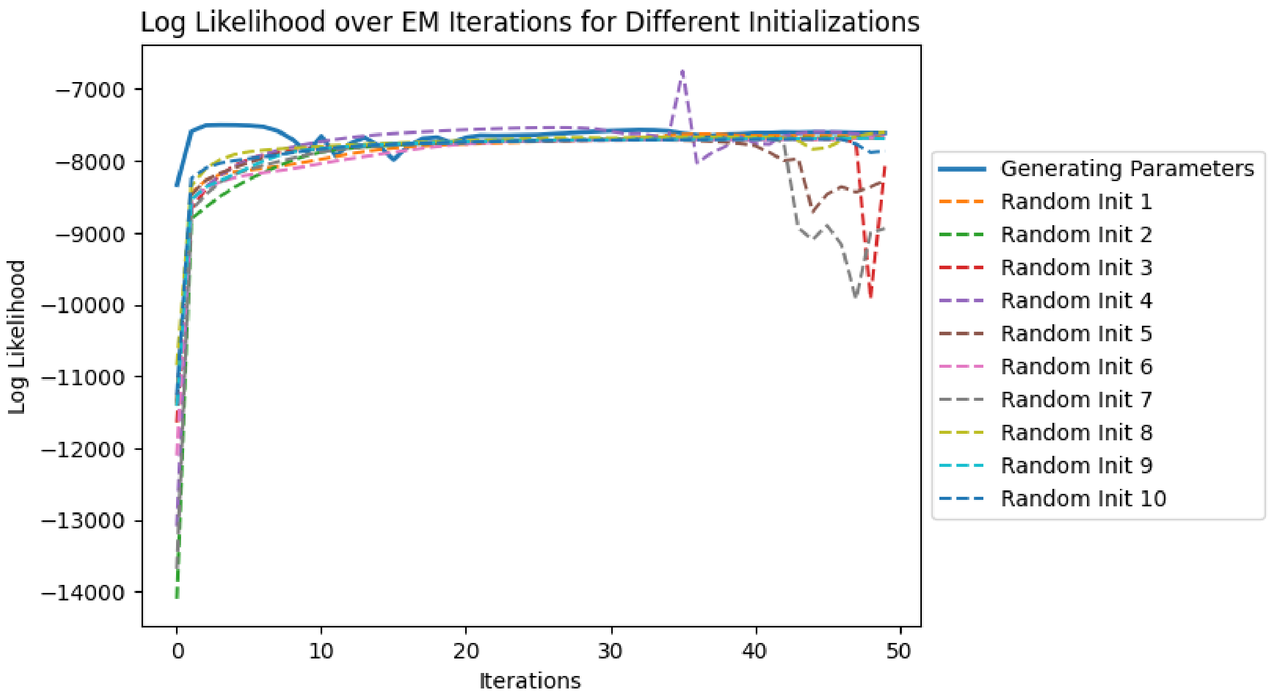


Figure 11: Log Likelihood over EM Iterations for Different Initializations

Here are some key observations from the plot:

1. **Convergence Behavior**: The plot shows that the log-likelihood generally increases over time, with most initializations converging within the first 30 iterations. This is consistent with the expectation that the EM algorithm should monotonically increase or maintain the log-likelihood.

2. **Generating Parameters**: The curve for the generating parameters achieves a higher likelihood compared to some random initializations. This behavior is expected, as the generating parameters should ideally be closer to the optimal solution.

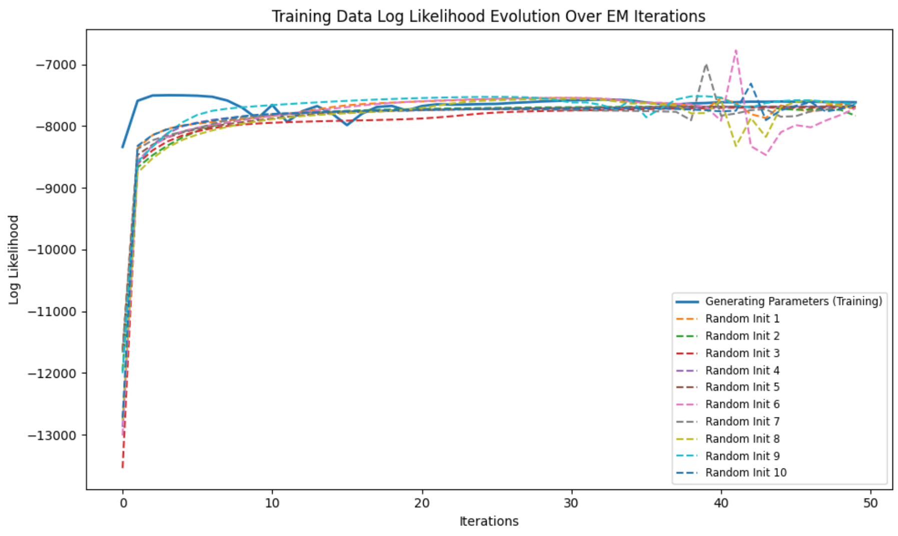
3. **Random Initializations**: Different random initializations converge to slightly different final log-likelihood values, indicating that some initializations may reach suboptimal local maxima. However, most random runs approach a similar likelihood, suggesting that the likelihood landscape is well-behaved.

4. **Fluctuations in Log-Likelihood**: Minor fluctuations in the log-likelihood in some random runs are observed, likely due to numerical instabilities. Adding small regularizations or adjusting convergence criteria could mitigate these fluctuations.

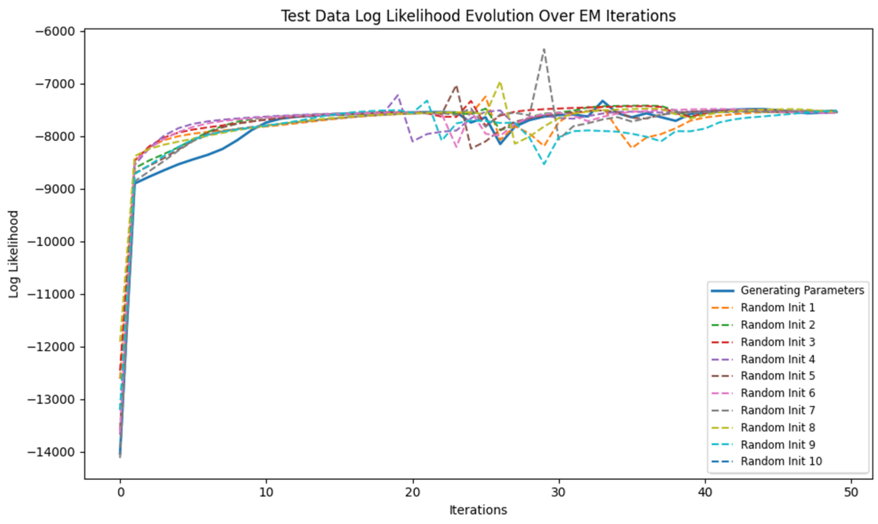
5. **Why EM Does Not Terminate Immediately**: EM requires multiple iterations because each step only partially maximizes the expected log-likelihood. Unlike gradient-based optimization, which directly seeks the maximum, EM iterates between expectation and maximization steps, gradually improving parameter estimates.

To further improve the model, numerical stability can be enhanced by implementing additional stability measures, such as regularization, to reduce minor fluctuations in the log-likelihood plot and improve convergence. Increasing the number of iterations beyond 50 may also contribute to stabilizing the log-likelihood, especially for random initializations that are close to convergence. Additionally, using prior knowledge for parameter initialization, as demonstrated with the generating parameters, can facilitate more efficient convergence by guiding the EM algorithm toward more optimal starting points.

(c) The results of test data under the true parameters, as well as the parameters obtained via EM initialization at the true parameters and random initializations, are shown in the plots provided. (code implemented: Appendix 4c



(a) Training Data Log Likelihood



(b) Test Data Log Likelihood

Figure 12: Log Likelihood Evolution Over EM Iterations for Training and Test Data. The left plot shows the training data, while the right plot shows the test data.

**Observations**

- The log-likelihoods for both the training and test datasets converge over EM iterations, regardless of initialization.
- **EM Initialized with True Parameters**: This initialization converges smoothly and achieves a likelihood very close to the maximum for both training and test data. This indicates that the EM algorithm successfully refines the model to fit the data.
- **Random Initializations**:
  - The likelihoods for random initializations exhibit variability in the first few iterations but generally converge to similar values for both training and test data.
  - Some random initializations lead to slower convergence or slightly suboptimal results, as seen in the test data plot where a few runs achieve lower final likelihoods.
- **Training vs. Test Data**: The training log-likelihoods are consistently higher than those for the test data, which is expected due to overfitting to the training data. However, the close proximity of test log-likelihoods indicates good generalization.

The evaluation shows that the EM algorithm is effective in optimizing model parameters, achieving high likelihoods for both training and test datasets. The results highlight the algorithm's robustness across initializations, with the true parameters providing a strong benchmark. The small generalization gap underscores the model's ability to perform well on unseen data.

## 5 Question 5: Decrypting Messages with MCMC

### (a) ML Estimates of Transition Probabilities and Stationary Distribution

We estimate the transition probabilities and the stationary distribution of symbols based on a large sample of English text (e.g., *War and Peace*).

Let $s_i$ denote the $i$-th symbol in the text. We aim to estimate:

**Stationary Distribution** $\phi(\gamma)$: the long-term probability of each symbol $\gamma$ in the text.

**Transition Probability** $\psi(\alpha, \beta) = p(s_i = \alpha \mid s_{i-1} = \beta)$: the probability that a symbol $\alpha$ follows symbol $\beta$.

**Formulae for ML Estimates**

Given counts of individual symbols and symbol pairs in the text, we can compute the ML estimates for the stationary distribution and transition probabilities.

- **Stationary Distribution** $\phi(\gamma)$: The stationary distribution for a symbol $\gamma$ represents the long-run probability of observing $\gamma$ in the text. Computationally, we approximate this by iteratively applying the transition matrix to an initial uniform distribution vector until convergence:

$$\phi(\gamma) \approx \lim_{n\to\infty} \pi^{(0)} P^n$$

  where $\pi^{(0)}$ is an initial uniform distribution vector.
- **Transition Probability** $\psi(\alpha, \beta)$: The ML estimate for the transition probability of symbol $\beta$ following symbol $\alpha$ is given by

$$\psi(\alpha, \beta) = \frac{C(\beta, \alpha)}{\sum_\gamma C(\beta, \gamma)}$$

  where $C(\beta, \alpha)$ is the count of transitions from symbol $\beta$ to symbol $\alpha$, and $\sum_\gamma C(\beta, \gamma)$ is the total count of occurrences of $\beta$ as a preceding symbol.

