Probabilistic and Unsupervised Learning Summative Assignment COMP0086

Richard Huang

Nov 13 2024

Question 1

(a) The dataset contains binary images where each pixel is either 0 or 1, representing black or white, respectively. A multivariate Gaussian distribution, however, assumes continuous values for each variable. Specifically, each element in a multivariate Gaussian vector can take any real value, which doesn't match the binary nature of this dataset.

Additionally, a multivariate Gaussian is defined by mean and covariance parameters and is used for data that tends to follow a continuous, bell-shaped distribution in each dimension. Since binary data (0 or 1) does not follow a continuous distribution, a Gaussian model would not capture the true distribution of the binary image data effectively. Therefore, a multivariate Gaussian model is not suitable for modeling binary vectors like these images.

(b) To find the ML estimate of $\mathbf{p}$, we maximize the likelihood of the data given $\mathbf{p}$. The likelihood function for a single sample $\mathbf{x}$ is given by:

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

For N images, the likelihood of the entire dataset becomes:

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

Taking the logarithm to simplify, we get the log-likelihood:

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

Define $s_d = \sum_{n=1}^{N} x_d^{(n)}$, the total count of 1s at pixel d across all images. Then, the log-likelihood simplifies to:

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

To maximize with respect to p_d, take the derivative of the log-likelihood with respect to p_d and set it to zero:

$$\frac{\partial}{\partial p_d}\log P = \frac{s_d}{p_d} - \frac{N-s_d}{1-p_d} = 0$$

Solving this equation for p_d, we find that:

$$p_{\text{MLE}} = \frac{1}{N}\sum_{n=1}^{N} x_d^{(n)}$$

Thus, the maximum likelihood estimate for each pixel parameter p_{MLE} is given by:

$$p_{\text{MLE}} = \frac{1}{N}\sum_{n=1}^{N} x_d^{(n)}$$

(c) To find the maximum a posteriori (MAP) estimator for $\mathbf{p}$, we assume independent Beta priors for each p_d with parameters α and β. The prior distribution for each p_d is given by:

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

where $B(\alpha, \beta)$ is the Beta function. The prior distribution for $\mathbf{p}$ is:

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

The likelihood function for the observed data is:

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

The posterior distribution $P(\mathbf{p}|\mathbf{x})$ is proportional to the product of the likelihood and the prior:

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

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

Simplifying for each p_d, we have:

$$P(p_d|\mathbf{x}) \propto p_d^{\sum_{n=1}^{N} x_d^{(n)}+\alpha-1}(1-p_d)^{N-\sum_{n=1}^{N} x_d^{(n)}+\beta-1}$$

To find the MAP estimate, we maximize $P(p_d|\mathbf{x})$ with respect to p_d, which is equivalent to maximizing the logarithm of $P(p_d|\mathbf{x})$:

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

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

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

Solving for p_d, we get:

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

Expanding and simplifying:

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

Thus, the MAP estimate for p_d is:

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

Therefore, the MAP estimator for $\mathbf{p}$ is:

$$p_d^{\text{MAP}} = \frac{\sum_{n=1}^{N} x_d^{(n)} + \alpha - 1}{N + \alpha + \beta - 2} \quad \text{for each } d = 1, \ldots, D.$$

(d) The visualization of the learned parameter vector for question(d) is shown in Figure 1

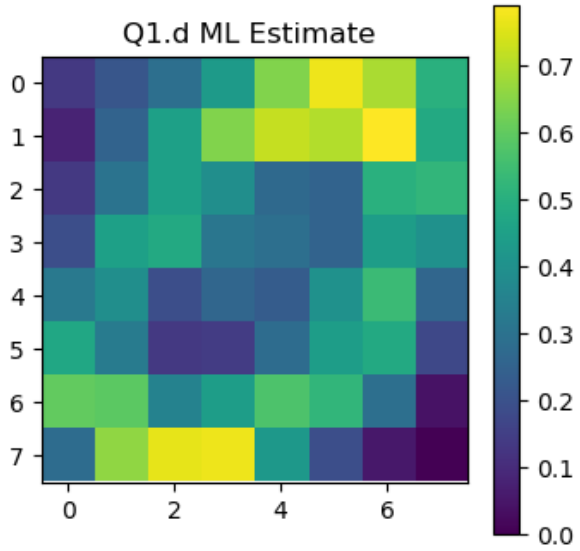

Figure 1: ML Parameters

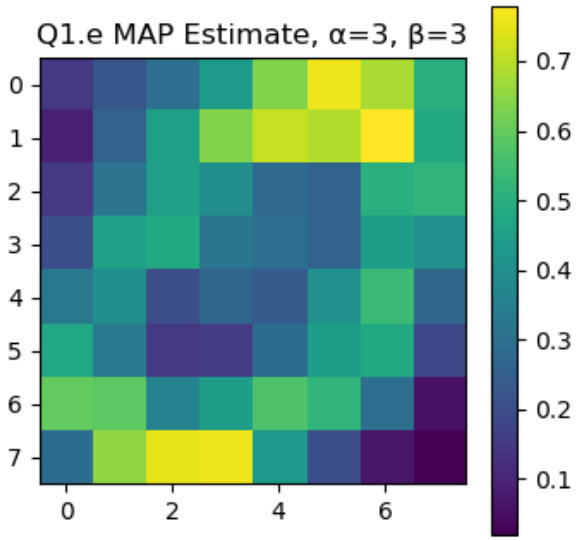

Figure 2: MAP Parameters

(e) The visualization of the learned parameter vector for question(d) is shown in Figure 2

The ML estimate for the multivariate Bernoulli model calculates each parameter p_d as the proportion of times the pixel is 1 across all images in the dataset. This approach directly reflects the observed data, giving an estimate that is unbiased by any prior beliefs. However, this estimate can lead to extreme values (close to 0 or 1) for pixels that are consistently off or on, particularly in small datasets. These extreme values might result in overfitting, where the model captures noise rather than general patterns.

In contrast, the MAP estimate incorporates a prior, specifically a Beta distribution with parameters $\alpha = 3$ and $\beta = 3$, which introduces regularization. This prior pulls each parameter p_d toward 0.5, a value indicating no strong preference for on or off. The MAP estimate is thus less influenced by rare occurrences or noise in the data, providing more conservative probabilities. This regularization effect is particularly useful in cases where there is limited data for certain pixels, as the prior stabilizes the estimate and prevents extreme values.

While MAP regularization helps avoid overfitting and can improve generalization to new data, it introduces bias. The prior's pull toward 0.5 means that the MAP estimate may understate genuine patterns in the data, especially when certain pixels are consistently on or off. In large datasets, where data patterns are clear and representative of the true distribution, the ML estimate is typically more accurate, as it directly reflects the observed data without the influence of a prior. However, in smaller or noisier datasets, the MAP estimate is often preferable, as it offers a balanced estimate that mitigates overfitting risks. Thus, while the MAP estimate may be more robust in uncertain data conditions, it can underrepresent true patterns compared to the ML estimate when ample data is available.

Python code for d&e:

```

import numpy as np
import matplotlib.pyplot as plt

#Load Data
data = np.loadtxt('/Users/richardhuang/Documents/GitHub/Projects/Probabilistc
    Summartive/binarydigits.txt')

def learn_ml_params(data):
    """
    Learn the Maximum Likelihood (ML) parameters for a multivariate Bernoulli distribution.
    This function computes the ML estimate and reshapes it to an 8x8 image format.
    """
    p_mle = np.mean(data, axis=0)
    return p_mle.reshape(8, 8)

def learn_map_params(data, alpha=3, beta=3):
    """
    Learn the Maximum A Posteriori (MAP) parameters for a multivariate Bernoulli distribution
    with Beta prior parameters alpha and beta. This function computes the MAP estimate and
    reshapes it to an 8x8 image format.
    """
    N = data.shape[0]
    pixel_sums = np.sum(data, axis=0)
    p_map = (pixel_sums + alpha - 1) / (N + alpha + beta - 2)
    return p_map.reshape(8, 8)

#Calculate ML and MAP parameters
p_mle_image = learn_ml_params(data)
p_map_image = learn_map_params(data, alpha=3, beta=3)

plt.figure(figsize=(4, 4))
plt.imshow(p_mle_image, cmap='viridis', interpolation='nearest')
plt.colorbar()
plt.title("Q1.d ML Estimate ")
plt.show()

plt.figure(figsize=(4, 4))
plt.imshow(p_map_image, cmap='viridis', interpolation='nearest')
plt.colorbar()
plt.title("Q1.e MAP_Estimate, \alpha = 3, \beta=3")
plt.show()

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

Listing 1: solutions/q1.py

Question 2

(a) In model (a), we assume that every pixel in the image is generated by a Bernoulli distribution with the same parameter $p_d = 0.5$ for all pixels. The probability mass function for a Bernoulli distribution is:

$$P(x_d|p_d) = p_d^{x_d}(1-p_d)^{1-x_d}$$

where $x_d \in \{0, 1\}$ represents the value of the d-th pixel in the image and $p_d = 0.5$ is the probability of the pixel being 1.

Since $p_d = 0.5$ for all pixels, the likelihood of observing a specific image $\mathbf{x}^{(n)}$ with D pixels is:

$$P(\mathbf{x}^{(n)}|p_d = 0.5) = \prod_{d=1}^{D} 0.5 = 0.5^D$$

The total likelihood for the entire dataset, which consists of N independent images, is:

$$P(\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \ldots, \mathbf{x}^{(N)}|p_d = 0.5) = \prod_{n=1}^{N} P(\mathbf{x}^{(n)}|p_d = 0.5) = \left(0.5^D\right)^N = 0.5^{N\times D}$$

Thus, the likelihood of the dataset for model (a) is:

$$P(\mathbf{x}|p) = 0.5^{N\times D}$$

(b) In model (b), we assume that each pixel in the image is generated by a Bernoulli distribution with an unknown parameter p_d, which is the same for all pixels in the image. The likelihood of observing an image $\mathbf{x}^{(n)}$ with D pixels is:

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

Since the images are independent, the total likelihood for the entire dataset of N images is:

$$P(\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \ldots, \mathbf{x}^{(N)}|p_d) = \prod_{n=1}^{N}\prod_{d=1}^{D} p_d^{x_d^{(n)}}(1-p_d)^{1-x_d^{(n)}}$$

Thus, the likelihood of the dataset for model (b) is:

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

(c) In model (c), we assume that each pixel has its own separate Bernoulli distribution with its own unknown parameter p_d. The likelihood of observing an image $\mathbf{x}^{(n)}$ is:

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

The total likelihood for the entire dataset of N images is:

$$P(\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \ldots, \mathbf{x}^{(N)}|\mathbf{p}) = \prod_{n=1}^{N} \prod_{d=1}^{D} p_d^{x_d^{(n)}} (1-p_d)^{1-x_d^{(n)}}$$

Thus, the likelihood of the dataset for model (c) is:

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

Posterior probabilities Since the priors for the models are uniform, we have:

$$P(M_a) = P(M_b) = P(M_c) = \frac{1}{3}$$

The marginal likelihood $P(\mathbf{x})$ is computed as:

$$P(\mathbf{x}) = P(\mathbf{x}|M_a) \cdot P(M_a) + P(\mathbf{x}|M_b) \cdot P(M_b) + P(\mathbf{x}|M_c) \cdot P(M_c)$$

Given the three models M_a, M_b, and M_c, the posterior probabilities can be computed using Bayes' Theorem:

$$P(M_i|\mathbf{x}) = \frac{P(\mathbf{x}|M_i) \cdot P(M_i)}{P(\mathbf{x})}$$

Where the evidence $P(\mathbf{x})$ is:

$$P(\mathbf{x}) = P(\mathbf{x}|M_a) \cdot P(M_a) + P(\mathbf{x}|M_b) \cdot P(M_b) + P(\mathbf{x}|M_c) \cdot P(M_c)$$

Thus, the posterior probabilities for each model are:

$$P(M_a|\mathbf{x}) = \frac{P(\mathbf{x}|M_a) \cdot P(M_a)}{P(\mathbf{x})}$$

$$P(M_b|\mathbf{x}) = \frac{P(\mathbf{x}|M_b) \cdot P(M_b)}{P(\mathbf{x})}$$

$$P(M_c|\mathbf{x}) = \frac{P(\mathbf{x}|M_c) \cdot P(M_c)}{P(\mathbf{x})}$$

Where the likelihoods for each model are as follows: - For model M_a, $P(\mathbf{x}|M_a) = 0.5^{N \times D}$, - For model M_b, $P(\mathbf{x}|M_b) = \prod_{d=1}^{D} p_d^{x_d^{(n)}} (1-p_d)^{1-x_d^{(n)}}$, - For model M_c, $P(\mathbf{x}|M_c) = \prod_{d=1}^{D} p_d^{x_d^{(n)}} (1-p_d)^{1-x_d^{(n)}}$.

Where p_d is the mean of the data for each pixel in the case of models (b) and (c).

The posterior probabilities for each model, calculated using Bayes' theorem, are as follows:

$$P(M_a|\mathbf{x}) = 0.37434997449465635$$

$$P(M_b|\mathbf{x}) = 0.3128250127526719$$

$$P(M_c|\mathbf{x}) = 0.3128250127526717$$

Model	**Posterior Probability**
Model (a)	0.37434997449465635
Model (b)	0.3128250127526719
Model (c)	0.3128250127526717

Table 1: Posterior probabilities for each model

Python code for Q2:

```

N, D = data.shape

# Model (a) - Log likelihood
P_x_given_Ma = N * D * np.log(0.5)

# Model (b) - Log likelihood
P_x_given_Mb = np.sum(np.log(np.power(p_d_b, np.sum(data, axis=0)) *
np.power(1 - p_d_b, N - np.sum(data, axis=0))))

# Model (c) - Log likelihood
P_x_given_Mc = np.sum(np.sum(np.log(np.power(p_d_b, data) * np.power(1 - p_d_b, 1 - data)),
    axis=0))

# Priors
P_Ma = P_Mb = P_Mc = 1/3

# Evidence (marginal likelihood)
P_x = P_x_given_Ma * P_Ma + P_x_given_Mb * P_Mb + P_x_given_Mc * P_Mc


if P_x == 0:
    print("Warning: The evidence (P_x) is zero!")

