COMP0086

NMWX9

October 28, 2024

Contents

1 Problem1: Models for Binary Vectors

1.1 Part (a)

A multivariate Gaussian distribution is defined over continuous variables and assumes that each variable can take any real value from $-\infty$ to ∞. However, in this data set, each pixel $x_d^{(n)}$ is binary, taking values in $0, 1$. Modeling binary data with a multivariate Gaussian is inappropriate for several reasons:

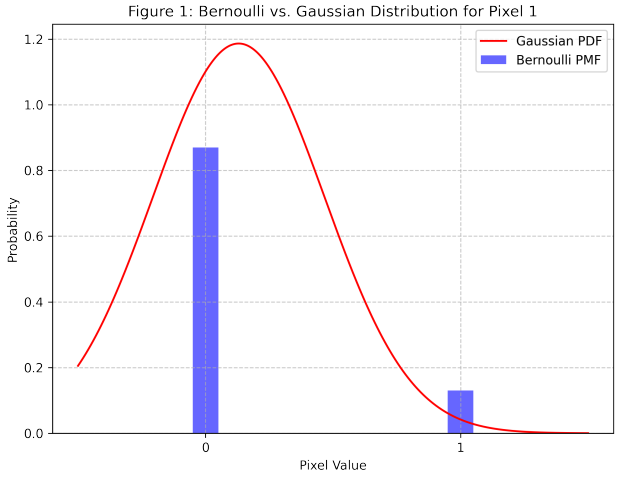


Figure 1: Example where each pixel has a 20% chance of being black (0) and 80% chance of being white (1). A multivariate Gaussian centers itself between the two outcomes with large tails, inadequately capturing the binary nature of the data.

1. **Support Mismatch**: The Gaussian distribution assigns probability density to all real numbers, including values outside the binary set $0, 1$, which is inconsistent with the data.
2. **Inappropriate Marginals**: The marginal distributions under a Gaussian are continuous and unbounded, whereas binary data requires Bernoulli marginals.
3. **Incorrect Covariance Structure**: The Gaussian distribution models linear relationships suitable for continuous variables, which may not capture the dependence structure in binary data.
4. **Example Illustration**: In the given binarydigits.txt dataset each pixel has a 20% chance of being black (0) and an 80% chance of being white (1). A multivariate Gaussian would center itself somewhere in between the two possible outcomes, with large enough tails to attempt to satisfy both outcomes, but failing to do so effectively. This is illustrated in Figure 1.

Therefore, a multivariate Bernoulli distribution, which is defined over binary variables, is a more appropriate model for this data set.

1.2 Part (b)

Given the multivariate Bernoulli distribution:

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

and a data set $\{\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(N)}\}$, the likelihood function is:

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

The log-likelihood is:

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

To find the ML estimate, take the derivative with respect to p_d and set it to zero:

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

Solving for p_d:

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

Thus, the ML estimate of p_d is the sample mean of the d-th pixel across all images.

1.3 Part (c)

The posterior distribution is proportional to the likelihood times the prior:

$$P(\mathbf{p}|\mathbf{X}) \propto L(\mathbf{p}) \cdot P(\mathbf{p})$$

The log-posterior is:

$$\log P(\mathbf{p}|\mathbf{X}) = \log L(\mathbf{p}) + \log P(\mathbf{p}) + \text{const}$$

Combining the log-likelihood and log-prior:

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

To find the MAP estimate, take the derivative with respect to p_d:

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

Solving for p_d:

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

1.4 Part (d)

MLE and Visualisation Python Code can be found in Appendix Problem 1, Part (d).

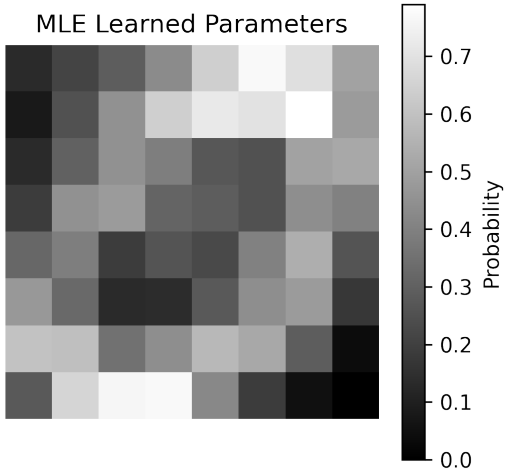


Figure 2: Maximum likelihood of a multivariate Bernoulli from binarydigits.txt: Bright pixels indicate higher probabilities being 1, while dark pixels indicate lower probabilities. it represents the average pattern from the data.

1.5 Part (e)

MAP and Visualisation Python Code can be found in Appendix Problem 1, Part (e).

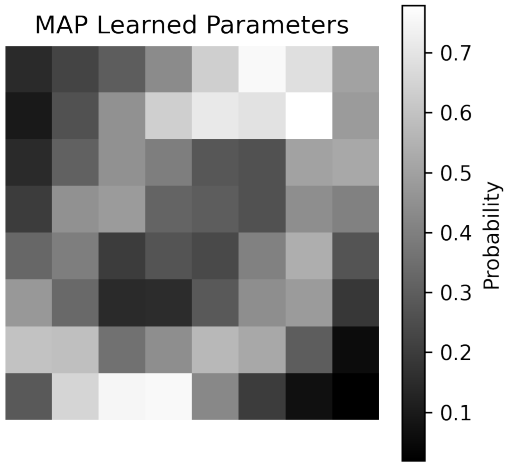


Figure 3: MAP of a multivariate Bernoulli from binarydigits.txt: It estimate image appears similar to the ML estimate but with slightly adjusted probabilities due to the influence of the prior. The prior pulls the probabilities towards 0.5, providing a smoothing effect.

Maximum likelihood (ML) estimation derives the best estimates based solely on the observed data, without incorporating any prior knowledge or beliefs about the parameters. In contrast, Maximum a Posteriori (MAP) estimation integrates prior information with the observed data to produce parameter

estimates. If no prior information is available about the model parameters, ML estimation is typically preferred because it relies purely on the data, avoiding potential biases introduced by incorrect or uninformative priors. However, when reliable prior information is available—such as priors derived from previous studies, expert knowledge, or conjugate priors that simplify computations—MAP estimation can yield better results. This is because the prior can guide the estimates towards more plausible values, especially in cases where the data may be limited or noisy.

Nonetheless, the effectiveness of MAP estimation depends on the accuracy and relevance of the prior. If the prior is well-chosen and reflects true underlying beliefs about the parameters, MAP can enhance the estimation by incorporating this additional information. Conversely, if the prior is poorly chosen or biased, MAP estimation may lead to less accurate or misleading results compared to ML estimation.

In summary, MAP estimation offers a way to incorporate prior knowledge into the parameter estimation process, potentially improving results when the prior is informative and accurate. However, in the absence of reliable prior information, ML estimation remains the safer and more objective choice.

2 Problem 2: Model Selection

2.1 Model (a): Fixed $p_d = 0.5$

Under Model (a), each pixel is independently generated with $p_d = 0.5$. The likelihood of the entire dataset is:

$$P(\mathbf{D}|M_a) = \prod_{n=1}^{N}\prod_{d=1}^{D} P(x_d^{(n)}|p_d = 0.5) = \prod_{n=1}^{N}\prod_{d=1}^{D} 0.5 = (0.5)^{ND}$$

Taking the natural logarithm for numerical stability:

$$\ln P(\mathbf{D}|M_a) = ND \ln 0.5 = -ND \ln 2$$

2.2 Model (b): Unknown, Identical $p_d = p$

Under Model (b), all D components share the same unknown probability p. The likelihood is:

$$P(\mathbf{D}|p, M_b) = p^S (1-p)^{ND-S}$$

where $S = \sum_{n=1}^{N}\sum_{d=1}^{D} x_d^{(n)}$ is the total number of ones in the dataset.
With a uniform prior $P(p) = 1$, the marginal likelihood is:

$$P(\mathbf{D}|M_b) = \int_0^1 p^S (1-p)^{ND-S} dp = \text{Beta}(S+1, ND-S+1)$$

Using the Beta function:

$$\text{Beta}(a,b) = \frac{\Gamma(a)\Gamma(b)}{\Gamma(a+b)}$$

Thus,

$$\ln P(\mathbf{D}|M_b) = \ln\Gamma(S+1) + \ln\Gamma(ND-S+1) - \ln\Gamma(ND+2)$$

2.3 Model (c): Unknown, Separate p_d

Under Model (c), each pixel d has its own unknown probability p_d. The likelihood is:

$$P(\mathbf{D}|M_c) = \prod_{d=1}^{D} P(\mathbf{D}_d|p_d, M_c) = \prod_{d=1}^{D} p_d^{S_d}(1-p_d)^{N-S_d}$$

where $S_d = \sum_{n=1}^{N} x_d^{(n)}$ is the number of ones for pixel d.
With uniform priors $P(p_d) = 1$, the marginal likelihood for each pixel is:

$$P(\mathbf{D}_d|M_c) = \text{Beta}(S_d+1, N-S_d+1)$$

Therefore, the total marginal likelihood is:

$$P(\mathbf{D}|M_c) = \prod_{d=1}^{D} \text{Beta}(S_d+1, N-S_d+1)$$

Taking the natural logarithm:

$$\ln P(\mathbf{D}|M_c) = \sum_{d=1}^{D} [\ln\Gamma(S_d+1) + \ln\Gamma(N-S_d+1) - \ln\Gamma(N+2)]$$

2.4 Stirling's Approximation

For large arguments, computing the Gamma function directly can lead to numerical overflow. We use Stirling's approximation to provide an efficient way to approximate the logarithm of the Gamma function:

$$\ln \Gamma(n) \approx n \ln n - n + \frac{1}{2} \ln(2\pi n)$$

Reason for Using Stirling's Approximation:
Stirling's approximation is utilized to compute $\ln \Gamma(n)$ efficiently for large n (100), ensuring numerical stability and preventing overflow during calculations.

2.5 Posterior Probabilities

Assuming equal prior probabilities for each model, $P(M_a) = P(M_b) = P(M_c) = \frac{1}{3}$, the posterior probabilities are proportional to the marginal likelihoods:

$$P(M_i|\mathbf{D}) \propto P(\mathbf{D}|M_i)$$

To compute the relative posterior probabilities, we calculate the marginal likelihoods for each model and normalize them using the log-sum-exp trick for numerical stability. The Python code implementation can be found in Appendix Problem 2.:

Output of the Code:

```
Log Likelihoods:
Log-likelihood for Model a: -4436.14195558365
Log-likelihood for Model b: -4283.721384281947
Log-likelihood for Model c: -3851.5098017975783

Log Denominator:
The log denominator: -3851.5098017975783

Log Nominators:
Log nominator for Model a: -4436.14195558365
Log nominator for Model b: -4283.721384281947
Log nominator for Model c: -3851.5098017975783

Log Posteriors:
Log posterior for Model a: -584.6321537860717
Log posterior for Model b: -432.21158248436905
Log posterior for Model c: 0.0

Posteriors:
Posterior for Model a: 1.2516464372281e-254
Posterior for Model b: 1.9628843497623033e-188
Posterior for Model c: 1.0
```

2.6 Interpretation of Results

From the output, we observe the following posterior probabilities for each model:

- **Posterior Probability of Model (a)**: $1.2516464372281 \times 10^{-254}$
- **Posterior Probability of Model (b)**: $1.9628843497623033 \times 10^{-188}$
- **Posterior Probability of Model (c)**: 1.0

These results indicate that Model (c) has a posterior probability of 1, while Models (a) and (b) have negligible probabilities approaching zero. This suggests that Model (c), where each component has its own separate, unknown probability p_d, is overwhelmingly the most probable model given the data in `binarydigits.txt`.

The posterior probability calculations demonstrate that Model (c) is the most suitable model for the given binary dataset. This outcome aligns with the expectation that allowing each pixel to have its own distinct probability of being white or black provides a more flexible and accurate representation of the data compared to Models (a) and (b).

3 Problem3: EM for Binary Data

3.1 Part (a)

Let $\pi_1, \ldots, \pi_K$ denote the mixing proportions, where $0 \leq \pi_k \leq 1$ and $\sum_{k=1}^{K} \pi_k = 1$. Let P be a $K \times D$ matrix, where p_{kd} denotes the probability that pixel d takes value 1 under mixture component k.

Assuming that images are independent and identically distributed (iid) and that pixels are independent of each other within each component, the likelihood of the data set $\mathbf{X} = \{\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(N)}\}$ is:

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

3.2 Part (b)

The responsibility r_{nk} is the probability that mixture component k is responsible for data vector $\mathbf{x}^{(n)}$, given the current parameters:

$$r_{nk} = P(s^{(n)} = k \,|\, \mathbf{x}^{(n)}, \pi, P) = \frac{P(s^{(n)} = k, \mathbf{x}^{(n)} \,|\, \pi, P)}{P(\mathbf{x}^{(n)} \,|\, \pi, P)} = \frac{\pi_k \prod_{d=1}^{D} p_{kd}^{x_d^{(n)}} (1 - p_{kd})^{1 - x_d^{(n)}}}{\sum_{j=1}^{K} \pi_j \prod_{d=1}^{D} p_{jd}^{x_d^{(n)}} (1 - p_{jd})^{1 - x_d^{(n)}}}$$

This computation provides the E-step for the EM algorithm.

3.3 Part (c)

We aim to maximize the expected complete data log-likelihood:

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

Update for Mixing Proportions π_k:

We maximize $Q(\pi, P)$ with respect to π_k, subject to the constraint $\sum_{k=1}^{K} \pi_k = 1$. Using a Lagrange multiplier λ, we have:

$$\frac{\partial}{\partial \pi_k} \left[Q(\pi, P) + \lambda \left(\sum_{k=1}^{K} \pi_k - 1 \right) \right] = 0$$

This yields:

$$\frac{\partial Q}{\partial \pi_k} + \lambda = 0 \implies \sum_{n=1}^{N} \frac{r_{nk}}{\pi_k} + \lambda = 0 \implies \pi_k = \frac{1}{N} \sum_{n=1}^{N} r_{nk}$$

Update for Bernoulli Parameters p_{kd}:

We maximize $Q(\pi, P)$ with respect to p_{kd}. Setting the derivative to zero:

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

Solving for p_{kd}:

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

Thus, the M-step updates are:

$$\pi_k^{\text{new}} = \frac{\sum_{n=1}^{N} r_{nk}}{N}, \quad p_{kd}^{\text{new}} = \frac{\sum_{n=1}^{N} r_{nk} x_d^{(n)}}{\sum_{n=1}^{N} r_{nk}}$$

3.4 Part (d)

The Python implementation of the EM algorithm, including the outputs for Part(e), is provided in Appendix Problem 3, Part (d) and Part (e). In this implementation, weak priors were not incorporated, as they did not lead to noticeable improvements in model fit or convergence rate during our implementation.

Output Results:

For each value of $K \in \{2, 3, 4, 7, 10\}$, the EM runs and outputs the log-likelihood plot against the number of iterations the π_k is displayed in the caption, p_{kd} are displayed in the next section in the images.

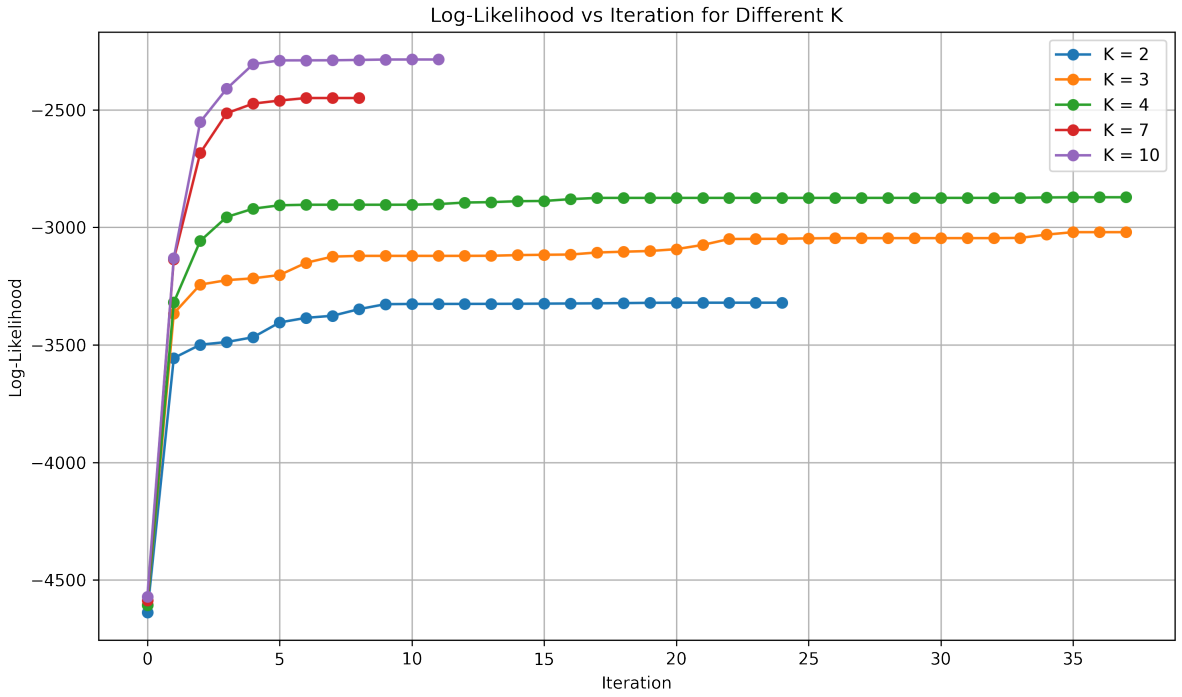


Figure 4: $\pi_{k=2} = [0.34037769459702377, 0.6596223054029764]$,
$\pi_{k=3} = [0.17999994954946813, 0.39999957327696867, 0.42000047717356315]$,
$\pi_{k=4} = [0.22998940643610763, 0.3299988503483383, 0.1599999999999964, 0.2800117432155577]$,
$\pi_{k=7} = [0.06999999999999872, 0.16011887453450613, 0.15999999999999903, 0.2500000000176119, 0.07999999999985732, 0.17988112544788426, 0.10000000000014268]$,
$\pi_{k=10} = [0.03, 0.12000318025426729, 0.16, 0.12001356796591468, 0.08, 0.09999885124170778, 0.08, 0.05999999999999165, 0.04998553552592256, 0.19999886501218855]$

3.5 Part (e)

After running the EM algorithm 50 times with different random initializations for a fixed number of mixture components K, we can see for each fixed K, the log-likelihood tends to converge to approximately the same value across different runs. This indicates that the algorithm reliably reaches a similar level of data fit, suggesting stability in the optimization process.