**Estimated Probabilities**

Using the text data, we computed the stationary distribution as below (code in Appendix 5a):

| Symbol | Stationary Probability |
|---|---|
| (space) | 1.816e-01 |
| ! | 1.401e-03 |
| " | 2.031e-04 |
| ’ | 1.980e-04 |
| ( | 4.168e-04 |
| ) | 6.249e-04 |
| * | 2.920e-04 |
| , | 1.286e-02 |
| - | 7.862e-04 |
| . | 1.033e-02 |
| / | 2.051e-04 |
| 0 | 7.583e-04 |
| 1 | 5.911e-04 |
| 2 | 3.727e-04 |
| 3 | 2.476e-04 |
| 4 | 2.509e-04 |
| 5 | 3.413e-04 |
| 6 | 3.609e-04 |
| 7 | 3.023e-04 |
| 8 | 5.062e-04 |
| 9 | 2.975e-04 |
| : | 5.195e-04 |
| ; | 5.580e-04 |
| = | 1.965e-04 |
| [ | 1.965e-04 |
| ] | 2.029e-04 |

Table 8: Sample of Estimated Stationary Probabilities for Symbols and Numbers

| Letter | Stationary Probability |
|---|---|
| a | 6.247e-02 |
| b | 1.080e-02 |
| c | 1.914e-02 |
| d | 3.636e-02 |
| e | 9.643e-02 |
| f | 1.709e-02 |
| g | 1.582e-02 |
| h | 5.167e-02 |
| i | 5.273e-02 |
| j | 9.826e-04 |
| k | 6.338e-03 |
| l | 2.942e-02 |
| m | 1.913e-02 |
| n | 5.586e-02 |
| o | 5.862e-02 |
| p | 1.436e-02 |
| q | 9.123e-04 |
| r | 4.567e-02 |
| s | 4.826e-02 |
| t | 6.911e-02 |
| u | 2.019e-02 |
| v | 7.948e-03 |
| w | 1.816e-02 |
| x | 1.557e-03 |
| y | 1.410e-02 |
| z | 7.400e-04 |

Table 9: Sample of Estimated Stationary Probabilities for Letters (a-z)

Here’s a heatmap that visualizes the transition matrix we calculated:

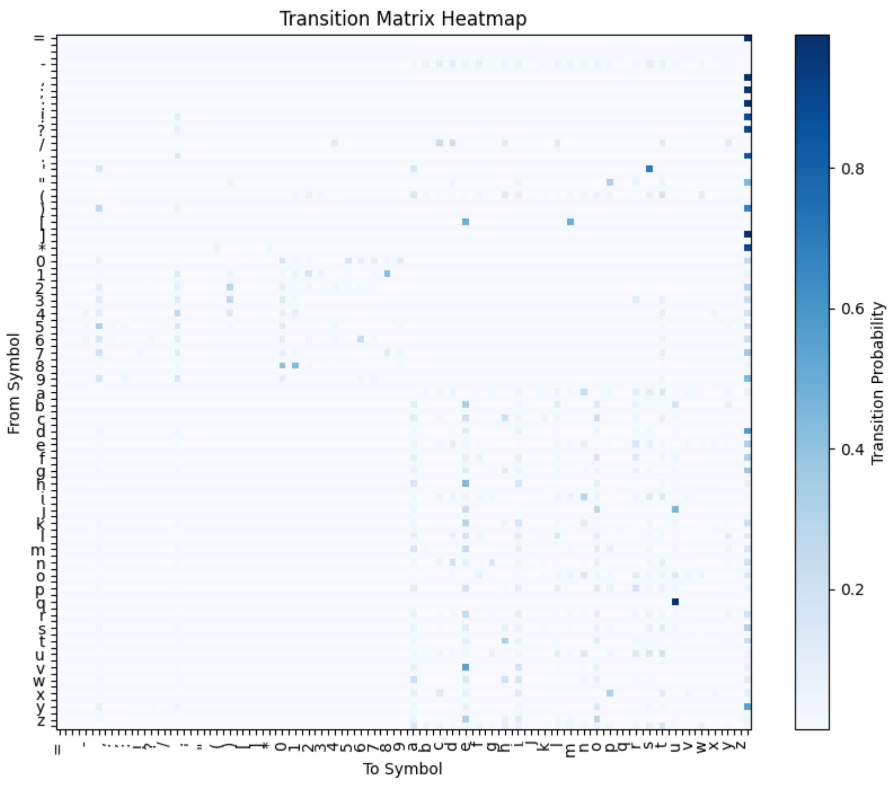


Figure 13: Estimated Transition Probabilities for Symbols

(b) **Independence of Latent Variables $\sigma(s)$**

The latent variables $\sigma(s)$, representing the mappings for each symbol $s$ in the encrypted text, are **not independent**. Since $\sigma$ represents a one-to-one mapping between symbols in the encrypted text and the original text, each symbol in the encrypted text must map to a unique symbol in the decrypted text. This constraint introduces dependencies among the mappings, as no two symbols can map to the same decrypted symbol.

**Joint Probability of $e_1 e_2 \cdots e_n$ Given $\sigma$**

Let $e_1 e_2 \cdots e_n$ be the encrypted text, where each $e_i$ represents an encrypted symbol. Let $s_1 s_2 \cdots s_n$ denote the corresponding sequence of symbols in the decrypted text, such that $s_i = \sigma^{-1}(e_i)$ for each $i$.

The probability of the sequence $s_1 s_2 \cdots s_n$ can be written as:

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

Given the permutation $\sigma$, the probability of observing the encrypted text $e_1 e_2 \cdots e_n$ is the same as the probability of observing the decrypted sequence $s_1 s_2 \cdots s_n$. Thus, we can write:

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

Substituting $s_i = \sigma^{-1}(e_i)$, we get:

$$p(e_1 e_2 \cdots e_n \mid \sigma) = p(\sigma^{-1}(e_1)) \prod_{i=2}^{n} p(\sigma^{-1}(e_i) \mid \sigma^{-1}(e_{i-1}))$$

where $p(\sigma^{-1}(e_1))$ represents the stationary probability of the first symbol in the decrypted sequence and $p(\sigma^{-1}(e_i) \mid \sigma^{-1}(e_{i-1}))$ represents the transition probability.

Therefore, the joint probability of the encrypted text $e_1 e_2 \cdots e_n$ given the permutation $\sigma$ is:

$$p(e_1 e_2 \cdots e_n \mid \sigma) = p(\sigma^{-1}(e_1)) \prod_{i=2}^{n} p(\sigma^{-1}(e_i) \mid \sigma^{-1}(e_{i-1}))$$

(c) **Proposal Probability $S(\sigma \to \sigma')$**

In the Metropolis-Hastings (MH) chain, the proposal involves choosing two symbols $s$ and $s'$ uniformly at random and swapping their corresponding mappings $\sigma(s)$ and $\sigma(s')$ in the permutation $\sigma$. This results in a new permutation $\sigma'$.

Since we are selecting any two symbols from a total of $N$ symbols, the probability of proposing a swap between any specific pair $(s, s')$ is uniform and independent of the current permutation $\sigma$. The number of ways to choose two symbols from $N$ is $\binom{N}{2} = \frac{N(N-1)}{2}$. Therefore, the proposal probability is:

$$S(\sigma \to \sigma') = \frac{1}{\binom{N}{2}} = \frac{2}{N(N-1)}$$

This probability is constant and does not depend on the specific permutations $\sigma$ and $\sigma'$.

**Metropolis-Hastings Acceptance Probability**

The acceptance probability in the Metropolis-Hastings algorithm is given by:

$$A(\sigma \to \sigma') = \min\left(1, \frac{p(\sigma' \mid e) S(\sigma' \to \sigma)}{p(\sigma \mid e) S(\sigma \to \sigma')}\right)$$

Since the proposal distribution $S(\sigma \to \sigma')$ is symmetric (i.e., $S(\sigma \to \sigma') = S(\sigma' \to \sigma)$), we can cancel these terms, simplifying the acceptance probability to:

$$A(\sigma \to \sigma') = \min\left(1, \frac{p(\sigma' \mid e)}{p(\sigma \mid e)}\right)$$

Assuming a uniform prior over permutations, we have $p(\sigma \mid e) \propto p(e \mid \sigma)$ by Bayes' theorem. The acceptance probability becomes:

$$A(\sigma \to \sigma') = \min\left(1, \frac{p(e \mid \sigma')}{p(e \mid \sigma)}\right)$$

where the likelihood $p(e \mid \sigma)$ of the encrypted text under permutation $\sigma$ is given by:

$$p(e \mid \sigma) = p(s_1) \prod_{i=2}^{n} p(s_i \mid s_{i-1})$$

which we have computed from (b).