# Posterior probabilities using Bayes' theorem
P_Ma_given_x = (P_x_given_Ma * P_Ma) / P_x if P_x != 0 else 0
P_Mb_given_x = (P_x_given_Mb * P_Mb) / P_x if P_x != 0 else 0
P_Mc_given_x = (P_x_given_Mc * P_Mc) / P_x if P_x != 0 else 0


print(f"Posterior probability for Model (a): {P_Ma_given_x}")
print(f"Posterior probability for Model (b): {P_Mb_given_x}")
print(f"Posterior probability for Model (c): {P_Mc_given_x}")
```

Listing 2: solutions/q1.py

Question 3

(a) To model a mixture of K multivariate Bernoulli distributions for the binary image dataset, we define the following parameters:

$\pi_1, \pi_2, \ldots, \pi_K$: the mixing proportions for the K components, where $0 \leq \pi_k \leq 1$ and $\sum_{k=1}^{K} \pi_k = 1$. - $\mathbf{P}$: a $K \times D$ matrix where each element p_{kd} represents the probability that pixel d takes the value 1 under mixture component k.

The likelihood for a single image $\mathbf{x} = (x_1, x_2, \ldots, x_D)$ under this mixture model is given by:

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

Since the images are independent and identically distributed (i.i.d.) under the model, the likelihood of the entire dataset $X = \{\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \ldots, \mathbf{x}^{(N)}\}$ is:

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

(b) To compute the responsibility of mixture component k for data vector $\mathbf{x}^{(n)}$, we define the responsibility r_{nk} as the posterior probability that the hidden variable $s^{(n)} = k$ given $\mathbf{x}^{(n)}, \pi, \mathbf{P}$.

This is expressed as:

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

Since $P(s^{(n)} = k|\pi) = \pi_k$ and $P(\mathbf{x}^{(n)}|s^{(n)} = k, \mathbf{P}) = \prod_{d=1}^{D} p_{kd}^{x_d^{(n)}} (1 - p_{kd})^{1-x_d^{(n)}}$, we can rewrite r_{nk} as:

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

This expression represents the responsibility of component k for data point $\mathbf{x}^{(n)}$, and it forms the E-step in the Expectation-Maximization (EM) algorithm. During this E-step, we compute the responsibilities r_{nk} for each data point and each component k given the current estimates of the parameters π and $\mathbf{P}$.

(c) To find the maximizing parameters for the expected log-joint likelihood with respect to the mixing proportions π and the Bernoulli parameters $\mathbf{P}$, we can proceed as follows:

The goal is to maximize the expected log-joint likelihood, which is given by:

$$\mathbb{E}_{q(\{s^{(n)}\})} \left[\sum_{n} \log P(\mathbf{x}^{(n)}, s^{(n)}|\pi, \mathbf{P}) \right] = \sum_{n} \sum_{k=1}^{K} r_{nk} \log P(\mathbf{x}^{(n)}, s^{(n)} = k|\pi, \mathbf{P})$$

where $r_{nk} = P(s^{(n)} = k|\mathbf{x}^{(n)}, \pi, \mathbf{P})$ is the responsibility assigned to component k for data point $\mathbf{x}^{(n)}$.

Maximizing with Respect to π (Mixing Proportions): The term involving π_k is:

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

To maximize with respect to π_k (subject to $\sum_{k=1}^{K} \pi_k = 1$), we use a Lagrange multiplier λ to enforce this constraint. The Lagrange function is:

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

Taking the partial derivative of $\mathcal{L}$ with respect to π_k and setting it to zero gives:

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

Rearranging, we find:

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

To satisfy the constraint $\sum_{k=1}^{K} \pi_k = 1$, we substitute $\pi_k = \frac{\sum_{n=1}^{N} r_{nk}}{\lambda}$ into the constraint, which gives:

$$\lambda = N$$

Therefore, the update rule for π_k becomes:

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

Maximizing with Respect to $\mathbf{P}$ (Bernoulli Parameters): For each Bernoulli parameter p_{kd}, the relevant term in the expected log-joint likelihood is:

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

To maximize with respect to p_{kd}, take the partial derivative with respect to p_{kd} and set it to zero:

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

Solving this equation gives the update rule:

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

This expression updates p_{kd} as the weighted average of the pixel values $x_d^{(n)}$ for all data points, where the weights are given by the responsibilities r_{nk}.

The M-step of the EM algorithm for this model involves updating the parameters π and $\mathbf{P}$ using the following rules:

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

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

These updates maximize the expected log-joint likelihood given the responsibilities computed in the E-step.

(d) Figure 3 displays the convergence of the log-likelihood over iterations for different values of K in the EM algorithm. Each subplot corresponds to a distinct value of K, specifically $K = 2, 3, 4, 7$, and 10, as labeled. For each K value, the log-likelihood is observed to increase as the number of iterations progresses, eventually reaching a plateau, which indicates that the EM algorithm has found a stable set of parameters and converged. This behavior is consistent across all values of K, with convergence generally achieved within approximately 15–20 iterations.

As K increases, the final log-likelihood at convergence improves. For example, the log-likelihood converges around -3300 for $K = 2$, whereas it reaches approximately -2400 for $K = 10$. This increase in the log-likelihood with higher values of K is expected, as models with more components possess greater flexibility to fit the data distribution more closely. Therefore, higher values of K allow the model to capture more patterns within the data, resulting in a better fit.

However, while the increasing log-likelihood values with higher K suggest an improved fit, this does not imply that the model with the highest K (here $K = 10$) is necessarily the best. Increasing K introduces more parameters into the model, which can lead to overfitting, where the model captures noise rather than genuine underlying patterns in the data.

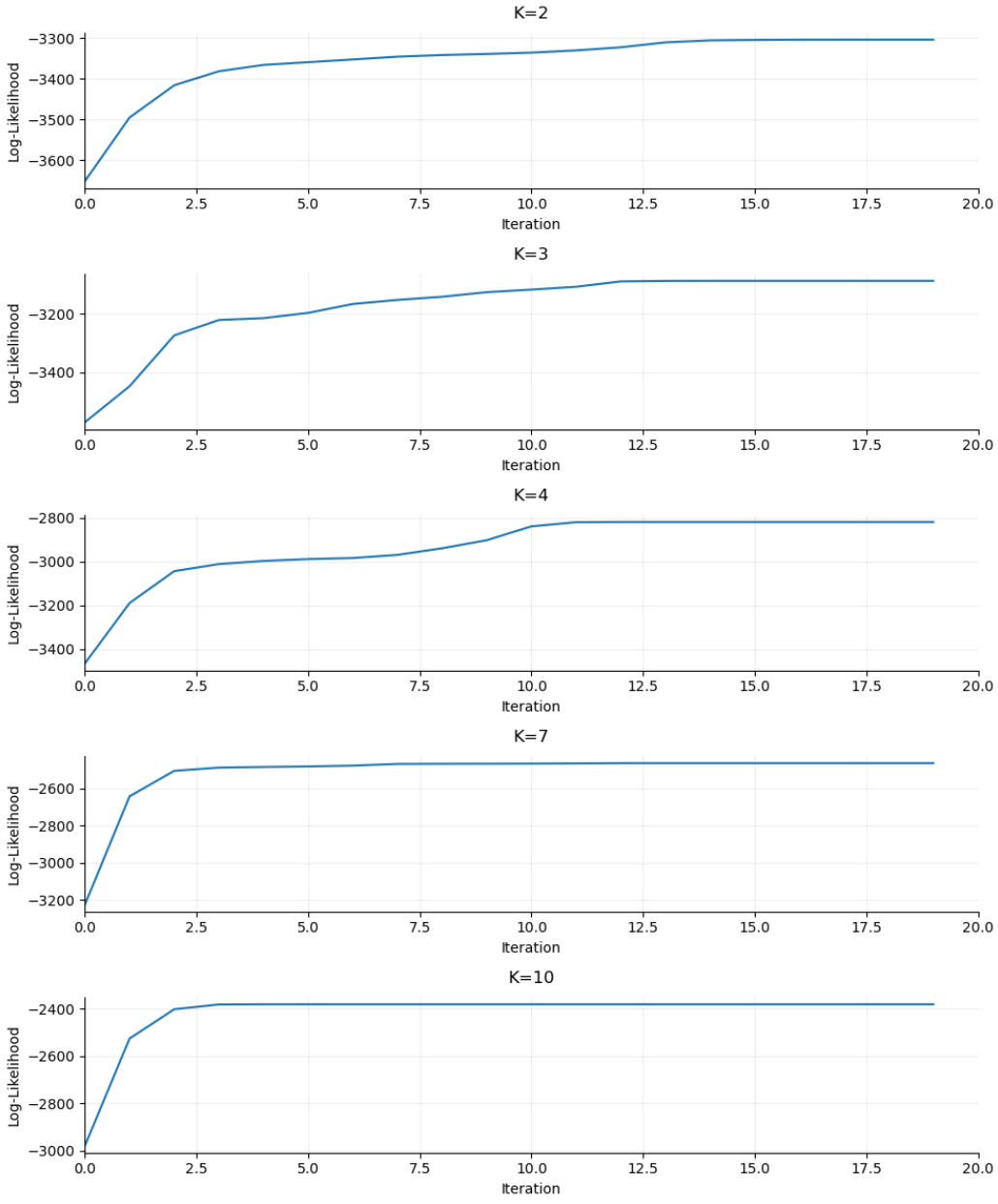

Figure 3: Log likelihood Function

The Figure 4 shows the learned parameters of a mixture model with varying values of K (the number of components). Each row corresponds to a specific K value, with each component represented by an 8×8 grid displaying pixel probabilities. Higher values (in yellow) indicate a greater likelihood of the pixel being "on." Each component's mixing proportion, π, is shown above the grid. As K increases, the model captures more diverse patterns, with finer structural details becoming apparent. However, for higher K, some components have low mixing proportions, indicating they contribute minimally to the mixture.

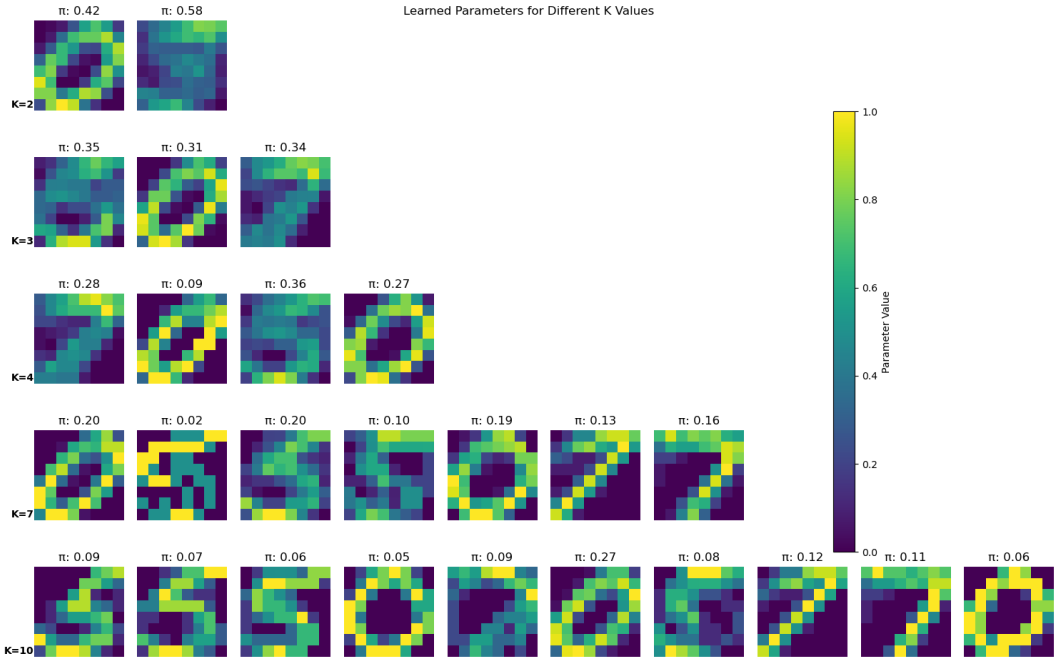

Figure 4: EM Learned Parameters for Different K Values

Python code for (d) (Log likelihood vs iteration number with different K):

```python
# Initialize the parameters for EM algorithm
def initialize_parameters(K, D):
    # Randomly initialize mixing proportions and ensure they sum to 1
    pi = np.random.dirichlet(alpha=np.ones(K), size=1)[0]

    P = 0.2 + 0.6 * np.random.rand(K, D)
    return pi, P

# Calculate the log-likelihood of the data
def compute_log_likelihood(X, pi, P):
    N, D = X.shape
    K = pi.shape[0]
    log_likelihood = 0
    for n in range(N):
        log_prob_sum = np.log(np.sum([
            pi[k] * np.prod(P[k]**X[n] * (1 - P[k])**(1 - X[n])) for k in range(K)
        ]))
        log_likelihood += log_prob_sum
    return log_likelihood

# E-step: calculate responsibilities
def e_step(X, pi, P):
    N, D = X.shape
    K = pi.shape[0]
    responsibilities = np.zeros((N, K))

    for n in range(N):
        log_resps = np.array([
```

```
            np.log(pi[k] + 1e-10) + np.sum(X[n] * np.log(P[k] + 1e-10) + (1 - X[n])
            * np.log(1 - P[k] + 1e-10))
            for k in range(K)
        ])
        max_log_resp = np.max(log_resps)
        log_resps -= max_log_resp
        responsibilities[n, :] = np.exp(log_resps)
        responsibilities[n, :] /= np.sum(responsibilities[n, :])

    return responsibilities

# M-step: update parameters based on responsibilities
def m_step(X, responsibilities):
    N, D = X.shape
    K = responsibilities.shape[1]
    pi_new = np.sum(responsibilities, axis=0) / N
    P_new = np.dot(responsibilities.T, X) / np.sum(responsibilities, axis=0)[:, None]
    return pi_new, P_new

# EM algorithm
def plot_log_likelihoods_subplots(data, K_values):
    """Plot log-likelihoods in the specified format"""
    fig, axes = plt.subplots(len(K_values), 1, figsize=(10, 12))



    for idx, K in enumerate(K_values):
        # Run EM
        pi, P, log_likelihoods = em_algorithm(data, K, max_iter=20)

        # Plot
        ax = axes[idx]
        ax.plot(np.arange(len(log_likelihoods)), log_likelihoods, '-', linewidth=1.5,
            color='#1f77b4')

        ax.set_title(f'K={K}', pad=10)
        ax.set_xlabel('Iteration')
        ax.set_ylabel('Log-Likelihood')

        ax.set_xlim(0, 50)

        ax.grid(True, linestyle='-', alpha=0.2)
        ax.spines['top'].set_visible(False)
        ax.spines['right'].set_visible(False)

        ax.tick_params(direction='out')

    plt.tight_layout()
    plt.show()

# Modified EM algorithm to ensure proper convergence behavior
def em_algorithm(X, K, max_iter=20):
    """EM algorithm with fixed number of iterations"""
    pi, P = initialize_parameters(K, X.shape[1])
    log_likelihoods = []

    for _ in range(max_iter):
        # E-step
        resp = e_step(X, pi, P)

        # M-step
        pi, P = m_step(X, resp)

        # Compute and store log likelihood
        log_likelihood = compute_log_likelihood(X, pi, P)
        log_likelihoods.append(log_likelihood)

    return pi, P, log_likelihoods

# Run with the specified K values
data = np.loadtxt('binarydigits.txt')
```

```
99
100 plot_log_likelihoods_subplots(data, [2, 3, 4, 7, 10])
```

Listing 3: src/solutions/q1.py

Python code for (d) (Display parameters found):