While the log-likelihoods converge to similar values, the mixing proportions π_k exhibit significant variability across different runs, especially for higher values of K. However, for lower values of K, the π_k values tend to be more consistent. This variability can be attributed to the presence of multiple local maxima in the likelihood surface, particularly as K increases. Below we show for a example of each value of $K \in \{2, 3, 4, 7, 10\}$, the p_{kd} as 8×8 images.

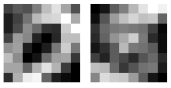


Figure 5: parameter matrices $\mathbf{P}$, when $K = 2$

Figure 6: parameter matrices $\mathbf{P}$, when $K = 3$

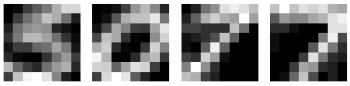


Figure 7: parameter matrices $\mathbf{P}$, when $K = 4$

Figure 8: parameter matrices $\mathbf{P}$, when $K = 7$

Figure 9: parameter matrices $\mathbf{P}$, when $K = 10$

The choice of K plays a crucial role in the performance of the EM algorithm, in our case:

- **Low K Values ($K < 3$):** For our datasets (which seems to only contain images of digits 0, 5, and 7), setting $K < 3$ is inadequate. The model fails to capture the distinct characteristics of each digit, leading to poor clustering and representation.
- **Optimal K Values ($2 < K < 7$):** Within this range, the algorithm effectively distinguishes between the different digits, assigning each to a separate mixture component. This range balances model complexity and data fit, preventing overfitting while adequately capturing the underlying structure.
- **High K Values ($K > 7$):** Larger values of K tend to lead to overfitting, where the model starts to capture noise in the data rather than meaningful patterns. This results in disparate π_k values across runs and diminished interpretability of the mixture components.

we run the algorithm for each value of $K \in \{2, 3, 4, 7, 10\}$ 50 times starting from randomly chosen initial conditions. we can see that Upon inspection of the learned Bernoulli parameter matrices $\mathbf{P}$, it is evident that the majority of components correspond to specific digits such as 0, 5, and 7. For instance, components with parameters closely resembling the patterns of these digits are consistently identified across different runs, validating the model's capability to cluster similar images effectively.

Examining the responsibilities matrix reveals that for each data point, one responsibility dominates while the others are negligible. This behavior indicates that the EM algorithm successfully assigns each image to a single, most probable mixture component, thereby achieving clear and distinct clusters.

Improvement Strategies:

- **Initialization Techniques:** Implementing better initialization methods, maybe initialise with k-means clustering, can enhance convergence speed and reduce variability across runs by providing a better starting point.
- **Regularization and Priors:** Introducing priors on the parameters can prevent overfitting and promote more robust estimates. For example, incorporating Dirichlet priors on the mixing proportions π_k can encourage a more balanced distribution.
- **Model Selection Criteria:**** Utilizing criteria like the Bayesian Information Criterion (BIC) or Akaike Information Criterion (AIC) can aid in selecting an optimal K that balances model fit and complexity.
- **Hierarchical Models:** Hierarchical mixture models can capture more nuanced variations within the data, potentially improving clustering performance without excessively increasing the number of parameters.

3.6 Part (f) [Bonus]

In this part, we express the negative log-likelihoods obtained from the EM algorithm in bits and compare these numbers to the length of the naive encoding of the binary data.

Given Outputs:

- file raw size: 103200 bits
- Naive encoding length: 6400 bits
- `gzip` compressed size: 8328 bits

For each value of K, we have:

K	Final Negative Log-Likelihood (bits)	Model Cost (bits)	Total Cost (bits)
2	4790.29	8256	13046.29
3	4357.38	12416	16773.38
4	4143.13	16576	20719.13
7	3534.04	29056	32590.04
10	3297.51	41536	44833.51

Explanation:
The negative log-likelihood (NLL) in bits represents the codelength required to encode the data given the model, according to Shannon's source coding theorem. Specifically, for data $\mathbf{X}$ and model parameters θ, the optimal codelength for encoding the data is:

$$L_{\text{data}} = -\log_2 P(\mathbf{X} \mid \theta)$$

Comparing the NLLs to the naive encoding length:

- The naive encoding length is 6400 bits, since each pixel (binary value) requires 1 bit, and there are $N = 100$ images with $D = 64$ pixels each.

- The NLLs for all values of K are less than the naive encoding length, indicating that our models provide a more efficient encoding of the data. For example, at $K = 2$, the NLL is 4790.29 bits, which is approximately 1609.71 bits less than the naive encoding length.

Comparison to `gzip`:
The `gzip` compressed size is 8328 bits, which is higher than the naive encoding length. This may seem counterintuitive since compression algorithms are expected to reduce file sizes. One possible explanation is that `gzip` may not be as effective on small datasets or datasets without sufficient redundancy. In our case, the NLLs obtained from our models for $K \geq 2$ are all less than the `gzip` compressed size. This suggests that our EM models are capturing structure in the data more efficiently than `gzip` for this specific dataset.

Why the Difference?

- `gzip` uses general-purpose lossless compression algorithms (DEFLATE) which is good for general purposes compression (it combines LZ77 and Huffman coding) but it is not optimized for the specific patterns in our binary images (for example, compares to PNG).
- Our EM is specifically tailored to the data, learning the underlying distribution and capturing dependencies among pixels within mixture components. It is also a lossy compression in the sense that it approximates the data distribution using a finite mixture model.
- Our EM exploit the statistical regularities in the data, leading to shorter codelengths as per the Minimum Description Length (MDL) principle.

Therefore, the shorter codelength achieved by our model comes at the cost of potentially losing some information about the data.

3.7 Part (g) [Bonus]

In this part, we consider the total cost of encoding both the model parameters and the data given the model. We analyze how this total cost compares to `gzip` and how it depends on K.

Given Outputs:
Refer to the table above for the model cost and total cost for each K.

Computing Model Cost:

- The model parameters include the mixing proportions π_k and the Bernoulli parameters p_{kd}.
- Number of parameters:

$$\text{Number of parameters} = (K - 1) + K \times D$$

  where $(K - 1)$ accounts for the mixing proportions (since they sum to 1), and $K \times D$ accounts for the Bernoulli parameters.

- Each parameter is stored with 64 bits of precision (float64 representation, NumPy default), the model cost is:

$$\text{Model Cost} = \text{Number of parameters} \times 64 \text{ bits}$$

Total Cost:

$$\text{Total Cost} = \text{Negative Log-Likelihood (bits)} + \text{Model Cost (bits)}$$

Analysis of Total Cost vs. K:
As K increases: The **negative log-likelihood** decreases, indicating a better fit to the data. This is expected because more components allow the model to capture more nuanced structures in the data. The **model cost** increases linearly with K, due to the additional parameters introduced. The **total cost** increases with K, indicating that the decrease in data encoding cost does not compensate for the increase in model complexity. For example: From $K = 2$ to $K = 10$, the NLL decreases from 4790.29 bits

to 3297.51 bits (a reduction of 1492.78 bits). However, the model cost increases from 8256 bits to 41536 bits (an increase of 33280 bits). Consequently, the total cost increases from 13046.29 bits to 44833.51 bits.

Comparison to `gzip`:

The total cost for all values of K exceeds the `gzip` compressed size of 8328 bits. This indicates that when accounting for the cost of storing the model parameters, our models are less efficient in terms of total codelength compared to `gzip`. One explanation is that `gzip` does not store an explicit model; instead, it uses adaptive coding techniques (DEFLATE) that implicitly capture redundancies in the data without a separate model cost.

Implications and Insights:

There is a trade-off between model complexity and data encoding cost. This is a fundamental concept in model selection and is related to the **Bias-Variance Trade-off**. Increasing K allows the model to fit the data more closely (lower NLL) but at the expense of higher model complexity (more parameters).

The **Minimum Description Length (MDL) Principle** suggests choosing the model that minimizes the total codelength (model cost + data cost). In our case, the total cost increases with K, indicating that simpler models (with smaller K) are preferable when considering both data fit and model complexity. The optimal value of K in terms of total cost might be the one that achieves the lowest total codelength. From the given outputs, $K = 2$ has the lowest total cost among the tested values. However we know this is not ideal as we have at least three types of digits in our dataset, indicated model mis-specification. By default, NumPy uses `float64` (64-bit floating-point numbers) for numerical computations, which supposed to provides high precision but is unnecessary for our data which are 8x8 binary images consisting of only 0s and 1s. We can reduce the model cost significantly by using lower-precision data types like `float32` or even integer types (`int8`, `uint8`), while still achieve results.

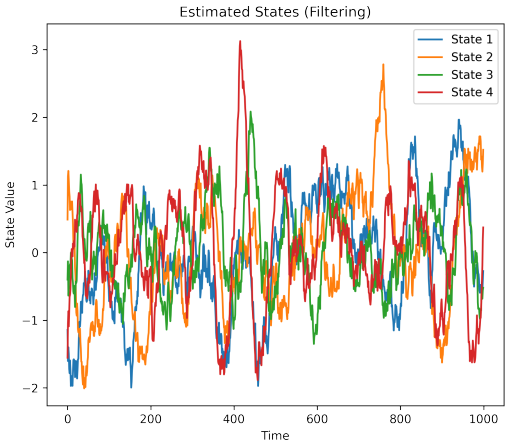
More General Theoretical Considerations:

- **Occam's Razor**: Simpler models are preferred unless the more complex model provides a significantly better fit to the data.
- **Information Criteria**: Model selection criteria such as the AIC or the BIC penalize model complexity to prevent overfitting.
- **Overfitting**: Models with high complexity (large K) may overfit the data, capturing noise rather than underlying patterns.
- **Generalization**: A model with a lower total cost is likely to generalize better to unseen data, as it captures the essential structure without overfitting.

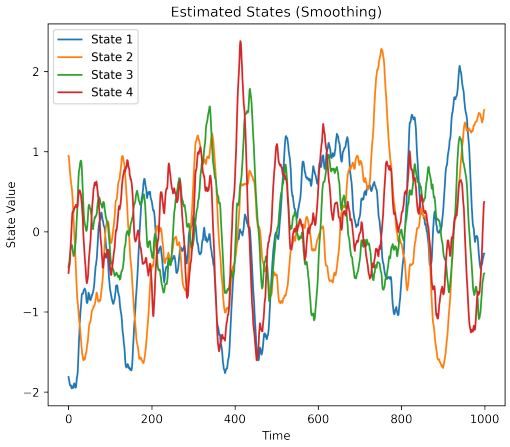
4 Problem4: LGSSMs, EM and SSID [BONUS]

4.1 Part (a)

The Python code for running the Kalman filter and smoother is provided in Appendix Problem4 Part (a) In the Kalman filter the filtered state estimates are influenced by data up to the current time step.



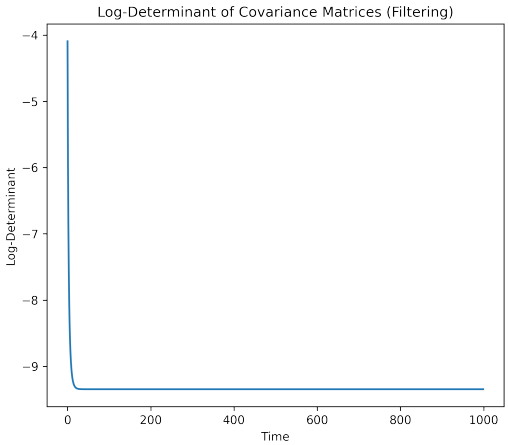
(a) The state estimates Y_t obtained using Kalman filter



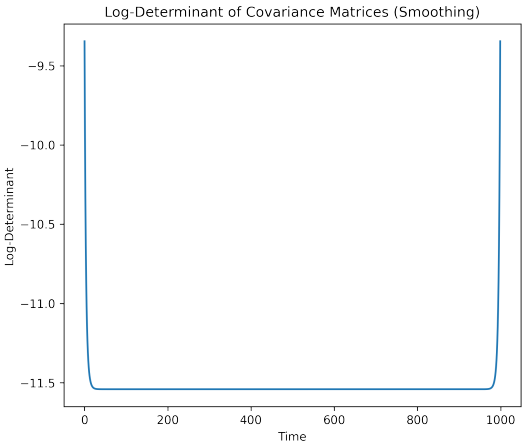
(b) The state estimates Y_t obtained using Kalman smoother

Figure 10: Y plot under "filt" (a) and "smooth" (b) modes

These estimates are noisier compared to the smoothed ones because the Kalman filter can only use past observations to estimate the current state. While in Kalman smoother the smoothed state estimates are less noisy compared to the filtered ones, as the smoother uses both past and future observations to estimate the state. This leads to smoother, more accurate state estimates. The smoothed estimates are more accurate and less noisy than the filtered ones because the smoother can utilize future data.



(a) The log-determinant of the state covariance matrix V_t during the filtering process



(b) The log-determinant of the state covariance matrix V_t during the smoothing process

Figure 11: V plot under "filt" (a) and "smooth" (b) modes

In the Kalman filter The log-determinant of the covariance matrix decreases rapidly at the beginning, reflecting that the filter becomes more confident in its state estimates as more observations are incorporated. After an initial sharp decrease, it stabilizes, indicating that the uncertainty has reached a steady-state. Similar to the filter, in the Kalman smoother the log-determinant of the covariance matrix

shows a sharp decrease at the beginning, but it stabilizes at a lower value than the filter. This reflects that the smoother has more information (future observations) and thus has more confidence in its state estimates.

For the smoother and filter we observe variance high at first/final points because there are not enough observations. The covariance matrices from the smoother show lower uncertainty (smaller values of the log-determinant) compared to the filter because the smoother can incorporate more information to reduce uncertainty.

4.2 Part (b)

We aim to learn the parameters A, Q, C, and R of a Linear Gaussian State-Space Model (LGSSM) using the Expectation-Maximization (EM) algorithm. We assume the initial state distribution is known. The Python code for EM with 1 initialisation with true parameters and 10 random initialisations are provided in Appendix Problem4 Part (b).

Updating C and R: The observation model is:

$$x_t = Cy_t + \epsilon_t, \quad \epsilon_t \sim \mathcal{N}(0, R).$$

The expected complete-data log-likelihood for the observations is:

$$\mathbb{E}[\log p(x_{1:T}|y_{1:T}, C, R)] = -\frac{T}{2}\log|R| - \frac{1}{2}\sum_{t=1}^{T}\mathbb{E}\left[(x_t - Cy_t)^\top R^{-1}(x_t - Cy_t)\right].$$

Setting the derivative with respect to C to zero:

$$\sum_{t=1}^{T} x_t y_t^\top = C_{\text{new}} \sum_{t=1}^{T}\left(y_t y_t^\top + V_t\right),$$

where $V_t = \text{Cov}[y_t|x_{1:T}]$.

Thus, the update for C is:

$$C_{\text{new}} = \left(\sum_{t=1}^{T} x_t y_t^\top\right)\left(\sum_{t=1}^{T}\left(y_t y_t^\top + V_t\right)\right)^{-1}.$$

Similarly, the update for R is:

$$\begin{aligned} R_{\text{new}} &= \frac{1}{T}\sum_{t=1}^{T}\mathbb{E}\left[(x_t - C_{\text{new}}y_t)(x_t - C_{\text{new}}y_t)^\top\right] \\ &= \frac{1}{T}\left(\sum_{t=1}^{T} x_t x_t^\top - C_{\text{new}}\sum_{t=1}^{T} x_t y_t^\top\right). \end{aligned}$$

Updating A and Q: The state transition model is:

$$y_t = Ay_{t-1} + \eta_t, \quad \eta_t \sim \mathcal{N}(0, Q).$$

The expected complete-data log-likelihood for the states is:

$$\mathbb{E}[\log p(y_{2:T}|y_{1:T-1}, A, Q)] = -\frac{T-1}{2}\log|Q| - \frac{1}{2}\sum_{t=2}^{T}\mathbb{E}\left[(y_t - Ay_{t-1})^\top Q^{-1}(y_t - Ay_{t-1})\right].$$

Setting the derivative with respect to A to zero:

$$\sum_{t=2}^{T} y_t y_{t-1}^\top = A_{\text{new}}\sum_{t=2}^{T}\left(y_{t-1}y_{t-1}^\top + V_{t-1}\right).$$

Thus, the update for A is:

$$A_{\text{new}} = \left(\sum_{t=2}^{T} y_t y_{t-1}^\top\right)\left(\sum_{t=2}^{T}\left(y_{t-1}y_{t-1}^\top + V_{t-1}\right)\right)^{-1}.$$

The update for Q is:

$$\begin{aligned} Q_{\text{new}} &= \frac{1}{T-1}\sum_{t=2}^{T}\mathbb{E}\left[(y_t - A_{\text{new}}y_{t-1})(y_t - A_{\text{new}}y_{t-1})^\top\right] \\ &= \frac{1}{T-1}\left(\sum_{t=2}^{T} y_t y_t^\top - A_{\text{new}}\sum_{t=2}^{T} y_t y_{t-1}^\top\right). \end{aligned}$$

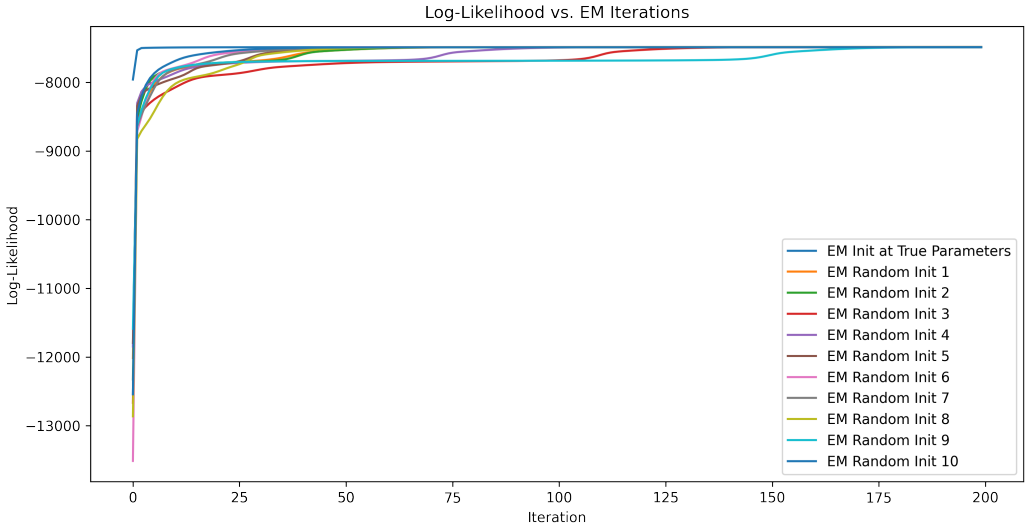


Figure 12: EM with 1 initialisation with the true parameters and 10 random initialisations (100 iterations)

Why EM Does Not Terminate Immediately with True Parameters? EM refines the estimates based on the empirical data, which includes noise. Hence, it does not terminate immediately, as it seeks to find a maximum likelihood estimate that better fits the noisy observations.