(d) To decrypt the provided encrypted text by sampling permutations of the symbols, I implemented the M-H sampler. The sampling process involves initial random permutation, proposing new permutations by randomly swapping two symbols and deciding whether to accept these proposals based on their likelihoods. I ran the sampler multiple times from 10000 to 30000 iterations, and on average the chain converged to a coherent message after round 25000 iterations. (code in Appendix 5d) Here are the first 60 decrypted symbols every 100 iterations:

```
Iteration 0: Decoded message: [:p1lpl]x:r?!pw:np1]!?pmx;:?!w.;?pl?w!gp1lp/ws-?!prwm?p1?pg]
Iteration 100: Decoded message: 1: jl le6:fd! k:n je!d u6;:d!k.;d ldk!2 jl pks-d! fkud jd 2e
Iteration 200: Decoded message: c: jf fe6:odu k:n jeud t6;:dukl;d fdku. jf pks-du oktd jd .e
Iteration 300: Decoded message: c: j6 6ef:odu k:n jeud tf;:dukl;d 6dku. j6 pks-du oktd jd .e
Iteration 400: Decoded message: t: j= =ef:odu k:n jeud cfh:dukihd =dkur j= pks-du okcd jd re
Iteration 500: Decoded message: t. j= =ef.cdu k.n jeud ofh.dukihd =dkur j= pks-du ckod jd re
Iteration 600: Decoded message: ty (= =efycdu kyn (eud ofhydukihd =dkur (= pks-du ckod (d re
Iteration 700: Decoded message: ay h= =mfycdu kyn hmud ofeydukied =dkur h= pks-du ckod hd rm
Iteration 800: Decoded message: ay h= =mfygdu kyn hmud ifeydukbed =dkur h= pkswdu gkid hd rm
Iteration 900: Decoded message: ay w= =mfygdu kyn wmud ifeydukbed =dkur w= pkshdu gkid wd rm
Iteration 1000: Decoded message: ay o= =mfygeu kyi omue nfdyeukbde =ekur o= pksheu gkne oe rm
Iteration 1100: Decoded message: ay o= =mfygew kyi omwe nfdyewkbde =ekwr o= pkshew gkne oe rm
Iteration 1200: Decoded message: ay ov vifygew kym oiwe nfdyewkcde vekwr ov pkshew gkne oe ri
Iteration 1300: Decoded message: ay ov vifyger kym oire nfdyerkcde vekrw ov pksher gkne oe wi
Iteration 1400: Decoded message: ay ou uifyger kym oire nfdyerkcde uekrw ou pksher gkne oe wi
Iteration 1500: Decoded message: ay ou uifyger dym oire nfkyerdbke uedrw ou pdsher gdne oe wi
Iteration 1600: Decoded message: am ou uifmger dmy oire nfkmerdbke uedrw ou pdsher gdne oe wi
Iteration 1700: Decoded message: am ou uinmfer dmy oire vnkmerdbke uedrw ou pdsher fdve oe wi
Iteration 1800: Decoded message: am ou uinmfer dmy oire vnkmerdbke uedrw ou pdsher fdve oe wi
Iteration 1900: Decoded message: am od dinmfer umy oire vnkmerubke deurw od pusher fuve oe wi
Iteration 2000: Decoded message: am ok kinmfer umy oire vndmerubde keurw ok pusher fuve oe wi
Iteration 2100: Decoded message: am ok kinmfer umy oire vndmerubde keurw ok pusher fuve oe wi
Iteration 2200: Decoded message: am ok kinmfer umy oire vndmerubde keurw ok pusher fuve oe wi
Iteration 2300: Decoded message: am ok kinmfer umy oire vndmerubde keurw ok pusher fuve oe wi
Iteration 2400: Decoded message: am ok kinmfed umy oide vnrmedubre keudw ok pushed fuve oe wi
Iteration 2500: Decoded message: am ok kinmfed umy oide vnrmedubre keudw ok pushed fuve oe wi
Iteration 2600: Decoded message: am ok kinmfed umy oide vnrmedubre keudw ok pushed fuve oe wi
Iteration 2700: Decoded message: am nk kiomfel umy nile vormelubre keulw nk pushel fuve ne wi
Iteration 2800: Decoded message: am nk kiomfel umy nile vormelubre keulw nk pushel fuve ne wi
Iteration 2900: Decoded message: um nk kiomfel amy nile vormelabre kealw nk pashel fave ne wi
Iteration 3000: Decoded message: um nk kiomfel amy nile vormelabre keald nk pashel fave ne di
Iteration 3100: Decoded message: um nk kiomfel amy nile vormelabre keald nk pashel fave ne di
Iteration 3200: Decoded message: um ck kiomfel amy cile vormelabre keald ck pashel fave ce di
Iteration 3300: Decoded message: um ck kiomfel amy cile vormelabre keald ck pashel fave ce di
Iteration 3400: Decoded message: um tk kiomfel amy tile vormelabre keald tk pashel fave te di
Iteration 3500: Decoded message: ut mk kiotfel aty mile vortelabre keald mk pashel fave me di
Iteration 3600: Decoded message: ut mk kiotfel aty mile vortelabre keald mk pashel fave me di
Iteration 3700: Decoded message: ut mk kiotfel aty mile vortelabre keald mk nashel fave me di
Iteration 3800: Decoded message: ut mk kiotfel aty mile vortelabre keald mk nashel fave me di
Iteration 3900: Decoded message: ut mk kiotfel aty mile vortelabre keald mk nashel fave me di
Iteration 4000: Decoded message: ul mk kiolfet aly mite vorletabre keatd mk nashet fave me di
Iteration 4100: Decoded message: ur mk kiorfet ary mite volretable keatd mk nashet fave me di
Iteration 4200: Decoded message: ur mk kiorfet ary mite volretable keatd mk nashet fave me di
Iteration 4300: Decoded message: ur mk kiorfet ary mite volretable keatd mk nashet fave me di
Iteration 4400: Decoded message: ur my yiorfet ark mite volretable yeatd my nashet fave me di
Iteration 4500: Decoded message: ur my yiorget ark mite volretable yeatd my nashet gave me di
Iteration 4600: Decoded message: ur my yiorget ark mite volretable yeatd my nashet gave me di
Iteration 4700: Decoded message: ur my yiorget ark mite volretable yeatd my nashet gave me di
Iteration 4800: Decoded message: ur my yiorget ark mite volretable yeatd my nashet gave me di
Iteration 4900: Decoded message: ur my yiorget ark mite volretable yeatd my nashet gave me di
Iteration 5000: Decoded message: ur my yiorget ark mite volretable yeatd my nashet gave me di
Iteration 5100: Decoded message: ur my yiorget ark mite volretable yeatd my nashet gave me di
Iteration 5200: Decoded message: ur my yiorget ark mite volretable yeatd my nashet gave me di
Iteration 5300: Decoded message: ur my yiorget ark mite volretable yeatd my nashet gave me di
Iteration 5400: Decoded message: ur my yiorget ark mite volretable yeatd my nashet gave me di
Iteration 5500: Decoded message: un my yionget ank mite volnetable yeatd my rashet gave me di
Iteration 5600: Decoded message: un my yionget ank mite volnetable yeatd my rashet gave me di
Iteration 5700: Decoded message: un my yionget ank mite rolnetable yeatd my vashet gare me di
Iteration 5800: Decoded message: un my yionket ang mite rolnetable yeatd my vashet kare me di
Iteration 5900: Decoded message: un my yionket ang mite rolnetable yeatd my vashet kare me di
Iteration 6000: Decoded message: un my yionket ang mite rolnetable yeatd my vashet kare me di
Iteration 6100: Decoded message: un my yionket ang mite rolnetable yeatd my fashet kare me di
Iteration 6200: Decoded message: un my yionket ang mite rolnetable yeatd my fashet kare me di
Iteration 6300: Decoded message: un my yionket ang mite rolnetable yeatd my fashet kare me di
Iteration 6400: Decoded message: un my yionket ang mite rolnetable yeatd my fashet kare me di
Iteration 6500: Decoded message: un my yionket ang mite rolnetable yeatd my fashet kare me di
Iteration 6600: Decoded message: un my yionket ang mite rolnetable yeatd my fashet kare me di
Iteration 6700: Decoded message: un my yionket ang mite rolnetable yeatd my fashet kare me di
Iteration 6800: Decoded message: un my yionket ang mite rolnetable yeatd my fashet kare me di
Iteration 6900: Decoded message: un my yionket ang mite rolnetable yeatd my fashet kare me di
Iteration 7000: Decoded message: un my yionket ang mite rolnetable yeatd my fashet kare me di
Iteration 7100: Decoded message: un my yionket ang mite rolnetable yeatd my fashet kare me di
Iteration 7200: Decoded message: un my yionket ang mite rolnetable yeatd my fashet kare me di
Iteration 7300: Decoded message: un my yionket ang mite rolnetable yeatd my fashet kare me di
Iteration 7400: Decoded message: un my yionket ang mite rolnetable yeatd my fashet kare me di
Iteration 7500: Decoded message: un my yionket ang mite rolnetable yeatd my fashet kare me di
Iteration 7600: Decoded message: un my yionket ang mite rolnetable yeatd my fashet kare me di
Iteration 7700: Decoded message: un my yionket ang mite rolnetable yeatd my fashet kare me di
Iteration 7800: Decoded message: un my yionket ang mite rolnetable yeatd my fashet kare me di
Iteration 7900: Decoded message: un my yionket ang mite rolnetable yeatd my fashet kare me di
Iteration 8000: Decoded message: un my yionket ang mite rolnetable yeatd my fashet kare me di
Iteration 8100: Decoded message: un my yionket ang mite rolnetable yeatd my fashet kare me di
Iteration 8200: Decoded message: un my yionket ang mite rolnetable yeatd my fashet kare me di
Iteration 8300: Decoded message: un my yionket ang mite rolnetable yeatd my fashet kare me di
Iteration 8400: Decoded message: un my yionket ang mite rolnetable yeatd my fashet kare me di
Iteration 8500: Decoded message: un my yionket ang mite rolnetable yeatd my fashet kare me di
Iteration 8600: Decoded message: un my yionket ang mite rolnetable yeatd my fashet kare me di
Iteration 8700: Decoded message: un my yionket ang mite rolnetable yeatd my fashet kare me di
Iteration 8800: Decoded message: un my yionket ang mite rolnetable yeatd my fashet kare me di
Iteration 8900: Decoded message: un my yionket ang mite rolnetable yeatd my fashet kare me di
Iteration 9000: Decoded message: un my yionket ang mite rolnetable yeatd my fashet kare me di
Iteration 9100: Decoded message: un my yionket ang mite rolnetable yeatd my fashet kare me di
Iteration 9200: Decoded message: un my yionket and mite rolnetable yeatg my fashet kare me gi
Iteration 9300: Decoded message: un my yionket and mite rolnetable yeatg my fashet kare me gi
Iteration 9400: Decoded message: un my yionket and mite rolnetable yeatg my fashet kare me gi
Iteration 9500: Decoded message: un my yionket and mite rolnetable yeatg my fashet kare me gi
Iteration 9600: Decoded message: un my yionket and mite rolnetable yeatg my fashet kare me gi
Iteration 9700: Decoded message: un my yionket and mite rolnetable yeatg my fashet kare me gi
Iteration 9800: Decoded message: un my yionket and mite rolnetable yeatg my fashet kare me gi
Iteration 9900: Decoded message: un my yionket and mite rolnetable yeatg my fashet kare me gi
Iteration 10000: Decoded message: un my yionket and mite rolnetable yeatg my fashet kare me gi
Iteration 10100: Decoded message: un my yionket and mite rolnetable yeatg my fashet kare me gi
Iteration 10200: Decoded message: un my yionket and mite rolnetable yeatg my fashet kare me gi
Iteration 10300: Decoded message: un my yionket and mite rolnetable yeatg my fashet kare me gi
Iteration 10400: Decoded message: un my yionket and mite rolnetable yeatg my fashet kare me gi
Iteration 10500: Decoded message: un my yionket and mite rolnetable yeatg my fashet kare me gi
Iteration 10600: Decoded message: un my yionket and mite rolnetable yeatg my fashet kare me gi
Iteration 10700: Decoded message: un my yionket and mite rolnetable yeatg my fashet kare me gi
Iteration 10800: Decoded message: un my yionker and mire tolnerable yearg my fasher kate me gi
Iteration 10900: Decoded message: un my yionker and mire tolnerable yearg my fasher kate me gi
Iteration 11000: Decoded message: un my yionker and mire tolnerable yearg my fasher kate me gi
Iteration 11100: Decoded message: un my yionker and mire tolnerable yearg my fasher kate me gi
Iteration 11200: Decoded message: un my yionker and mire tolnerable yearg my fasher kate me gi
Iteration 11300: Decoded message: un my yionker and mire tolnerable yearg my fasher kate me gi
Iteration 11400: Decoded message: un my yionker and mire tolnerable yearg my fasher kate me gi
Iteration 11500: Decoded message: un my yionker and mire tolnerable yearg my fasher kate me gi
Iteration 11600: Decoded message: un my yionker and mire tolnerable yearg my fasher kate me gi
Iteration 11700: Decoded message: un my yionker and mire tolnerable yearg my fasher kate me gi
Iteration 11800: Decoded message: un my yionter and mire kolnerable yearg my fasher take me gi
Iteration 11900: Decoded message: un my yionter and mire kolnerable yearg my fasher take me gi
Iteration 12000: Decoded message: un my yionter and mire kolnerable yearg my fasher take me gi
Iteration 12100: Decoded message: un my yionter and mire kolnerable yearg my fasher take me gi
Iteration 12200: Decoded message: un my yionter and mire kolnerable yearg my fasher take me gi