```
1
2  plt.figure(figsize=(15, 10))
3
4  max_K = 10
5  subplot_idx = 1
6
7  for K in [2, 3, 4, 7, 10]:
8      # Run EM algorithm
9      pi, P, log_likelihoods = em_algorithm(data, K, 20)
10
11     # Plot each component for current K
12     for k in range(K):
13         subplot_pos = (subplot_idx - 1) * max_K + k + 1
14         plt.subplot(5, max_K, subplot_pos)
15
16         # Display the image
17         component_params = P[k].reshape(8, 8)
18         im = plt.imshow(component_params,
19                         cmap='viridis',
20                         vmin=0,
21                         vmax=1,
22                         interpolation='nearest')
23         plt.axis('off')
24
25         # Add pi value above each component
26         plt.title(f'  : {pi[k]:.2f}')
27
28         # Add K value to bottom-left corner of first component in each row
29         if k == 0:
30             plt.text(-0.5, 7, f'K={K}',
31                     color='black',
32                     fontweight='bold',
33                     fontsize=10,
34                     ha='right',
35                     va='center')
36
37     subplot_idx += 1
38
39 plt.colorbar(im, ax=plt.gcf().axes, shrink=0.8, label='Parameter Value')
40
41 plt.suptitle('Learned Parameters for Different K Values', y=0.95)
42 plt.tight_layout()
43 plt.show()
```

Listing 4: src/solutions/q1.py

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

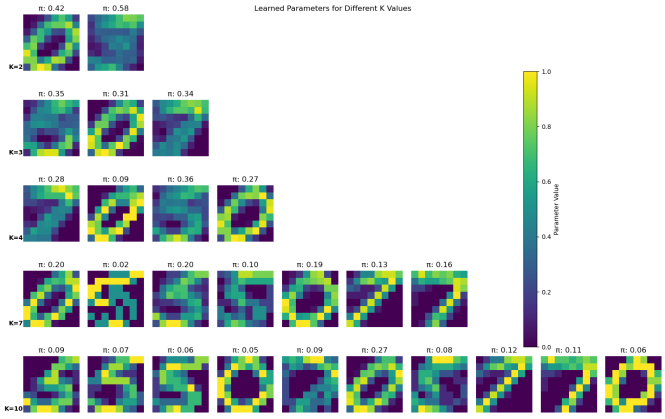

Figure 5: EM Learned Parameters Trial 0

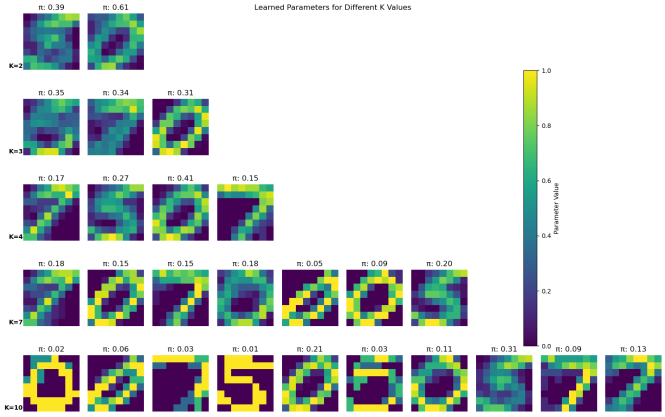

Figure 6: EM Learned Parameters Trial 1

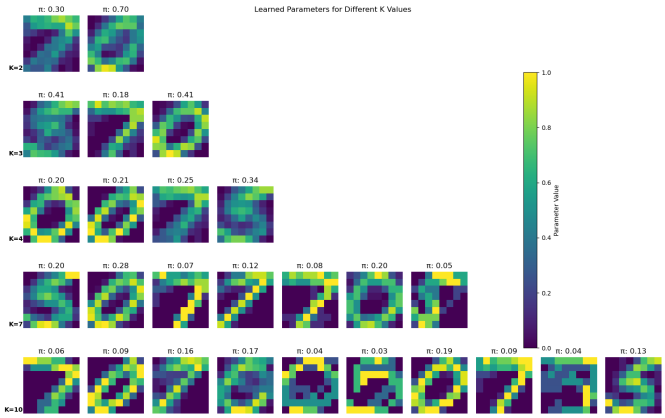

Figure 7: EM Learned Parameters Trial 2

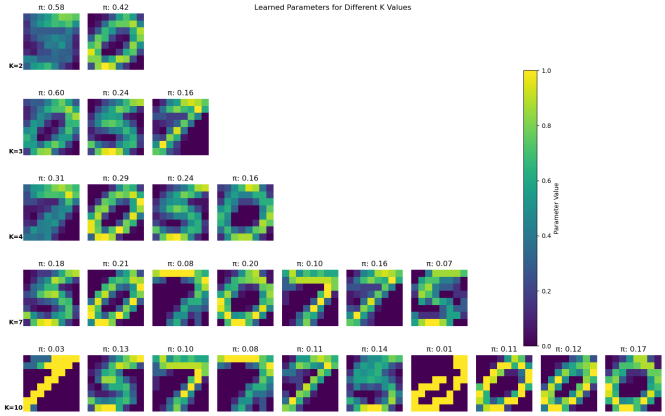

Figure 8: EM Learned Parameters Trial 3

The learned parameters from the EM algorithm indicate varying degrees of distinctiveness across different values of K. For lower values of K (e.g., $K = 2$ and $K = 3$), the clusters exhibit recognizable patterns, suggesting that the algorithm effectively identifies distinct groups in the data. The images for these clusters show clear features, indicating successful clustering.

As K increases (e.g., from $K = 4$ to $K = 10$), the clusters become more complex and less distinct. This may indicate that the algorithm is attempting to fit noise in the data rather than capturing meaningful patterns, leading to less interpretable results.

The mixing proportions provide insight into the relative size of each cluster:

For $K = 2$, one cluster has a significantly higher proportion ($\pi \approx 0.66$) compared to the other ($\pi \approx 0.34$). This suggests that the algorithm has identified a dominant pattern in the data.

As K increases, the proportions become more evenly distributed, which can be a sign of overfitting, where the model attempts to create too many clusters for the available data.

The responsibilities indicate how likely each data point belongs to each cluster. If the responsibilities are concentrated (i.e., a data point has a high probability of belonging to one cluster), it suggests that the clusters are well-defined. Conversely, if responsibilities are spread out, it may indicate overlapping clusters or poor separation, which can complicate the interpretation of the results.

Each learned parameter matrix can be interpreted as a representation of specific features or patterns in the data. For example, if the data consists of binary images of digits, certain clusters may represent specific digits or variations of digits.

The patterns in the learned parameters can be analyzed to understand what features are being captured by each cluster. Clusters that show circular patterns may correspond to digits like '0' or '6'.

To enhance the performance of the EM algorithm, the following improvements can be considered:

- Better Initialization: Using k-means clustering to initialize the parameters can lead to better convergence and

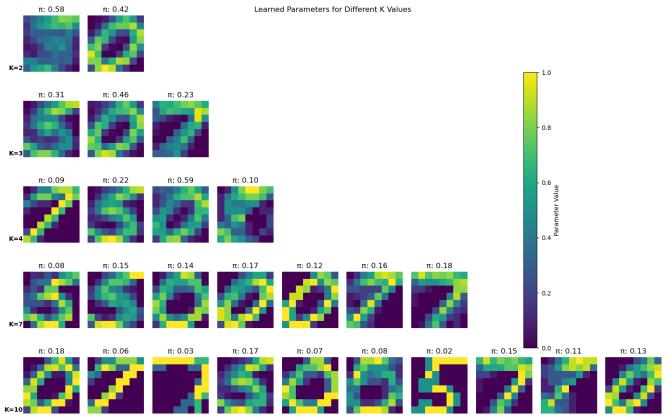

Figure 9: EM Learned Parameters Trial 4

more distinct clusters.

- Regularization: Adding regularization to the parameters can help prevent overfitting, especially if the model is too complex for the data.
- Feature Engineering: Transforming the data or adding features may help the algorithm better capture the underlying structure.
- More Iterations: Allowing for more iterations or implementing a more sophisticated convergence criterion can help the algorithm find better solutions.

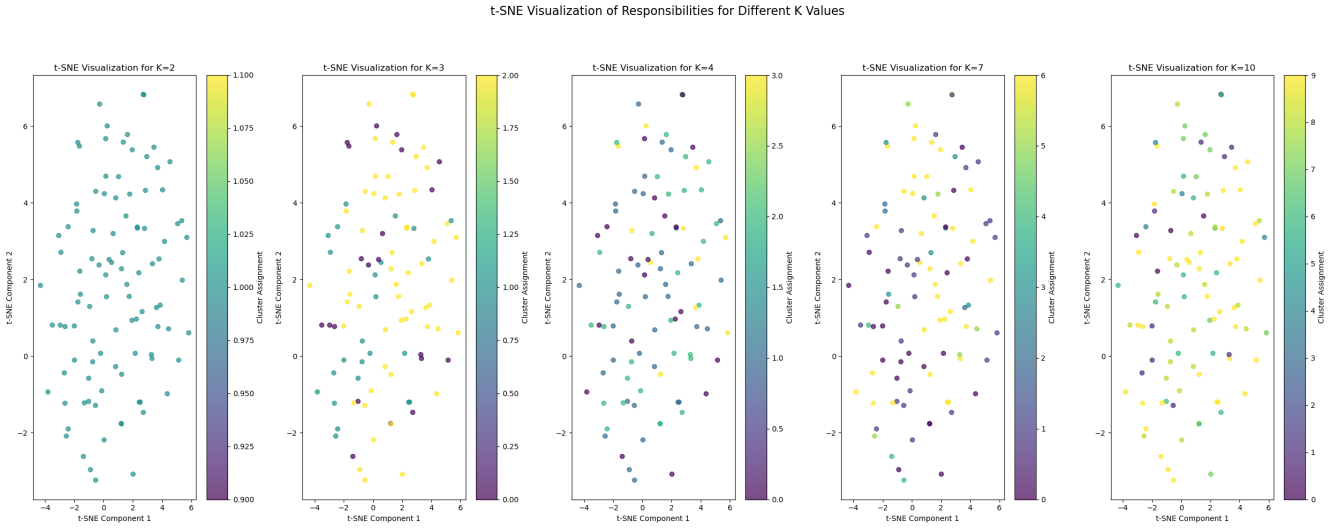

Figure 10: t-SNE Visualization of Representations for Different K values

Moreover, the resulting cluster responsibilities were visualized using t-SNE. Each t-SNE plot in Figure 10 provides a two-dimensional representation of the high-dimensional responsibility distributions, where each point corresponds to a data instance. The colors indicate the cluster assignments derived from the responsibilities, reflecting the degree of membership in each cluster.

The t-SNE visualizations for different values of K demonstrate the following:

The structure and separability of the clusters depend significantly on the choice of K. For smaller K (e.g., $K = 2$), the data points are grouped into broader clusters, resulting in less granular distinctions between responsibilities. As K increases (e.g., $K = 7, 10$), the clusters become more refined, with smaller, more localized groupings. However, overly large K may result in overfitting, where clusters start representing noise or minor variations in the data.

The t-SNE plots indicate that the clusters are reasonably well-formed for moderate values of K (e.g., $K = 3, 4, 7$). For smaller K, the clusters appear overly broad, potentially failing to capture the finer details of the data structure. For larger K, while the clusters become more distinct, some may overlap or fragment, suggesting that the increased complexity does not necessarily correspond to better interpretability or meaningful groupings.

The color gradients in the t-SNE plots reflect the responsibilities assigned to each cluster. These responsibilities indicate the likelihood of each data point belonging to a specific cluster. For $K = 2$, most points are strongly associated with one cluster or the other. As K increases, the responsibilities are distributed more finely, reflecting a higher degree of uncertainty or overlapping memberships for some points.

Python code for (e) (tsne visualization for responsibilities):

```python
from sklearn.manifold import TSNE

def visualize_responsibilities_tsne(X, pi_list, P_list, k_values):
    """
    Visualize the responsibilities for different K values using t-SNE.

    Parameters:
    X : numpy.ndarray
        The input data, shape (N, D) where N is the number of data points and D is the number of features.
    pi_list : list of numpy.ndarray
        List of mixing proportions for each K, each shape (K,).
    P_list : list of numpy.ndarray
        List of cluster parameters for each K, each shape (K, D).
    k_values : list of int
        List of K values corresponding to pi_list and P_list.
    """
    num_k = len(k_values)
    plt.figure(figsize=(25, 10))

    for idx, K in enumerate(k_values):
        # Compute responsibilities for the current K
        responsibilities = e_step(X, pi_list[idx], P_list[idx])

        # Apply t-SNE
        tsne = TSNE(n_components=2, random_state=42)
        data_tsne = tsne.fit_transform(X)

        # Get cluster assignments based on maximum responsibility
        cluster_assignments = np.argmax(responsibilities, axis=1)

        plt.subplot(1, num_k, idx + 1)
        plt.scatter(data_tsne[:, 0], data_tsne[:, 1], c=cluster_assignments, cmap='viridis', alpha=0.7)
        plt.title(f't-SNE Visualization for K={K}')
        plt.xlabel('t-SNE Component 1')
        plt.ylabel('t-SNE Component 2')
        plt.colorbar(label='Cluster Assignment')

    plt.suptitle('t-SNE Visualization of Responsibilities for Different K Values', fontsize=16)
    plt.tight_layout()
    plt.subplots_adjust(top=0.85)
    plt.show()

k_values = [2, 3, 4, 7, 10]
pi_list = [np.random.dirichlet(np.ones(K), size=1)[0] for K in k_values]
P_list = [np.random.rand(K, D) for K in k_values]

visualize_responsibilities_tsne(X, pi_list, P_list, k_values)
```

Listing 5: src/solutions/q1.py

(f) To express the log-likelihoods obtained in bits and relate it to the length of the naive encoding.

First, consider the log-likelihood of the dataset $\{x^{(n)}\}_{n=1}^N$ given model parameters θ. This is defined as:

$$\text{log-likelihood} = \log(P(\{x^{(n)}\}_{n=1}^N|\theta)).$$

Since probabilities $P(\{x^{(n)}\}_{n=1}^N|\theta)$ are typically values between 0 and 1, their logarithms are negative values (because $\log(p) < 0$ when $0 < p < 1$).

To represent this log-likelihood as an information cost in bits, we need to convert the log-likelihood from base e (natural log) to base 2. This conversion is achieved by dividing the log-likelihood by $\ln(2)$:

$$\log_2(P(\{x^{(n)}\}_{n=1}^N|\theta)) = \frac{\log(P(\{x^{(n)}\}_{n=1}^N|\theta))}{\ln(2)}.$$

To interpret this result as the total number of bits required to encode the dataset under the model, we use the negative form:

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

The negative sign is necessary because the log-likelihood is generally negative, and using the negative form converts it to a positive quantity. This positive value represents the information cost in bits or the compressed encoding length of the data according to the model.

For the naive encoding length of binary data, where each pixel or feature is represented by a single bit (0 or 1), we calculate the total bit length as:

$$\text{naive encoding length} = N \times D,$$

where N is the number of data points and D is the number of binary features per data point. This length represents the uncompressed encoding of the entire dataset.

The compression rate measures how much the model-based encoding reduces the bit length compared to the naive encoding. It is defined as:

$$\text{compression rate} = 1 - \frac{-\log_2(P(\{x^{(n)}\}_{n=1}^N|\theta))}{N \times D}.$$

In this formula: $-\log_2(P(\{x^{(n)}\}_{n=1}^N|\theta))$ represents the information cost in bits using the model. $N \times D$ is the naive encoding length.

The negative sign in the log-likelihood expression ensures that we obtain a positive bit length. This bit length can then be compared to the naive encoding length to compute the compression rate. A positive compression rate indicates that the model provides compression relative to naive encoding, and a rate close to 1 suggests a high degree of compression.

K	**Final Log-Likelihood in Bits**	**Compression Rate**
2	-4766	0.26
3	-4559	0.29
4	-4209	0.34
7	-3729	0.42
10	-3375	0.47

Table 2: Final Log-Likelihood in Bits and Compression Rate for Different Values of K

From Table 2, as K increases, both the final log-likelihood in bits and the compression rate improve. The log-likelihood becomes less negative, indicating a better model fit as the number of clusters grows, while the compression rate approaches 0.5 for $K = 10$. This trend shows that the model captures more structure in the data with higher K, resulting in improved compression relative to naive encoding.

Compared to standard compression algorithms like gzip, the compression rates achieved by this model are lower. gzip leverages exact redundancy and repeated patterns in the data, allowing for highly efficient compression, especially on structured or repetitive data. In contrast, our model-based compression relies on capturing underlying probabilistic patterns rather than exact sequence redundancy. While the model captures the statistical structure in the data, it does not reach the compression efficiency of gzip on simple repetitive data, which explains the difference in compression performance.

Python code for (f) (log-likelihood in bits and compression rates):

```python

import numpy as np

def em_algorithm_compression_rate(X, K, max_iter=20):
    """
    Parameters:
    - X: Data matrix (N x D), where N is the number of data points and D is the dimensionality.
    - K: Number of clusters/components.
    - max_iter: Maximum number of iterations for the EM algorithm.

    Returns:
    - final_log_likelihood_bits: Final log-likelihood value in bits (base 2).
    - compression_rate: Compression rate compared to the naive encoding.
    """
    # Initialize parameters
    N, D = X.shape
    pi, P = initialize_parameters(K, D)
    log_likelihoods = []

    for _ in range(max_iter):
        # E-step
        resp = e_step(X, pi, P)

        # M-step
        pi, P = m_step(X, resp)

        # Compute and store log-likelihood in natural log
        log_likelihood = compute_log_likelihood(X, pi, P)
        log_likelihoods.append(log_likelihood)

    # Convert the final log-likelihood to bits
    final_log_likelihood_bits = log_likelihoods[-1] / np.log(2)

    # Calculate naive encoding length in bits
    naive_encoding_length = N * D

    # Calculate compression rate
    compression_rate = 1 - (-final_log_likelihood_bits / naive_encoding_length)

    return final_log_likelihood_bits, compression_rate

def compute_compression_rates_for_different_K(X, K_values, max_iter=20):
    """
    Parameters:
    - X: Data matrix (N x D).
    - K_values: List of integers, where each integer is a value of K (number of clusters) to
        evaluate.
    - max_iter: Maximum number of iterations for the EM algorithm.