Features We Can Observe: For both true parameters initialisation and random initialisations, the log-likelihood increases monotonically, EM guarantees that the likelihood should not decrease after each iteration. initialise with true parameters bring us closer to the highest log-likelihood, and reach this bound quicker. random initialisations starts a further from the optima but rapidly for the first 10 iteration, and then slowly converges to the near the same bound as true parameters initialisation at different pace. The EM runs initialized with random parameters show more variation in their paths. the fluctuations are due to the alternating E/M steps

4.3 Part (c)

Python code provided in Appendix Problem4 Part (c).

Table 1: Training and Test Log-Likelihoods for Different Models

Model	**Training Log-Likelihood**	**Test Log-Likelihood**
True Parameters	−7958.252479	−8031.964223
EM Init at True Parameters	−7486.158003	−7431.271450
EM Random Init 1	−7486.491128	−7440.906146
EM Random Init 2	−7485.265550	−7439.424476
EM Random Init 3	−7486.278643	−7435.145651
EM Random Init 4	−7486.516645	−7434.929224
EM Random Init 5	−7486.547517	−7442.206142
EM Random Init 6	−7485.077281	−7444.416476
EM Random Init 7	−7485.341470	−7435.895431
EM Random Init 8	−7485.794999	−7435.611981
EM Random Init 9	−7487.046042	−7440.961086
EM Random Init 10	−7487.279754	−7437.143238

True Parameters Training log-likelihood (LL) is −7958.25, and test LL is −8031.96. This serves as a baseline. The test likelihood is slightly worse than the training likelihood, as expected, due to the different random seed used in generating the test data.

EM Initialized at True Parameters Training LL is −7486.16, and test LL is −7431.27. The EM algorithm improves the training likelihood significantly, indicating better parameter estimates. The test likelihood also improves, suggesting that the model generalizes well and does not overfit.

EM Runs with Random Initializations Training LL ranges approximately from −7485 to −7487, and test LL ranges approximately from −7430 to −7444. All EM runs, despite different initializations, converge to similar parameter estimates, as evidenced by the comparable log-likelihoods. This indicates robustness of the EM implementation and convergence to a global or good local maximum.

5 Problem5: Decrypting Messages with MCMC

5.1 Part(a)

Estimating the Transition Probabilities

Let $s = s_1 s_2 \dots s_n$ represent a sequence of symbols (plaintext) from the English language. We model the sequence using a first-order Markov chain, where the probability of the sequence is given by:

$$P(s) = P(s_1) \prod_{i=2}^{n} P(s_i \mid s_{i-1}) = P(s_1) \prod_{i=2}^{n} \psi(s_i, s_{i-1})$$

where $\psi(s_i, s_{i-1}) = P(s_i \mid s_{i-1})$ is the transition probability from symbol s_{i-1} to s_i.

Our goal is to estimate $\psi(\alpha, \beta)$ for all pairs of symbols α, β and the initial symbol probabilities $\phi(\gamma) = P(s_1 = \gamma)$.

Maximum Likelihood Estimation (MLE)

Given a large corpus of text, we can estimate the transition probabilities using the Maximum Likelihood Estimator.

1. Unigram Probabilities (Initial Probabilities): The MLE for the initial symbol probabilities $\phi(\gamma)$ is:

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

where:

- $C(\gamma)$ is the count of symbol γ appearing as the first symbol in the sequences.
- N is the total number of sequences (or occurrences of first symbols) in the corpus.

In practice, since we have a single long sequence of text, we can approximate $\phi(\gamma)$ using the overall frequency of γ in the text.

2. Bigram Probabilities (Transition Probabilities): The MLE for the transition probabilities $\psi(\alpha, \beta)$ is:

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

where:

- $C(\beta, \alpha)$ is the count of times symbol β is followed by symbol α in the corpus.
- $C(\beta)$ is the total count of symbol β in the corpus.

Derivation Using Lagrange Multipliers

We can formalize the estimation of $\psi(\alpha, \beta)$ by maximizing the log-likelihood of the observed data subject to the constraints that the transition probabilities sum to one.

Objective Function:

$$\text{Maximize } \log P(s) = \log P(s_1) + \sum_{i=2}^{n} \log \psi(s_i, s_{i-1}) = \log P(s_1) + \sum_{\alpha,\beta} N(\alpha, \beta) \log \psi(\alpha, \beta)$$

where $N(\alpha, \beta)$ is the number of times the transition from β to α occurs in the corpus.

Constraints:

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

Lagrangian:

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

Taking Partial Derivatives: For each $\psi(\alpha,\beta)$:

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

Solving for $\psi(\alpha,\beta)$:

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

Applying the constraint:

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

Therefore, $\lambda_\beta = C(\beta)$, and:

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

which confirms our earlier MLE formula.

Applying Laplace Smoothing

To avoid zero probabilities (which can cause issues in computations), we apply Laplace smoothing (add-one smoothing):

$$N'(\alpha,\beta) = N(\alpha,\beta) + 1$$
$$C'(\beta) = C(\beta) + K$$

where K is the number of possible symbols.
Then, the smoothed transition probabilities are:

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

Similarly, we adjust the unigram counts:

$$C'(\gamma) = C(\gamma) + 1$$
$$N' = N + K$$

and the smoothed initial probabilities are:

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

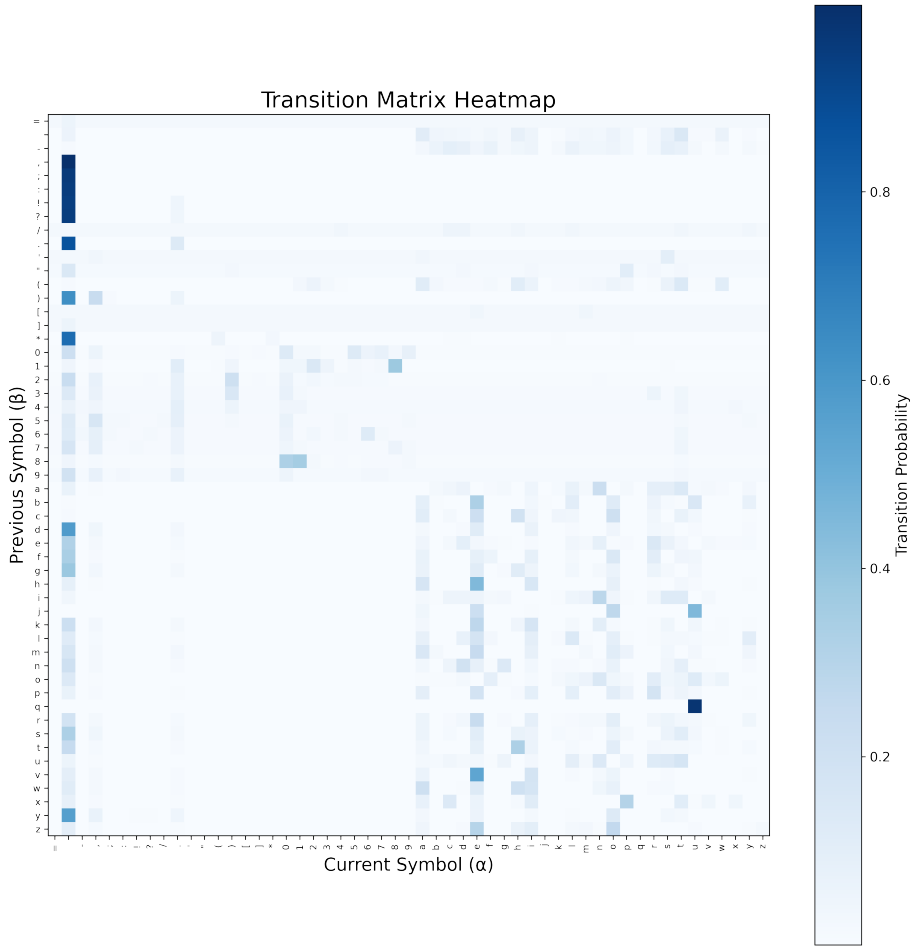


Figure 13: Heatmap representation of the transition matrix of the English language, sampled from the English translation of War and Peace by Tolstoy.

5.2 Part (b)

Question 1: Are the latent variables $\sigma(s)$ for different s independent?
No, the latent variables $\sigma(s)$ are **not independent**. The mapping σ is a permutation, which is a bijective function from the set of symbols to itself. This means that if $\sigma(s)$ maps symbol s to a particular encrypted symbol, then no other symbol s' can be mapped to the same encrypted symbol. Therefore, the assignments of $\sigma(s)$ for different s are constrained by the permutation, making them dependent.
Question 2: Write down the joint probability $P(e_1e_2 \dots e_n \mid \sigma)$ in terms of σ and the transition probabilities.

Let $e = e_1e_2 \dots e_n$ be the encrypted message, and $s = s_1s_2 \dots s_n$ be the plaintext message. The mapping σ maps plaintext symbols to encrypted symbols, so $e_i = \sigma(s_i)$. Equivalently, $s_i = \sigma^{-1}(e_i)$.

The joint probability of the encrypted text given σ is:

$$P(e_1e_2 \dots e_n \mid \sigma) = P(s_1s_2 \dots s_n) = \phi(s_1) \prod_{i=2}^{n} \psi(s_i, s_{i-1})$$

Since $s_i = \sigma^{-1}(e_i)$ and $s_{i-1} = \sigma^{-1}(e_{i-1})$, we have:

$$P(e_1e_2 \dots e_n \mid \sigma) = \phi(\sigma^{-1}(e_1)) \prod_{i=2}^{n} \psi(\sigma^{-1}(e_i), \sigma^{-1}(e_{i-1}))$$

5.3 Part (c)

Question 1: How does the proposal probability $S(\sigma \to \sigma')$ depend on σ and σ'?

In the Metropolis-Hastings algorithm, we propose a new permutation σ' by swapping two randomly chosen symbols in the current permutation σ. Since there are K symbols, there are $\binom{K}{2}$ possible pairs of symbols to swap.

The proposal probability $S(\sigma \to \sigma')$ is:

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

This is because each possible swap is equally likely, and the total number of possible swaps is $\binom{K}{2}$.
Symmetry of the Proposal Distribution:
The proposal distribution is symmetric:

$$S(\sigma \to \sigma') = S(\sigma' \to \sigma)$$

since swapping symbols s and s' in σ yields σ', and swapping them back in σ' yields σ, with the same probability.
Question 2: What is the Metropolis-Hastings acceptance probability for a given proposal?

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

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

Since $S(\sigma \to \sigma') = S(\sigma' \to \sigma)$, the acceptance probability simplifies to:

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

This is because the prior over permutations is uniform, and the proposal distribution is symmetric.

5.4 Part (d)

The full implementation in Python is given in the Appendix Problem 5, Part (d), showing full code in the main text affect readability. Below is a overview of the implementation details:

1. **Initialization:**
   - "Intelligent" initial permutation of σ by matching the frequency of symbols in the encrypted message to those in the corpus.
   - Map the most frequent encrypted symbols to the most frequent plaintext symbols.
2. **Log-Likelihood Function:**
   - Compute the log-likelihood $\ell(\sigma)$ of the encrypted message given the permutation:

$$\ell(\sigma) = \log \phi(\sigma^{-1}(e_1)) + \sum_{i=2}^{n} \log \psi(\sigma^{-1}(e_i), \sigma^{-1}(e_{i-1}))$$

3. **Metropolis-Hastings Sampling Loop:**
   - At each iteration:
     (a) Propose a new permutation σ' by swapping two randomly chosen symbols.
     (b) Compute $\Delta\ell = \ell(\sigma') - \ell(\sigma)$.
     (c) Accept σ' with probability $A = \min(1, e^{\Delta\ell})$.
4. **Monitoring Progress:**
   - Every few iterations, decrypt the first 60 symbol of the message using the current permutation and display it to monitor progress.

Implementation Details:

- **Handling Zero Probabilities:**
  - Apply Laplace smoothing to ensure that all probabilities $\phi(\gamma)$ and $\psi(\alpha, \beta)$ are non-zero.
- **Optimization:**
  - Store logarithms of probabilities to avoid underflow and improve computational efficiency.
- **"Intelligent" Initialization:**
  - Improves convergence by starting from a permutation close to the true mapping. (we used the Laplace Smoothing of the true mapping to handle 0 probabilities)

Output:

```
Iteration 100: on vw wmlnyes inh vmse fldnesipde weisa vw gitres yife ve am
Iteration 300: on vr rslnyem anh vsme flcnemapce reami vr gatwem yafe ve is
Iteration 400: on vr rilnyed ank vide flcnedapce reads vr gatwed yafe ve si
Iteration 600: on vy yilnder ank vire flcnerajce years vy gatwer dafe ve si
Iteration 700: on vy yilnder ang vire hlcnerajce years vy katwer dahe ve si
Iteration 800: on vy yilnger and vire hlcnerafce years vy katwer gahe ve si
Iteration 900: on vy yilnger and vire hlcnerafce years vy katwer gahe ve si
Iteration 1000: on vy yilnger and vire hlcnerafce years vy katwer gahe ve si
Iteration 1100: on vy yilnger and vire hlcnerafce years vy katwer gahe ve si
Iteration 1200: on vy yilnger and vire hlcnerafce years vy katwer gahe ve si
Iteration 1300: in hy younger and hore vucnerafce years hy katwer gave he so
Iteration 1400: in hy younger and hore vucnerabce years hy katwer gave he so
Iteration 1500: in hy younger and hore vucnerabce years hy katwer gave he so
Iteration 1600: in hy younger and hore vucnerabce years hy fatwer gave he so
Iteration 1700: in hy younger and hore vucnerabce years hy fatwer gave he so
```

```
Iteration 1800: in hy younger and hore vucnerabce years hy fatler gave he so
Iteration 2100: in hy younger and hore vumnerabme years hy fatler gave he so
Iteration 2200: in hy younger and hore vumnerabme years hy fatler gave he so
Iteration 2300: in hy younger and hore vulnerable years hy fatmer gave he so
Iteration 2400: in hy younger and hore vulnerable years hy fatmer gave he so
Iteration 2500: in hy younger and hore vulnerable years hy fatmer gave he so
Iteration 2600: in hy younger and hore vulnerable years hy fatmer gave he so
Iteration 2700: in my younger and more vulnerable years my father gave me so
Iteration 2800: in my younger and more vulnerable years my father gave me so
...
```

The decrypted text did not change after iteration 2800, below we show how log-likelihood improves over iteration, :

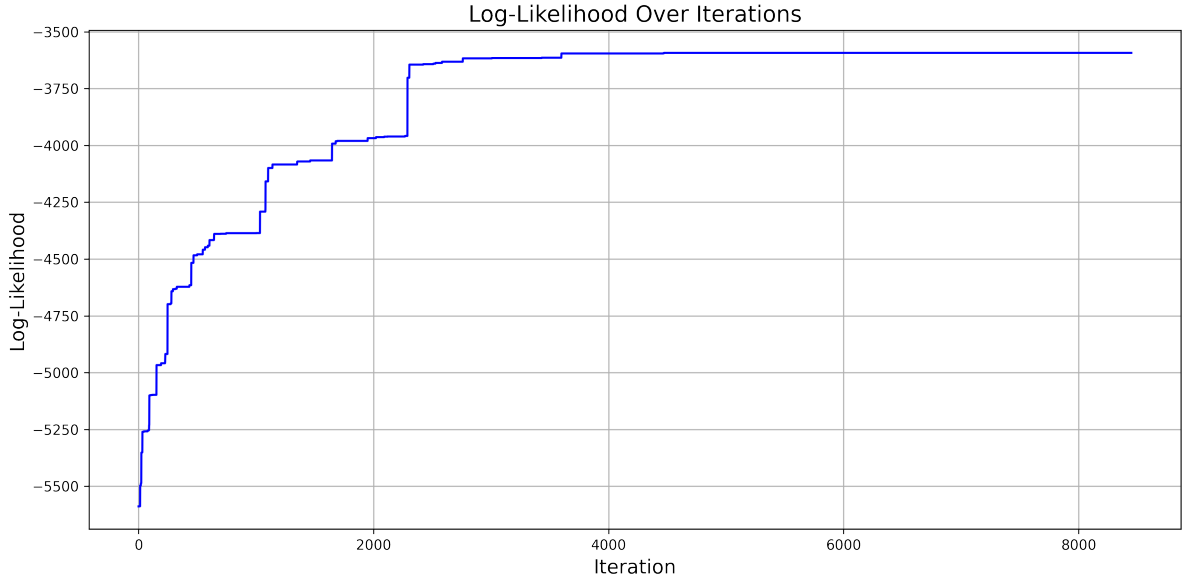


Figure 14: The log-likelihood stops improving around 3700 iterations, but the decrypted text remains unchanged after approximately 2800 iterations. This is acceptable because, by the 2800th iteration, the log-likelihood has already reached a value close to convergence, meaning that further iterations don't significantly affect the decrypted text.

5.5 Part (e)

Effect of Zero Transition Probabilities: Zero values in $\psi(\alpha, \beta)$ imply that certain transitions between symbols have zero probability under the model. If the decrypted message under a permutation σ results in bigrams with zero transition probabilities, the likelihood $P(e \mid \sigma)$ becomes zero. This leads to an acceptance probability $A(\sigma \to \sigma') = 0$, preventing the chain from moving to certain states.

Impact on Ergodicity:

- The Markov chain becomes **non-ergodic** because it is not possible to reach all states (permutations) from any starting state due to zero probabilities.
- The chain lacks irreducibility, which is a condition for ergodicity.

Restoring Ergodicity:

- **Solution:** Apply Laplace smoothing to the bigram counts to ensure that all transition probabilities are positive.

$$N'(\alpha, \beta) = N(\alpha, \beta) + 1$$

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

- This adjustment ensures that $\psi(\alpha, \beta) > 0$ for all α, β, making the Markov chain irreducible and thus ergodic.