Iteration 12300: Decoded message: un my yionter and mire volnerable yearg my fasher tave me gi
Iteration 12400: Decoded message: un my yionter and mire volnerable yearg my fasher tave me gi
Iteration 12500: Decoded message: un my yionter and mire volnerable yearg my fasher tave me gi
Iteration 12600: Decoded message: un my yionter and mire volnerable yearg my fasher tave me gi
Iteration 12700: Decoded message: un my yionter and mire volnerable yearg my fasher tave me gi
Iteration 12800: Decoded message: un my yionser and mire volnerable yearg my father save me gi
Iteration 12900: Decoded message: un my yionser and mire volnerable yearg my father save me gi
Iteration 13000: Decoded message: un my yionser and mire volnerable yearg my father save me gi
Iteration 13100: Decoded message: un my yionser and mire volnerable yearg my father save me gi
Iteration 13200: Decoded message: un my yionser and mire volnerable yearg my father save me gi
Iteration 13300: Decoded message: un my yionser and mire volnerable yearg my father save me gi
Iteration 13400: Decoded message: un my yionser and mire volnerable yearg my father save me gi
Iteration 13500: Decoded message: un my yionser and mire volnerable yearg my father save me gi
Iteration 13600: Decoded message: un my yionser and mire volnerable yearg my father save me gi
Iteration 13700: Decoded message: un my yionser and mire volnerable yearg my father save me gi
Iteration 13800: Decoded message: un my yionser and mire volnerable yearg my father save me gi
Iteration 13900: Decoded message: un my yionser and mire volnerable yearg my father save me gi
Iteration 14000: Decoded message: un my yionser and mire volnerable yearg my father save me gi
Iteration 14100: Decoded message: un my yionser and mire volnerable yearg my father save me gi
Iteration 14200: Decoded message: un my yionser and mire volnerable yearg my father save me gi
Iteration 14300: Decoded message: un my yionser and mire volnerable yearg my father save me gi
Iteration 14400: Decoded message: un my yionser and mire volnerable yearg my father save me gi
Iteration 14500: Decoded message: un my yionser and mire volnerable yearg my father save me gi
Iteration 14600: Decoded message: un my yionser and mire volnerable yearg my father save me gi
Iteration 14700: Decoded message: un my yionser and mire volnerable yearg my father save me gi
Iteration 14800: Decoded message: un my yionser and mire volnerable yearg my father save me gi
Iteration 14900: Decoded message: un my yionser and mire volnerable yearg my father save me gi
Iteration 15000: Decoded message: un my yoinser and more vilnerable yearg my father save me go
Iteration 15100: Decoded message: un my yoinser and more vilnerable yearg my father save me go
Iteration 15200: Decoded message: un my yoinser and more vilnerable yearg my father save me go
Iteration 15300: Decoded message: un my yoinser and more vilnerable yearg my father save me go
Iteration 15400: Decoded message: un my yoinser and more vilnerable yearg my father save me go
Iteration 15500: Decoded message: un my yoinser and more vilnerable yearg my father save me go
Iteration 15600: Decoded message: un my yoinser and more vilnerable yearg my father save me go
Iteration 15700: Decoded message: un my yoinser and more vilnerable yearg my father save me go
Iteration 15800: Decoded message: un my yoinser and more vilnerable yearg my father save me go
Iteration 15900: Decoded message: un my yoinser and more vilnerable yearg my father save me go
Iteration 16000: Decoded message: un my yoinser and more vilnerable yearg my father save me go
Iteration 16100: Decoded message: un my yoinser and more vilnerable yearg my father save me go
Iteration 16200: Decoded message: un my yoinser and more vilnerable yearg my father save me go
Iteration 16300: Decoded message: un my yoinser and more vilnerable yearg my father save me go
Iteration 16400: Decoded message: un my yoinser and more vilnerable yearg my father save me go
Iteration 16500: Decoded message: un my yoinser and more vilnerable yearg my father save me go
Iteration 16600: Decoded message: un my yoinser and more vilnerable yearg my father save me go
Iteration 16700: Decoded message: un my yoinser and more vilnerable yearg my father save me go
Iteration 16800: Decoded message: un my yoinser and more vilnerable yearg my father save me go
Iteration 16900: Decoded message: un my yoinser and more vilnerable yearg my father save me go
Iteration 17000: Decoded message: un my yoinser and more vilnerable yearg my father save me go
Iteration 17100: Decoded message: un my yoinser and more vilnerable yearg my father save me go
Iteration 17200: Decoded message: un my yoinser and more vilnerable yearg my father save me go
Iteration 17300: Decoded message: un my yoinser and more vilnerable yearg my father save me go
Iteration 17400: Decoded message: un my yoinser and more vilnerable yearg my father save me go
Iteration 17500: Decoded message: un my yoinser and more vilnerable yearg my father save me go
Iteration 17600: Decoded message: un my yoinser and more vilnerable yearg my father save me go
Iteration 17700: Decoded message: un my yoinser and more vilnerable yearg my father save me go
Iteration 17800: Decoded message: un my yoinser and more vilnerable yearg my father save me go
Iteration 17900: Decoded message: un my yoinser and more vilnerable yearg my father save me go
Iteration 18000: Decoded message: un my yoinser and more vilnerable yearg my father save me go
Iteration 18100: Decoded message: un my yoinser and more vilnerable yearg my father save me go
Iteration 18200: Decoded message: un my yoinser and more vilnerable yearg my father save me go
Iteration 18300: Decoded message: un my yoinser and more vilnerable yearg my father save me go
Iteration 18400: Decoded message: un my yoinser and more vilnerable yearg my father save me go
Iteration 18500: Decoded message: un my yoinser and more vilnerable yearg my father save me go
Iteration 18600: Decoded message: un my yoinser and more vilnerable yearg my father save me go
Iteration 18700: Decoded message: un my yoinser and more vilnerable yearg my father save me go
Iteration 18800: Decoded message: un my yoinser and more vilnerable yearg my father save me go
Iteration 18900: Decoded message: un my yoinser and more vilnerable yearg my father save me go
Iteration 19000: Decoded message: un my yoinser and more vilnerable yearg my father save me go
Iteration 19100: Decoded message: un my yoinser and more vilnerable yearg my father save me go
Iteration 19200: Decoded message: un my yoinser and more vilnerable yearg my father save me go
Iteration 19300: Decoded message: un my yoinser and more vilnerable yearg my father save me go
Iteration 19400: Decoded message: un my yoinger and more vilnerable years my father gave me so
Iteration 19500: Decoded message: un my yoinger and more vilnerable years my father gave me so
Iteration 19600: Decoded message: un my yoinger and more vilnerable years my father gave me so
Iteration 19700: Decoded message: un my yoinger and more vilnerable years my father gave me so
Iteration 19800: Decoded message: un my yoinger and more vilnerable years my father gave me so
Iteration 19900: Decoded message: un my yoinger and more vilnerable years my father gave me so
Iteration 20000: Decoded message: un my yoinger and more vilnerable years my father gave me so
Iteration 20100: Decoded message: un my yoinger and more vilnerable years my father gave me so
Iteration 20200: Decoded message: un my yoinger and more vilnerable years my father gave me so
Iteration 20300: Decoded message: un my yoinger and more vilnerable years my father gave me so
Iteration 20400: Decoded message: un my yoinger and more vilnerable years my father gave me so
Iteration 20500: Decoded message: un my yoinger and more vilnerable years my father gave me so
Iteration 20600: Decoded message: un my yoinger and more vilnerable years my father gave me so
Iteration 20700: Decoded message: un my yoinger and more vilnerable years my father gave me so
Iteration 20800: Decoded message: un my yoinger and more vilnerable years my father gave me so
Iteration 20900: Decoded message: un my yoinger and more vilnerable years my father gave me so
Iteration 21000: Decoded message: un my yoinger and more vilnerable years my father gave me so
Iteration 21100: Decoded message: un my yoinger and more vilnerable years my father gave me so
Iteration 21200: Decoded message: un my yoinger and more vilnerable years my father gave me so
Iteration 21300: Decoded message: un my yoinger and more vilnerable years my father gave me so
Iteration 21400: Decoded message: un my yoinger and more vilnerable years my father gave me so
Iteration 21500: Decoded message: un my yoinger and more vilnerable years my father gave me so
Iteration 21600: Decoded message: un my yoinger and more vilnerable years my father gave me so
Iteration 21700: Decoded message: un my yoinger and more vilnerable years my father gave me so
Iteration 21800: Decoded message: un my yoinger and more vilnerable years my father gave me so
Iteration 21900: Decoded message: un my yoinger and more vilnerable years my father gave me so
Iteration 22000: Decoded message: un my yoinger and more vilnerable years my father gave me so
Iteration 22100: Decoded message: un my yoinger and more vilnerable years my father gave me so
Iteration 22200: Decoded message: un my yoinger and more vilnerable years my father gave me so
Iteration 22300: Decoded message: un my yoinger and more vilnerable years my father gave me so
Iteration 22400: Decoded message: un my yoinger and more vilnerable years my father gave me so
Iteration 22500: Decoded message: un my yoinger and more vilnerable years my father gave me so
Iteration 22600: Decoded message: un my yoinger and more vilnerable years my father gave me so
Iteration 22700: Decoded message: un my yoinger and more vilnerable years my father gave me so
Iteration 22800: Decoded message: un my yoinger and more vilnerable years my father gave me so
Iteration 22900: Decoded message: un my yoinger and more vilnerable years my father gave me so
Iteration 23000: Decoded message: un my yoinger and more vilnerable years my father gave me so
Iteration 23100: Decoded message: un my yoinger and more vilnerable years my father gave me so
Iteration 23200: Decoded message: un my yoinger and more vilnerable years my father gave me so
Iteration 23300: Decoded message: un my yoinger and more vilnerable years my father gave me so
Iteration 23400: Decoded message: un my yoinger and more vilnerable years my father gave me so
Iteration 23500: Decoded message: un my yoinger and more vilnerable years my father gave me so
Iteration 23600: Decoded message: un my yoinger and more vilnerable years my father gave me so
Iteration 23700: Decoded message: un my yoinger and more vilnerable years my father gave me so
Iteration 23800: Decoded message: un my yoinger and more vilnerable years my father gave me so
Iteration 23900: Decoded message: un my yoinger and more vilnerable years my father gave me so
Iteration 24000: Decoded message: un my yoinger and more vilnerable years my father gave me so
Iteration 24100: Decoded message: un my yoinger and more vilnerable years my father gave me so
Iteration 24200: Decoded message: un my yoinger and more vilnerable years my father gave me so
Iteration 24300: Decoded message: un my yoinger and more vilnerable years my father gave me so
Iteration 24400: Decoded message: un my yoinger and more vilnerable years my father gave me so
Iteration 24500: Decoded message: un my yoinger and more vilnerable years my father gave me so
Iteration 24600: Decoded message: un my yoinger and more vilnerable years my father gave me so
Iteration 24700: Decoded message: un my yoinger and more vilnerable years my father gave me so
Iteration 24800: Decoded message: in my younger and more vulnerable years my father gave me so
Iteration 24900: Decoded message: in my younger and more vulnerable years my father gave me so
Iteration 25000: Decoded message: in my younger and more vulnerable years my father gave me so
```