    Returns:
    - results: Dictionary where keys are K values and values are tuples
    (final_log_likelihood_bits, compression_rate).
    """
    results = {}
    for K in K_values:
        final_log_likelihood_bits, compression_rate = em_algorithm_compression_rate(X, K,
            max_iter)
        results[K] = (final_log_likelihood_bits, compression_rate)
        print(f"K={K}: Final log-likelihood in bits = {final_log_likelihood_bits},
        Compression rate = {compression_rate}")
    return results

K_values = [2, 3, 4, 7, 10]
results = compute_compression_rates_for_different_K(data, K_values, max_iter=20)

print(results)
```

Listing 6: src/solutions/q1.py

(g) The total cost of encoding both the data and model parameters can be computed as follows:

$$\text{Total Cost} = -\log_2 \left(P \left(\{\mathbf{x}^{(n)}\}_{n=1}^{N} \mid \theta \right) \right) + M \cdot (K - 1 + K \cdot D)$$

where:

- $-\log_2 \left(P \left(\{\mathbf{x}^{(n)}\}_{n=1}^{N} \mid \theta \right) \right)$ represents the final log-likelihood in bits, corresponding to the data encoding cost.
- M is the bit cost of storing a single floating-point value (64 bits for a `float64`).
- K is the number of clusters.
- D is the dimensionality of each data point.
- $(K - 1 + K \cdot D)$ accounts for the number of model parameters, which includes: -
- $K - 1$ mixing proportions (since they sum to 1).
- $K \cdot D$ parameters for the Bernoulli distributions associated with each cluster.

The total compression rate is then defined as the ratio of the total cost to the naive encoding length:

$$\text{Total Compression Rate} = \frac{\text{Total Cost}}{N \cdot D}$$

where: $N \cdot D$ represents the naive encoding length, assuming each data point can be represented by a single bit per pixel (for binary data).

Thus, the final outputs are the total cost and the total compression rate for each value of K.

K	**Total Cost**	**Total Compression Rate**
2	11523.68	1.8006
3	15700.66	2.4532
4	19808.04	3.0950
7	32117.22	5.0183
10	44521.13	6.9564

Table 3: Total Cost and Total Compression Rate for Different Values of K

The Table 3 shows that the total cost and compression rate increase as K becomes larger. This indicates that the model complexity and storage requirements grow with the number of clusters, as each additional cluster adds more parameters that must be stored. Compared to a compression algorithm like gzip, which is optimized for minimizing redundancy without explicitly modeling clusters, the EM model with higher K values is less efficient because it encodes both data and parameters.

As K increases, the total compression rate exceeds 1, meaning this method requires more bits than naive encoding due to the overhead of storing model parameters. This suggests that while increasing K provides a finer-grained model, it incurs higher storage costs. This result highlights a trade-off: increasing K might improve the model's fit but reduces compression efficiency, showing the balance between model complexity and storage efficiency.

Python code for (g) (Total Cost and Total Compression Rate):

```

import numpy as np
import pandas as pd

def em_algorithm_total_cost_and_compression_rate_single_run(X, K_values, max_iter=20, M=64):
    """

    Parameters:
    - X: Data matrix (N x D), where N is the number of data points and D is the dimensionality.
    - K_values: List of integers, the different values of K (number of clusters) to evaluate.
    - max_iter: Maximum number of iterations for the EM algorithm.
    - M: Cost in bits to store each parameter (default is 64 bits for a float64).

    Returns:
    - results_df: DataFrame with total compression rates for each K value.
    """
    N, D = X.shape
    naive_encoding_length = N * D
    results = {"K": [], "Total Cost": [], "Total Compression Rate": []}

    for K in K_values:
        pi, P = initialize_parameters(K, D)
        log_likelihoods = []

        # EM Algorithm
        for _ in range(max_iter):
            resp = e_step(X, pi, P)

            pi, P = m_step(X, resp)

            log_likelihood = compute_log_likelihood(X, pi, P)
            log_likelihoods.append(log_likelihood)

        # Final log-likelihood in bits
        final_log_likelihood_nats = log_likelihoods[-1]
        final_log_likelihood_bits = final_log_likelihood_nats / np.log(2)

        # Cost of encoding the model parameters
        num_params = (K - 1) + K * D
        model_cost_bits = num_params * M

        # Total cost in bits
        total_cost = -final_log_likelihood_bits + model_cost_bits

        # Total compression rate
        total_compression_rate = total_cost / naive_encoding_length

        results["K"].append(K)
        results["Total Cost"].append(total_cost)
        results["Total Compression Rate"].append(total_compression_rate)

    results_df = pd.DataFrame(results)

    return results_df

K_values = [2, 3, 4, 7, 10]

# Run the computation
results_df = em_algorithm_total_cost_and_compression_rate_single_run(X, K_values, max_iter=20,
    M=64)

print(results_df)
```

Listing 7: src/solutions/q1.py

Question 4

(a) The filtered states exhibit more variability and respond quickly to changes in observations as shown at Figure 11. This is because each filtered estimate is based only on data up to the current time step, making it more sensitive to local fluctuations. The filtered estimates appear noisier, with more frequent and sharper changes in the state values. This reflects the uncertainty in the estimate, as it does not take future information into account. The filtered states are suitable for real-time tracking, as they only require past and current observations, but they may not provide the most accurate state estimate.

Figure 11: Kalman filtering vs Kalman Smoothing Y

The smoothed states are notably smoother than the filtered states, showing fewer sharp fluctuations as shown at Figure 12. This is because the smoother considers all observations (both past and future) to estimate each state, leading to a more refined trajectory. With the benefit of future data, the smoothed estimates can correct past

fluctuations that might have been due to noise, resulting in a cleaner and more stable pattern over time.

The smoothed states are generally closer to the true underlying states, as they incorporate more information. This makes them more reliable for post-hoc analysis.

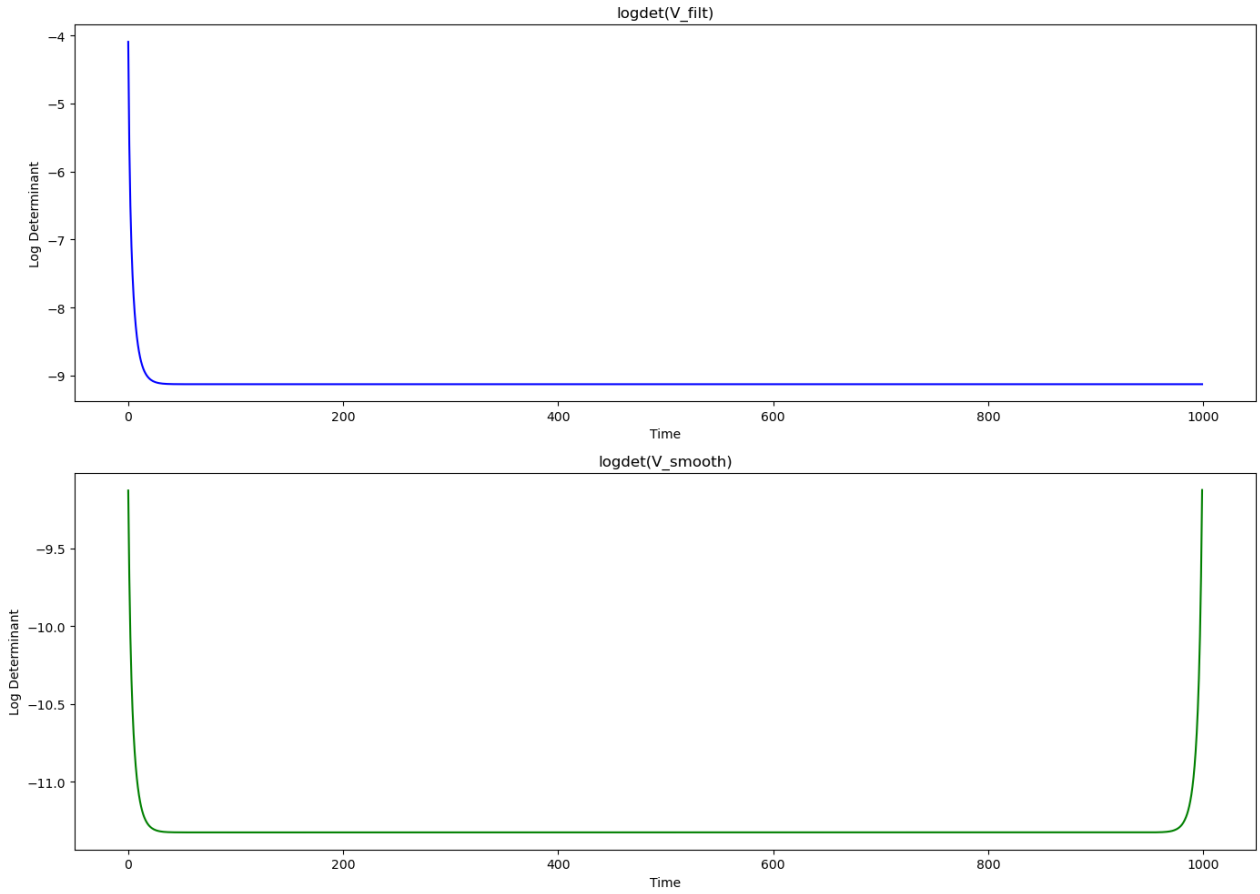

Figure 12: Kalman filtering vs Kalman Smoothing logdet V

The smoothed estimates (Y_{smooth}) are generally more accurate and certain than the filtered estimates (Y_{filt}), as they incorporate information from the entire sequence of observations.

The filtered states (Y_{filt}) are more sensitive to observation noise and local fluctuations, while the smoothed states (Y_{smooth}) average out these fluctuations, providing a better estimate of the underlying state.

The filtered states are suitable for real-time applications where only past and current data is available. In contrast, the smoothed states are more useful for retrospective analysis, where the entire dataset can be used to improve the accuracy of state estimates.

Python code for (a):

```

# Define parameters
X = np.loadtxt('/Users/richardhuang/Documents/GitHub/Projects/Probabilistc
    Summartive/ssm_spins.txt').T

# Define given 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)]
])

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

Q = np.eye(4) - A @ A.T
R = np.eye(5)
y_init = np.zeros(4)
Q_init = np.eye(4)

def logdet(A):
    return 2 * np.sum(np.log(np.diag(np.linalg.cholesky(A))))


Y_filt, V_filt, _, L_filt = run_ssm_kalman(X, y_init, Q_init, A, Q, C, R , 'filt' )
y_smooth, V_smooth, V_joint, L_smooth = run_ssm_kalman(X, y_init, Q_init, A, Q, C, R , 'smooth' )

# Generate the plots
plt.figure(figsize=(12, 10))

plt.subplot(2, 1, 1)
plt.plot(y_filt.T)
plt.title("Filtered States (Y_filt)")
plt.xlabel("Time")
plt.ylabel("State Value")

plt.subplot(2, 1, 2)
plt.plot(y_smooth.T)
plt.title("Smoothed States (Y_smooth)")
plt.xlabel("Time")
plt.ylabel("State Value")

# Calculate log determinants for V_filt and V_smooth
logdet_V_filt = [logdet(V_filt[t]) for t in range(len(V_filt))]
logdet_V_smooth = [logdet(V_smooth[t]) for t in range(len(V_smooth))]

plt.figure(figsize=(14, 10))
plt.subplot(2,1, 1)
plt.plot(logdet_V_filt, color='blue')
plt.title("logdet(V_filt)")
plt.xlabel("Time")
plt.ylabel("Log Determinant")

plt.subplot(2,1, 2)
plt.plot(logdet_V_smooth, color='green')
plt.title("logdet(V_smooth)")
plt.xlabel("Time")
plt.ylabel("Log Determinant")

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

Listing 8: src/solutions/q1.py

(b) To derive the M-step update equations for R_{new} and Q_{new}.

First, consider R_{new}, The observation model is given by:

$$\mathbf{x}_t = C\mathbf{y}_t + \mathbf{v}_t, \quad \mathbf{v}_t \sim \mathcal{N}(0, R)$$

where R is the observation noise covariance matrix. The goal is to update R using the current estimates of the latent states $\mathbf{y}_t$ and the observations $\mathbf{x}_t$.

The optimal R_{new} can be derived by computing the expected squared difference between $\mathbf{x}_t$ and $C\mathbf{y}_t$, averaged over time. This expectation is:

$$R_{\text{new}} = \frac{1}{T}\sum_{t=1}^{T} \mathbb{E}\left[(\mathbf{x}_t - C\mathbf{y}_t)(\mathbf{x}_t - C\mathbf{y}_t)^\top\right].$$

Expanding this expectation:

$$R_{\text{new}} = \frac{1}{T}\left(\sum_{t=1}^{T}\mathbf{x}_t\mathbf{x}_t^\top - \sum_{t=1}^{T}\mathbf{x}_t\langle\mathbf{y}_t\rangle^\top C^\top - C\sum_{t=1}^{T}\langle\mathbf{y}_t\rangle\mathbf{x}_t^\top + C\sum_{t=1}^{T}\langle\mathbf{y}_t\mathbf{y}_t^\top\rangle C^\top\right).$$

However, by choosing C_{new} such that $C_{\text{new}} = \left(\sum_{t=1}^{T}\mathbf{x}_t\langle\mathbf{y}_t\rangle^\top\right)\left(\sum_{t=1}^{T}\langle\mathbf{y}_t\mathbf{y}_t^\top\rangle\right)^{-1}$, we simplify R_{new} to:

$$R_{\text{new}} = \frac{1}{T}\left(\sum_{t=1}^{T}\mathbf{x}_t\mathbf{x}_t^\top - \left(\sum_{t=1}^{T}\mathbf{x}_t\langle\mathbf{y}_t\rangle^\top\right)C_{\text{new}}^\top\right).$$

For Q_{new}, The state transition model is given by:

$$\mathbf{y}_t = A\mathbf{y}_{t-1} + \mathbf{w}_t, \quad \mathbf{w}_t \sim \mathcal{N}(0, Q)$$

where Q is the process noise covariance matrix. To update Q, we compute the expected squared difference between $\mathbf{y}_t$ and $A\mathbf{y}_{t-1}$, averaged over time. This expectation is:

$$Q_{\text{new}} = \frac{1}{T-1}\sum_{t=2}^{T} \mathbb{E}\left[(\mathbf{y}_t - A\mathbf{y}_{t-1})(\mathbf{y}_t - A\mathbf{y}_{t-1})^\top\right].$$

Expanding this expectation:

$$Q_{\text{new}} = \frac{1}{T-1}\left(\sum_{t=2}^{T}\langle\mathbf{y}_t\mathbf{y}_t^\top\rangle - \sum_{t=2}^{T}\langle\mathbf{y}_t\rangle\langle\mathbf{y}_{t-1}^\top\rangle A^\top - A\sum_{t=2}^{T}\langle\mathbf{y}_{t-1}\rangle\langle\mathbf{y}_t\rangle^\top + A\sum_{t=2}^{T}\langle\mathbf{y}_{t-1}\mathbf{y}_{t-1}^\top\rangle A^\top\right).$$

By choosing A_{new} such that $A_{\text{new}} = \left(\sum_{t=2}^{T}\langle\mathbf{y}_t\rangle\langle\mathbf{y}_{t-1}\rangle^\top\right)\left(\sum_{t=2}^{T}\langle\mathbf{y}_{t-1}\mathbf{y}_{t-1}^\top\rangle\right)^{-1}$, we simplify Q_{new} to:

$$Q_{\text{new}} = \frac{1}{T-1}\left(\sum_{t=2}^{T}\langle\mathbf{y}_t\mathbf{y}_t^\top\rangle - \left(\sum_{t=2}^{T}\langle\mathbf{y}_t\rangle\langle\mathbf{y}_{t-1}^\top\rangle\right)A_{\text{new}}^\top\right).$$

Thus, we obtain the simplified M-step update equations for R and Q:

$$R_{\text{new}} = \frac{1}{T}\left(\sum_{t=1}^{T}\mathbf{x}_t\mathbf{x}_t^\top - \left(\sum_{t=1}^{T}\mathbf{x}_t\langle\mathbf{y}_{t-1}^\top\rangle\right)C_{\text{new}}^\top\right)$$

$$Q_{\text{new}} = \frac{1}{T-1}\left(\sum_{t=2}^{T}\langle\mathbf{y}_t\mathbf{y}_t^\top\rangle - \left(\sum_{t=2}^{T}\langle\mathbf{y}_t\rangle\langle\mathbf{y}_{t-1}^\top\rangle\right)A_{\text{new}}^\top\right).$$

The two plots below display the progression of the log-likelihood over 100 iterations of the Expectation-Maximization (EM) algorithm under two different initialization conditions. In the first plot, the EM algorithm is initialized with the generating (true) parameters, while in the second plot, it is initialized with 10 different sets of random parameters.