5.6 Part (f)

Question 1: Would symbol probabilities alone (unigram frequencies) be sufficient to decode the message?

No, using symbol probabilities alone (unigram frequencies) would not be sufficient for accurate decoding. Unigram frequencies ignore any kind of dependencies (orders) in the contextual information provided by symbol sequences (bigrams or higher-order dependencies). English text has many symbols and words that appear with similar frequencies, and without considering the order of symbols, we cannot capture the structure and grammar of the language. Therefore, relying solely on unigram frequencies would result in a poor decryption.

Question 2: If we used a second-order Markov chain for English text, what problems might we encounter?

A second-order Markov chain models the probability of a symbol based on the two preceding symbols:

$$P(s_i \mid s_{i-1}, s_{i-2})$$

Potential Problems:

- **Data Sparsity:** The number of possible trigram combinations increases exponentially with the number of symbols, leading to many combinations with zero or very low counts in the corpus. (the curse of dimensionality).
- **Computational Complexity:** Storing and computing probabilities for all possible trigrams requires significant memory and processing power.
- **Overfitting:** The model may overfit to the training corpus, reducing its generalization to other texts.
- **Non-Ergodicity:** Increased zero probabilities exacerbate the issues with ergodicity, making it more difficult for the MCMC sampler to explore the space of permutations.

Question 3: Will the approach work if the encryption scheme allows two symbols to be mapped to the same encrypted value (i.e., the mapping is not one-to-one)?

No, the approach relies on the encryption mapping being a bijection (one-to-one and onto). If multiple plaintext symbols map to the same encrypted symbol, the mapping is not invertible, and we cannot uniquely recover the plaintext symbols from the encrypted text. The permutation σ cannot represent such a mapping, and the MCMC sampler would not be able to find a valid permutation to decrypt the message.

Question 4: Would the approach work for a language like Chinese with over 10,000 symbols?

Applying this approach to a language with a very large number of symbols, such as Chinese, would be impractical due to several reasons:

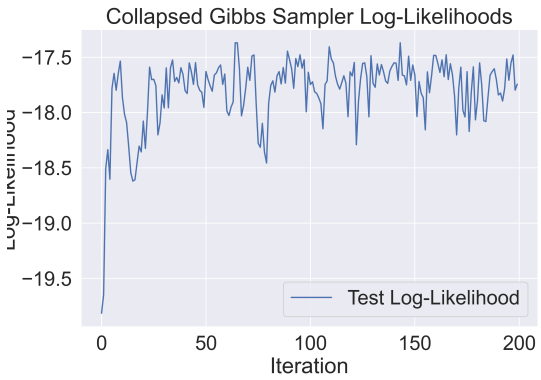
- **Permutation Space Size:** The number of possible permutations is $10,000!$, which is computationally infeasible to explore using MCMC.
- **Data Requirements:** Estimating reliable transition probabilities for all symbol pairs would require an enormous amount of text data.
- **Computational Resources:** Storing and processing transition probability matrices of size $10,000 \times 10,000$ exceeds typical memory capacities.
- **Sampling Efficiency:** The MCMC sampler would have difficulty converging in such a vast space, and the likelihood of finding the correct mapping would be negligible (again the curse of dimensionality) .

6 Problem6: Implementing Gibbs sampling for LDA [BONUS]

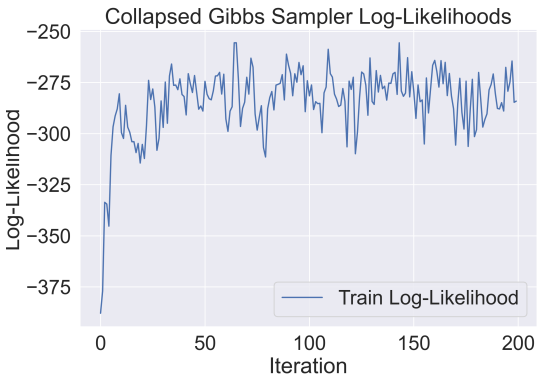
The provided Python code for the Gibbs samplers uses outdated versions of Numpy and Seaborn. Its default visualizations were uninformative, and the variable names did not match those specified in the README and MATLAB code. I have reconstructed most of the code to enhance visualization and readability. The original code is structured into two large classes. We followed this structure in our implementation, but this significantly impacts readability when presenting the code in the main text. Therefore, we will include only the necessary figures and a summary of the code implementation in the main text. Links to the code be provided, directing to the relevant code sections for each part of the problem. The full implementation is available in the appendix.

6.1 Part (a)

The full implementation in Python is given in the Appendix Problem 6, Part (a) and (b)

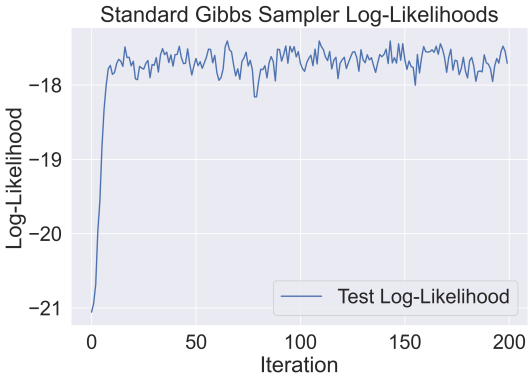


(a) collapsed Gibbs log pred (test)



(b) collapsed Gibbs log joint (train)

Figure 15: collapsed Gibbs sampler



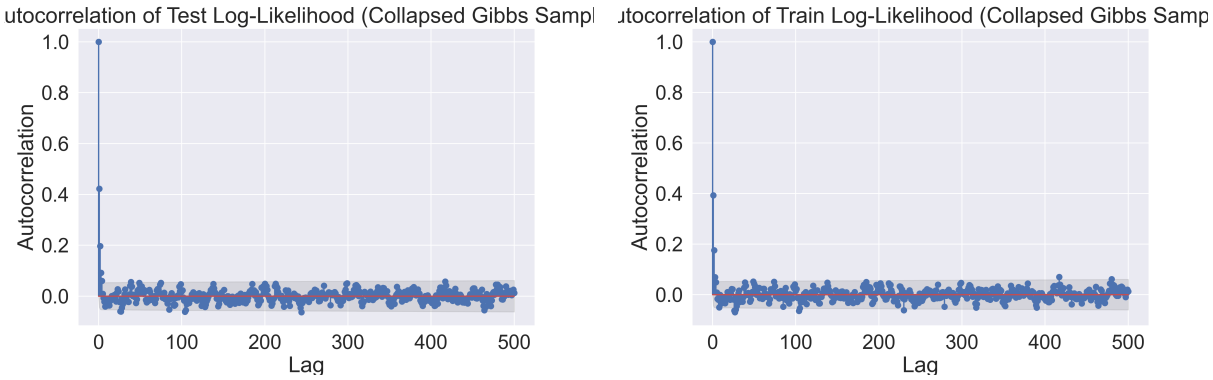
(a) standard Gibbs log pred (test)

(b) standard Gibbs log joint (train)

Figure 16: standard Gibbs sampler

6.2 Part (b)

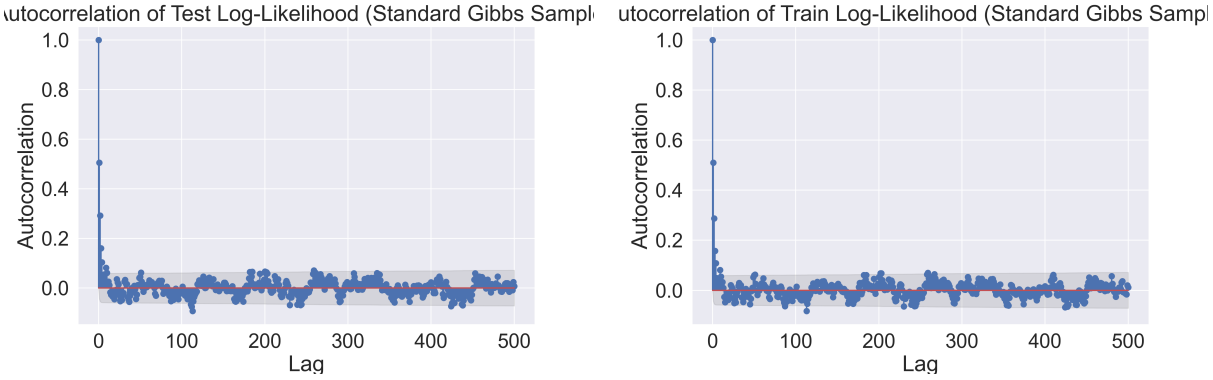
The full implementation in Python is given in the Appendix Problem 6, Part (a) and (b)



(a) collapsed Gibbs autocorr pred (test)

(b) collapsed Gibbs autocorr joint (train)

Figure 17: collapsed Gibbs sampler



(a) standard Gibbs autocorr pred (test)

(b) standard Gibbs autocorr joint (train)

Figure 18: standard Gibbs sampler

For both standard and collapsed Gibbs samplers, the effective "burn-in" period is approximately 25 iterations. We ran 2000 iterations to generate the autocorrelation plots, where the periodic trend is clearly visible. This periodicity reflects the base frequency of the original joint likelihood arrays (with periodicity of 50), indicating that the likelihood arrays themselves also exhibit periodic behavior. Hence the same periodicity of 50 must also occour in the sample of the posterior so I say 50 samples can roughly represent the set of samples from the posterior

6.3 Part (c)

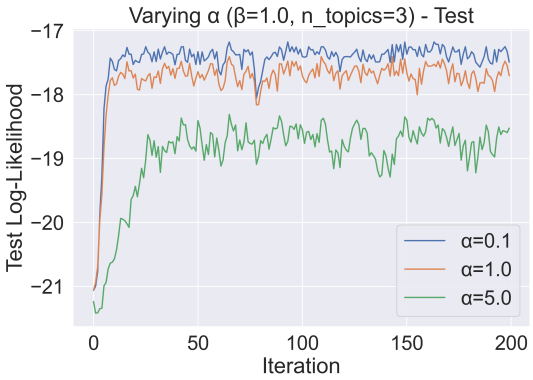
The collapsed Gibbs sampler converges faster than the standard Gibbs sampler, as it reaches a steady state more quickly and exhibits lower periodicity in the autocorrelation plots. This reduced periodicity means that fewer samples are needed to accurately represent the posterior distribution, allowing for more efficient transitions between samples. Consequently, the collapsed Gibbs sampler achieves convergence with fewer iterations, demonstrating greater efficiency compared to the standard Gibbs sampler.

6.4 Part (d)

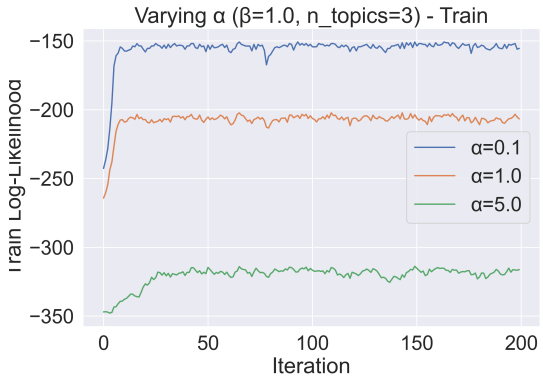
The full implementation in Python is given in the Appendix Problem 6, Part (d)

We performed a parameter sweep by varying α, β, and the number of topics n_{topics} while keeping the other parameters fixed. Specifically, we explored the following ranges:

- α values: {0.1, 1.0, 5.0}
- β values: {0.1, 1.0, 5.0}
- n_{topics} values: {2, 3, 5}

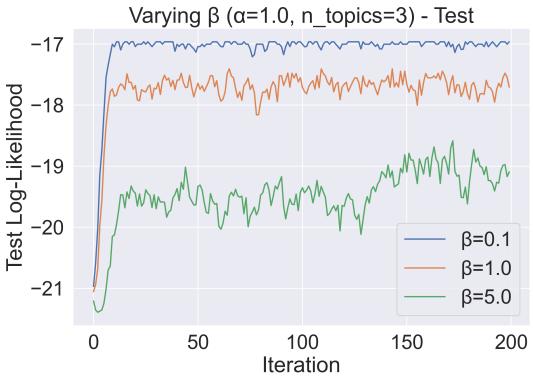


(a) collapsed Gibbs autocorr pred (test)

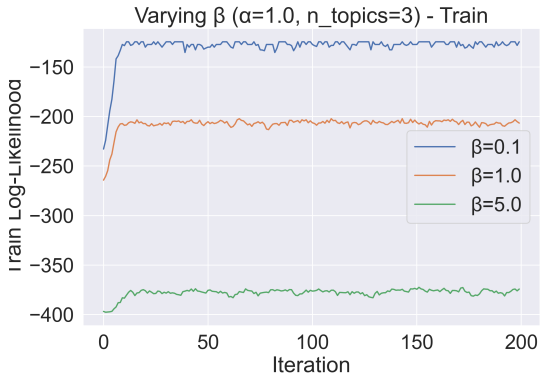


(b) collapsed Gibbs autocorr joint (train)

Figure 19: collapsed Gibbs sampler

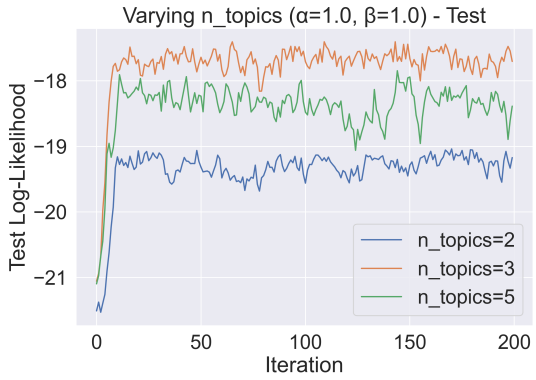


(a) collapsed Gibbs autocorr pred (test)

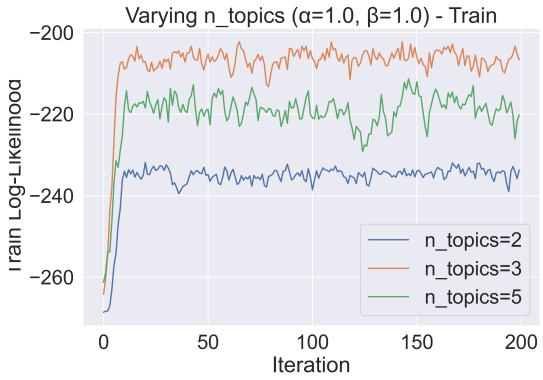


(b) collapsed Gibbs autocorr joint (train)

Figure 20: collapsed Gibbs sampler



(a) collapsed Gibbs autocorr pred (test)



(b) collapsed Gibbs autocorr joint (train)

Figure 21: collapsed Gibbs sampler

Both α and β exhibit similar characteristics in the log-predictive and joint likelihoods. A smaller α or β yields higher probabilities, as α and β controls the prior on θ, the document-topic distribution. In a Dirichlet distribution, larger α results in more concentrated θ values (low variance), while smaller α allows θ to spread more (high variance). Since documents typically focus on a few specific topics rather than all, a smaller α is more suitable as a prior.

For both joint probabilities and log-predictive likelihoods, $n_{topics} = 3$ yields the highest probabilities, followed by $n_{topics} = 5$, and $n_{topics} = 2$ having the lowest. Since the true data is generated from 3 topics, it is expected that $n_{topics} = 3$ performs best. While $n_{topics} = 5$ represents a more complex model, it performs worse, possibly due to limited data, making it harder to train a well-converged model. Conversely, $n_{topics} = 2$ is too simplistic to fully capture the topic structure.

6.5 Part (e)

The full implementation in Python is given in the Appendix Problem 6, Part (e) and (f) Tf-idf balances the frequency of a word in a document (term frequency) with how unique that word is across all documents (inverse document frequency).

Term Frequency (TF) measures how frequently a term occurs in a document, while Inverse Document Frequency (IDF) measures how important a term is across the corpus. By combining TF and IDF, tf-idf highlights words that are both frequent in a document and unique across the corpus.

I computed the tf-idf scores for all words in the vocabulary and then averaged these scores across all documents to get a single tf-idf score per word. I sorted the words based on these scores and selected the top 500 words.

To ensure that important keywords like "Bayesian," "graphical," "Gaussian," "support," "vector," "kernel," "representation," "regression," and "classification" were included, I inspected the top words and confirmed their presence. Most of these keywords naturally had high tf-idf scores due to their importance in the NeurIPS corpus.

6.6 Part (f)

The full implementation in Python is given in the Appendix Problem 6, Part (e) and (f) After reducing the vocabulary, I ran Latent Dirichlet Allocation (LDA) using a Collapsed Gibbs Sampler on the reduced dataset. I experimented with various settings of the hyperparameters α (alpha), β (beta), and the number of topics K to find reasonable topics.

I tested the following settings:

- Alpha (α): 0.1, 1.0
- Beta (β): 0.1, 1.0
- Number of Topics (K): 5, 10, 15

For each combination, I ran the Gibbs sampler for 300 iterations and observed the top words for each topic.

Observations

Effect of Number of Topics (K):

With $K = 5$, the topics were broad and encompassed multiple themes. For example, words related to "model," "data," and "algorithm" appeared together. As K increased to 10, the topics became more specific, and distinct themes emerged, such as "speech recognition," "neural networks," and "control systems." With $K = 15$, the topics were even more granular, capturing niche areas like "circuit design," "reinforcement learning," and "theoretical bounds."

Effect of Alpha (α):

A lower α (0.1) encourages documents to use fewer topics, resulting in more specific and coherent topics. A higher α (1.0) allows documents to mix more topics, making the topics broader and the distinctions between them less clear.

Effect of Beta (β):

A lower β (0.1) encourages topics to use fewer words, leading to more distinctive top words. A higher β (1.0) allows topics to use more words, resulting in overlap between topics and less distinctiveness.

Qualitative Changes in Topics

As α or β increases, topics become less sharp, and words common across multiple topics appear more frequently. Documents are modeled as mixtures of more topics, which can blur the distinctions between topics.

As K increases, the model captures more nuanced and specific topics. However, if K is too large, some topics may become too narrow or capture noise.