Decrypted Text: 'in my younger and more vulnerable years my father gave me some advice that ixve been turning over in my mind ever since. *whenever you feel like criticizing any one,* he told me, *just remember that all the people in this world havenxt had the advantages that youxve had.* he didnxt say any more but wexve always been unusually communicative in a reserved way, and i understood that he meant a great deal more than that. in consequence ixm inclined to reserve all judgments, a habit that has opened up many curious natures to me and also made me the victim of not a few veteran bores. the abnormal mind is quick to detect and attach itself to this quality when it appears in a normal person, and so it came about that in college i was unjustly accused of being a politician, because i was privy to the secret griefs of wild, unknown men. most of the confidences were unsought–frequently i have feigned sleep, preoccupation, or a hostile levity when i realized by some unmistakable sign that an intimate revelation was quivering on the horizon–for the intimate revelations of young men or at least the terms in which they express them are usually plagiaristic and marred by obvious suppressions. reserving judgments is a matter of infinite hope. i am still a little afraid of missing something if i forget that, as my father snobbishly suggested, and i snobbishly repeat a sense of the fundamental decencies is parcelled out unequally at birth.'

(e) A Markov chain is considered ergodic if it is both irreducible and aperiodic. When some transition probabilities $\psi(\alpha, \beta)$ are zero, certain pairs of states $(\alpha, \beta)$ lack a direct transition between them at a single step. This lack of direct transitions could impact irreducibility:

- If these zero entries prevent access to certain states from others, then the chain would not be irreducible and, consequently, would not be ergodic.
- However, if these zero entries only indicate the absence of direct transitions, while still allowing access to all states indirectly (via intermediate states), the chain could remain irreducible and thus ergodic.

If the chain remains ergodic despite some zero entries, this means:

- For any two states, there exists a sequence of transitions that allows the chain to move from one state to another over multiple steps.
- Consequently, the chain can still explore the entire state space, maintaining ergodicity. In this case, no modification to the transition matrix is necessary.

If ergodicity is lost due to zero entries (i.e., certain states become inaccessible), it can be restored by regularizing the transition matrix. This involves adding a small probability $\epsilon > 0$ to each zero entry, ensuring that all states are reachable:

- By adding a small non-zero probability for each transition, the chain becomes irreducible, as every state can be reached from any other state, either directly or indirectly.
- This adjustment also ensures that, assuming aperiodicity, the chain is ergodic.

(f) In analyzing the proposed decoding approach, we have several considerations:

- **Symbol Probabilities Alone:** Utilizing symbol probabilities alone would likely be NOT sufficient, as it ignores the context provided by neighboring symbols, leading to potential ambiguities in the decoded message. In natural language, the meaning of a symbol often depends on the preceding/following symbols, especially those with complex syntax or grammar.
- **Second Order Markov Chain Challenges:** While a second order Markov chain offers more contextual information by considering the previous two symbols, it introduces problems such as:
  - *Data Sparsity:* The increase in context complexity may lead to insufficient data to accurately estimate the probabilities for every possible sequence of two preceding symbols.
  - *Computational Complexity:* The size of the state space grows with the order, complicating computation and storage of transition probabilities.
  - *Overfitting:* With more parameters to estimate, there's a risk of overfitting to the training data, which might not generalize well to unseen data.
- **Mapping Two Symbols to the Same Encrypted Value:** If the encryption scheme permits two different symbols to be encoded as the same value, this leads to ambiguity during decoding. The algorithm may struggle to uniquely determine which of the two original symbols corresponds to the encrypted value, complicating the decoding process and potentially resulting in incorrect interpretations.
- **Applicability to Chinese Text:** Implementing this approach for Chinese text, which has a character set exceeding 10,000 symbols, presents additional challenges:
  - The increased number of symbols exacerbates data sparsity and computational complexity.
  - The absence of spaces between characters complicates the analysis of sequences, making it difficult for the Markov model to effectively capture character dependencies, especially without adequate training data.

## 6 Question 6: Implementing Gibbs Sampling for LDA.

## 7 Question 7: Optimization.

(a) Given the function:

$$f(x,y) = x + 2y$$

subject to the constraint:

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

Use the method of Lagrange multipliers. Define the Lagrange multiplier $\lambda$ and set up the equation:

$$\nabla f = \lambda \nabla g$$

where:

$$\nabla f = \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}\right) = (1,2)$$

$$\nabla g = \left(\frac{\partial g}{\partial x}, \frac{\partial g}{\partial y}\right) = (y, 2y + x)$$

Equate the gradients:

$$(1,2) = \lambda(y, 2y + x)$$

This gives us the system of equations:

$$1 = \lambda y$$

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

From the first equation, solve for $\lambda$:

$$\lambda = \frac{1}{y} \quad \text{(assuming } y \neq 0\text{)}$$

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

$$2y = 2y + x \implies x = 0$$

Substitute $x = 0$ into the constraint:

$$y^2 + 0 \cdot y - 1 = 0 \implies y^2 = 1 \implies y = \pm 1$$

Thus, the critical points are:

$$(x, y) = (0, 1) \quad \text{and} \quad (0, -1)$$

Evaluate $f(x, y)$ at these points:

$$f(0, 1) = 0 + 2 \cdot 1 = 2$$
$$f(0, -1) = 0 + 2 \cdot (-1) = -2$$

To conclude, the local extrema are:

$$(0, 1) \text{ with } f(0, 1) = 2 \quad \text{and} \quad (0, -1) \text{ with } f(0, -1) = -2$$

(b) To compute $\ln(a)$ for $a \in \mathbb{R}^+$ using Newton's method:

(i) Need to find $x$ such that:

$$x = \ln(a) \implies a = e^x$$
$$f(x, a) = e^x - a$$

We are looking for the root of $f(x, a) = 0$, which corresponds to $x = \ln(a)$.

(ii) Newton's method is given by:

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

Compute the derivative:

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

## 8 Question 8: Eigenvalues as solutions of an optimization problem.

(a) To show that $\sup_{x \in \mathbb{R}^n} R_A(x)$ is attained, we proceed as follows:

Given the Rayleigh quotient is defined as

$$R_A(x) = \frac{x^\top A x}{x^\top x} = \frac{q_A(x)}{\|x\|^2},$$

where $q_A(x) = x^\top A x$.

First, we establish that $R_A(x)$ is a continuous function. This is true because both the numerator $x^\top A x$ and the denominator $x^\top x$ are continuous functions, and $x^\top x \neq 0$ for all $x \neq 0$.

The set $\mathbb{R}^n$ is not compact, so we need to transform the problem to a compact set. Consider the unit sphere

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

The set $S$ is compact because it is both closed and bounded.

Next, we show the equivalence of the supremum over $\mathbb{R}^n$ and the supremum over $S$. For any nonzero vector $x \in \mathbb{R}^n$, we can construct the normalized vector $\tilde{x} = \frac{x}{\|x\|}$, which lies on $S$. We have

$$R_A\left(\frac{x}{\|x\|}\right) = \frac{\left(\frac{x}{\|x\|}\right)^\top A \left(\frac{x}{\|x\|}\right)}{\left(\frac{x}{\|x\|}\right)^\top \left(\frac{x}{\|x\|}\right)} = R_A(x).$$

This shows that the value of $R_A(x)$ remains unchanged when $x$ is normalized to lie on the unit sphere $S$. So, we have

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

By the Extreme Value Theorem, which states that a continuous function on a compact set attains its maximum and minimum values, $R_A(x)$ must attain its supremum on the compact set $S$. Since $R_A(x)$ is continuous on $S$, we conclude that $R_A(x)$ attains its maximum on $S$.

We have shown that $\sup_{x \in \mathbb{R}^n} R_A(x)$ is attained, as required.

(b) Given that the eigenvalues $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_n$ of the symmetric matrix $A$ are enumerated in decreasing order, with corresponding eigenvectors $\xi_1, \ldots, \xi_n$ forming an orthonormal basis (ONB) of $\mathbb{R}^n$. Any vector $x \in \mathbb{R}^n$ can be represented as

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

The Rayleigh quotient is defined as

$$R_A(x) = \frac{x^\top Ax}{x^\top x}.$$

To expand $x^\top Ax$, we use the eigen-decomposition of $A$. Since $A\xi_i = \lambda_i \xi_i$ for each $i$, we have:

$$x^\top Ax = \left(\sum_{i=1}^{n} (\xi_i^\top x)\xi_i\right)^\top A \left(\sum_{j=1}^{n} (\xi_j^\top x)\xi_j\right).$$

Using the linearity of $A$ and the orthonormality of $\xi_i$, we get:

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

Similarly,

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

Hence, the Rayleigh quotient becomes

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

The expression $R_A(x)$ is a weighted average of the eigenvalues $\lambda_i$, where the weights $(\xi_i^\top x)^2$ are non-negative and sum to 1:

$$\sum_{i=1}^{n} \frac{(\xi_i^\top x)^2}{\sum_{j=1}^{n} (\xi_j^\top x)^2} = 1.$$

Since $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_n$, the largest value of this weighted average is $\lambda_1$. Therefore, we have

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

(c) Given that there are several linearly independent eigenvectors $\xi_1, \ldots, \xi_k$ corresponding to the eigenvalue $\lambda_1$, where $k \leq n$. The span of these eigenvectors is

$$\operatorname{span}\{\xi_1, \ldots, \xi_k\}.$$

We need to show: if $x \in \mathbb{R}^n$ is not contained in $\operatorname{span}\{\xi_1, \ldots, \xi_k\}$, then $R_A(x) < \lambda_1$.

Since the eigenvectors $\{\xi_1, \ldots, \xi_n\}$ form an orthonormal basis of $\mathbb{R}^n$, we can express any vector $x \in \mathbb{R}^n$ as

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

If $x \notin \operatorname{span}\{\xi_1, \ldots, \xi_k\}$, then there exists at least one coefficient $(\xi_i^\top x) \neq 0$ for some $i > k$.

The Rayleigh quotient is

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

Since $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_n$, and $x$ has a nonzero component $(\xi_i^\top x)$ for some $i > k$, the terms $\lambda_i (\xi_i^\top x)^2$ for $i > k$ contribute to $x^\top Ax$. These terms are associated with eigenvalues $\lambda_i$ that are strictly less than $\lambda_1$.