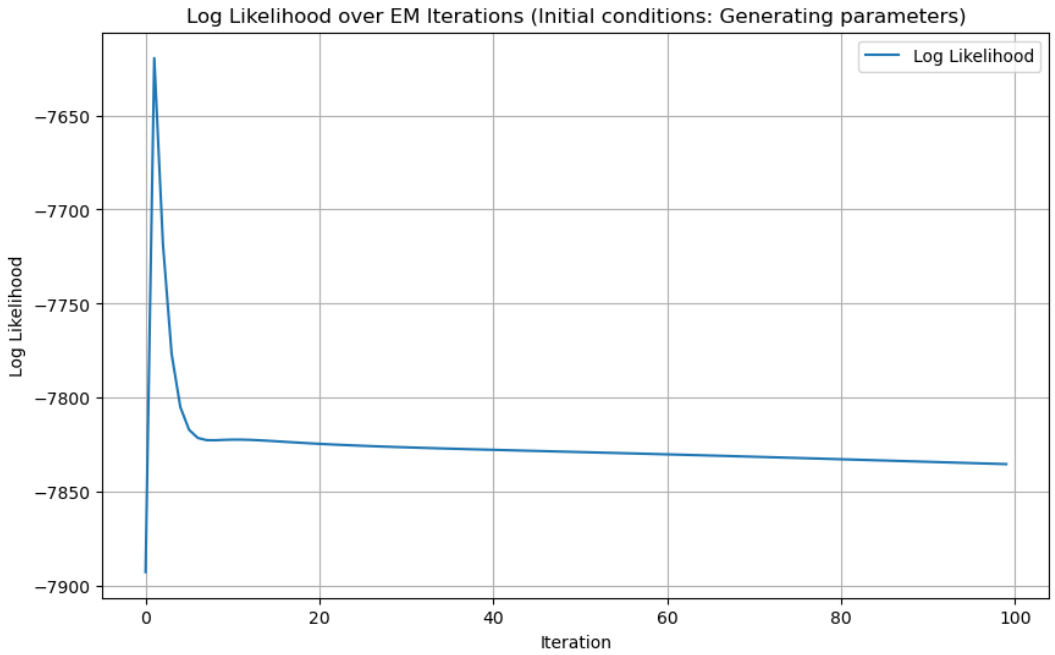

Figure 13: EM Iterations (Initial Conditions: Generating parameters)

In the first Figure 13, we observe a rapid increase in log-likelihood within the initial few iterations, followed by a gradual convergence to a stable value. This behavior suggests that starting with parameters close to the true values allows the EM algorithm to quickly reach a high log-likelihood, as it starts near an optimal point in the parameter space. After the initial rapid increase, the log-likelihood stabilizes, indicating that the algorithm has likely converged and further iterations provide minimal improvement.

The second Figure 14, where EM is initialized with random parameters for 10 different runs, shows a different pattern. Each line represents a separate run with a unique random initialization. Here, the log-likelihoods begin at much lower values, reflecting the random starting points far from the optimum. Over the iterations, each run's log-likelihood gradually increases, but at a slower rate compared to the first plot. Despite the varied starting points, all runs eventually converge to similar log-likelihood values, demonstrating the robustness of the EM algorithm in finding a high-likelihood solution regardless of initialization. However, the convergence process is slower and less direct compared to starting from the true parameters.

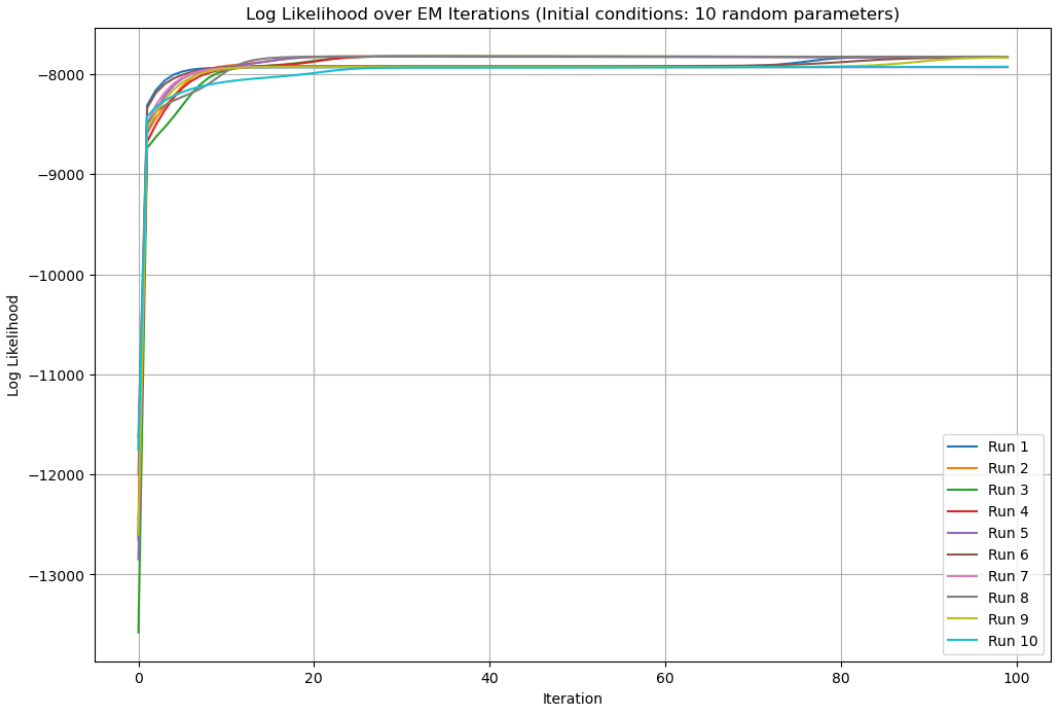

Figure 14: EM Iterations (Initial Conditions: 10 random parameters)

Python code for (b)

```
1
2  def cellsum(C):
3      """
4      Sums a list of matrices element-wise.
5      :param C: List of matrices (numpy arrays) to sum
6      :return: Element-wise sum of all matrices in C
7      """
8      return np.sum(np.stack(C, axis=2), axis=2)
9
10 def em_algorithm(X, A_init, Q_init, C_init, R_init, y_init, Q_0, num_iterations=100):
11     """
12     EM algorithm for learning the parameters A, Q, C, and R.
13     :param X: Observation data, shape [d, T]
14     :param A_init: Initial A matrix
15     :param Q_init: Initial Q matrix
16     :param C_init: Initial C matrix
17     :param R_init: Initial R matrix
18     :param y_init: Initial state vector
19     :param Q_0: Initial state covariance
20     :param num_iterations: Number of EM iterations
21     :return: Updated A, Q, C, R matrices and log-likelihoods
22     """
23     A, Q, C, R = A_init, Q_init, C_init, R_init
24     T = X.shape[1]
25     log_likelihoods = []
26
27     for i in range(num_iterations):
28         y_hat, V_hat, V_joint, log_likelihood = run_ssm_kalman(X, y_init, Q_0, A, Q, C, R,
               mode='smooth')
29         log_likelihoods.append(np.sum(log_likelihood))
30
31         # M-step: Update parameters A, Q, C, and R
32
33         # Update C
34         C_new = (cellsum([np.outer(X[:, t], y_hat[:, t]) for t in range(T)]) @
35                  np.linalg.inv(cellsum([V_hat[t] + np.outer(y_hat[:, t], y_hat[:, t])
```

```
                for t in range(T)])))

        # Update R
        R_new = (1 / T) * (cellsum([np.outer(X[:, t], X[:, t]) for t in range(T)]) -
                           cellsum([np.outer(X[:, t], y_hat[:, t]) for t in range(T)]) @ C_new.T)

        # Update A
        A_new = (cellsum([np.outer(y_hat[:, t + 1], y_hat[:, t]) for t in range(T - 1)]) @
                 np.linalg.inv(cellsum([V_hat[t] + np.outer(y_hat[:, t], y_hat[:, t])
                 for t in range(T - 1)])))

        # Update Q
        Q_new = (1 / (T - 1)) * (cellsum([V_hat[t + 1] + np.outer(y_hat[:, t + 1], y_hat[:, t +
            1])
        for t in range(T - 1)]) -
        cellsum([np.outer(y_hat[:, t + 1], y_hat[:, t]) for t in range(T - 1)]) @ A_new.T)
        A, Q, C, R = A_new, Q_new, C_new, R_new

    return A, Q, C, R, log_likelihoods

A_em, Q_em, C_em, R_em, log_likelihoods_para =
em_algorithm(X, A, Q, C, R, y_init, Q_0)

plt.figure(figsize=(10, 6))
plt.plot(log_likelihoods_para, label='Log Likelihood')
plt.xlabel('Iteration')
plt.ylabel('Log Likelihood')
plt.title('Log Likelihood over EM Iterations (Initial conditions: Generating parameters)')
plt.legend()
plt.grid()
plt.show()

# Run EM with 10 random initializations
all_log_likelihoods = []

for i in range(10):
    A_rand = np.random.randn(*A.shape)
    Q_rand = np.eye(A.shape[0])
    C_rand = np.random.randn(*C.shape)
    R_rand = np.eye(X.shape[0])

    A_ra, Q_ra, C_ra, R_ra, log_likelihoods_rand
    = em_algorithm(X, A_rand, Q_rand, C_rand, R_rand, y_init, Q_0)

    all_log_likelihoods.append(log_likelihoods_rand)

plt.figure(figsize=(12, 8))

for i, log_likelihoods in enumerate(all_log_likelihoods):
    plt.plot(log_likelihoods, label=f'Run {i+1}')

plt.xlabel('Iteration')
plt.ylabel('Log Likelihood')
plt.title('Log Likelihood over EM Iterations (Initial conditions: 10 random parameters)')
plt.legend()
plt.grid()
plt.show()
```

Listing 9: src/solutions/q1.py

(c)

Question 5

(a) To derive the Maximum Likelihood (ML) estimates for the transition probabilities $\psi(\alpha, \beta)$ and the stationary distribution $\phi(\gamma)$, we proceed as follows:

The transition probability $\psi(\alpha, \beta)$ represents the probability of observing symbol α immediately after symbol β.

Estimate $\psi(\alpha, \beta) = p(s_i = \alpha | s_{i-1} = \beta)$ based on observed data by maximizing the likelihood of observed transitions.

Let:

- $N(\alpha, \beta)$: the count of observed pairs where β is followed by α,
- $N(\beta)$: the total number of times β appears as a preceding symbol in any pair.

To maximize the likelihood of the observed data, we define the likelihood function for all observed transitions as:

$$L(\psi) = \prod_{\alpha,\beta} \psi(\alpha, \beta)^{N(\alpha,\beta)}$$

where $\psi(\alpha, \beta)$ is the probability of observing symbol α given that the previous symbol was β.

Taking the logarithm of the likelihood function, we get the log-likelihood:

$$\log L(\psi) = \sum_{\alpha,\beta} N(\alpha, \beta) \log \psi(\alpha, \beta)$$

Since the probabilities must sum to 1 for each preceding symbol β, we have the constraint:

$$\sum_{\alpha} \psi(\alpha, \beta) = 1, \quad \forall\, \beta$$

To maximize $\log L(\psi)$ subject to the above constraint, we introduce a Lagrange multiplier λ_β for each β:

$$\mathcal{L} = \sum_{\alpha,\beta} N(\alpha, \beta) \log \psi(\alpha, \beta) + \sum_{\beta} \lambda_\beta \left(1 - \sum_{\alpha} \psi(\alpha, \beta)\right)$$

Taking the derivative of $\mathcal{L}$ with respect to $\psi(\alpha, \beta)$ and setting it to zero:

$$\frac{\partial \mathcal{L}}{\partial \psi(\alpha, \beta)} = \frac{N(\alpha, \beta)}{\psi(\alpha, \beta)} - \lambda_\beta = 0$$

Rearranging, we find:

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

Using the constraint $\sum_\alpha \psi(\alpha, \beta) = 1$, we find:

$$\sum_{\alpha} \frac{N(\alpha, \beta)}{\lambda_\beta} = 1$$

$$\lambda_\beta = N(\beta)$$

Thus, the ML estimate for $\psi(\alpha, \beta)$ is:

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

The stationary distribution $\phi(\gamma)$ represents the long-term probability of observing symbol γ in the text. Let:

- $N(\gamma)$: the total count of occurrences of symbol γ,
- N: the total number of symbols in the text.

The probability $\phi(\gamma)$ can be estimated by the relative frequency of γ in the text:

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

where:

- $N(\gamma)$ is the count of symbol γ in the entire text.
- $N = \sum_{\gamma} N(\gamma)$, the total number of symbols in the text.

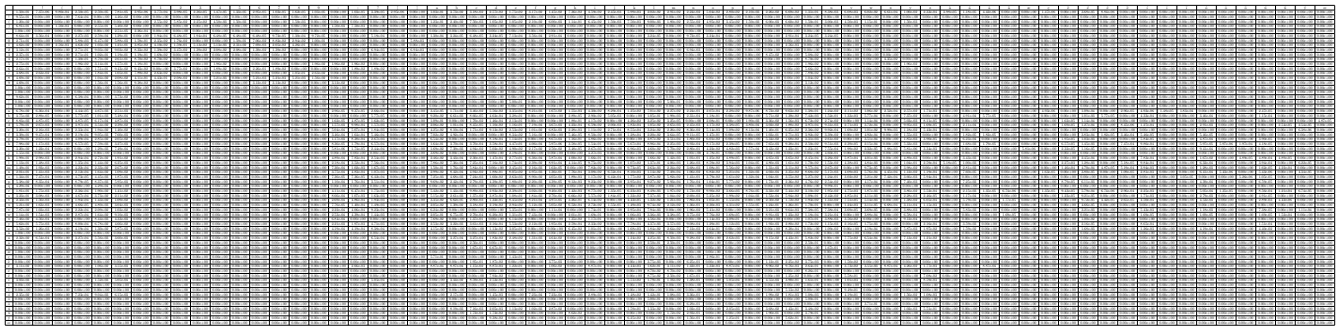

Table 4: Transition Matrix Ψ (Sorry.. the numbers are too many to display here)

(b) The latent variables $\sigma(s)$ represent the mappings of each symbol in the encrypted text to symbols in the decrypted text. Since we assume a uniform prior distribution over all possible permutations, each permutation is equally likely, which means there is no inherent dependency structure enforced by the prior.

However, **the latent variables $\sigma(s)$ are not independent**. This is because they represent a permutation. In a permutation, choosing one mapping (e.g., $\sigma(a) = s$) affects the choices for the remaining mappings (e.g., $\sigma(b)$ cannot also be s if it's already mapped to a). Therefore, while the prior is uniform, the constraint that each symbol maps to a unique counterpart means that the latent variables $\sigma(s)$ are not independent.

The joint probability of the encrypted sequence $e_1 e_2 \cdots e_n$ given the permutation σ can be expressed in terms of the transition probabilities ψ and the stationary distribution ϕ, which were estimated from English text in part (a).

Define $d_i = \sigma^{-1}(e_i)$ to be the decrypted symbol corresponding to e_i under the permutation σ.

The joint probability of the sequence $e_1 e_2 \cdots e_n$ given σ is:

$$p(e_1 e_2 \cdots e_n | \sigma) = p(d_1 d_2 \cdots d_n) = \phi(d_1) \prod_{i=2}^{n} \psi(d_i | d_{i-1})$$

where:

- $\phi(d_1)$ is the stationary probability of the first decrypted symbol d_1,
- $\psi(d_i | d_{i-1})$ is the transition probability from d_{i-1} to d_i.

This joint probability captures the likelihood of observing the decrypted sequence $d_1 d_2 \cdots d_n$, which corresponds to the encrypted sequence $e_1 e_2 \cdots e_n$ under the permutation σ.

(c) Here, the proposal $\sigma \to \sigma'$ is defined by randomly choosing two symbols s and s' and swapping their mappings in σ. Since both s and s' are chosen randomly, each pair of symbols has an equal probability of being selected.

Let N be the total number of symbols. The probability of selecting any specific pair (s, s') is:

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

because there are $\binom{N}{2}$ ways to choose two distinct symbols from N symbols.

This proposal probability $S(\sigma \to \sigma')$ does not depend on the specific configurations of σ or σ'; it is only dependent on the number of symbols N.