Sample Topics

With $\alpha = 0.1$, $\beta = 0.1$, $K = 10$:

- **Topic 1**: Signal processing (signal, noise, information, component, filter, channel, frequency, analysis, system, source)
- **Topic 7**: Neural networks (network, unit, input, weight, output, neural, layer, hidden, learning, training)
- **Topic 10**: Reinforcement learning (learning, algorithm, problem, action, function, step, optimal, task, reinforcement, policy)

With $\alpha = 1.0$, $\beta = 1.0$, $K = 15$:

- **Topic 2**: Image recognition (image, object, images, features, recognition, pixel, feature, region, visual, view)
- **Topic 5**: Classification algorithms (training, set, data, classification, classifier, performance, pattern, class, test, network)
- **Topic 13**: Probabilistic models (model, data, distribution, parameter, gaussian, probability, mean, likelihood, prior, mixture)

By adjusting α, β, and K, I observed significant qualitative changes in the topics produced by LDA. Lower values of α and β led to more distinct and coherent topics, while higher values resulted in broader topics with more word overlap. Increasing K allowed the model to capture more specific themes within the corpus.

Overall, using tf-idf to reduce the vocabulary made the dataset manageable for LDA without losing critical information. Experimenting with LDA hyperparameters demonstrated the importance of tuning α, β, and K to obtain meaningful topics. The insights gained align with the theoretical expectations of how these parameters influence topic modeling.

Sample Output for $\alpha = 0.1$, $\beta = 0.1$, $K = 10$

```
Running LDA on reduced NeurIPS data

Running with =0.1, =0.1, n_topics=10
Collapsed Gibbs Sampling: 100%
```

```
Top words for =0.1, =0.1, n_topics=10
Topic 1: signal, noise, information, component, filter, channel, frequency, analysis, system, source
Topic 2: set, training, data, error, method, classifier, algorithm, classification, test, performanc
Topic 3: image, object, visual, images, model, map, field, motion, direction, representation
Topic 4: model, data, distribution, parameter, gaussian, probability, mean, algorithm, likelihood, m
Topic 5: neuron, cell, input, model, circuit, synaptic, neural, pattern, network, chip
Topic 6: recognition, word, speech, system, training, context, hmm, sequence, character, speaker
Topic 7: network, unit, input, weight, output, neural, layer, hidden, learning, training
Topic 8: model, system, network, control, dynamic, neural, learning, point, motor, memory
Topic 9: function, vector, error, weight, learning, linear, case, result, bound, point
Topic 10: learning, algorithm, problem, action, function, step, optimal, task, reinforcement, policy
```

Sample Output for $\alpha = 1.0$, $\beta = 1.0$, $K = 15$

```
Running LDA on reduced NeurIPS data

Running with =1.0, =1.0, n_topics=15
Collapsed Gibbs Sampling: 100%
Top words for =1.0, =1.0, n_topics=15
Topic 1: signal, circuit, output, system, chip, analog, input, current, filter, noise
Topic 2: image, object, images, features, recognition, pixel, feature, region, visual, view
Topic 3: node, nodes, representation, memory, tree, rules, level, set, rule, learning
Topic 4: vector, data, space, function, matrix, component, linear, point, kernel, dimensional
Topic 5: training, set, data, classification, classifier, performance, pattern, class, test, network
Topic 6: network, neural, system, dynamic, model, recurrent, equation, point, order, parameter
Topic 7: learning, control, action, task, system, reinforcement, policy, model, function, step
Topic 8: function, bound, number, theorem, result, approximation, case, threshold, set, probability
Topic 9: error, training, learning, weight, prediction, set, generalization, noise, function, input
Topic 10: neuron, cell, model, input, synaptic, activity, pattern, response, spike, firing
Topic 11: field, model, motion, direction, position, map, eye, system, movement, motor
Topic 12: algorithm, problem, learning, gradient, function, method, solution, point, convergence, co
Topic 13: model, data, distribution, parameter, gaussian, probability, mean, likelihood, prior, mixt
Topic 14: word, recognition, speech, system, context, training, hmm, character, sequence, model
Topic 15: network, unit, input, output, weight, layer, hidden, neural, net, learning
```

7 Problem7: Optimization

7.1 Part (a)

We are asked to find the local extrema of $f(x, y) = x+2y$ subject to the constraint $g(x, y) = y^2+xy-1 = 0$ using Lagrange multipliers. We introduce the Lagrangian:

$$\mathcal{L}(x, y, \lambda) = f(x, y) - \lambda(g(x, y))$$

Substitute the expressions for $f(x, y)$ and $g(x, y)$:

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

Now, we compute the partial derivatives of L with respect to x, y, and λ:

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial x} &= 1 - \lambda y = 0 \quad \Rightarrow \quad \lambda y = 1 \quad (1) \\
\frac{\partial \mathcal{L}}{\partial y} &= 2 - \lambda(2y + x) = 0 \quad (2) \\
\frac{\partial \mathcal{L}}{\partial \lambda} &= -(y^2 + xy - 1) = 0 \quad \Rightarrow \quad y^2 + xy = 1 \quad (3)
\end{aligned}$$

From equation (1), we solve for λ:

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

Substitute $\lambda = \frac{1}{y}$ into equation (2):

$$2 - \frac{1}{y}(2y + x) = 0$$

Simplify the equation:

$$2 - \frac{2y + x}{y} = 0 \quad \Rightarrow \quad 2 - \left(2 + \frac{x}{y}\right) = 0$$

This simplifies to:

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

Substitute $x = 0$ into the constraint equation (3):

$$y^2 + 0 \cdot y = 1 \quad \Rightarrow \quad y^2 = 1 \quad \Rightarrow \quad y = \pm 1$$

Thus, the critical points are $(x, y) = (0, 1)$ and $(x, y) = (0, -1)$.
Conclusion: The local extrema occur at the points:

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

7.2 part (b)

We aim to find $x = \ln(a)$ using Newton's method, given that we have access to a routine to compute $\exp(x)$.

(i) Deriving a function for Newton's method

We want to solve the equation $x = \ln(a)$, which is equivalent to solving:

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

Thus, we define the function $f(x, a)$ as:

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

(ii) Newton's update equation

Newton's method update rule is given by:

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

First, compute the derivative of $f(x, a)$:

$$f'(x, a) = \frac{d}{dx}[\exp(x) - a] = \exp(x)$$

Substituting this into the update rule:

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

This is the iterative update formula to compute $\ln(a)$ using Newton's method.

8 Problem8: Eigenvalues as solutions of an optimization problem. [BONUS]

8.1 Part (a)

We are given a symmetric matrix $A \in \mathbb{R}^{n\times n}$ and the Rayleigh quotient $R_A(x) = \dfrac{x^\top Ax}{x^\top x}$ for $x \in \mathbb{R}^n$. Our goal is to show that the supremum of $R_A(x)$ over $\mathbb{R}^n$ is attained.

Firstly, observe that for any $x \neq 0$, the value of $R_A(x)$ depends only on the direction of x, not its magnitude. Specifically, for any scalar $\alpha \neq 0$,

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

Therefore, it suffices to consider vectors x on the unit sphere $S = \{x \in \mathbb{R}^n \mid \|x\| = 1\}$.

Now, define the function $f : S \to \mathbb{R}$ by

$$f(x) = x^\top Ax = q_A(x).$$

Since A is symmetric and q_A is continuous, f is continuous on S.

The unit sphere S in $\mathbb{R}^n$ is a closed and bounded subset of $\mathbb{R}^n$, and hence it is compact by the Heine–Borel theorem.

By the extreme value theorem, a continuous function on a compact set attains its maximum and minimum values on that set. Therefore, f attains its maximum on S, say at some $x^* \in S$.

Since $\|x^*\| = 1$, we have

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

which shows that the supremum of $R_A(x)$ over $\mathbb{R}^n$ is indeed attained at x^*.

8.2 Part (b)

Let $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_n$ be the eigenvalues of A, and let $\{\xi_1, \xi_2, \ldots, \xi_n\}$ be the corresponding orthonormal eigenvectors forming an orthonormal basis (ONB) of $\mathbb{R}^n$.

Any vector $x \in \mathbb{R}^n$ can be expressed in terms of this ONB:

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

Define $\alpha_i = \xi_i^\top x$, so $x = \sum_{i=1}^n \alpha_i \xi_i$.

Compute $x^\top Ax$:

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

Since $A\xi_j = \lambda_j\xi_j$ and $\xi_i^\top\xi_j = \delta_{ij}$ (Kronecker delta), we have

$$\xi_i^\top A\xi_j = \xi_i^\top(\lambda_j\xi_j) = \lambda_j\xi_i^\top\xi_j = \lambda_j\delta_{ij}.$$

Therefore,

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

Similarly, compute $x^\top x$:

$$x^\top x = \left(\sum_{i=1}^{n} \alpha_i \xi_i^\top\right)\left(\sum_{j=1}^{n} \alpha_j \xi_j\right) = \sum_{i=1}^{n} \alpha_i^2 \xi_i^\top\xi_i = \sum_{i=1}^{n} \alpha_i^2.$$

Therefore, the Rayleigh quotient becomes

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

Since $\lambda_i \leq \lambda_1$ for all i, and $\alpha_i^2 \geq 0$, we have

$$\sum_{i=1}^n \alpha_i^2 \lambda_i \leq \sum_{i=1}^n \alpha_i^2 \lambda_1 = \lambda_1 \sum_{i=1}^n \alpha_i^2.$$

Thus,

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

Therefore, $R_A(x) \leq \lambda_1$ for all $x \in \mathbb{R}^n$.
Moreover, if we choose $x = \xi_1$, then $\alpha_1 = \xi_1^\top \xi_1 = 1$ and $\alpha_i = 0$ for $i \neq 1$. Then,

$$R_A(\xi_1) = \frac{\alpha_1^2 \lambda_1}{\alpha_1^2} = \lambda_1.$$

Thus, $R_A(\xi_1) = \lambda_1$, showing that the maximum value λ_1 is attained by ξ_1.

8.3 Part (c)

Suppose λ_1 has multiplicity $k \geq 1$, and the corresponding eigenvectors are $\{\xi_1, \xi_2, \ldots, \xi_k\}$. Let $x \in \mathbb{R}^n$ be such that $x \notin \text{span}\{\xi_1, \xi_2, \ldots, \xi_k\}$.

Express x in terms of the orthonormal basis:

$$x = \sum_{i=1}^n \alpha_i \xi_i,$$

with $\alpha_i = \xi_i^\top x$.

Since $x \notin \text{span}\{\xi_1, \ldots, \xi_k\}$, there exists at least one $\alpha_j \neq 0$ for some $j > k$. That is, x has a non-zero component in the direction of an eigenvector corresponding to an eigenvalue $\lambda_j < \lambda_1$.

Compute $R_A(x)$:

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

Let $S = \sum_{i=1}^n \alpha_i^2$, which is positive since $x \neq 0$. We can write

$$R_A(x) = \lambda_1 \left(\frac{\sum_{i=1}^n \alpha_i^2 \frac{\lambda_i}{\lambda_1}}{S} \right).$$

Note that for $i > k$, $\lambda_i < \lambda_1$, so $\dfrac{\lambda_i}{\lambda_1} < 1$.

Split the sums:

$$R_A(x) = \lambda_1 \left(\frac{\sum_{i=1}^k \alpha_i^2 \frac{\lambda_i}{\lambda_1} + \sum_{i=k+1}^n \alpha_i^2 \frac{\lambda_i}{\lambda_1}}{S} \right).$$

Since $\lambda_i = \lambda_1$ for $i = 1, \ldots, k$, we have $\dfrac{\lambda_i}{\lambda_1} = 1$ for $i \leq k$.

Therefore,

$$R_A(x) = \lambda_1 \left(\frac{\sum_{i=1}^k \alpha_i^2 + \sum_{i=k+1}^n \alpha_i^2 \frac{\lambda_i}{\lambda_1}}{S} \right).$$

Since $\dfrac{\lambda_i}{\lambda_1} < 1$ for $i > k$, and $\alpha_i^2 \geq 0$, it follows that

$$\sum_{i=k+1}^{n} \alpha_i^2 \frac{\lambda_i}{\lambda_1} < \sum_{i=k+1}^{n} \alpha_i^2.$$

Therefore,

$$R_A(x) < \lambda_1 \left(\frac{\sum_{i=1}^{k} \alpha_i^2 + \sum_{i=k+1}^{n} \alpha_i^2}{S} \right) = \lambda_1 \frac{S}{S} = \lambda_1.$$

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

A Python Code Implementations

A.1 Problem1

Part (d)

MLE and Visualisation Python Code:

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

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

# Compute ML estimates
p_ml = np.mean(data, axis=0)

# Reshape and display the parameters as an 8x8 image
p_ml_image = p_ml.reshape((8, 8)).T  # Transpose for correct orientation

plt.imshow(p_ml_image, cmap='gray', interpolation='nearest')
plt.title('MLE Estimate of p')
plt.colorbar(label='Probability')
plt.show()
```

Listing 1: MLE Python Code Implementations

Part (e)

MAP and Visualisation Python Code:

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

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

# Hyperparameters
alpha = 3
beta = 3

# Compute MAP estimates
S = np.sum(data, axis=0)
p_map = (S + alpha - 1) / (N + alpha + beta - 2)

# Reshape and display the parameters as an 8x8 image
p_map_image = p_map.reshape((8, 8)).T  # Transpose for correct orientation

plt.imshow(p_map_image, cmap='gray', interpolation='nearest')
plt.title('MAP Estimate of p (alpha=3, beta=3)')
plt.colorbar(label='Probability')
plt.show()
```

Listing 2: MAP Python Code Implementations with $\alpha = \beta = 3$

A.2 Problem2

```
import numpy as np
# Load the dataset from the given text file
X = np.loadtxt('binarydigits.txt')
# Retrieve the number of samples (N) and number of features/pixels (D)
N, D = X.shape
# Step 2: Compute the statistics
S = np.sum(X)               # Total number of ones in the dataset
S_d = np.sum(X, axis=0)    # Number of ones for each pixel across all samples
# Step 3: Calculate the marginal likelihoods

# Model (a): All D components are generated from a Bernoulli distribution with p_d = 0.5
total_data_points = N * D
loglikelihood_a = total_data_points * np.log(0.5)  # Log-likelihood for Model (a)

def ln_gamma(n):
    """
    Compute the natural logarithm of the Gamma function using Stirling's approximation.
    """
    epsilon = 1e-10
    return (n - 0.5) * np.log(n + epsilon) - n + 0.5 * np.log(2 * np.pi)

# Model (b): Shared unknown Bernoulli parameter p
a_b = S + 1
b_b = total_data_points - S + 1
loglikelihood_b = ln_gamma(a_b) + ln_gamma(b_b) - ln_gamma(a_b + b_b)

# Model (c): Independent unknown Bernoulli parameters p_d for each pixel
loglikelihood_c = 0.0
for d in range(D):
    a_c = S_d[d] + 1
    b_c = N - S_d[d] + 1
    loglikelihood_c += ln_gamma(a_c) + ln_gamma(b_c) - ln_gamma(a_c + b_c)

# Compute log denominators using log-sum-exp for numerical stability
log_nominator_a = loglikelihood_a  # Assuming prior log P(Ma) is 0
log_nominator_b = loglikelihood_b  # Assuming prior log P(Mb) is 0
log_nominator_c = loglikelihood_c  # Assuming prior log P(Mc) is 0
# Stack log-nominators for log-sum-exp
log_nominators = np.array([log_nominator_a, log_nominator_b, log_nominator_c])
# Compute the log denominator using the log-sum-exp trick
max_log = np.max(log_nominators)
log_denominator = max_log + np.log(np.sum(np.exp(log_nominators - max_log)))
# Compute log posteriors
log_posterior_a = log_nominator_a - log_denominator
log_posterior_b = log_nominator_b - log_denominator
log_posterior_c = log_nominator_c - log_denominator
# Convert log posteriors to actual posterior probabilities
posterior_a = np.exp(log_posterior_a)
posterior_b = np.exp(log_posterior_b)
posterior_c = np.exp(log_posterior_c)
# Output results
print("Log Likelihoods:")
print(f"Log-likelihood for Model a: {loglikelihood_a}")
print(f"Log-likelihood for Model b: {loglikelihood_b}")
print(f"Log-likelihood for Model c: {loglikelihood_c}\n")
print("Log Denominator:")
print(f"The log denominator: {log_denominator}\n")
print("Log Nominators:")
print(f"Log nominator for Model a: {log_nominator_a}")
print(f"Log nominator for Model b: {log_nominator_b}")
print(f"Log nominator for Model c: {log_nominator_c}\n")
print("Log Posteriors:")
print(f"Log posterior for Model a: {log_posterior_a}")
print(f"Log posterior for Model b: {log_posterior_b}")
print(f"Log posterior for Model c: {log_posterior_c}\n")
print("Posteriors:")
print(f"Posterior for Model a: {posterior_a}")
print(f"Posterior for Model b: {posterior_b}")
print(f"Posterior for Model c: {posterior_c}")
```

Listing 3: Posterior Probability Computation

A.3 Problem3

Part(d) and Part(e)

EM algorithm Python Code:

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


def em_algorithm(X, K, max_iter=100, tol=1e-6):
    """
    Perform the Expectation-Maximization (EM) algorithm for a Bernoulli mixture model.

    Parameters:
        X (np.ndarray): The data matrix of shape (N, D), where N is the number of
            samples
                        and D is the dimensionality of each sample.
        K (int): The number of mixture components (clusters).
        max_iter (int): The maximum number of iterations to run the algorithm.
        tol (float): The tolerance for convergence based on log-likelihood improvement.