So, the numerator $\sum_{i=1}^{n} \lambda_i (\xi_i^\top x)^2$ is a weighted average where not all weights are associated with $\lambda_1$. Consequently, the Rayleigh quotient $R_A(x)$ is strictly less than $\lambda_1$, as there are contributions from eigenvalues $\lambda_i < \lambda_1$.

We have shown that if $x \notin \operatorname{span}\{\xi_1, \ldots, \xi_k\}$, then $R_A(x) < \lambda_1$.

# Appendix

## 1(d) - Code

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

```python
# Load the data set directly from binarydigits.txt
Y = np.loadtxt('binarydigits.txt')
N, D = Y.shape

# Compute the ML parameters
ml_params = np.mean(Y, axis=0)

# Reshape the ML parameters into an 8x8 image
ml_image = ml_params.reshape(8, 8)

# Display the learned parameter vector as an image
plt.imshow(ml_image, cmap='gray')
plt.title("ML Parameters as 8x8 Image")
plt.colorbar()
plt.show()
```

## 1(e) - Code

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

```python
# Load the data set directly from binarydigits.txt
Y = np.loadtxt('binarydigits.txt')
N, D = Y.shape

# Maximum A Posteriori (MAP) estimation with alpha = beta = 3
alpha = 3
beta = 3

# Compute the MAP parameters:
p_MAP = (np.sum(Y, axis=0) + alpha - 1) / (N + alpha + beta - 2)

# Reshape the parameter vector into an 8x8 image
p_MAP_image = p_MAP.reshape((8, 8))

# Visualize the learned parameter vector as an 8x8 image
plt.imshow(p_MAP_image, cmap='gray')
plt.title('MAP Parameters (alpha=3, beta=3)')
plt.colorbar()
plt.show()
```

## 2 - Code

```python
import numpy as np
import scipy.special as sp
```

```python
# Load the binary data
Y = np.loadtxt('binarydigits.txt')
N, D = Y.shape

# Calculate the relative log-probability of Model (a)
P_X_given_Ma = -N * D * np.log(2)

# Calculate the relative log-probability of Model (b)
alpha_b = 1 + np.sum(Y)
beta_b = 1 + N * D - np.sum(Y)
```

```python
P_X_given_Mb = sp.betaln(alpha_b, beta_b)

# Calculate the relative log-probability of Model (c)
x_sum = np.sum(Y, axis=0)
alpha_c = 1 + x_sum
beta_c = 1 + N - x_sum
P_X_given_Mc = np.sum(sp.betaln(alpha_c, beta_c))

# Calculate relative probabilities by exponentiating and scaling
relative_probs = [
    np.exp(4000 + P_X_given_Ma),
    np.exp(4000 + P_X_given_Mb),
    np.exp(4000 + P_X_given_Mc)
]

# Normalize to get posterior probabilities
total = sum(relative_probs)
normalized_probs = [p / total for p in relative_probs]
```

## 3(d) - Code

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

```python
    data = np.loadtxt('binarydigits.txt')
    tensor_data = torch.tensor(data).double()
    num_samples, num_features = tensor_data.shape
```

```python
# Functions for Initialization

# Create the initial Pi vector with equal probabilities for each Bernoulli component
def initialize_pi(num_components):
    """Generates a tensor of shape (num_components,) where each element is 1/num_components."""
    return torch.full((num_components,), 1 / num_components).double()

# Create the initial P matrix with values sampled uniformly from the range [0, 1]
def initialize_P(num_components, num_features):
    """Generates a matrix of shape (num_components, num_features) filled with random values in [0, 1]."""
    return torch.rand((num_components, num_features)).double()
```

```python
# Functions for Probability Calculation

# Calculate the log-probability of the observations given the model parameters
def calculate_prob_data_given_model(P, X):
    """Computes the probability of the observations based on the provided model parameters."""
    diag_X = torch.diag_embed(X)
    diag_I_X = torch.diag_embed(torch.ones(X.size()) - X)

    prob_tensor = ((P @ diag_X[0]) + (torch.ones(P.size()) - P) @ diag_I_X[0]).unsqueeze(0)

    for i in range(1, num_samples):
        A = ((P @ diag_X[i]) + (torch.ones(P.size()) - P) @ diag_I_X[i]).unsqueeze(0)
        prob_tensor = torch.cat((prob_tensor, A), 0)

    final_prob_matrix = torch.prod(prob_tensor, dim=2)
    return final_prob_matrix

# Calculate the joint probability of the observations and the Bernoulli components
def calculate_prob_data(P, X, pi):
    """Computes the joint probability of the data and the model parameters."""
    return calculate_prob_data_given_model(P, X) @ torch.diag(pi)

# Calculate the sum across components for normalization of responsibilities
def sum_across_components(P, X, pi):
    """Sums the probabilities across components for normalization purposes."""
    return torch.sum(calculate_prob_data(P, X, pi), dim=1)
```

```python
    # Functions for E-step Updates

     # Calculate responsibilities (E-step of the EM algorithm)
def compute_responsibility(P, X, pi):
    """Calculates the responsibilities for each component based on the current model parameters."""
    normalization_factors = torch.div(torch.ones(sum_across_components(P, X, pi).size()), sum_across_components(P,
    normalization_matrix = torch.diag(normalization_factors)
    return normalization_matrix @ calculate_prob_data(P, X, pi)

# Compute the log-likelihood of the data given the current model parameters
def compute_log_likelihood(P, X, pi):
    """Calculates the log-likelihood of the observations given the model parameters."""
    diag_X = torch.diag_embed(X).float()
    diag_I_X = torch.diag_embed(torch.ones(X.size()) - X).float()
    repeated_P = (P.unsqueeze(0)).repeat(num_samples, 1, 1).float()
    repeated_I_P = ((torch.ones(P.size()) - P).unsqueeze(0)).repeat(num_samples, 1, 1).float()
    prob_tensor = (torch.bmm(repeated_P, diag_X) + torch.bmm(repeated_I_P, diag_I_X)).double()
    prob_matrix = torch.prod(prob_tensor, dim=2)
    log_prob_per_image = torch.sum((prob_matrix @ torch.diag(pi)), dim=1)

    return torch.sum(torch.log(log_prob_per_image))
```

```python
# Functions for M-step Updates

# Update the Pi vector during the M-step
def update_pi_vector(R):
    """Calculates the new Pi vector as the average of responsibilities across all images."""
    return torch.sum(R, dim=0) / num_samples

# Update the P matrix during the M-step
def update_P_matrix(X, R):
    """Updates the P matrix based on the responsibilities."""
    numerator = torch.t(R) @ X
    normalization_matrix = torch.diag(1.0 / torch.sum(R, dim=0))
    return torch.t(torch.t(numerator) @ normalization_matrix)
```

```python
# Implementation of the EM Algorithm

# Execute the EM algorithm for a specified number of iterations
def run_EM_algorithm(num_components, X, max_iterations, tol=1e-6):
    """Runs the EM algorithm for the specified number of components and iterations."""
    num_features = X.shape[1]
    pi = initialize_pi(num_components)
    P = initialize_P(num_components, num_features)
    R = compute_responsibility(P, X, pi)

    log_likelihoods = []

    for iteration in range(max_iterations):
        R = compute_responsibility(P, X, pi)
        pi = update_pi_vector(R)
        P = update_P_matrix(X, R)
        likelihood = compute_log_likelihood(P, X, pi)
        log_likelihoods.append(likelihood.item())

        # Early stopping condition based on likelihood convergence
        if iteration > 0 and abs(log_likelihoods[-1] - log_likelihoods[-2]) < tol:
            break  # Terminate early if the change is smaller than the tolerance

    return log_likelihoods, pi, P, R
```

```python
# Running the EM Algorithm for Various K Values

K_options = [2, 3, 4, 7, 10]
for k in K_options:
    log_likelihoods, pi, P, R = run_EM_algorithm(k, tensor_data, 30)
    plt.plot(log_likelihoods, label=f'K={k}')

plt.xlabel('Iterations')
plt.ylabel('Log Likelihood')
plt.legend(loc='center left', bbox_to_anchor=(1, 0.5))
plt.savefig('log_likelihood_plot.png')
plt.show()
```

## 3(e) - Code

```
# Function to execute the EM algorithm multiple times with different initial conditions
def execute_multiple_EM(num_components, tensor_data, max_iterations, num_runs=5):
    pi_values_all_runs = []  # List to store the Pi values for each run
    P_values_all_runs = []    # List to store the P values for each run

    # Create a figure to plot all runs for each value of K in a consolidated format
    fig, axs = plt.subplots(num_runs, num_components, figsize=(num_components * 3, num_runs * 3))
    fig.suptitle(f'Learned Probability Images for K={num_components}', fontsize=16)

    for run in range(num_runs):
        # Execute the EM algorithm
        _, pi, P, _ = EM_algorithm(num_components, tensor_data, max_iterations)

        # Store the mixing proportions (pi) and probability matrices (P)
        pi_values_all_runs.append(pi.numpy())  # Convert to numpy for easier export if needed
        P_values_all_runs.append(P.numpy())     # Convert to numpy for easier export if needed

        # Print the mixing proportions for the current run
        print(f"Run {run + 1} - Mixing Proportions (pi):", pi)

        # Visualize the learned probability vectors as images for each cluster in the subplot grid
        for k in range(num_components):
            ax = axs[run, k] if num_runs > 1 else axs[k]  # Handle case where num_runs = 1
            ax.imshow(P[k].reshape(8, 8), cmap='gray')
            ax.set_title(f'Cluster {k + 1} (Run {run + 1})')
            ax.axis('off')  # Disable axis display

    # Adjust layout and save the plot for the current number of components
    plt.tight_layout(rect=[0, 0, 1, 0.96])
    plt.savefig(f'Learned_Probability_Images_K_{num_components}.png')
    plt.show()

    # Return all mixing proportions (pi) and probability matrices (P) for each run
    return pi_values_all_runs, P_values_all_runs
# Define the different values of K to test
K_options = [2, 3, 4, 7, 10]
max_iterations = 30
num_runs = 7\textbf
# Dictionary to store the Pi values for each K
pi_values_for_all_K = {}

# Execute the EM algorithm multiple times for each value of K and save results
for num_components in K_options:
    print(f"\nRunning EM algorithm for K={num_components} with {num_runs} different initializations...")
    pi_list, P_list = execute_multiple_EM(num_components, tensor_data, max_iterations, num_runs)
    pi_values_for_all_K[num_components] = pi_list  # Save the pi values for this K
# Display Pi values for all runs in a readable format
print("\nMixing Proportions (pi) for each K and each run:")
for num_components, pi_values in pi_values_for_all_K.items():
    print(f"\nK={num_components}:")
    for i, pi in enumerate(pi_values):
        print(f"  Run {i + 1}: {pi}")
```