The Metropolis-Hastings acceptance probability $A(\sigma \to \sigma')$ is given by:

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

where:

- $\pi(\sigma)$ is the target probability (posterior) of the current permutation σ,
- $S(\sigma \to \sigma')$ is the proposal probability for moving from σ to σ',
- $S(\sigma' \to \sigma)$ is the reverse proposal probability from σ' back to σ.

Since $S(\sigma \to \sigma') = S(\sigma' \to \sigma) = \frac{2}{N(N-1)}$ (symmetric proposal), the acceptance probability simplifies to:

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

Thus, the acceptance probability depends only on the ratio of the posterior probabilities $\pi(\sigma')$ and $\pi(\sigma)$.

- If $\pi(\sigma') > \pi(\sigma)$, then σ' has a higher probability than σ, and the move is always accepted.
- If $\pi(\sigma') < \pi(\sigma)$, the move is accepted with probability $\frac{\pi(\sigma')}{\pi(\sigma)}$.

Therefore,
$S(\sigma \to \sigma') = \frac{2}{N(N-1)}$, independent of σ and σ'.
$A(\sigma \to \sigma') = \min\left(1, \frac{\pi(\sigma')}{\pi(\sigma)}\right)$, dependent only on the ratio $\frac{\pi(\sigma')}{\pi(\sigma)}$.

(d) To implement the Metropolis-Hastings (MH) sampler for decrypting the provided encrypted text. Below is a description of the process:

Initialization

Defined a `Decrypter` class that takes a decryption dictionary to map encrypted symbols to their decrypted counterparts. The `initialize_parameters_by_frequency` function was used to create an initial mapping based on the frequency of symbols in the encrypted text. This mapping was derived from the frequency of letters in the English language, which helps in starting the decryption process with a reasonable guess.

Transition Probability Calculation

The `calculate_transition_probs` function was implemented to compute the transition probabilities between symbols in the training text. This involved counting the occurrences of each symbol and the pairs of consecutive symbols, applying Laplace smoothing to avoid zero probabilities.

Log-Likelihood Calculation

The `compute_log_likelihood` function was created to evaluate the likelihood of the decrypted text based on the transition probabilities. This function sums the log probabilities of the transitions in the decrypted text.

Metropolis-Hastings Algorithm

The `metropolis_hastings_decrypt` function was implemented to run the MH algorithm. It starts with an initial mapping and iteratively proposes new mappings by swapping two random symbols. For each proposed mapping, the new decrypted text is generated, and its log-likelihood is computed. The new mapping is accepted if it improves the score or based on a probability that considers the temperature of the system (simulated annealing). The temperature is gradually decreased to allow the algorithm to converge over time.

Logging Progress

The algorithm was run for 5000 times, after every 100 iterations, the current decryption of the first 60 symbols is logged.

MH Iteration	Current Decryption
0	fwxm3x3hdwt1 x2wrxmh 1xydqw1 2eq1x312 gxm3x42ln1 xt2y1xm1xgh
100	o2w4243joa102nor2w3012mjdo10ned1241n0g2w42qnli102anm12w12g3
200	e3 s4 4gj3r10 n32 sg01 mjd310nid1 41n0o s4 qnla10 rnm1 s1 og
300	e3 s4 4gh3r0y n32 sgy0 1hd30ynid0 40nyo s4 fnla0y rn10 s0 og
400	e0 s4 4ch0o3y n0d scy3 1h203yni23 43nyt s4 knla3y on13 s3 tc
500	e4 s1 1ch43of n4d scfo rhg4ofnigo 1onft s1 knlaof 3nro so tc
600	o4 s1 1ai4vef n4d safe rig4efnpge 1enft s1 kncuef vnre se ta
700	o3 ml lai3jes n3d mase vig3esnbge lenst ml kncues jnve me ta
800	o3 mg gai3res n3d mase vil3esnble genst mg knches rnve me ta
900	on mg gainres und mase vilnesuble geust mg kuches ruve me ta
1000	on py yaincer und pare vilnerufle yeurt py kusher cuve pe ta
1100	on my yaincer und mare vilneruble yeurt my kusher cuve me ta
1200	an my yoincer und more vilneruble yeurt my kusher cuve me to
1300	an my yoin1er und more vilneruble yeurt my cusher 1uve me to
1400	an cy yoin1er und core vilneruble yeurt cy musher 1uve ce to
1500	an cy yoinker und core vilneruble yeurt cy musher kuve ce to
1600	an cy yoinker und core vilneruble yeurt cy musher kuve ce to
1700	an by yoinker und bore vilneruple yeurt by musher kuve be to
1800	an by yuinker ond bure vilnerople yeort by mosher kove be tu
1900	an by yuinger ond bure vilneromle yeort by posher gove be tu
2000	an by yuinger ond bure vilneromle yeort by fosher gove be tu
2100	an by yuinger ond bure vilneromle yeort by fosher gove be tu
2200	on my yuinger and mure vilnerable years my father gave me su
2300	on my yuinger and mure vilnerable years my father gave me su
2400	on my yuinger and mure vilnerable years my father gave me su
2500	on my yuinger and mure vilnerable years my father gave me su
2600	on my yuinger and mure vilnerable years my father gave me su
2700	on my yuinger and mure vilnerable years my father gave me su
2800	on my yiunger and mire vulnerable years my father gave me si
2900	in my younger and more vulnerable years my father gave me so
3000	in my younger and more vulnerable years my father gave me so
3100	in my younger and more vulnerable years my father gave me so
3200	in my younger and more vulnerable years my father gave me so
3300	in my younger and more vulnerable years my father gave me so
3400	in my younger and more vulnerable years my father gave me so
3500	in my younger and more vulnerable years my father gave me so
3600	in my younger and more vulnerable years my father gave me so
3700	in my younger and more vulnerable years my father gave me so
3800	in my younger and more vulnerable years my father gave me so
3900	in my younger and more vulnerable years my father gave me so
4000	in my younger and more vulnerable years my father gave me so
4100	in my younger and more vulnerable years my father gave me so
4200	in my younger and more vulnerable years my father gave me so
4300	in my younger and more vulnerable years my father gave me so
4400	in my younger and more vulnerable years my father gave me so
4500	in my younger and more vulnerable years my father gave me so
4600	in my younger and more vulnerable years my father gave me so
4700	in my younger and more vulnerable years my father gave me so
4800	in my younger and more vulnerable years my father gave me so
4900	in my younger and more vulnerable years my father gave me so
5000	in my younger and more vulnerable years my father gave me so

Table 5: Decryption Results

Python code for (d): Report the current decryption

```

from collections import defaultdict, Counter
import random
from collections import Counter, defaultdict

url = 'https://www.gutenberg.org/files/2600/2600-0.txt'
response = requests.get(url)
text = response.text

random.seed(42)

class Decrypter:
    def __init__(self, decryption_dict):
        self.decryption_dict = decryption_dict

    def decrypt(self, encrypted_message):
        return ''.join(self.decryption_dict.get(char, char) for char in encrypted_message)

def initialize_parameters_by_frequency(encrypted_text):
    """Initialize mapping based on English letter frequencies."""
    # English letter frequencies (including space, punctuation, and numbers)
    english_freq_order = " etaoinshrdlcumwfgypbvkjxqz0123456789.,!?"

    all_symbols = set(encrypted_text)

    mapping = {}

    # First map the symbols that exist in our frequency order
    available_targets = list(english_freq_order)
    for symbol in all_symbols:
        if len(available_targets) > 0:
            mapping[symbol] = available_targets.pop(0)
        else:
            mapping[symbol] = symbol

    return mapping

def calculate_transition_probs(text):
    """Calculate transition probabilities with smoothing."""
    transitions = defaultdict(lambda: defaultdict(lambda: 0.01))
    char_counts = defaultdict(lambda: 0)

    # Count occurrences
    for i in range(len(text)-1):
        curr, next_char = text[i], text[i+1]
        transitions[curr][next_char] += 1
        char_counts[curr] += 1

    # Normalize to get probabilities
    for char in transitions:
        total = sum(transitions[char].values())
        for next_char in transitions[char]:
            transitions[char][next_char] /= total

    return transitions

def compute_log_likelihood(text, transitions):
    """Compute log likelihood of text using transition probabilities."""
    log_prob = 0
    for i in range(len(text)-1):
        curr, next_char = text[i], text[i+1]
        prob = transitions[curr][next_char]
        log_prob += np.log(max(prob, 1e-10))
    return log_prob

def metropolis_hastings_decrypt(encrypted_text, transitions, n_iterations=10000,
    report_interval=100):
    """Run Metropolis-Hastings algorithm with simulated annealing."""
    current_mapping = initialize_parameters_by_frequency(encrypted_text)
```

```
    current_decrypter = Decrypter(current_mapping)

    current_decrypt = current_decrypter.decrypt(encrypted_text)
    current_score = compute_log_likelihood(current_decrypt, transitions)

    best_score = current_score
    best_mapping = current_mapping.copy()
    best_decrypt = current_decrypt

    decryption_log = [(0, current_decrypt[:60])]

    temp = 1.0
    cooling_rate = 0.9999

    symbols = list(set(encrypted_text))

    for i in range(n_iterations):
        new_mapping = current_mapping.copy()
        s1, s2 = random.sample(symbols, 2)
        new_mapping[s1], new_mapping[s2] = new_mapping[s2], new_mapping[s1]

        new_decrypter = Decrypter(new_mapping)
        new_decrypt = new_decrypter.decrypt(encrypted_text)
        new_score = compute_log_likelihood(new_decrypt, transitions)

        # Accept or reject new mapping
        score_diff = new_score - current_score
        if score_diff > 0 or random.random() < np.exp(score_diff / temp):
            current_mapping = new_mapping
            current_score = new_score
            current_decrypt = new_decrypt

            if current_score > best_score:
                best_score = current_score
                best_mapping = current_mapping.copy()
                best_decrypt = current_decrypt

        if (i + 1) % report_interval == 0:
            decryption_log.append((i + 1, current_decrypt[:60]))

        temp *= cooling_rate

    return best_mapping, best_decrypt, decryption_log

def run_decryption(encrypted_message, training_text):
    """Run the decryption process and display progress."""
    # Calculate transition probabilities from training text
    transitions = calculate_transition_probs(training_text)

    # Run Metropolis-Hastings for 5000 iterations
    _, decrypted_text, log = metropolis_hastings_decrypt(
        encrypted_message,
        transitions,
        n_iterations=5000,
        report_interval=100)
    print("\nMH Iteration | Current Decryption")
    print("-" * 80)
    for iteration, text in log:
        print(f"{iteration:>10} | {text}")

    return decrypted_text

file_path = '/Users/richardhuang/Documents/GitHub/Projects/Probabilistc Summartive/message.txt'
with open(file_path, 'r') as file:
    encrypted_message = file.read().strip()

decrypted_result = run_decryption(
    encrypted_message=encrypted_message,
    training_text=text )
```

Listing 10: solutions/q1.py

(e) The presence of zero values in $\psi(\alpha, \beta)$ can affect the ergodicity of the Markov chain. If these zero values lead to isolated states, the chain becomes reducible and thus non-ergodic, as it would not converge to a unique stationary distribution.

To prove ergodicity, we must show that the chain remains irreducible and aperiodic. If the chain is still ergodic, we can demonstrate that all states can be reached from any other state, and the greatest common divisor of the lengths of all possible return paths is one.

If the chain is found to be non-ergodic, we can restore ergodicity by adjusting transition probabilities to ensure all states are reachable, introducing dummy states to connect isolated states, or reparameterizing α and β to ensure all $\psi(\alpha, \beta)$ values are positive.

(f) In analyzing the decoding approach, it becomes clear that relying solely on symbol probabilities, rather than transition probabilities, may not be sufficient for effective decoding. While symbol probabilities provide insights into the likelihood of individual symbols appearing in the text, they fail to account for the context in which these symbols occur. For instance, let $P(x)$ represent the probability of a symbol x occurring in the text. This probability alone does not capture the relationships between symbols. Language is inherently sequential, and the meaning of a symbol often depends on the symbols that precede or follow it. Therefore, using only symbol probabilities can lead to ambiguities and incorrect interpretations.

If we were to employ a second-order Markov chain, which considers the probabilities of pairs of consecutive symbols, we would enhance the contextual information captured in the model. This approach allows for the influence of the previous two symbols on the current symbol to be taken into account. Mathematically, this can be represented as:

$$P(x_t|x_{t-1}, x_{t-2}) = \frac{C(x_{t-2}, x_{t-1}, x_t)}{C(x_{t-2}, x_{t-1})}$$

where $C(a, b)$ denotes the count of occurrences of the symbols a and b together. However, several challenges may arise with this method. The state space grows significantly, necessitating more data to accurately estimate the transition probabilities between pairs of symbols, which can lead to sparse data issues, especially in smaller datasets. Additionally, the increased number of states and transitions results in higher computational costs during both training and inference phases, and there is a risk of overfitting the model to the training data, which may not generalize well to unseen text.

Moreover, if the encryption scheme permits two different symbols to be mapped to the same encrypted value, this introduces ambiguity in the decoding process. For example, if symbols a and b both map to the same encrypted value y, we cannot distinguish between them during decoding. This situation can lead to multiple valid interpretations of the decrypted text, making it challenging to determine the correct original symbols. Consequently, the model may struggle to disambiguate between the possible original symbols, resulting in incorrect or nonsensical outputs.

When considering languages like Chinese, which have a vast number of unique symbols (over 10,000), the challenges become even more pronounced. The high dimensionality increases the complexity of the model, making it difficult to estimate probabilities accurately without a large corpus of text. For instance, if we denote the set of symbols as S and the number of unique symbols as $|S| > 10000$, the number of possible pairs in a second-order Markov chain becomes $|S|^2$, leading to a combinatorial explosion in the state space. Additionally, Chinese characters can have different meanings based on context, and a simple Markov model may not effectively capture these nuances. The likelihood of encountering certain pairs or sequences of symbols decreases, leading to sparse data issues that can hinder the model's performance.

In summary, while symbol probabilities provide a foundational understanding of the text, they are insufficient for effective decoding without considering the context provided by transitions. A second-order Markov chain can improve context capture but introduces complexity and potential overfitting. Allowing multiple symbols to map to the same encrypted value creates ambiguity, complicating the decoding process. Finally, applying this approach to languages with a large symbol set, such as Chinese, presents significant challenges in terms of data sparsity and contextual understanding.

Question 6

(a) The sample plots produced by toysample with 200 and 500 iterations are shown below in Figure 15 and Figure 16. The code for the four files are also shown below.