    Returns:
        pi_k (np.ndarray): Mixing coefficients of shape (K,), representing the prior
            probability
                           of each mixture component.
        P_kd (np.ndarray): Bernoulli parameters of shape (K, D), representing the
            probability
                           of each feature being 1 in each mixture component.
        log_likelihoods (list): Log-likelihood values at each iteration, useful for
            monitoring
                                convergence.
    """
    N, D = X.shape  # Number of samples and dimensionality
    # Initialize mixing coefficients uniformly: pi_k = 1/K
    pi_k = np.full(K, 1 / K)
    # Initialize Bernoulli parameters randomly in [0.25, 0.75]: P_kd
    np.random.seed(0)  # Seed for reproducibility replace when run multiple times
    P_kd = np.random.rand(K, D) * 0.5 + 0.25
    log_likelihoods = []  # To store log-likelihood at each iteration

    for iteration in range(max_iter):
        # E-step: Compute responsibilities gamma_nk = P(k | x_n)
        log_gamma_nk = np.zeros((N, K))  # Log responsibilities for numerical stability
        for k in range(K):
            # Compute log probability of data under component k
            # Using Bernoulli log-likelihood: x_n * log(p_kd) + (1 - x_n) * log(1 - p_kd
                )
            log_prob = (
                X @ np.log(P_kd[k] + 1e-10)  # Prevent log(0)
                + (1 - X) @ np.log(1 - P_kd[k] + 1e-10)
            )
            # Add log mixing coefficient
            log_gamma_nk[:, k] = np.log(pi_k[k] + 1e-10) + log_prob

        # Normalize responsibilities using log-sum-exp for numerical stability
        max_log_gamma = np.max(log_gamma_nk, axis=1, keepdims=True)
        log_gamma_nk -= max_log_gamma  # Shift for numerical stability
        gamma_nk = np.exp(log_gamma_nk)  # Convert log to normal scale
        gamma_nk /= gamma_nk.sum(axis=1, keepdims=True)  # Normalize to sum to 1

        # M-step: Update parameters based on responsibilities
        N_k = gamma_nk.sum(axis=0)  # Effective number of samples per component
        pi_k = N_k / N  # Update mixing coefficients: pi_k = N_k / N
        P_kd = (gamma_nk.T @ X) / N_k[:, np.newaxis]  # Update Bernoulli parameters

        # Compute log-likelihood for convergence check
        log_likelihood = np.sum(
            max_log_gamma.flatten()
            + np.log(np.sum(np.exp(log_gamma_nk), axis=1) + 1e-10)
        )
        log_likelihoods.append(log_likelihood)

        # Check for convergence based on the improvement in log-likelihood
```

```
        if iteration > 0 and np.abs(log_likelihood - log_likelihoods[-2]) < tol:
            print(f"Converged at iteration {iteration} for K = {K}")
            break

    return pi_k, P_kd, log_likelihoods


def plot_mixture_components(P_kd):
    """
    Visualize the Bernoulli parameters of each mixture component as 8x8 grayscale images
        .

    Each mixture component's parameters are reshaped into an 8x8 grid, allowing
        visualization
    of the feature probabilities as images.

    Parameters:
        P_kd (np.ndarray): Bernoulli parameters of shape (K, D), where K is the number
            of
                           mixture components and D is the dimensionality of each
                               component.
    """
    K, D = P_kd.shape  # Number of components and dimensionality
    plt.figure(figsize=(2 * K, 2))  # Adjust figure size based on K
    for k in range(K):
        plt.subplot(1, K, k + 1)
        plt.imshow(P_kd[k].reshape(8, 8), cmap='gray', interpolation='nearest')
        plt.axis('off')  # Hide axes for clarity
        plt.title(f'Component {k + 1}')  # Title each component
    plt.tight_layout()
    plt.show()


def plot_log_likelihoods(log_likelihood_dict):
    """
    Plot the log-likelihood curves for different K values on a single graph.

    This visualization helps in assessing the convergence behavior and comparing
    the performance across different numbers of mixture components.

    Parameters:
        log_likelihood_dict (dict): A dictionary where keys are K values and values are
                                    lists of log-likelihoods per iteration.
    """
    plt.figure(figsize=(10, 6))
    for K, log_likelihoods in log_likelihood_dict.items():
        plt.plot(
            log_likelihoods,
            marker='o',
            label=f'K = {K}'
        )
    plt.title('Log-Likelihood vs Iteration for Different K')
    plt.xlabel('Iteration')
    plt.ylabel('Log-Likelihood')
    plt.legend()
    plt.grid(True)
    plt.tight_layout()
    plt.show()


def main():
    """
    Main function to execute the EM algorithm on the binary digits dataset for various K
        values.

    The function performs the following steps:
    1. Loads the binary digits data.
    2. Defines a set of K values (number of mixture components) to evaluate.
    3. Runs the EM algorithm for each K, storing log-likelihoods and visualizing
        components.
    4. Plots all log-likelihood progressions on a single graph for comparison.
    """
    # Load the binary digits data from a text file
```

```
    try:
        data = np.loadtxt('binarydigits.txt')
    except IOError:
        print("Error: 'binarydigits.txt' not found. Check working directory.")
        return

    # Define the different values of K (number of mixture components) to evaluate
    K_values = [2, 3, 4, 7, 10]
    # Dictionary to store log-likelihoods for each K
    log_likelihood_dict = {}

    # Iterate over each K and perform the EM algorithm
    for K in K_values:
        print(f"\nRunning EM algorithm for K = {K}")
        # Execute the EM algorithm
        pi_k, P_kd, log_likelihoods = em_algorithm(data, K)

        # Store log-likelihoods in the dictionary
        log_likelihood_dict[K] = log_likelihoods

        # Visualize the learned mixture components
        plot_mixture_components(P_kd)

    # After processing all K's, plot all log-likelihoods on a single plot
    plot_log_likelihoods(log_likelihood_dict)


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

Listing 4: EM

A.4 Problem4

Part (a)

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

# Load training and test data
X_train = np.loadtxt('ssm_spins.txt').T
X_test = np.loadtxt('ssm_spins_test.txt').T

# Define A
theta1 = 2 * np.pi / 180
theta2 = 2 * np.pi / 90
A_rotation = np.array([
    [np.cos(theta1), -np.sin(theta1), 0, 0],
    [np.sin(theta1), np.cos(theta1), 0, 0],
    [0, 0, np.cos(theta2), -np.sin(theta2)],
    [0, 0, np.sin(theta2), np.cos(theta2)]
])
A = 0.99 * A_rotation

# Define Q
Q = np.eye(4) - A @ A.T

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

# Define R
R = np.eye(5)

# Initial state and covariance
y_init = np.zeros(4)
Q_init = np.eye(4)

# Run the Kalman filter
Y_filt, V_filt, _, L_filt = run_ssm_kalman(X_train, y_init, Q_init, A, Q, C, R, mode='
    filt')

# Plot the estimated states
plt.figure(figsize=(7, 6))
for i in range(Y_filt.shape[0]):
    plt.plot(Y_filt[i, :], label=f'State {i+1}')
plt.title('Estimated States (Filtering)')
plt.xlabel('Time')
plt.ylabel('State Value')
plt.legend()
plt.savefig('Y_fit.png', dpi=300)
plt.show()

# Plot the estimated states
plt.figure(figsize=(7, 6))
for i in range(Y_filt.shape[0]):
    plt.plot(Y_filt[i, :], label=f'State {i+1}')
plt.title('Estimated States (Filtering)')
plt.xlabel('Time')
plt.ylabel('State Value')
plt.legend()
plt.savefig('Y_fit.png', dpi=300)
plt.show()

# Run the Kalman smoother
Y_smooth, V_smooth, _, L_smooth = run_ssm_kalman(X_train, y_init, Q_init, A, Q, C, R,
    mode='smooth')

# Plot the estimated states
```

```
plt.figure(figsize=(7, 6))
for i in range(Y_smooth.shape[0]):
    plt.plot(Y_smooth[i, :], label=f'State {i+1}')
plt.title('Estimated States (Smoothing)')
plt.xlabel('Time')
plt.ylabel('State Value')
plt.legend()
plt.savefig('Y_smooth.png', dpi=300)
plt.show()

# Compute log-determinant of V_smooth
logdet_V_smooth = [np.linalg.slogdet(V_smooth[t])[1] for t in range(V_smooth.shape[0])]

plt.figure(figsize=(7, 6))
plt.plot(logdet_V_smooth)
plt.title('Log-Determinant of Covariance Matrices (Smoothing)')
plt.xlabel('Time')
plt.ylabel('Log-Determinant')
plt.savefig('V_smooth.png', dpi=300)
plt.show()
```

Listing 5: run Kalman filter and smoother

Part (b)

```
def em_algorithm(X, y_init, Q_init, A_init, Q_init_param, C_init, R_init, max_iters=50):
    """
    Run the EM algorithm to estimate parameters A, Q, C, R.
    """
    T = X.shape[1]
    k = A_init.shape[0]
    d = X.shape[0]

    # Initialize parameters
    A = A_init.copy()
    Q_param = Q_init_param.copy()
    C = C_init.copy()
    R = R_init.copy()

    log_likelihoods = []

    for iteration in range(max_iters):
        # E-step: Run Kalman smoother
        Y_smooth, V_smooth, V_joint, L = run_ssm_kalman(X, y_init, Q_init, A, Q_param, C
            , R, mode='smooth')
        total_log_likelihood = np.sum(L)
        log_likelihoods.append(total_log_likelihood)

        # Compute sufficient statistics

        # Sum over time of expected state outer products
        sum_y_yT = Y_smooth @ Y_smooth.T + np.sum(V_smooth, axis=0)

        # Sum over time of expected state at t and t-1 outer products
        sum_y_yPrevT = Y_smooth[:, 1:] @ Y_smooth[:, :-1].T + np.sum(V_joint[1:], axis
            =0)

        # Sum over time of expected previous state outer products
        sum_yPrev_yPrevT = Y_smooth[:, :-1] @ Y_smooth[:, :-1].T + np.sum(V_smooth[:-1],
             axis=0)

        # Sum over time of observations and state estimates
        sum_x_yT = X @ Y_smooth.T

        # Sum over time of observation outer products
        sum_x_xT = X @ X.T

        # Update C
        C_new = sum_x_yT @ np.linalg.inv(sum_y_yT)

        # Update R
        R_new = (sum_x_xT - C_new @ sum_x_yT.T) / T
        R_new = (R_new + R_new.T) / 2  # Ensure symmetry

        # Update A
        A_new = sum_y_yPrevT @ np.linalg.inv(sum_yPrev_yPrevT)

        # Update Q
        sum_y_yT_next = Y_smooth[:, 1:] @ Y_smooth[:, 1:].T + np.sum(V_smooth[1:], axis
            =0)
        Q_new = (sum_y_yT_next - A_new @ sum_y_yPrevT.T) / (T - 1)
        Q_new = (Q_new + Q_new.T) / 2  # Ensure symmetry

        # Update parameters
        A = A_new
        Q_param = Q_new
        C = C_new
        R = R_new

    return A, Q_param, C, R, log_likelihoods

em_results_random = []

# Initialize with true parameters
A_init = A.copy()
Q_init_param = Q.copy()
```

```
C_init = C.copy()
R_init = R.copy()

# Run EM
A_em_true, Q_em_true, C_em_true, R_em_true, ll_em_true = em_algorithm(
    X_train, y_init, Q_init, A_init, Q_init_param, C_init, R_init, max_iters=100

for i in range(10):
    # Random initialization
    A_init = np.random.randn(4, 4)
    Q_init_param = np.eye(4)
    C_init = np.random.randn(5, 4)
    R_init = np.eye(5)

    # Run EM
    A_em_rand, Q_em_rand, C_em_rand, R_em_rand, ll_em_rand = em_algorithm(
        X_train, y_init, Q_init, A_init, Q_init_param, C_init, R_init, max_iters=100
    )

    em_results_random.append({
        'A': A_em_rand,
        'Q': Q_em_rand,
        'C': C_em_rand,
        'R': R_em_rand,
        'log_likelihoods': ll_em_rand
    })

plt.figure(figsize=(12, 6))

# Plot for EM initialized at true parameters
plt.plot(ll_em_true, label='EM Init at True Parameters')

# Plot for EM runs with random initializations
for idx, result in enumerate(em_results_random):
    plt.plot(result['log_likelihoods'], label=f'EM Random Init {idx+1}')

plt.title('Log-Likelihood vs. EM Iterations')
plt.xlabel('Iteration')
plt.ylabel('Log-Likelihood')
plt.legend()
plt.savefig('runs.png', dpi=300)
plt.show()
```

Listing 6: get log-likelihood from test data

Part (c)

```
import pandas as pd

# Load the test data and transpose to match expected dimensions
X_test = np.loadtxt('ssm_spins_test.txt').T  # Shape: [d, T_test]

def compute_log_likelihood(X, y_init, Q_init, A, Q, C, R):
    _, _, _, likelihood = run_ssm_kalman(X, y_init, Q_init, A, Q, C, R, mode='filt')
    return np.sum(likelihood)

# Compute likelihood for True Parameters
ll_train_true = compute_log_likelihood(X_train, y_init, Q_init, A, Q, C, R)
ll_test_true = compute_log_likelihood(X_test, y_init, Q_init, A, Q, C, R)

# Compute likelihood for EM initialized at true parameters
ll_train_em_true = compute_log_likelihood(X_train, y_init, Q_init, A_em_true, Q_em_true,
     C_em_true, R_em_true)
ll_test_em_true = compute_log_likelihood(X_test, y_init, Q_init, A_em_true, Q_em_true,
    C_em_true, R_em_true)

# Compute likelihood for EM runs with random initializations
ll_train_em_random = []
ll_test_em_random = []

for result in em_results_random:
    A_em_rand = result['A']
    Q_em_rand = result['Q']
    C_em_rand = result['C']
    R_em_rand = result['R']

    ll_train = compute_log_likelihood(X_train, y_init, Q_init, A_em_rand, Q_em_rand,
        C_em_rand, R_em_rand)
    ll_test = compute_log_likelihood(X_test, y_init, Q_init, A_em_rand, Q_em_rand,
        C_em_rand, R_em_rand)

    ll_train_em_random.append(ll_train)
    ll_test_em_random.append(ll_test)

# If you have SSID parameters, include them similarly:
# ll_train_ssid = compute_log_likelihood(X_train, y_init, Q_init, A_ssid, Q_ssid, C_ssid
    , R_ssid)
# ll_test_ssid = compute_log_likelihood(X_test, y_init, Q_init, A_ssid, Q_ssid, C_ssid,
    R_ssid)

# Create a dictionary to store results
data = {
    'Model': ['True Parameters', 'EM Init at True Parameters'] + [f'EM Random Init {i+1}
        ' for i in range(10)],
    'Training Log-Likelihood': [ll_train_true, ll_train_em_true] + ll_train_em_random,
    'Test Log-Likelihood': [ll_test_true, ll_test_em_true] + ll_test_em_random
}