## 3(f) - Code

```python
# Convert log-likelihood to bits
def log_likelihood_in_bits(log_likelihood):
    return log_likelihood / math.log(2)
```

```python
# Calculate naive encoding length in bits
def naive_encoding_length(X):
    N, D = X.shape
    return N * D
```

```python
# Compress the dataset using gzip
def gzip_compression_size(file_path):
    with open(file_path, 'rb') as f_in:
        compressed_file = file_path + '.gz'
        with gzip.open(compressed_file, 'wb') as f_out:
            f_out.writelines(f_in)
    compressed_size = os.path.getsize(compressed_file) * 8
    os.remove(compressed_file)
    return compressed_size
```

```python
naive_bits = naive_encoding_length(X.numpy())
compressed_bits = gzip_compression_size('binarydigits.txt')
```

```python
    for k in K_values:
    losses, _, _, _ = EM_algorithm(k, X, 30)
    final_log_likelihood = losses[-1]
    log_likelihood_bits = log_likelihood_in_bits(final_log_likelihood)

    print(f"K={k}:")
    print(f"  Log-likelihood (bits): {log_likelihood_bits:.2f}")
    print(f"  Naive encoding length (bits): {naive_bits}")
    print(f"  gzip compressed length (bits): {compressed_bits}")
    print()
```

## 3(g) - Code

```python
# Calculate the total cost of encoding
def parameter_cost(K, D, precision_levels=256):
    pi_cost = K * math.log2(precision_levels)  # Mixing coefficients
    P_cost = K * D * math.log2(precision_levels)  # Bernoulli parameters
    return pi_cost + P_cost

# Total encoding cost
def total_encoding_cost(K, log_likelihood_bits, D, precision_levels=256):
    param_cost = parameter_cost(K, D, precision_levels)
    return param_cost - log_likelihood_bits  # Negative log-likelihood given it's a 'cost'


for k in K_values:
    losses, _, _, _ = EM_algorithm(k, X, 30)
    final_log_likelihood = losses[-1]
    log_likelihood_bits = log_likelihood_in_bits(final_log_likelihood)
    param_cost = parameter_cost(k, D, 256)  # Assume 256 precision levels here
    total_cost = total_encoding_cost(k, log_likelihood_bits, D)

    print(f"K={k}:")
    print(f"  Parameter cost (bits): {param_cost:.2f}")
    print(f"  Total encoding cost (bits): {total_cost:.2f}")
    print()
```

## 4(a) - Code

```python
import numpy as np
import matplotlib.pyplot as plt
import torch
from ssm_kalman import run_ssm_kalman
```

```python
# Load X (observation data) from 'ssm_spins.txt'
X = np.loadtxt('ssm_spins.txt').T  # Transpose to match expected shape [d, t_max]

# Define initial values and matrices based on question's information
Y0 = np.zeros(4)  # Initial latent state, assuming a 4-dimensional state
Q0 = np.eye(4)    # Initial covariance (identity matrix for simplicity)

# Define the given A matrix
A = 0.99 * np.array([
    [np.cos(2 * np.pi / 180), -np.sin(2 * np.pi / 180), 0, 0],
    [np.sin(2 * np.pi / 180),  np.cos(2 * np.pi / 180), 0, 0],
    [0, 0, np.cos(2 * np.pi / 90), -np.sin(2 * np.pi / 90)],
    [0, 0, np.sin(2 * np.pi / 90),  np.cos(2 * np.pi / 90)]
])

# Define the C matrix
C = np.array([
    [1, 0, 1, 0],
    [0, 1, 0, 1],
    [1, 0, 0, 0],
    [0, 1, 0, 0],
    [0.5, 0.5, 0.5, 0.5]
])

# Define Q and R matrices
Q = np.eye(4) - A @ A.T  # Q = I - A * A^T
R = np.eye(5)            # R is identity matrix of shape [d, d]
```

```python
# Define log determinant function
def logdet(A):
    return 2 * torch.sum(torch.log(torch.diag(torch.cholesky(A))))
```

```python
# Run the Kalman filter and smoother to estimate the latent states and covariances given the observations
# 1000 time steps, 4-dimensional latent state, 5-dimensional observation
# Filtered results
Y, V, _, L = run_ssm_kalman(X, Y0, Q0, A, Q, C, R, mode='filt')

# Plot Y for the filtered case
plt.figure(figsize=(15, 5))
plt.plot(Y.T)
plt.title("Filtered Y")
plt.xlabel("Time")
plt.ylabel("Y values")
plt.show()

# Plot log-determinant of V for the filtered case
plt.figure(figsize=(15, 5))
plt.plot([logdet(torch.tensor(v)) for v in V])
plt.title("Log Determinant of V (Filtered)")
plt.xlabel("Time")
plt.ylabel("logdet(V)")
plt.show()
```

```python
# Smoothed results
Y, V, Vj, L = run_ssm_kalman(X, Y0, Q0, A, Q, C, R, mode='smooth')

# Plot Y for the smoothed case
# adjust the size of the figure
plt.figure(figsize=(15, 5))
plt.plot(Y.T)
plt.title("Smoothed Y")
plt.xlabel("Time")
plt.ylabel("Y values")
plt.show()

# Plot log-determinant of V for the smoothed case
plt.figure(figsize=(15, 5))
plt.plot([logdet(torch.tensor(v)) for v in V])
plt.title("Log Determinant of V (Smoothed)")
plt.xlabel("Time")
plt.ylabel("logdet(V)")
plt.show()
```

## 4(b) - Code

```
import numpy as np
import matplotlib.pyplot as plt
from ssm_kalman import run_ssm_kalman
```

```
# Function to initialize parameters A, Q, C, R
def initialize_parameters():
    A = 0.99 * np.array([[np.cos(2 * np.pi / 180), -np.sin(2 * np.pi / 180), 0, 0],
                         [np.sin(2 * np.pi / 180), np.cos(2 * np.pi / 180), 0, 0],
                         [0, 0, np.cos(2 * np.pi / 90), -np.sin(2 * np.pi / 90)],
                         [0, 0, np.sin(2 * np.pi / 90), np.cos(2 * np.pi / 90)]])

    Q = np.eye(4) - A @ A.T  # Q = I - AA^T

    C = np.array([[1, 0, 1, 0],
                  [0, 1, 0, 1],
                  [1, 0, 0, 0],
                  [0, 0, 1, 0],
                  [0.5, 0.5, 0.5, 0.5]])

    R = np.eye(5)

    return A, Q, C, R
```

```
# Function for random initialization of parameters
def random_initialize_parameters():
    A = np.random.randn(4, 4)
    Q = np.eye(4)  # Keeping Q positive semi-definite
    C = np.random.randn(5, 4)
    R = np.eye(5)  # Keeping R positive semi-definite
    return A, Q, C, R
```

```
# EM algorithm to learn A, Q, C, R
def EM_algorithm(X, Y0, Q0, num_iterations=50, A=None, Q=None, C=None, R=None):
    # Initialize parameters if not provided
    if A is None or Q is None or C is None or R is None:
        A, Q, C, R = initialize_parameters()
    k, T = Y0.shape[0], X.shape[1]
    log_likelihoods = []

    # Main EM loop
    for iteration in range(num_iterations):
        # E-step: Run the Kalman smoother
        Y, V, V_joint, log_likelihood = run_ssm_kalman(X, Y0, Q0, A, Q, C, R, mode='smooth')

        # Compute sufficient statistics for the M-step
        sum_xt_xtT = np.sum([np.outer(X[:, t], X[:, t]) for t in range(T)], axis=0)
        sum_xt_YtT = np.sum([np.outer(X[:, t], Y[:, t]) for t in range(T)], axis=0)
        sum_Yt_YtT = np.sum([np.outer(Y[:, t], Y[:, t]) + V[t] for t in range(1, T)], axis=0)
        sum_Yt_Ytm1T = np.sum([np.outer(Y[:, t], Y[:, t-1]) + V_joint[t] for t in range(1, T)], axis=0)

        # M-step: Update parameters
        C_new = sum_xt_YtT @ np.linalg.inv(sum_Yt_YtT)
        R_new = (1 / T) * (sum_xt_xtT - C_new @ sum_xt_YtT.T)
        A_new = sum_Yt_Ytm1T @ np.linalg.inv(sum_Yt_YtT)
        Q_new = (1 / (T - 1)) * (sum_Yt_YtT - A_new @ sum_Yt_Ytm1T.T)

        # Update parameters
        A, Q, C, R = A_new, Q_new, C_new, R_new

        # Store log-likelihood
        log_likelihoods.append(np.sum(log_likelihood))

    return A, Q, C, R, log_likelihoods
```

```
# Load data
X = np.loadtxt('ssm_spins.txt').T  # Transpose to have data vectors in columns
Y0 = np.zeros(4)  # Initial latent state
Q0 = np.eye(4)  # Initial state covariance
```

```
# Run EM with different initializations
num_iterations = 50
num_random_runs = 10
all_log_likelihoods = []

# Run EM with generating parameters
A, Q, C, R = initialize_parameters()
_, _, _, _, log_likelihoods = EM_algorithm(X, Y0, Q0, num_iterations, A, Q, C, R)
all_log_likelihoods.append(log_likelihoods)

# Run EM with random initializations
for i in range(num_random_runs):
    A, Q, C, R = random_initialize_parameters()
    _, _, _, _, log_likelihoods = EM_algorithm(X, Y0, Q0, num_iterations, A, Q, C, R)
    all_log_likelihoods.append(log_likelihoods)
```

```
# Plot log-likelihoods for each run
for i, log_likelihoods in enumerate(all_log_likelihoods):
    if i == 0:
        plt.plot(range(num_iterations), log_likelihoods, label="Generating Parameters", linewidth=2)
    else:
        plt.plot(range(num_iterations), log_likelihoods, label=f"Random Init {i}", linestyle='--')

plt.xlabel('Iterations')
plt.ylabel('Log Likelihood')
plt.title('Log Likelihood over EM Iterations for Different Initializations')
plt.legend()
plt.show()
```

```
# Load the test dataset
X_test = np.loadtxt('ssm_spins_test.txt').T
```

```
# Compute log-likelihood for a given dataset and parameter set
def compute_log_likelihood(X, Y0, Q0, A, Q, C, R):
    _, _, _, log_likelihood = run_ssm_kalman(X, Y0, Q0, A, Q, C, R, mode='filt')
    return np.sum(log_likelihood)
```

```
# Evaluate test data likelihoods
test_likelihoods = []

# 1. True parameters
A_true, Q_true, C_true, R_true = initialize_parameters()
test_likelihoods.append(("True Parameters", compute_log_likelihood(X_test, Y0, Q0, A_true, Q_true, C_true, R_true)

# 2. Parameters from generating initialization
test_likelihoods.append(("EM (Gen Params)", compute_log_likelihood(X_test, Y0, Q0, A, Q, C, R)))

# 3. Parameters from random initializations
for i in range(num_random_runs):
    A, Q, C, R = random_initialize_parameters()
    A_est, Q_est, C_est, R_est, _ = EM_algorithm(X, Y0, Q0, num_iterations, A, Q, C, R)
    likelihood = compute_log_likelihood(X_test, Y0, Q0, A_est, Q_est, C_est, R_est)
    test_likelihoods.append((f"EM (Random Init {i+1})", likelihood))
```

```
# Match the training and test likelihood sets
training_likelihoods = [("True Parameters", all_log_likelihoods[0][-1])]
training_likelihoods += [(f"EM (Random Init {i+1})", all_log_likelihoods[i + 1][-1]) for i in range(num_random_run

test_likelihoods = [("True Parameters", compute_log_likelihood(X_test, Y0, Q0, A_true, Q_true, C_true, R_true))]
test_likelihoods += [(f"EM (Random Init {i+1})", compute_log_likelihood(X_test, Y0, Q0, A, Q, C, R)) for i in rang

# Ensure equal lengths
assert len(training_likelihoods) == len(test_likelihoods), "Mismatch in training and test likelihoods!"
```

```
# Plot Training Log-Likelihoods
plt.figure(figsize=(10, 6))

for i, log_likelihoods in enumerate(all_log_likelihoods):
    if i == 0:
        plt.plot(range(num_iterations), log_likelihoods, label="Generating Parameters", linewidth=2)
    else:
        plt.plot(range(num_iterations), log_likelihoods, label=f"Random Init {i}", linestyle='--')

plt.xlabel("Iterations")
plt.ylabel("Log Likelihood")
plt.title("Training Data Log Likelihood Evolution Over EM Iterations")
plt.legend(loc="lower right", fontsize="small")
plt.tight_layout()
plt.show()
```

```
# Plot Test Log-Likelihoods
plt.figure(figsize=(10, 6))

for i in range(num_random_runs):
    A, Q, C, R = random_initialize_parameters()
    _, _, _, _, test_log_likelihoods = EM_algorithm(X_test, Y0, Q0, num_iterations, A, Q, C, R)
    if i == 0:
        plt.plot(range(num_iterations), test_log_likelihoods, label="Generating Parameters", linewidth=2)
    else:
        plt.plot(range(num_iterations), test_log_likelihoods, label=f"Random Init {i}", linestyle='--')

plt.xlabel("Iterations")
plt.ylabel("Log Likelihood")
plt.title("Test Data Log Likelihood Evolution Over EM Iterations")
plt.legend(loc="lower right", fontsize="small")
plt.tight_layout()
plt.show()
```