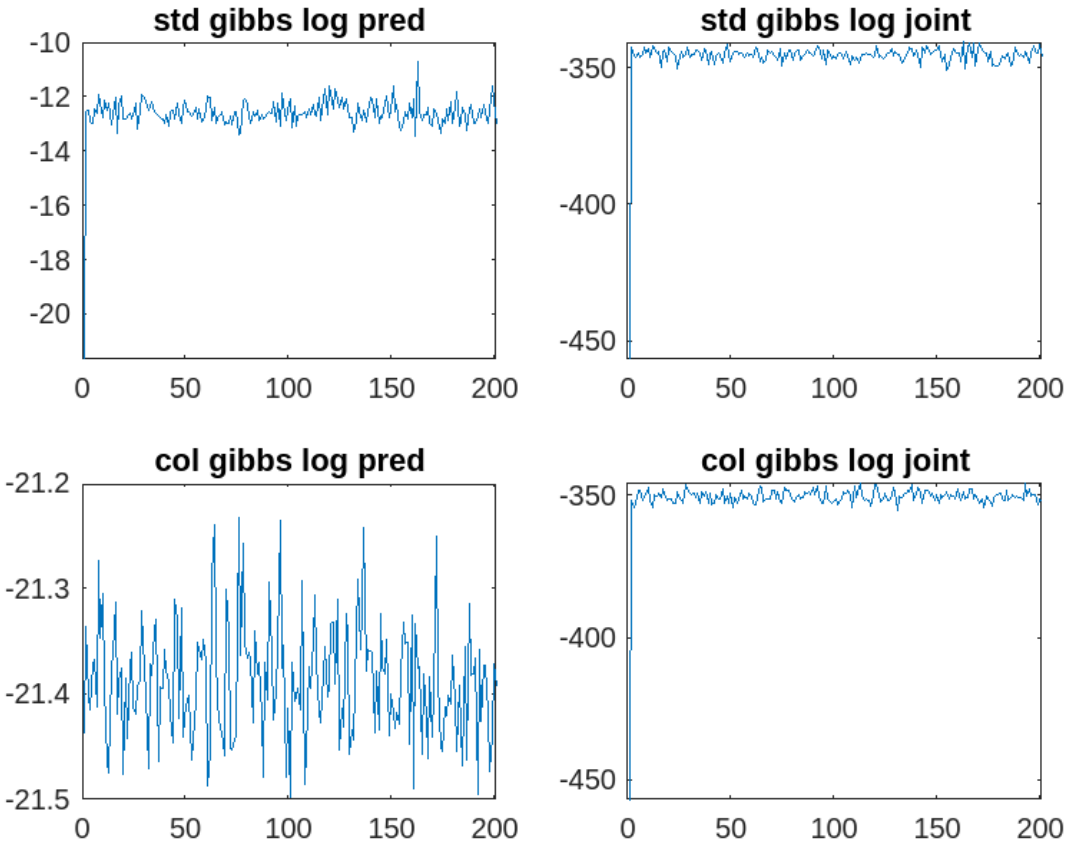

Figure 15: Sample plots produced by toyexample try 200 iterations

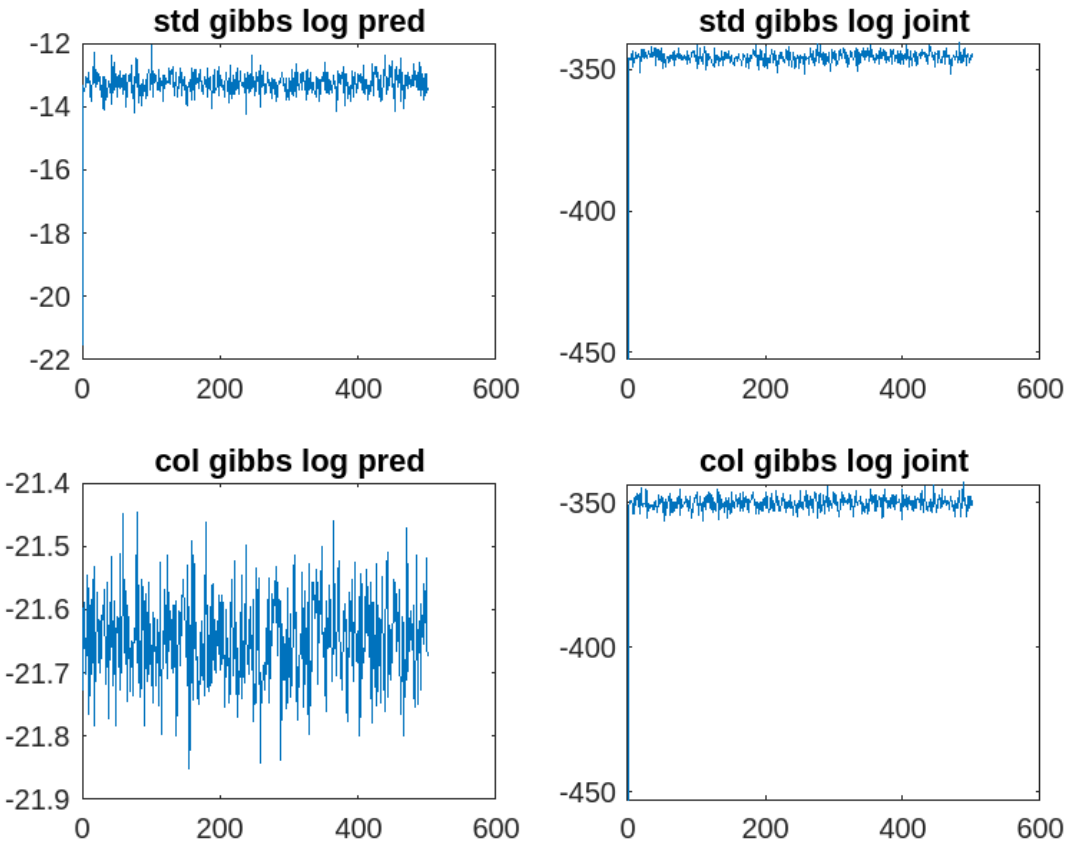

Figure 16: Sample plots produced by toyexample try 500 iterations

MATLAB code for (a): four files codes

```
function log_joint = stdgibbs_logjoint(theta, phi, Adk, Bkw, Mk, I, D, K, W, di, wi,
    ci, citest, Id, Iw, Nd, alpha, beta)
    % theta: Document-topic distribution (D x K)
    % phi: Topic-word distribution (K x W)
    % Adk: Document-topic count matrix (D x K)
    % Bkw: Topic-word count matrix (K x W)
    % Mk: Total number of words assigned to each topic (K x 1)
    % I: Number of (document, word) pairs in training/test sets
    % D: Number of documents
    % K: Number of topics
    % W: Number of unique words
    % di, wi, ci, citest, Id, Iw, Nd: Various input parameters (details provided in
        README)
    % alpha: Dirichlet prior for document-topic distribution
    % beta: Dirichlet prior for topic-word distribution

    log_joint = 0;

    for d = 1:D
        theta_d = theta(d, :); % Get theta for document d
        Adk_d = Adk(d, :) + alpha; % Dirichlet parameters with prior for document d
        log_joint = log_joint + sum(gammaln(Adk_d)) - gammaln(sum(Adk_d));
    end
    log_joint = log_joint - D * K * gammaln(alpha) + D * gammaln(K * alpha);

    % Contribution from phi (topic-word distributions)
    for k = 1:K
        phi_k = phi(k, :); % Get phi for topic k
        Bkw_k = Bkw(k, :) + beta; % Dirichlet parameters with prior for topic k
        log_joint = log_joint + sum(gammaln(Bkw_k)) - gammaln(sum(Bkw_k));
    end
    log_joint = log_joint - K * W * gammaln(beta) + K * gammaln(W * beta);

end
```

Listing 11: Code in 'stdgibbs logjoint' file

```
function [zi, theta, phi, Adk, Bkw, Mk] = stdgibbs_update(zi, theta, phi, Adk, Bkw, Mk,
    I, D, K, W, di, wi, ci, citest, Id, Iw, Nd, alpha, beta)
    % zi: Current topic assignments (cell array with topic for each word in each
        document)
    % theta: Document-topic distribution (D x K)
    % phi: Topic-word distribution (K x W)
    % Adk: Document-topic count matrix (D x K)
    % Bkw: Topic-word count matrix (K x W)
    % Mk: Total count of words assigned to each topic (K x 1)
    % I, D, K, W: Number of (doc, word) pairs, documents, topics, and vocabulary size
    % di, wi: Document and word indices
    % ci, citest: Training and test counts (if applicable)
    % Id, Iw, Nd: Document-word pairs and document word counts
    % alpha, beta: Dirichlet priors for document-topic and topic-word distributions

    for i = 1:I
        d = di(i);
        w = wi(i);
        current_topic = zi{i};

        Adk(d, current_topic) = Adk(d, current_topic) - 1;
        Bkw(current_topic, w) = Bkw(current_topic, w) - 1;
        Mk(current_topic) = Mk(current_topic) - 1;

        % Sample theta and phi explicitly using Dirichlet distributions
        theta(d, :) = dirichrnd(Adk(d, :) + alpha);
        phi(:, w) = dirichrnd(Bkw(:, w) + beta);

        % Compute conditional probabilities for each topic
        topic_probs = theta(d, :) .* phi(:, w)';

        % Normalize to get probabilities
        topic_probs = topic_probs / sum(topic_probs);

        % Sample new topic
        new_topic = find(mnrnd(1, topic_probs));

        % Update counts with the new topic assignment
        Adk(d, new_topic) = Adk(d, new_topic) + 1;
        Bkw(new_topic, w) = Bkw(new_topic, w) + 1;
        Mk(new_topic) = Mk(new_topic) + 1;

        % Update the topic assignment
        zi{i} = new_topic;
    end
end

function x = dirichrnd(alpha)
    x = gamrnd(alpha, 1);
    x = x / sum(x);
end
```

Listing 12: Code in 'stdgibbs update' file

```
function [zi, Adk, Bkw, Mk] = colgibbs_update(zi, Adk, Bkw, Mk, I, D, K, W, di, wi, ci,
    citest, Id, Iw, Nd, alpha, beta)
    % zi: Current topic assignments (cell array with topic for each word in each
        document)
    % Adk: Document-topic count matrix (D x K)
    % Bkw: Topic-word count matrix (K x W)
    % Mk: Total count of words assigned to each topic (K x 1)
    % I, D, K, W: Number of (doc, word) pairs, documents, topics, and vocabulary size
    % di, wi, ci, citest, Id, Iw, Nd: Various input parameters
    % alpha, beta: Dirichlet priors for document-topic and topic-word distributions

    % Loop through each (document, word) pair
    for i = 1:I
        d = di(i);
        w = wi(i);
        current_topic = zi{i}; % Current topic assignment

        Adk(d, current_topic) = Adk(d, current_topic) - 1;
        Bkw(current_topic, w) = Bkw(current_topic, w) - 1;
        Mk(current_topic) = Mk(current_topic) - 1;

        % Calculate conditional probabilities for each topic
        topic_probs = (Adk(d, :) + alpha) .* (Bkw(:, w)' + beta) ./ (Mk + W * beta);

        % Normalize to get probabilities
        topic_probs = topic_probs / sum(topic_probs);

        % Sample new topic
        new_topic = find(mnrnd(1, topic_probs));

        % Update counts with the new topic assignment
        Adk(d, new_topic) = Adk(d, new_topic) + 1;
        Bkw(new_topic, w) = Bkw(new_topic, w) + 1;
        Mk(new_topic) = Mk(new_topic) + 1;

        % Update the topic assignment
        zi{i} = new_topic;
    end
end
```

Listing 13: Code in 'colgibbs update' file

```
function log_joint = colgibbs_logjoint(Adk, Bkw, Mk, I, D, K, W, di, wi, ci, citest,
    Id, Iw, Nd, alpha, beta)
    % Adk: Document-topic count matrix (D x K)
    % Bkw: Topic-word count matrix (K x W)
    % Mk: Total number of words assigned to each topic (K x 1)
    % I, D, K, W: Number of (doc, word) pairs, documents, topics, and vocabulary size
    % di, wi, ci, citest, Id, Iw, Nd: Various input parameters (details provided in
        README)
    % alpha, beta: Dirichlet priors for document-topic and topic-word distributions

    log_joint = 0;

    % Contribution from document-topic counts (Adk)
    for d = 1:D
        Adk_d = Adk(d, :) + alpha;
        log_joint = log_joint + sum(gammaln(Adk_d)) - gammaln(sum(Adk_d));
    end
    log_joint = log_joint - D * K * gammaln(alpha) + D * gammaln(K * alpha);

    % Contribution from topic-word counts (Bkw)
    for k = 1:K
        Bkw_k = Bkw(k, :) + beta;
        log_joint = log_joint + sum(gammaln(Bkw_k)) - gammaln(sum(Bkw_k));
    end
    log_joint = log_joint - K * W * gammaln(beta) + K * gammaln(W * beta);
end
```

Listing 14: Code in 'colgibbs logjoint' file

(b) Based on the autocorrelation plots for 200 and 500 iterations in Figure 17 and Figure 18, the plots with 500 iterations show significantly reduced noise in the autocorrelation values compared to those with 200 iterations. Specifically, the autocorrelation values for both the standard and collapsed Gibbs samplers tend to approach zero more consistently within the lag range for 500 iterations, indicating that the chains have decorrelated more effectively. This suggests that increasing the number of iterations helps in reducing the noise in the autocorrelations, thereby providing a more reliable representation of the posterior.

For a representative set of samples from the posterior, it is essential to discard the burn-in period and select samples that are sufficiently uncorrelated. The 500-iteration results indicate that a lag of approximately 10 is sufficient to ensure minimal autocorrelation, based on the rapid drop in the autocorrelation values. Therefore, it is recommended to thin the samples by taking every 10th sample after the burn-in period to construct a representative posterior. This decision is justified as it balances computational efficiency with the need for decorrelation, as evidenced by the reduced noise and consistent trends in the 500-iteration plots.

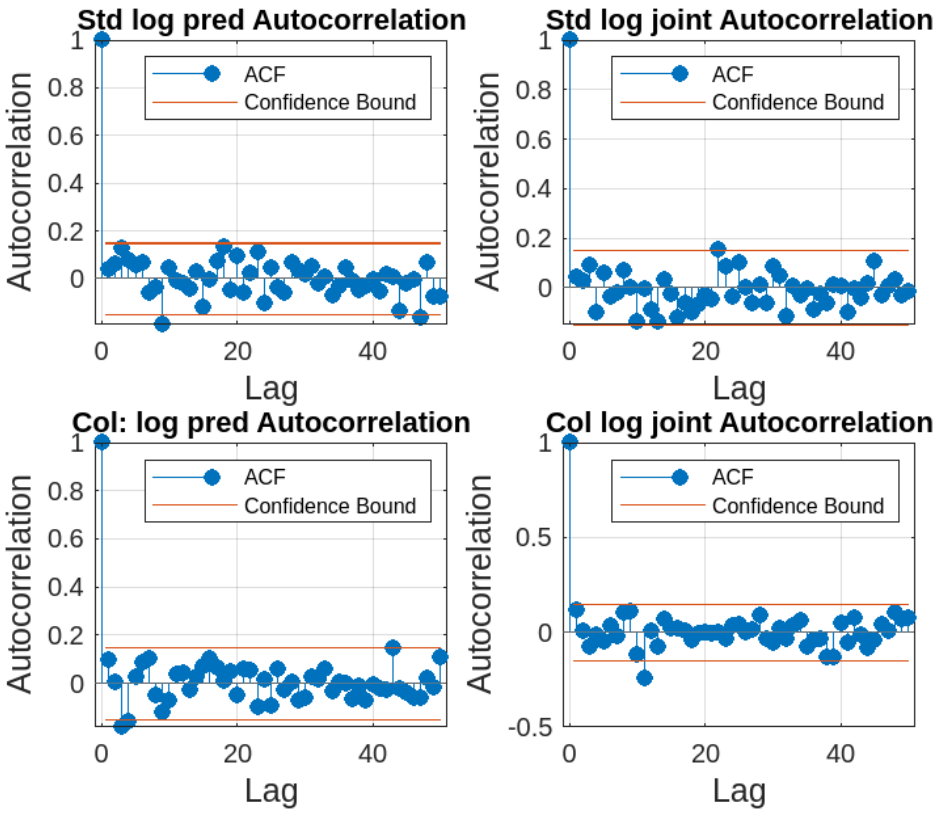

Figure 17: Autocorrelation plot by 200 iterations

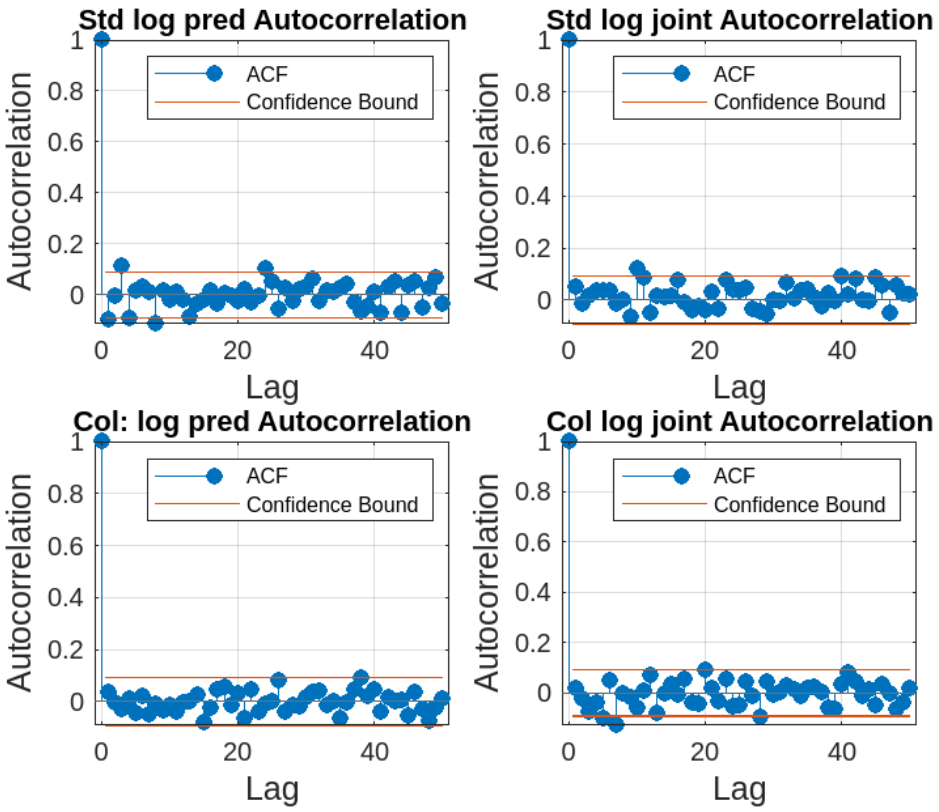

Figure 18: Autocorrelation plot by 500 iterations

(c) Based on the plots, the collapsed Gibbs sampler appears to converge faster compared to the standard Gibbs sampler, particularly in terms of the stability of the log predictive and log joint probabilities. The collapsed Gibbs sampler exhibits less variability in the log joint probabilities and log predictive probabilities after a few iterations, whereas the standard Gibbs sampler shows slightly more fluctuation.