# Create a DataFrame
results_df = pd.DataFrame(data)
print(results_df)

# Set up the figure
plt.figure(figsize=(14, 8))
```

Listing 7: run EM with true parameters initialisation

A.5 Problem5

Part (d)

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


def load_symbols(symbols_file):
    """
    Load symbols from a file, replacing empty lines with a space character.

    Args:
        symbols_file (str): Path to the symbols file.

    Returns:
        list: A list of symbols.
    """
    with open(symbols_file, 'r') as f:
        symbols = [line.rstrip('\n') for line in f]
    # Replace empty strings with space character to represent '    space    '
    symbols = [symbol if symbol else '␣' for symbol in symbols]
    return symbols


# Load the list of symbols from 'symbols.txt'
symbols = load_symbols('symbols.txt')

# Create dictionaries to map symbols to indices and vice versa
symbol_to_index = {symbol: idx for idx, symbol in enumerate(symbols)}
index_to_symbol = {idx: symbol for idx, symbol in enumerate(symbols)}
N_symbols = len(symbols)  # Total number of unique symbols

# FOR DEGUGGING Verify that the space character is present in the symbol mappings
if '␣' not in symbol_to_index:
    raise ValueError("Space␣character␣'␣'␣is␣not␣present␣in␣symbols.txt.")

# Load the encrypted message from 'message.txt'
with open('message.txt', 'r') as f:
    encrypted_message = f.read().strip()

# FOR DEGUGGING Identify all unique symbols present in the encrypted message
unique_encrypted_symbols = set(encrypted_message)

# FOR DEGUGGING Check for any symbols in the encrypted message that are not in 'symbols.
    txt'
missing_symbols = unique_encrypted_symbols - set(symbols)
if missing_symbols:
    raise ValueError(f"Missing␣symbols␣in␣symbols.txt:␣{missing_symbols}")

# Convert the encrypted message symbols to their corresponding indices
encrypted_indices = np.array([symbol_to_index[s] for s in encrypted_message], dtype=np.
    int32)


def load_corpus(corpus_file):
    """
    Load and preprocess the corpus text by retaining only allowed symbols.

    Args:
        corpus_file (str): Path to the corpus file.

    Returns:
        np.ndarray: Array of symbol indices representing the corpus.
    """
    with open(corpus_file, 'r', encoding='utf-8') as f:
        text = f.read().lower()
    allowed_chars = set(symbols)
    # Replace any character not in allowed_chars with a space
    text = ''.join([char if char in allowed_chars else '␣' for char in text])
    # Convert text to indices
    text_indices = np.array([symbol_to_index[char] for char in text], dtype=np.int32)
    return text_indices
```

```
# Load and preprocess the corpus
corpus_indices = load_corpus('war_and_peace.txt')

# Vectorized unigram counts
unigram_counts = np.bincount(corpus_indices, minlength=N_symbols)

# Vectorized bigram counts
indices1 = corpus_indices[:-1]
indices2 = corpus_indices[1:]
bigram_counts = np.zeros((N_symbols, N_symbols), dtype=np.int32)
np.add.at(bigram_counts, (indices1, indices2), 1)

# Apply Laplace (add-one) smoothing to unigram and bigram counts to avoid zero
    probabilities
unigram_counts += 1
bigram_counts += 1

# Compute unigram probabilities (stationary distribution) by normalizing unigram counts
phi = unigram_counts / np.sum(unigram_counts)

# Compute bigram probabilities (transition probabilities) by normalizing bigram counts
bigram_totals = bigram_counts.sum(axis=1, keepdims=True)  # Sum of bigrams starting with
     each symbol
psi = bigram_counts / bigram_totals

# Precompute logarithms
log_phi = np.log(phi)
log_psi = np.log(psi)

# Vectorized encrypted unigram counts
encrypted_unigram_counts = np.bincount(encrypted_indices, minlength=N_symbols)

# Vectorized encrypted bigram counts
encrypted_indices1 = encrypted_indices[:-1]
encrypted_indices2 = encrypted_indices[1:]
encrypted_bigram_counts = np.zeros((N_symbols, N_symbols), dtype=np.int32)
np.add.at(encrypted_bigram_counts, (encrypted_indices1, encrypted_indices2), 1)

# Intelligent Initialization
# Step 1: Initial mapping based on unigram frequencies
# Sort plaintext symbols by descending unigram counts (most frequent first)
plaintext_indices_sorted = np.argsort(-unigram_counts)
# Sort ciphertext symbols by descending unigram counts in the encrypted message
ciphertext_indices_sorted = np.argsort(-encrypted_unigram_counts)

# Initialize the permutation array where index corresponds to ciphertext symbol
# and value corresponds to plaintext symbol
initial_permutation = np.zeros(N_symbols, dtype=np.int32)
initial_permutation[ciphertext_indices_sorted] = plaintext_indices_sorted

def permutation(permutation, plaintext_bigram_freq, ciphertext_bigram_freq, iterations
    =5):
    """
    Initial permutation by swapping symbol mappings to better align bigram frequencies.

    Args:
        permutation (np.ndarray): Current permutation array mapping ciphertext to
            plaintext indices.
        plaintext_bigram_freq (np.ndarray): Normalized bigram frequencies from the
            corpus.
        ciphertext_bigram_freq (np.ndarray): Normalized bigram frequencies from the
            encrypted message.
        iterations (int): Number of refinement iterations to perform.

    Returns:
        np.ndarray: Refined permutation array.
    """
    N_symbols = len(permutation)
    for _ in range(iterations):
        # Iterate over each ciphertext symbol
        for cipher_idx in range(N_symbols):
```

```
            # Get the current plaintext symbol mapped to this ciphertext symbol
            plain_idx = permutation[cipher_idx]
            # Compute the current score based on the absolute difference in bigram
                frequencies
            current_score = np.sum(
                np.abs(ciphertext_bigram_freq[cipher_idx] - plaintext_bigram_freq[
                    plain_idx])
            )
            # Try swapping with other ciphertext symbols to find a better mapping
            for other_cipher_idx in range(N_symbols):
                if other_cipher_idx == cipher_idx:
                    continue  # Skip swapping with itself
                other_plain_idx = permutation[other_cipher_idx]
                # Swap the plaintext mappings
                permutation[cipher_idx], permutation[other_cipher_idx] = other_plain_idx
                    , plain_idx
                # Compute the new score after swapping
                new_score = (
                    np.sum(
                        np.abs(ciphertext_bigram_freq[cipher_idx] -
                            plaintext_bigram_freq[other_plain_idx])
                    )
                    + np.sum(
                        np.abs(ciphertext_bigram_freq[other_cipher_idx] -
                            plaintext_bigram_freq[plain_idx])
                    )
                )
                # If the new score is better (lower), keep the swap
                if new_score < current_score:
                    current_score = new_score
                    plain_idx = other_plain_idx
                else:
                    # Revert the swap if it does not improve the score
                    permutation[cipher_idx], permutation[other_cipher_idx] = plain_idx,
                        other_plain_idx
    return permutation

# Normalize bigram counts to frequencies for comparison
plaintext_bigram_freq = bigram_counts / bigram_counts.sum()
ciphertext_bigram_freq = encrypted_bigram_counts / encrypted_bigram_counts.sum()

# Refine the initial permutation to better align bigram frequencies
current_permutation = initial_permutation.copy()
current_permutation = permutation(
    current_permutation, plaintext_bigram_freq, ciphertext_bigram_freq, iterations=3
)

# Precompute decrypted indices
decrypted_indices = current_permutation[encrypted_indices]

# Precompute initial log-likelihood
def compute_initial_log_likelihood(decrypted_indices, log_phi, log_psi):
    """
    Compute the initial log-likelihood of the decrypted message.

    Args:
        decrypted_indices (np.ndarray): Array of decrypted symbol indices.
        log_phi (np.ndarray): Logarithm of unigram probabilities.
        log_psi (np.ndarray): Logarithm of bigram probabilities.

    Returns:
        float: Log-likelihood value.
    """
    first_idx = decrypted_indices[0]
    log_likelihood = log_phi[first_idx]
    prev_indices = decrypted_indices[:-1]
    curr_indices = decrypted_indices[1:]
    log_likelihood += np.sum(log_psi[prev_indices, curr_indices])
    return log_likelihood

current_log_likelihood = compute_initial_log_likelihood(decrypted_indices, log_phi,
    log_psi)
best_permutation = current_permutation.copy()
```

```
best_log_likelihood = current_log_likelihood

# Metropolis-Hastings sampler parameters
num_iterations = 10000    # Total number of MH iterations
print_interval = 100     # Interval at which to print decrypted text

# Lists to store log-likelihoods and acceptance rates for plotting
log_likelihood_history = []
acceptance_history = []

# Counter for accepted proposals to calculate acceptance rate
accepted = 0

# Run the Metropolis-Hastings sampler
for iteration in range(1, num_iterations + 1):
    # Propose a new permutation by swapping two randomly selected symbols
    i, j = np.random.choice(N_symbols, 2, replace=False)

    # Create a proposal permutation
    proposal_permutation = current_permutation.copy()
    proposal_permutation[i], proposal_permutation[j] = proposal_permutation[j],
        proposal_permutation[i]

    # Update decrypted indices only at positions where the swapped symbols occur
    affected_positions = np.where((encrypted_indices == i) | (encrypted_indices == j))
        [0]

    # If there are no affected positions, continue to next iteration
    if affected_positions.size == 0:
        continue

    # Compute the difference in log-likelihood
    delta_log_likelihood = 0.0

    # Positions to consider for delta computation (affected positions and their
        neighbors)
    positions = np.unique(np.concatenate([affected_positions - 1, affected_positions,
        affected_positions + 1]))
    positions = positions[(positions >= 0) & (positions < len(encrypted_indices))]

    # Current decrypted indices at these positions
    current_dec_indices = decrypted_indices[positions]

    # Proposed decrypted indices at these positions
    proposed_dec_indices = current_dec_indices.copy()
    mapping = {current_permutation[i]: proposal_permutation[i], current_permutation[j]:
        proposal_permutation[j]}
    for k in range(len(proposed_dec_indices)):
        idx = proposed_dec_indices[k]
        if idx in mapping:
            proposed_dec_indices[k] = mapping[idx]

    # Compute log-likelihood for current and proposed decrypted indices
    # For efficiency, we only compute the log-likelihood difference at the affected
        positions
    # Current log-likelihood
    current_ll = 0.0
    for pos in positions:
        idx = decrypted_indices[pos]
        if pos == 0:
            current_ll += log_phi[idx]
        else:
            idx_prev = decrypted_indices[pos - 1]
            current_ll += log_psi[idx_prev, idx]

    # Proposed log-likelihood
    proposed_ll = 0.0
    for idx, pos in zip(proposed_dec_indices, positions):
        if pos == 0:
            proposed_ll += log_phi[idx]
        else:
            idx_prev = proposed_dec_indices[np.where(positions == pos - 1)[0][0]] if pos
                - 1 in positions else decrypted_indices[pos - 1]
```

```
            proposed_ll += log_psi[idx_prev, idx]

    delta_log_likelihood = proposed_ll - current_ll

    # Compute acceptance probability using the Metropolis criterion
    acceptance_prob = min(1, np.exp(delta_log_likelihood))

    # Decide whether to accept the proposed permutation
    if np.random.rand() < acceptance_prob:
        # Accept the proposal
        current_permutation = proposal_permutation
        decrypted_indices[affected_positions] = proposed_dec_indices[np.isin(positions,
            affected_positions)]
        current_log_likelihood += delta_log_likelihood
        accepted += 1

        # Update the best permutation found if the new one has a higher log-likelihood
        if current_log_likelihood > best_log_likelihood:
            best_permutation = current_permutation.copy()
            best_log_likelihood = current_log_likelihood

    # Record log-likelihood and acceptance status for plotting
    log_likelihood_history.append(best_log_likelihood)
    acceptance_history.append(accepted / iteration)  # Running acceptance rate

    # Print the decrypted text of the first 60 symbols at specified intervals
    if iteration % print_interval == 0:
        partial_decrypted_indices = best_permutation[encrypted_indices[:60]]  # Decrypt
            first 60 symbols
        decrypted_text = ''.join([index_to_symbol[idx] for idx in
            partial_decrypted_indices])
        print(f"Iteration {iteration}: {decrypted_text}")

# After sampling, compute the final acceptance rate
final_acceptance_rate = accepted / num_iterations
print(f"\nFinal Acceptance Rate: {final_acceptance_rate:.2f}")

# Output the final decrypted message using the best permutation found
full_decrypted_indices = best_permutation[encrypted_indices]
full_decrypted_text = ''.join([index_to_symbol[idx] for idx in full_decrypted_indices])

print("\nDecrypted Message:")
print(full_decrypted_text)

# Plotting the Transition Matrix Heatmap
plt.figure(figsize=(12, 12))
plt.imshow(psi, cmap='Blues', interpolation='nearest')
plt.title('Transition Matrix Heatmap', fontsize=20)
plt.xlabel('Current Symbol (  )', fontsize=16)
plt.ylabel('Previous Symbol (  )', fontsize=16)

# Set ticks to symbols with rotation for better readability
plt.xticks(ticks=np.arange(N_symbols), labels=symbols, rotation=90, fontsize=8)
plt.yticks(ticks=np.arange(N_symbols), labels=symbols, fontsize=8)

# Add colorbar with label
cbar = plt.colorbar()
cbar.set_label('Transition Probability', fontsize=14)
cbar.ax.tick_params(labelsize=12)

plt.tight_layout()
plt.savefig('transition_matrix_heatmap.png', dpi=300)
plt.show()

# Plotting the Log-Likelihood Over Iterations
plt.figure(figsize=(12, 6))
plt.plot(log_likelihood_history, color='blue')
plt.title('Log-Likelihood Over Iterations', fontsize=16)
plt.xlabel('Iteration', fontsize=14)
plt.ylabel('Log-Likelihood', fontsize=14)
plt.grid(True)
plt.tight_layout()
plt.savefig('log_likelihood_over_iterations.png', dpi=300)
```

```
plt.show()

# Plotting the Acceptance Rate Over Iterations
plt.figure(figsize=(12, 6))
plt.plot(acceptance_history, color='green')
plt.title('Acceptance Rate Over Iterations', fontsize=16)
plt.xlabel('Iteration', fontsize=14)
plt.ylabel('Acceptance Rate', fontsize=14)
plt.grid(True)
plt.tight_layout()
plt.savefig('acceptance_rate_over_iterations.png', dpi=300)
plt.show()

# Plotting Symbol Frequency Comparison
# Compute decrypted unigram counts
decrypted_unigram_counts = np.bincount(full_decrypted_indices, minlength=N_symbols)

# Normalize counts to get frequencies
decrypted_freq = decrypted_unigram_counts / decrypted_unigram_counts.sum()
corpus_freq = unigram_counts / unigram_counts.sum()

# Plotting comparison of frequencies
plt.figure(figsize=(14, 7))
indices = np.arange(N_symbols)
width = 0.35  # Width of the bars

plt.bar(indices - width/2, corpus_freq, width, label='Corpus Frequency', alpha=0.7)
plt.bar(indices + width/2, decrypted_freq, width, label='Decrypted Frequency', alpha
    =0.7)

plt.xlabel('Symbol Index', fontsize=14)
plt.ylabel('Frequency', fontsize=14)
plt.title('Comparison of Symbol Frequencies: Corpus vs. Decrypted Message', fontsize=16)
plt.legend()
plt.tight_layout()
plt.savefig('symbol_frequency_comparison.png', dpi=300)
plt.show()
```