## 5(a) - Code

```
import torch
from collections import Counter
import matplotlib.pyplot as plt
```

```
# Load and Preprocess Text Data

# Function to read text from a file and return it as a list of characters
def load_text_to_list(file_path):
    with open(file_path, encoding='utf8') as file:
        text = list(file.read())
    # Replace newline characters with spaces, convert text to lowercase
    text = [' ' if char == '\n' else char.lower() for char in text]
    return text

# Load the primary text, symbol list, and the encrypted message as lists
english_text = load_text_to_list("war_and_peace.txt")
symbols = list(filter(lambda x: x != '\n', load_text_to_list("symbols.txt")))
encrypted_message = load_text_to_list("message.txt")
```

```python
# Clean the Text of Consecutive Symbols

# Function to remove consecutive spaces from the text
def remove_consecutive_spaces(text):
    cleaned_text = [text[0]]
    for i in range(1, len(text)):
        if not (text[i] == ' ' and text[i] == cleaned_text[-1]):
            cleaned_text.append(text[i])
    return cleaned_text

# Clean the English text to eliminate consecutive spaces
cleaned_english_text = remove_consecutive_spaces(english_text)
```

```python
# Count Symbol and Transition Frequencies

# Function to count the frequency of symbols and transitions between symbols
def count_symbol_frequencies_and_transitions(text, symbols):
    symbol_frequency = Counter(text)
    transition_frequency = Counter(zip(text, text[1:]))
    valid_symbols = set(symbols)

    # Filter counts to include only valid symbols and transitions
    filtered_symbol_counts = {sym: symbol_frequency[sym] for sym in valid_symbols if sym in symbol_frequency}
    filtered_transition_counts = {pair: transition_frequency[pair] for pair in transition_frequency if pair[0] in

    return filtered_symbol_counts, filtered_transition_counts

# Count symbols and transitions in the English text
symbol_frequency, transition_frequency = count_symbol_frequencies_and_transitions(english_text, symbols)
```

```python
# Create and Normalize the Transition Matrix


# Function to build a transition matrix based on transition counts
def build_transition_matrix(symbols, transition_counts):
    num_symbols = len(symbols)
    symbol_index = {sym: idx for idx, sym in enumerate(symbols)}
    transition_matrix = torch.zeros((num_symbols, num_symbols))

    for (from_symbol, to_symbol), count in transition_counts.items():
        i, j = symbol_index[from_symbol], symbol_index[to_symbol]
        transition_matrix[i, j] = count

    # Normalize rows to ensure they sum to 1
    row_sums = transition_matrix.sum(dim=1, keepdim=True)
    normalized_matrix = transition_matrix / row_sums.clamp(min=1e-10)  # Prevent division by zero
    return normalized_matrix

# Create the normalized transition matrix
transition_matrix = build_transition_matrix(symbols, transition_frequency)
```

```python
# Regularize the Transition Matrix

# Function to make the transition matrix irreducible and aperiodic
def regularize_transition_matrix(matrix, epsilon=1e-4):
    regularized_matrix = matrix + epsilon * torch.ones_like(matrix)
    row_sums = regularized_matrix.sum(dim=1, keepdim=True)
    return regularized_matrix / row_sums  # Normalize again to ensure each row sums to 1

# Apply regularization to the transition matrix
transition_matrix = regularize_transition_matrix(transition_matrix)
```

```python
# Compute the Stationary Distribution

# Function to compute the stationary distribution of the Markov chain
def calculate_stationary_distribution(matrix, iterations=100):
    num_states = matrix.size(0)
    distribution = torch.full((num_states,), 1.0 / num_states)
    for _ in range(iterations):
        distribution = distribution @ matrix

    return distribution

# Calculate the stationary distribution
stationary_distribution = calculate_stationary_distribution(transition_matrix)
```

```
# Map Symbols to Their Stationary Probabilities

# Function to create a dictionary mapping symbols to their stationary probabilities
def map_symbols_to_stationary_probabilities(symbols, distribution):
    return {symbols[i]: distribution[i].item() for i in range(len(symbols))}

stationary_probabilities_dict = map_symbols_to_stationary_probabilities(symbols, stationary_distribution)
```

```
# Display Results

# Visualize the transition matrix as a heatmap
plt.figure(figsize=(10, 8))
plt.imshow(transition_matrix, cmap='Blues', interpolation='nearest')
plt.colorbar(label='Transition Probability')
plt.xticks(ticks=range(len(symbols)), labels=symbols, rotation=90)
plt.yticks(ticks=range(len(symbols)), labels=symbols)
plt.title('Transition Matrix Heatmap')
plt.xlabel('To Symbol')
plt.ylabel('From Symbol')
plt.show()

# Print the stationary distribution
print("Symbol".ljust(8), "Stationary Probability")
for symbol, prob in stationary_probabilities_dict.items():
    print(f"{symbol:<8} {prob:.3e}")
```

## 5(d) - Code

```
import random
import math
```

```
# Map Symbols to Indices

# Create a dictionary that maps each symbol to its index in the transition matrix
symbol_index_mapping = {symbol: idx for idx, symbol in enumerate(symbols)}
```

```
# Step 2: Calculate Log-Likelihood

# Function to compute the log-likelihood of the encrypted message based on a symbol permutation
def compute_log_likelihood(permutation, message_text, transition_matrix, stationary_distribution):
    log_probability = 0.0
    # Get the stationary probability of the initial symbol
    initial_symbol = permutation[message_text[0]]
    log_probability += torch.log(stationary_distribution[symbol_index_mapping[initial_symbol]])

    # Calculate transition probabilities for subsequent symbols
    for i in range(1, len(message_text)):
        previous_symbol = permutation[message_text[i - 1]]
        current_symbol = permutation[message_text[i]]
        log_probability += torch.log(transition_matrix[symbol_index_mapping[previous_symbol], symbol_index_mapping

    return log_probability
```

```
# Symbol Swapping for Proposals

# Function to swap two symbols in the current permutation to generate a new proposal
def generate_new_permutation(permutation):
    new_permutation = permutation.copy()
```

```python
    symbol_a, symbol_b = random.sample(symbols, 2)  # Randomly select two symbols to swap
    new_permutation[symbol_a], new_permutation[symbol_b] = new_permutation[symbol_b], new_permutation[symbol_a]
    return new_permutation
```

```python
# Acceptance Probability Calculation

# Function to determine whether to accept the new permutation based on log-likelihood
def calculate_acceptance_probability(current_log_prob, new_log_prob):
    if new_log_prob > current_log_prob:
        return 1.0
    else:
        return math.exp(new_log_prob - current_log_prob)
```

```python
# Initialize Random Permutation

# Function to create an initial random permutation of symbols
def initialize_permutation():
    permutation = {symbol: symbol for symbol in symbols}  # Start with the identity permutation
    random.shuffle(list(permutation.keys()))
    return permutation
```

```python
# MCMC Sampling Procedure

# Function to perform MCMC sampling for message decoding
def perform_mcmc_sampling(message_text, transition_matrix, stationary_distribution, num_iterations=5000, report_in
    # Initialize the permutation and calculate its log-likelihood
    current_permutation = initialize_permutation()
    current_log_likelihood = compute_log_likelihood(current_permutation, message_text, transition_matrix, stationa

    # MCMC iteration loop
    for iteration in range(num_iterations):
        # Propose a new permutation by swapping two symbols
        proposed_permutation = generate_new_permutation(current_permutation)
        new_log_likelihood = compute_log_likelihood(proposed_permutation, message_text, transition_matrix, station

        # Determine the acceptance probability
        acceptance_prob = calculate_acceptance_probability(current_log_likelihood, new_log_likelihood)

        # Accept or reject the proposed permutation
        if random.uniform(0, 1) < acceptance_prob:
            current_permutation = proposed_permutation
            current_log_likelihood = new_log_likelihood

        # Every report_interval iterations, print the current decrypted message (first 60 symbols)
        if iteration % report_interval == 0:
            decrypted_message_preview = ''.join([current_permutation[char] for char in message_text[:60]])
            print(f"Iteration {iteration}: Decoded message (first 60 symbols): {decrypted_message_preview}")

    # Output the final decoded message after all iterations
    final_decoded_message = ''.join([current_permutation[char] for char in message_text])
    print("\nFinal decoded message (first 60 symbols):", final_decoded_message[:60])
    return final_decoded_message
```

```python
# Step 7: Execute MCMC Sampling

# Run the MCMC sampling process
perform_mcmc_sampling(message_text, transition_matrix, stationary_distribution, num_iterations=30000)
```