The collapsed Gibbs sampler integrates out parameters during sampling, which reduces the dimensionality of the sampling process and often leads to faster mixing and convergence. This advantage is particularly evident in smaller datasets, such as the toy example used here, where collapsed Gibbs benefits from the reduced variance in the sampling process. The faster convergence is observed in the stability of the collapsed Gibbs plots compared to the standard Gibbs.

(d) In Figure 19 and Figure 20, when varying α, β, and K, the effects on the posterior and predictive performance of the model become evident. Increasing K, the number of topics, leads to more granular topic modeling. While this can improve the model's capacity to capture finer details in the data, setting K too high may result in overfitting, as the model tries to assign topics to smaller and less meaningful partitions of the data. For instance, with $K = 5$, the model demonstrates stable convergence, and the log predictive probabilities exhibit smoother patterns compared to when K is lower.

The hyperparameter α, which governs the Dirichlet prior over topic distributions, plays a crucial role in determining how topics are distributed within documents. Larger values of α encourage documents to exhibit multiple topics more uniformly, thereby promoting diversity in topic assignments. This is reflected in the smoother predictive log probability plots observed with higher α, as the regularization effect helps the model generalize better across the dataset. Similarly, β, the hyperparameter for the Dirichlet prior over words within topics, impacts the diversity of words associated with each topic. A higher β ensures that word distributions within topics are more balanced, reducing the prominence of overly specific words. This smoothing effect contributes to more consistent and stable log joint and log predictive scores.

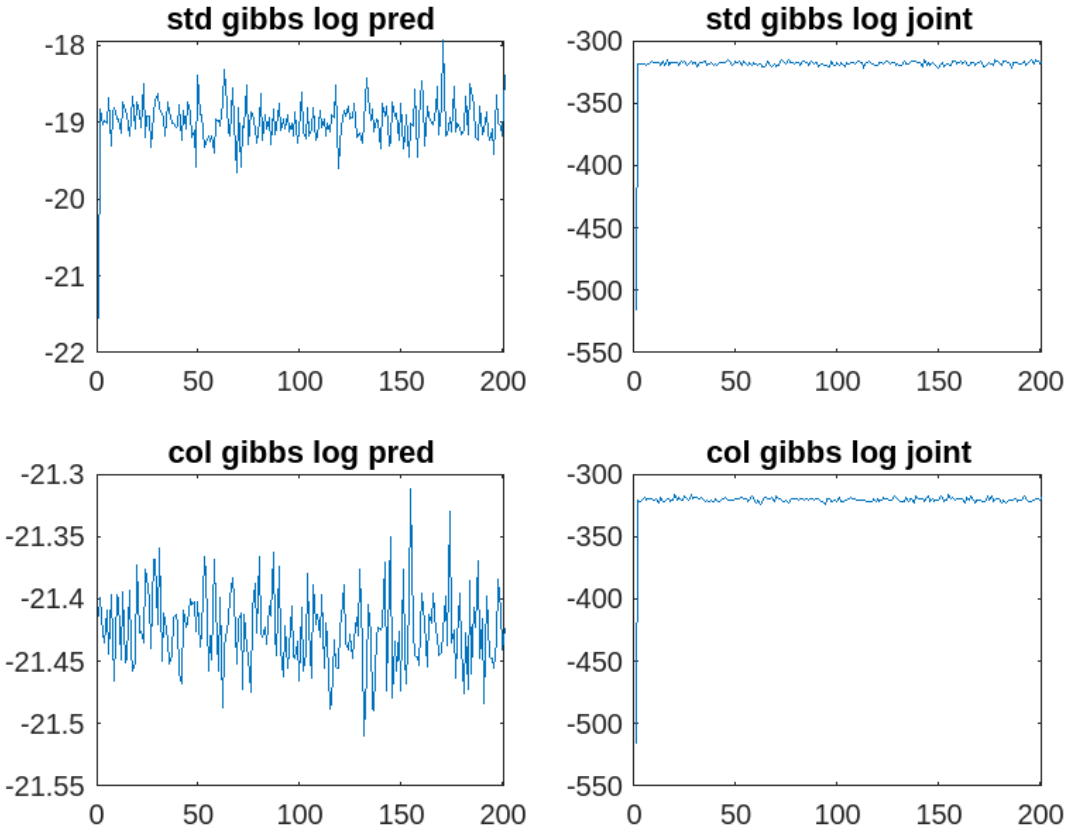

Figure 19: K = 5, alpha = 5, beta = 5

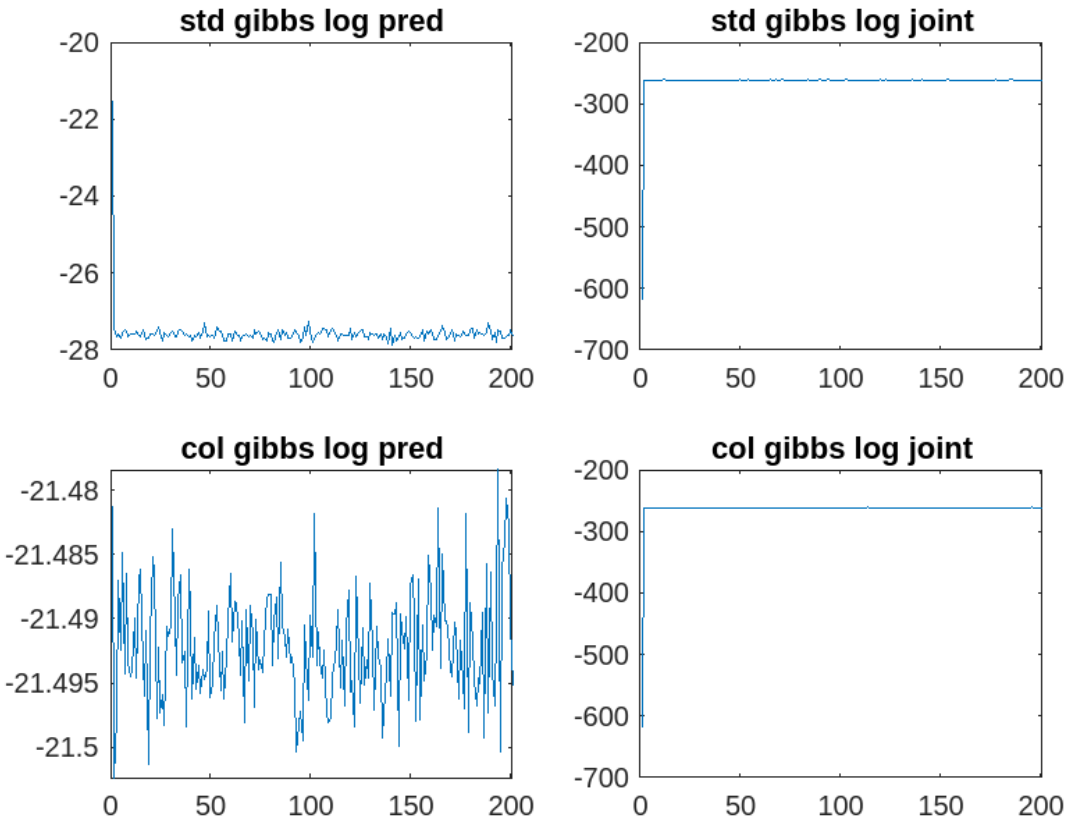

Figure 20: K = 10, alpha = 10, beta = 10

Question 7

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

We define the Lagrangian function $\mathcal{L}(x, y, \lambda)$ as follows:

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

where λ is the Lagrange multiplier.

Next, we find the partial derivatives of $\mathcal{L}$ with respect to x, y, and λ, and set them equal to zero.

Partial derivative with respect to x:

$$\frac{\partial \mathcal{L}}{\partial x} = 1 - \lambda y = 0$$

From this, we get:

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

Partial derivative with respect to y:

$$\frac{\partial \mathcal{L}}{\partial y} = 2 - 2\lambda y - \lambda x = 0$$

Substituting $\lambda = \frac{1}{y}$ into this equation, we get:

$$2 - 2 \cdot \frac{1}{y} \cdot y - \frac{x}{y} = 0$$

Simplifying, this becomes:

$$2 - 2 - \frac{x}{y} = 0 \Rightarrow x = 0$$

Partial derivative with respect to λ:

$$\frac{\partial \mathcal{L}}{\partial \lambda} = 1 - y^2 - xy = 0$$

Substituting $x = 0$, we obtain:

$$1 - y^2 = 0 \Rightarrow y^2 = 1$$

Thus, $y = \pm 1$.

From the equations above, we find the following critical points: When $y = 1$, $x = 0$, giving the point $(0, 1)$. When $y = -1$, $x = 0$, giving the point $(0, -1)$.

The local extrema of $f(x, y) = x + 2y$ subject to the constraint $y^2 + xy = 1$ occur at the points $(0, 1)$ and $(0, -1)$. These are the points at which the local extrema occur, as required by the problem.

(b) To compute $\ln(a)$ using Newton's method, we need to define a function $f(x, a)$ such that the root of this function will give $x = \ln(a)$.

Define the Function $f(x, a)$ We know that if $x = \ln(a)$, then $e^x = a$. Thus, we can define the function $f(x, a)$ as:

$$f(x, a) = e^x - a$$

The root of this function (i.e., where $f(x, a) = 0$) will give $x = \ln(a)$.

Derive the Update Equation for Newton's Method Newton's method for finding the root of a function $f(x)$ involves the iterative update:

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

To apply this to our function $f(x, a) = e^x - a$, we first need to compute $f'(x, a)$.

Compute $f'(x, a)$:

$$f'(x, a) = \frac{d}{dx}(e^x - a) = e^x$$

Substitute into the Newton's Method Update Equation: Using $f(x, a) = e^x - a$ and $f'(x, a) = e^x$, the update equation becomes:

$$x_{n+1} = x_n - \frac{e^{x_n} - a}{e^{x_n}}$$

Simplifying this expression, we get:

$$x_{n+1} = x_n - 1 + \frac{a}{e^{x_n}}$$

or equivalently,

$$x_{n+1} = x_n + \frac{a - e^{x_n}}{e^{x_n}}$$

Thus, to compute $\ln(a)$ using Newton's method:

i. Define the function $f(x, a) = e^x - a$.

ii. Use the iterative update equation:

$$x_{n+1} = x_n + \frac{a - e^{x_n}}{e^{x_n}}$$

Starting with an initial guess x_0, we can iterate using this formula until convergence to obtain $x = \ln(a)$.

Question 8

(a) Note that for any nonzero scalar α and vector $x \neq 0$, we have:

$$R_A(\alpha x) = \frac{(\alpha x)^T A(\alpha x)}{(\alpha x)^T(\alpha x)} = \frac{\alpha^2 x^T A x}{\alpha^2 x^T x} = R_A(x).$$

This shows that $R_A(x)$ is homogeneous of degree 0, meaning it is sufficient to consider vectors x with $\|x\| = 1$ because scaling x by a nonzero constant does not change $R_A(x)$. Therefore,

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

where $S = \{x \in \mathbb{R}^n \mid \|x\| = 1\}$ is the unit sphere.

The unit sphere S in $\mathbb{R}^n$ is a compact set. Since $q_A(x)$ is continuous (as given), $R_A(x)$ is also continuous on S because it is the ratio of two continuous functions (and $\|x\| = 1$ ensures the denominator is nonzero).

By the extreme value theorem, a continuous function on a compact set attains its maximum and minimum. Since $R_A(x)$ is continuous on the compact set S, there exists some $x^* \in S$ such that

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

Thus, x^* maximizes $R_A(x)$ on $\mathbb{R}^n$, and $R_A(x^*)$ is the largest eigenvalue of A, with x^* as a corresponding eigenvector.

(b) To show that $R_A(x) \leq \lambda_1$ for any vector $x \in \mathbb{R}^n$.

Let A be a symmetric matrix with eigenvalues $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_n$ and corresponding orthonormal eigenvectors $\xi_1, \xi_2, \ldots, \xi_n$, which form an orthonormal basis (ONB) of $\mathbb{R}^n$. Any vector $x \in \mathbb{R}^n$ can be represented as:

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

The quadratic form $q_A(x) = x^T A x$ can be expanded as follows:

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

Expanding the transpose, we get:

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

Since ξ_i are eigenvectors of A with corresponding eigenvalues λ_i, we have $A\xi_i = \lambda_i \xi_i$. Therefore,

$$\xi_i^\top A \xi_j = \lambda_i \xi_i^\top \xi_j.$$

Because $\{\xi_i\}$ form an orthonormal basis, $\xi_i^\top \xi_j = 0$ for $i \neq j$ and $\xi_i^\top \xi_i = 1$. This means that only terms where $i = j$ will survive in the summation, simplifying the expression to:

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

The squared norm $\|x\|^2$ is defined as:

$$\|x\|^2 = x^\top x.$$

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

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

Expanding the inner product, we get:

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

Since $\{\xi_i\}$ form an orthonormal basis, $\xi_i^\top \xi_j = 0$ for $i \neq j$ and $\xi_i^\top \xi_i = 1$, we get:

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

Now, $R_A(x) = \frac{q_A(x)}{\|x\|^2}$. The norm $\|x\|^2$ is given by:

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

Thus:

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

Since λ_1 is the largest eigenvalue, we have $\lambda_i \leq \lambda_1$ for all i. Therefore:

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

(c) To show that if $x \in \mathbb{R}^n$ is not contained in the span of the eigenvectors corresponding to the largest eigenvalue λ_1, then $R_A(x) < \lambda_1$.

Since A is symmetric, it has a complete set of orthonormal eigenvectors $\{\xi_1, \ldots, \xi_k, \xi_{k+1}, \ldots, \xi_n\}$.

Since x is not contained in the span of $\{\xi_1, \ldots, \xi_k\}$, it must have a non-zero component in the directions of the eigenvectors $\{\xi_{k+1}, \ldots, \xi_n\}$ associated with eigenvalues less than λ_1. This means that:

$$\sum_{i=k+1}^{n} (\xi_i^\top x)^2 > 0.$$

In the expression for $R_A(x)$, the numerator $\sum_{i=1}^{n} \lambda_i (\xi_i^\top x)^2$ includes terms with eigenvalues $\lambda_{k+1}, \ldots, \lambda_n$ that are strictly less than λ_1. Since $x \notin \text{span}\{\xi_1, \ldots, \xi_k\}$, there exists some components of x in the directions of eigenvectors $\{\xi_{k+1}, \ldots, \xi_n\}$ associated with eigenvalues smaller than λ_1. This results in the inequality:

$$\sum_{i=1}^{n} \lambda_i (\xi_i^\top x)^2 < \lambda_1 \sum_{i=1}^{n} (\xi_i^\top x)^2 = \lambda_1 \|x\|^2.$$

Since the Rayleigh quotient $R_A(x)$ is defined as

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

and we have shown that

$$\sum_{i=1}^{n} \lambda_i (\xi_i^\top x)^2 < \lambda_1 \|x\|^2,$$

it follows that:

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

Therefore, if $x \notin \text{span}\{\xi_1, \ldots, \xi_k\}$, then $R_A(x) < \lambda_1$, as required.