Listing 8: MH

A.6 Problem6

Part (a) and (b)

```
# -*- coding: utf-8 -*-

"""
    File name: gibbs_sampler.py
    Description: A re-implementation of the Gibbs sampler for LDA
    Author: Python: Roman Pogodin, MATLAB (original): Yee Whye Teh and Maneesh Sahani
    Implementation: Yuan Lu
    Date created: October 2018
    Python version: 3.6 and above
"""

import numpy as np
import pandas as pd
from scipy.special import gammaln
import matplotlib.pyplot as plt
from statsmodels.tsa.stattools import acf  # For autocorrelation computation
import seaborn as sns
from tqdm import tqdm  # Import tqdm for progress bars this is useful for nips modelling
sns.set(font_scale=2)


class GibbsSampler:
    def __init__(self, n_docs, n_topics, n_words, alpha, beta, random_seed=None):
        """
        Standard Gibbs Sampler for LDA.

        :param n_docs:          Number of documents
        :param n_topics:        Number of topics
        :param n_words:         Number of words in vocabulary
        :param alpha:           Dirichlet parameter on topic mixing proportions
        :param beta:            Dirichlet parameter on topic word distributions
        :param random_seed:     Random seed for reproducibility
        """
        self.n_docs = n_docs
        self.n_topics = n_topics
        self.n_words = n_words
        self.alpha = alpha
        self.beta = beta
        self.rand_gen = np.random.RandomState(random_seed)

        # Data placeholders
        self.docs_words = None
        self.docs_words_test = None
        self.do_test = False

        # Count matrices
        self.A_dk = np.zeros((self.n_docs, self.n_topics))  # Document-topic counts
        self.B_kw = np.zeros((self.n_topics, self.n_words))  # Topic-word counts

        # Parameters
        self.theta = np.zeros((self.n_docs, self.n_topics))
        self.phi = np.zeros((self.n_topics, self.n_words))

        # Topic assignments
        self.zi = []

        # Log-likelihoods
        self.loglike = None
        self.loglike_test = None

    def init_sampling(self, docs_words, docs_words_test=None, n_iter=0, save_loglike=
        False):
        assert np.all(docs_words.shape == (self.n_docs, self.n_words)), "docs_words␣
            shape=%s␣must␣be␣(%d,␣%d)" % (
            docs_words.shape, self.n_docs, self.n_words)
        self.docs_words = docs_words
        self.docs_words_test = docs_words_test

        self.do_test = docs_words_test is not None
```

```
        if save_loglike:
            self.loglike = np.zeros(n_iter)
            if self.do_test:
                self.loglike_test = np.zeros(n_iter)

        # Initialize counts
        self.A_dk.fill(0)
        self.B_kw.fill(0)

        # Initialize topic assignments and counts
        self.zi = []
        for d in range(self.n_docs):
            doc_topics = []
            for w in range(self.n_words):
                count = int(self.docs_words[d, w])
                if count > 0:
                    topics = self.rand_gen.randint(0, self.n_topics, size=count)
                    doc_topics.append(topics)
                    # Update counts
                    for k in topics:
                        self.A_dk[d, k] += 1
                        self.B_kw[k, w] += 1
                else:
                    doc_topics.append(np.array([], dtype=int))
            self.zi.append(doc_topics)

        # Initialize theta and phi
        self.update_theta_phi()

    def run(self, docs_words, docs_words_test=None, n_iter=100, save_loglike=False):
        self.init_sampling(docs_words, docs_words_test, n_iter=n_iter, save_loglike=
            save_loglike)

        for iteration in tqdm(range(n_iter), desc="Gibbs␣Sampling", unit="iter"):
            self.update_params()
            if save_loglike:
                self.update_loglike(iteration)

        return self.theta, self.phi

    def update_params(self):
        # Sample topic assignments
        for d in range(self.n_docs):
            for w in range(self.n_words):
                topics = self.zi[d][w]
                for n in range(len(topics)):
                    k = topics[n]
                    # Decrement counts
                    self.A_dk[d, k] -= 1
                    self.B_kw[k, w] -= 1

                    # Compute conditional distribution
                    p_z = self.theta[d, :] * self.phi[:, w]
                    p_z /= np.sum(p_z)

                    # Sample new topic
                    new_k = self.rand_gen.choice(self.n_topics, p=p_z)
                    topics[n] = new_k

                    # Increment counts
                    self.A_dk[d, new_k] += 1
                    self.B_kw[new_k, w] += 1

        # Sample theta and phi
        self.update_theta_phi()

    def update_theta_phi(self):
        # Sample theta from Dirichlet distribution
        self.theta = self.A_dk + self.alpha
        self.theta /= np.sum(self.theta, axis=1, keepdims=True)

        # Sample phi from Dirichlet distribution
```

```
        self.phi = self.B_kw + self.beta
        self.phi /= np.sum(self.phi, axis=1, keepdims=True)

    def update_loglike(self, iteration):
        # Compute log-likelihood for training data
        ll = np.sum((self.alpha - 1) * np.log(self.theta))
        ll += np.sum((self.beta - 1) * np.log(self.phi))
        for d in range(self.n_docs):
            for w in range(self.n_words):
                count = self.docs_words[d, w]
                if count > 0:
                    prob = np.dot(self.theta[d, :], self.phi[:, w])
                    ll += count * np.log(prob + 1e-12)
        self.loglike[iteration] = ll

        if self.do_test:
            # Compute log-likelihood for test data
            ll_test = 0
            for d in range(self.n_docs):
                for w in range(self.n_words):
                    count = self.docs_words_test[d, w]
                    if count > 0:
                        prob = np.dot(self.theta[d, :], self.phi[:, w])
                        ll_test += count * np.log(prob + 1e-12)
            self.loglike_test[iteration] = ll_test

    def get_loglike(self):
        if self.do_test:
            return self.loglike, self.loglike_test
        else:
            return self.loglike


class GibbsSamplerCollapsed:
    def __init__(self, n_docs, n_topics, n_words, alpha, beta, random_seed=None):
        """
        Collapsed Gibbs Sampler for LDA.

        :param n_docs:          Number of documents
        :param n_topics:        Number of topics
        :param n_words:         Number of words in vocabulary
        :param alpha:           Dirichlet parameter on topic mixing proportions
        :param beta:            Dirichlet parameter on topic word distributions
        :param random_seed:     Random seed for reproducibility
        """
        self.n_docs = n_docs
        self.n_topics = n_topics
        self.n_words = n_words
        self.alpha = alpha
        self.beta = beta
        self.rand_gen = np.random.RandomState(random_seed)

        # Data placeholders
        self.docs_words = None
        self.docs_words_test = None
        self.do_test = False

        # Count matrices
        self.A_dk = np.zeros((self.n_docs, self.n_topics))  # Document-topic counts
        self.B_kw = np.zeros((self.n_topics, self.n_words))  # Topic-word counts

        # Topic assignments
        self.zi = []

        # Log-likelihoods
        self.loglike = None
        self.loglike_test = None

    def init_sampling(self, docs_words, docs_words_test=None, n_iter=0, save_loglike=
        False):
        assert np.all(docs_words.shape == (self.n_docs, self.n_words)), "docs_words␣
            shape=%s␣must␣be␣(%d,␣%d)" % (
            docs_words.shape, self.n_docs, self.n_words)
```

```
        self.docs_words = docs_words
        self.docs_words_test = docs_words_test

        self.do_test = docs_words_test is not None

        if save_loglike:
            self.loglike = np.zeros(n_iter)
            if self.do_test:
                self.loglike_test = np.zeros(n_iter)

        # Initialize counts
        self.A_dk.fill(0)
        self.B_kw.fill(0)

        # Initialize topic assignments
        self.zi = []
        for d in range(self.n_docs):
            doc_topics = []
            for w in range(self.n_words):
                count = int(self.docs_words[d, w])
                if count > 0:
                    topics = self.rand_gen.randint(0, self.n_topics, size=count)
                    doc_topics.append(topics)
                    # Update counts
                    for k in topics:
                        self.A_dk[d, k] += 1
                        self.B_kw[k, w] += 1
                else:
                    doc_topics.append(np.array([], dtype=int))
            self.zi.append(doc_topics)

    def run(self, docs_words, docs_words_test=None, n_iter=100, save_loglike=False):
        self.init_sampling(docs_words, docs_words_test, n_iter=n_iter, save_loglike=
            save_loglike)

        for iteration in tqdm(range(n_iter), desc="Collapsed Gibbs Sampling", unit="iter
            "):
            self.update_params()
            if save_loglike:
                self.update_loglike(iteration)

        return self.A_dk, self.B_kw

    def update_params(self):
        # Sample topic assignments
        for d in range(self.n_docs):
            for w in range(self.n_words):
                topics = self.zi[d][w]
                for n in range(len(topics)):
                    k = topics[n]
                    # Decrement counts
                    self.A_dk[d, k] -= 1
                    self.B_kw[k, w] -= 1

                    # Compute conditional distribution (collapsed)
                    p_z = (self.A_dk[d, :] + self.alpha) * (self.B_kw[:, w] + self.beta)
                    denom = (np.sum(self.B_kw, axis=1) + self.n_words * self.beta)
                    p_z /= denom
                    p_z /= np.sum(p_z)

                    # Sample new topic
                    new_k = self.rand_gen.choice(self.n_topics, p=p_z)
                    topics[n] = new_k

                    # Increment counts
                    self.A_dk[d, new_k] += 1
                    self.B_kw[new_k, w] += 1

    def update_loglike(self, iteration):
        # Compute joint log-likelihood (excluding constants)
        ll = 0
        # Document-topic distributions
        ll += np.sum(gammaln(self.A_dk + self.alpha) - gammaln(self.alpha))
```

```
        ll -= np.sum(gammaln(np.sum(self.A_dk + self.alpha, axis=1)) - gammaln(self.
            n_topics * self.alpha))
        # Topic-word distributions
        ll += np.sum(gammaln(self.B_kw + self.beta) - gammaln(self.beta))
        ll -= np.sum(gammaln(np.sum(self.B_kw + self.beta, axis=1)) - gammaln(self.
            n_words * self.beta))
        self.loglike[iteration] = ll

        if self.do_test:
            # Compute predictive log-likelihood for test data
            phi = (self.B_kw + self.beta)
            phi /= np.sum(phi, axis=1)[:, np.newaxis]

            theta_test = (self.A_dk + self.alpha)
            theta_test /= np.sum(theta_test, axis=1)[:, np.newaxis]

            ll_test = 0
            for d in range(self.n_docs):
                for w in range(self.n_words):
                    count = self.docs_words_test[d, w]
                    if count > 0:
                        prob = np.dot(theta_test[d, :], phi[:, w])
                        ll_test += count * np.log(prob + 1e-12)  # Avoid log(0)
            self.loglike_test[iteration] = ll_test

    def get_loglike(self):
        if self.do_test:
            return self.loglike, self.loglike_test
        else:
            return self.loglike


def read_data(filename):
    """
    Reads the text data and splits into train/test.
    """
    data = pd.read_csv(filename, dtype=int, sep=' ', names=['doc', 'word', 'train', '
        test'])

    n_docs = np.amax(data.loc[:, 'doc'])
    n_words = np.amax(data.loc[:, 'word'])

    docs_words_train = np.zeros((n_docs, n_words), dtype=int)
    docs_words_test = np.zeros((n_docs, n_words), dtype=int)

    docs_words_train[data.loc[:, 'doc'] - 1, data.loc[:, 'word'] - 1] = data.loc[:, '
        train']
    docs_words_test[data.loc[:, 'doc'] - 1, data.loc[:, 'word'] - 1] = data.loc[:, 'test
        ']

    return docs_words_train, docs_words_test


def main():
    print('Running toyexample.data with the standard Gibbs sampler')

    # Read data
    docs_words_train, docs_words_test = read_data('toyexample.data')
    n_docs, n_words = docs_words_train.shape
    n_topics = 3
    alpha = 1.0
    beta = 1.0
    random_seed = 0
    n_iter = 200

    # Standard Gibbs Sampler
    sampler = GibbsSampler(n_docs=n_docs, n_topics=n_topics, n_words=n_words,
                           alpha=alpha, beta=beta, random_seed=random_seed)

    theta, phi = sampler.run(docs_words_train, docs_words_test, n_iter=n_iter,
        save_loglike=True)

    like_train, like_test = sampler.get_loglike()
```

```
    # Plot log-likelihoods
    plt.figure(figsize=(9, 6))
    plt.plot(like_train, label='Train Log-Likelihood')
    plt.xlabel('Iteration')
    plt.ylabel('Log-Likelihood')
    plt.title('Standard Gibbs Sampler Log-Likelihoods')
    plt.legend()
    plt.savefig('std_gibbs_train.png', dpi=300)
    plt.show()

    plt.figure(figsize=(9, 6))
    plt.plot(like_test, label='Test Log-Likelihood')
    plt.xlabel('Iteration')
    plt.ylabel('Log-Likelihood')
    plt.title('Standard Gibbs Sampler Log-Likelihoods')
    plt.legend()
    plt.savefig('std_gibbs_test.png', dpi=300)
    plt.show()

    print('Running toyexample.data with the collapsed Gibbs sampler')

    # Collapsed Gibbs Sampler
    sampler_collapsed = GibbsSamplerCollapsed(n_docs=n_docs, n_topics=n_topics, n_words=
        n_words,
                                              alpha=alpha, beta=beta, random_seed=
                                                  random_seed)

    sampler_collapsed.run(docs_words_train, docs_words_test, n_iter=n_iter, save_loglike
        =True)

    like_train_collapsed, like_test_collapsed = sampler_collapsed.get_loglike()

    # Plot log-likelihoods
    plt.figure(figsize=(9, 6))
    plt.plot(like_train_collapsed, label='Train Log-Likelihood')
    plt.xlabel('Iteration')
    plt.ylabel('Log-Likelihood')
    plt.title('Collapsed Gibbs Sampler Log-Likelihoods')
    plt.legend()
    plt.savefig('col_gibbs_train.png', dpi=300)
    plt.show()

    plt.figure(figsize=(9, 6))
    plt.plot(like_test_collapsed, label='Test Log-Likelihood')
    plt.xlabel('Iteration')
    plt.ylabel('Log-Likelihood')
    plt.title('Collapsed Gibbs Sampler Log-Likelihoods')
    plt.legend()
    plt.savefig('col_gibbs_test.png', dpi=300)
    plt.show()


    # Compute and plot autocorrelations (after burn-in)
    burn_in = 25
    plot_autocorrelation(like_train[burn_in:], 'Autocorrelation of Train Log-Likelihood (
        Standard Gibbs Sampler)')
    plot_autocorrelation(like_test[burn_in:], 'Autocorrelation of Test Log-Likelihood (
        Standard Gibbs Sampler)')

    plot_autocorrelation(like_train_collapsed[burn_in:], 'Autocorrelation of Train Log-
        Likelihood (Collapsed Gibbs Sampler)')
    plot_autocorrelation(like_test_collapsed[burn_in:], 'Autocorrelation of Test Log-
        Likelihood (Collapsed Gibbs Sampler)')


def plot_autocorrelation(log_values, title):
    """
    Computes and plots the autocorrelation of a time series.
    """
    autocorr = acf(log_values, nlags=50, fft=False)
    plt.figure(figsize=(9, 6))
    plt.stem(range(len(autocorr)), autocorr, linefmt='b-', markerfmt='bo', basefmt='r-')
```

```
    plt.xlabel('Lag')
    plt.ylabel('Autocorrelation')
    plt.title(title)
    plt.savefig(f'{title}.png', dpi=300)
    plt.show()


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

Listing 9: gibbs

Part (d)

```
def main():
    print('Experimenting␣with␣varying␣  ,␣  ,␣and␣n_topics')

    # Read data
    docs_words_train, docs_words_test = read_data('toyexample.data')
    n_docs, n_words = docs_words_train.shape
    random_seed = 0
    n_iter = 200

    # Define parameter ranges
    alpha_values = [0.1, 1.0, 5.0]
    beta_values = [0.1, 1.0, 5.0]
    n_topics_values = [2, 3, 5]

    # Fixed parameters
    fixed_beta = 1.0
    fixed_n_topics = 3
    fixed_alpha = 1.0

    # Store results
    results = []

    # Vary alpha while fixing beta and n_topics
    print('\nVarying␣  ␣while␣fixing␣  ␣and␣n_topics')

    # Dictionaries to store log-likelihoods
    loglike_train_alpha = {}
    loglike_test_alpha = {}

    for alpha in alpha_values:
        beta = fixed_beta
        n_topics = fixed_n_topics
        print(f'Running␣with␣  ={alpha},␣  ={beta},␣n_topics={n_topics}')

        # Standard Gibbs Sampler
        sampler = GibbsSampler(n_docs=n_docs, n_topics=n_topics, n_words=n_words,
                               alpha=alpha, beta=beta, random_seed=random_seed)

        theta, phi = sampler.run(docs_words_train, docs_words_test, n_iter=n_iter,
            save_loglike=True)

        like_train, like_test = sampler.get_loglike()

        # Store the log-likelihoods
        loglike_train_alpha[alpha] = like_train
        loglike_test_alpha[alpha] = like_test

        # Record final log-likelihoods
        results.append({
            'alpha': alpha,
            'beta': beta,
            'n_topics': n_topics,
            'train_loglike': like_train[-1],
            'test_loglike': like_test[-1]
        })

    # Plot train log-likelihoods
    plt.figure(figsize=(9, 6))
    for alpha in alpha_values:
        plt.plot(loglike_train_alpha[alpha], label=f'  ={alpha}')
    plt.xlabel('Iteration')
    plt.ylabel('Train␣Log-Likelihood')
    plt.title(f'Varying␣  ␣(  ={fixed_beta},␣n_topics={fixed_n_topics})␣-␣Train')
    plt.legend()
    plt.savefig(f'varying_alpha_train_beta{fixed_beta}_topics{fixed_n_topics}.png', dpi
        =300)
    plt.show()

    # Plot test log-likelihoods
    plt.figure(figsize=(9, 6))
    for alpha in alpha_values:
```

```
        plt.plot(loglike_test_alpha[alpha], label=f'  ={alpha}')
    plt.xlabel('Iteration')
    plt.ylabel('Test Log-Likelihood')
    plt.title(f'Varying     (  ={fixed_beta}, n_topics={fixed_n_topics}) - Test')
    plt.legend()
    plt.savefig(f'varying_alpha_test_beta{fixed_beta}_topics{fixed_n_topics}.png', dpi
        =300)
    plt.show()

    # Vary beta while fixing alpha and n_topics
    print('\nVarying     while fixing     and n_topics')

    # Dictionaries to store log-likelihoods
    loglike_train_beta = {}
    loglike_test_beta = {}

    for beta in beta_values:
        alpha = fixed_alpha
        n_topics = fixed_n_topics
        print(f'Running with     ={alpha},     ={beta}, n_topics={n_topics}')

        # Standard Gibbs Sampler
        sampler = GibbsSampler(n_docs=n_docs, n_topics=n_topics, n_words=n_words,
                               alpha=alpha, beta=beta, random_seed=random_seed)

        theta, phi = sampler.run(docs_words_train, docs_words_test, n_iter=n_iter,
            save_loglike=True)

        like_train, like_test = sampler.get_loglike()

        # Store the log-likelihoods
        loglike_train_beta[beta] = like_train
        loglike_test_beta[beta] = like_test

        # Record final log-likelihoods
        results.append({
            'alpha': alpha,
            'beta': beta,
            'n_topics': n_topics,
            'train_loglike': like_train[-1],
            'test_loglike': like_test[-1]
        })

    # Plot train log-likelihoods
    plt.figure(figsize=(9, 6))
    for beta in beta_values:
        plt.plot(loglike_train_beta[beta], label=f'  ={beta}')
    plt.xlabel('Iteration')
    plt.ylabel('Train Log-Likelihood')
    plt.title(f'Varying     (  ={fixed_alpha}, n_topics={fixed_n_topics}) - Train')
    plt.legend()
    plt.savefig(f'varying_beta_train_alpha{fixed_alpha}_topics{fixed_n_topics}.png', dpi
        =300)
    plt.show()

    # Plot test log-likelihoods
    plt.figure(figsize=(9, 6))
    for beta in beta_values:
        plt.plot(loglike_test_beta[beta], label=f'  ={beta}')
    plt.xlabel('Iteration')
    plt.ylabel('Test Log-Likelihood')
    plt.title(f'Varying     (  ={fixed_alpha}, n_topics={fixed_n_topics}) - Test')
    plt.legend()
    plt.savefig(f'varying_beta_test_alpha{fixed_alpha}_topics{fixed_n_topics}.png', dpi
        =300)
    plt.show()

    # Vary n_topics while fixing alpha and beta
    print('\nVarying n_topics while fixing     and     ')

    # Dictionaries to store log-likelihoods
    loglike_train_n_topics = {}
    loglike_test_n_topics = {}
```

```
    for n_topics in n_topics_values:
        alpha = fixed_alpha
        beta = fixed_beta
        print(f'Running␣with␣  ={alpha},␣  ={beta},␣n_topics={n_topics}')

        # Standard Gibbs Sampler
        sampler = GibbsSampler(n_docs=n_docs, n_topics=n_topics, n_words=n_words,
                               alpha=alpha, beta=beta, random_seed=random_seed)

        theta, phi = sampler.run(docs_words_train, docs_words_test, n_iter=n_iter,
            save_loglike=True)

        like_train, like_test = sampler.get_loglike()

        # Store the log-likelihoods
        loglike_train_n_topics[n_topics] = like_train
        loglike_test_n_topics[n_topics] = like_test

        # Record final log-likelihoods
        results.append({
            'alpha': alpha,
            'beta': beta,
            'n_topics': n_topics,
            'train_loglike': like_train[-1],
            'test_loglike': like_test[-1]
        })

    # Plot train log-likelihoods
    plt.figure(figsize=(9, 6))
    for n_topics in n_topics_values:
        plt.plot(loglike_train_n_topics[n_topics], label=f'n_topics={n_topics}')
    plt.xlabel('Iteration')
    plt.ylabel('Train␣Log-Likelihood')
    plt.title(f'Varying␣n_topics␣(  ={fixed_alpha},␣  ={fixed_beta})␣-␣Train')
    plt.legend()
    plt.savefig(f'varying_n_topics_train_alpha{fixed_alpha}_beta{fixed_beta}.png', dpi
        =300)
    plt.show()

    # Plot test log-likelihoods
    plt.figure(figsize=(9, 6))
    for n_topics in n_topics_values:
        plt.plot(loglike_test_n_topics[n_topics], label=f'n_topics={n_topics}')
    plt.xlabel('Iteration')
    plt.ylabel('Test␣Log-Likelihood')
    plt.title(f'Varying␣n_topics␣(  ={fixed_alpha},␣  ={fixed_beta})␣-␣Test')
    plt.legend()
    plt.savefig(f'varying_n_topics_test_alpha{fixed_alpha}_beta{fixed_beta}.png', dpi
        =300)
    plt.show()

    # Convert results to DataFrame
    results_df = pd.DataFrame(results)
    print('\nSummary␣of␣Results:')
    print(results_df)


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

Listing 10: parameter sweep

Part (e) and (f)

```
def reduce_vocabulary(docs_words, vocab, top_n=500):
    """
    Reduces the vocabulary to the top_n words based on tf-idf scores.

    :param docs_words: Document-word matrix (n_docs x n_words)
    :param vocab: List of words in the vocabulary
    :param top_n: Number of top words to retain
    :return: Reduced docs_words matrix and updated vocabulary
    """
    from sklearn.feature_extraction.text import TfidfTransformer

    # Calculate tf-idf scores
    transformer = TfidfTransformer()
    tfidf = transformer.fit_transform(docs_words)
    tfidf_scores = np.asarray(tfidf.mean(axis=0)).ravel()

    # Get indices of top_n words
    top_indices = np.argsort(tfidf_scores)[::-1][:top_n]

    # Filter docs_words and vocab
    reduced_docs_words = docs_words[:, top_indices]
    reduced_vocab = [vocab[i] for i in top_indices]

    return reduced_docs_words, reduced_vocab

def main():
    print('Running␣LDA␣on␣reduced␣NeurIPS␣data')

    # Read and reduce data
    docs_words_train, docs_words_test = read_data('nips.data')
    n_docs, n_words = docs_words_train.shape

    # Read vocabulary
    with open('nips.vocab', 'r') as f:
        vocab = [line.strip() for line in f]

    # Reduce vocabulary
    top_n = 500
    reduced_docs_words_train, reduced_vocab = reduce_vocabulary(docs_words_train, vocab,
        top_n=top_n)
    reduced_docs_words_test = docs_words_test[:, :top_n]

    n_docs, n_words = reduced_docs_words_train.shape
    random_seed = 0
    n_iter = 300  # Increased iterations for better convergence

    # Experiment with various settings
    alpha_values = [0.1, 1.0]
    beta_values = [0.1, 1.0]
    n_topics_values = [5, 10, 15]

    for alpha in alpha_values:
        for beta in beta_values:
            for n_topics in n_topics_values:
                print(f'\nRunning␣with␣  ={alpha},␣  ={beta},␣n_topics={n_topics}')

                # Collapsed Gibbs Sampler
                sampler = GibbsSamplerCollapsed(n_docs=n_docs, n_topics=n_topics,
                    n_words=n_words,
                                                alpha=alpha, beta=beta, random_seed=
                                                    random_seed)

                sampler.run(reduced_docs_words_train, reduced_docs_words_test, n_iter=
                    n_iter, save_loglike=True)

                # Get topic-word distributions
                phi = (sampler.B_kw + beta)
                phi /= np.sum(phi, axis=1)[:, np.newaxis]

                # Display top words for each topic
                print(f'Top␣words␣for␣  ={alpha},␣  ={beta},␣n_topics={n_topics}')
```

```
                for k in range(n_topics):
                    top_word_indices = np.argsort(phi[k, :])[::-1][:10]
                    top_words = [reduced_vocab[i] for i in top_word_indices]
                    print(f'Topic {k + 1}: {", ".join(top_words)}')

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

Listing 11: nips