Probabilistic & Unsupervised Learning Summative Assignment

Gatsby Computational Neuroscience Unit - Peter Orbanz

Student ID: 23205123

MSc Computational Statistics & Machine Learning

Department of Computer Science
University College London
London
15 November 2023

Contents

Problem 1: Models for binary vectors

a) Inappropriate model: Multivariate Gaussians

The multivariate Gaussian (also known as multivariate normal) distribution is used to model continuous data that can take any value in a continuous range. However, our binary images have pixels that can only take on the values 0 or 1. Therefore:

- Value Constraints: In the binary image, pixel values are either 0 (black) or 1 (white). In contrast, a multivariate Gaussian can produce any real-valued number as an output, which doesn't fit with the binary nature of our data.
- No Physical Interpretation: Any value other than 0 or 1 has no physical interpretation in the context of binary images. For example, what would a pixel value of 0.57 mean for a black and white image?
- Invalid Assumptions: The multivariate Gaussian assumes that the data follows a bell-shaped curve (normal distribution). This assumption doesn't hold for binary data which is bimodal (having two peaks).

b) ML estimator for p_d

Images modelled as i.i.d. samples from a D-dimensional multivariate Bernoulli distribution with parameter $p = (p_1, ..., p_D)$, which has form:

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

For the given data, the likelihood is given by the product of probabilities for each individual image for the vector space p.

$$L(p|x_1, ...x_N) = \Pi_{n=1}^{N}\Pi_{d=1}^{D} p_d^{x_d^n}(1 - p_d)^(1 - x_d^n) \tag{2}$$

Log likelihood function then becomes;

$$ln(L(p)) = \Sigma_{n=1}^{N}\Sigma_{d=1}^{D}[x_d^n ln(p_d) + (1 - x_d^n)ln(1 - p_d))] \tag{3}$$

Differentiating the log likelihood with respect to p_d, For $x_d^n ln(p_d)$:

$$\frac{\delta}{\delta p_d} x_d^n ln(p_d) = x_d^n \frac{1}{p_d}$$

For $(1 - x_d^n)ln(1 - p_d)$:

$$\frac{\delta}{\delta p_d}(1 - x_d^n)ln(1 - p_d) = (1 - x_d^n)\frac{-1}{1 - p_d}$$

Substituting these into our log-likelihood expression:

$$\frac{\delta ln(L(p))}{\delta p_d} = \Sigma_{n=1}^{N}[x_d^n \frac{1}{p_d} - (1 - x_d^n)\frac{1}{1 - p_d}]$$

$$\Sigma_{n=1}^{N}[x_d^n \frac{1}{p_d} - (1 - x_d^n)\frac{1}{1 - p_d}] = 0$$

Now, we'll solve for p_d:

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

Multiplying through by $p_d(1 - p_d)$:

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

Expanding this equation:

$$\Sigma_{n=1}^{N} x_d^n - p_d\Sigma_{n=1}^{N} x_d^n = p_d N - p_d\Sigma_{n=1}^{N} x_d^n$$

$$p_d = \frac{1}{N}\Sigma_{n=1}^{N} x_d^n$$

c) MAP estimator for p_d

Maximum A Posteriori (MAP) estimate for p_d given the data. Likelihood is given by the Bernoulli distribution for the binary data:

$$P(D|\theta) = \Pi_{n=1}^{N}\Pi_{d=1}^{D} p_d^{x_d^n}(1-p_d)^{1-x_d^n}$$

Prior is given by:

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

In order to the MAP estimate(from lecture slides):

$$\theta_{MAP} = argmax P(\theta|D)$$

$$\theta_{MAP} = argmax P(\theta)P(D|\theta)$$

Taking the logarithm:

$$lnP(\theta|D) = lnP(D|\theta) + lnP(\theta)$$

Differentiating with respect to p_d, and setting it to zero to give us maximum. For $lnP(D|\theta)$:

$$\frac{\delta}{\delta p_d}\Sigma_{n=1}^{N}[x_d^{(n)}ln(p_d) + (1-x_d^{(n)}ln(1-p_d))]$$

For $lnP(\theta)$:

$$\frac{\delta}{\delta p_d}[(\alpha-1)lnp_d + (\beta-1)ln(1-p_d)]$$

Adding these together and setting to zero:

$$\Sigma_{n=1}^{N}x_d^{(n)}ln(p_d) + (1-x_d^{(n)}ln(1-p_d)) + (\alpha-1)lnp_d + (\beta-1)ln(1-p_d) = 0$$

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

Therefore, the MAP estimate for p_d is:

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

d) ML parameters of Multivariate Bernoulli

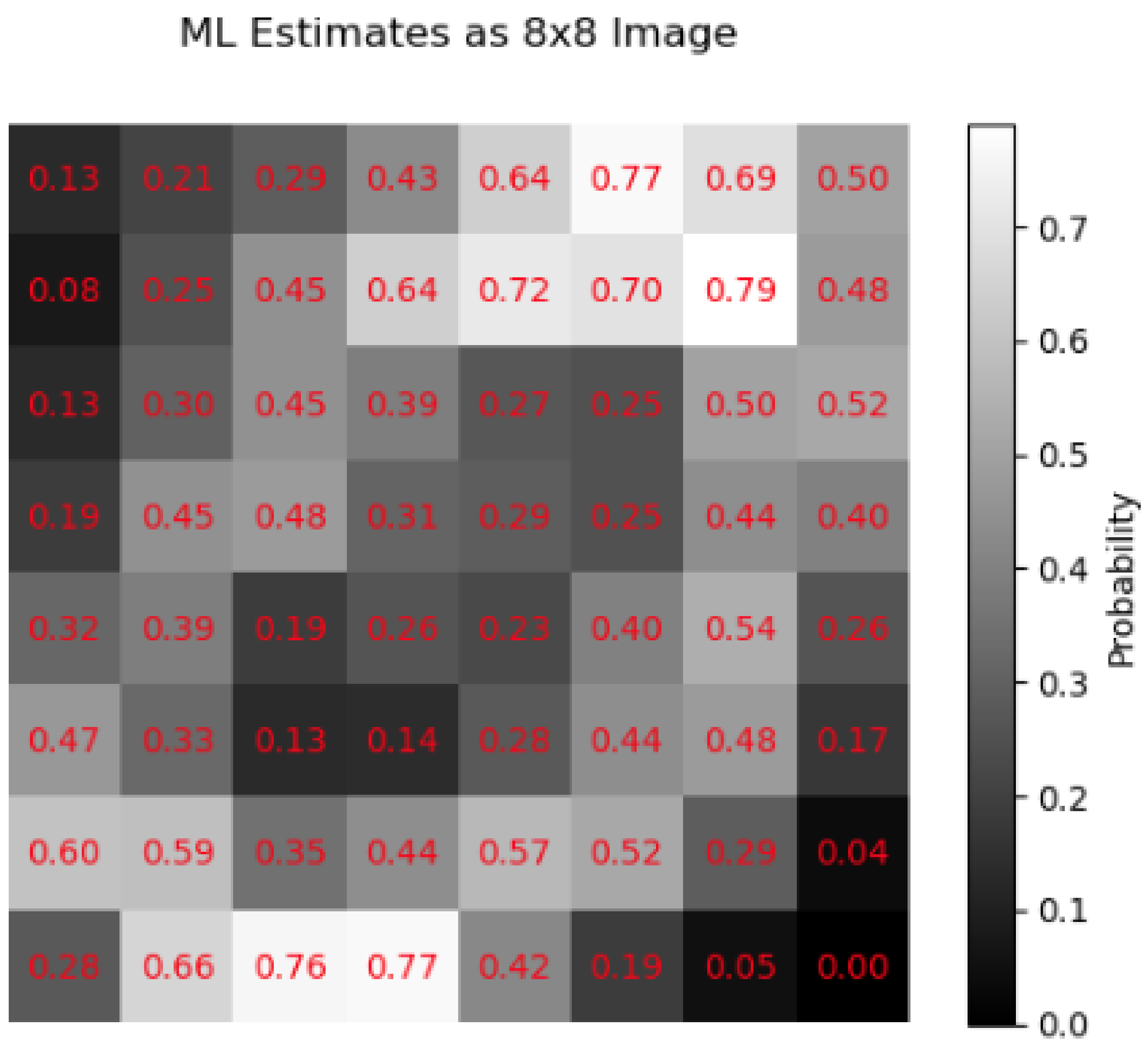


Figure 1: MAP Estimates as 8×8 image

e) MAP parameters: $\alpha = \beta = 3$

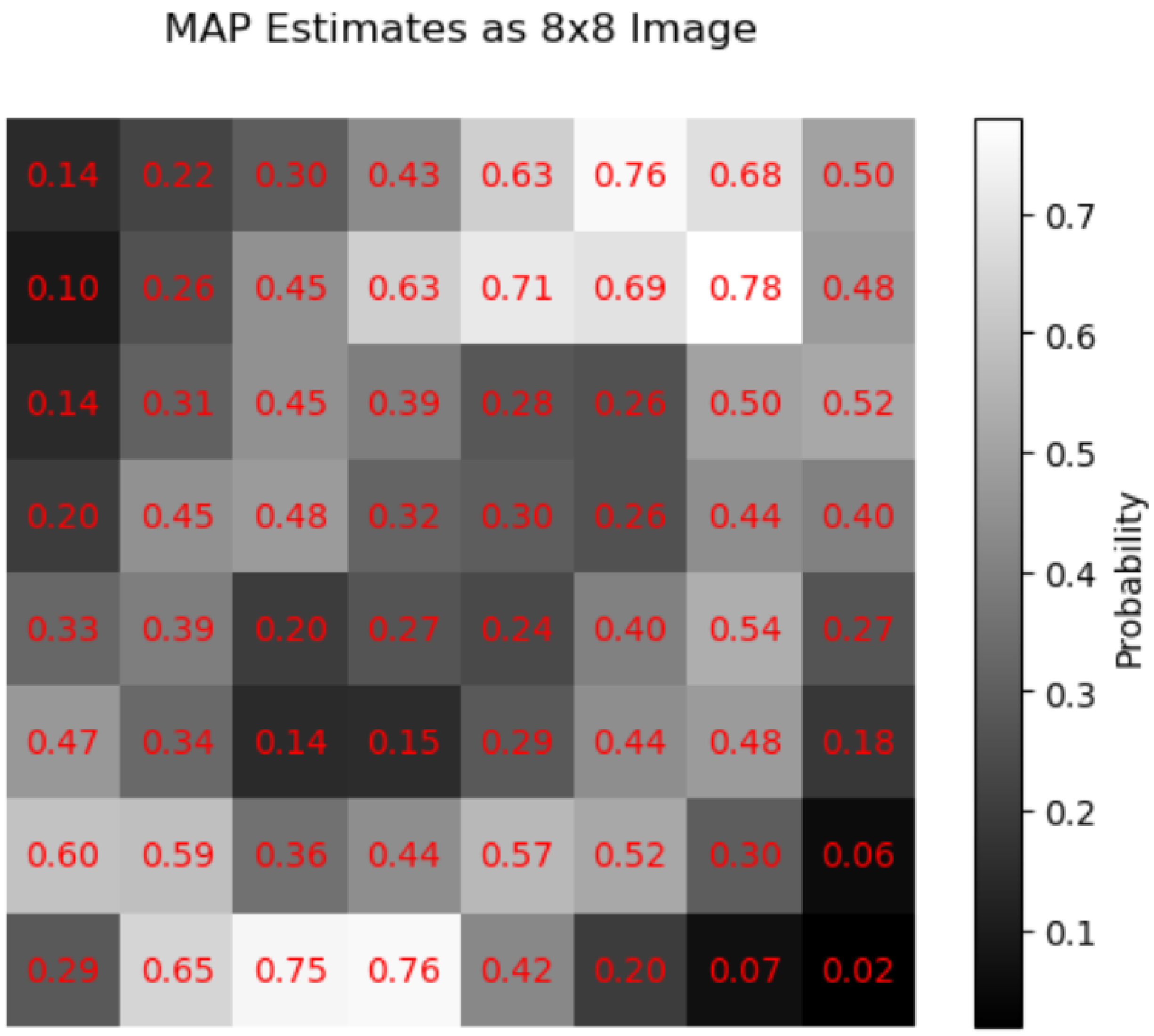


Figure 2: MAP Estimates as 8×8 image with $\alpha = \beta = 3$

MAP estimate performs better than the ML estimate. The ML estimate may not perform well with sparse data for certain pixels. If a pixel is rarely 'on'(almost always 0), ML could suggest a very low probability, effectively ignoring the rare occurrences where pixel is '1'. This might lead to underestimating the true likelihood of such events. MAP, with a prior that is not too strong, can potentially solve this issue by smoothing the probabilities and ensuring that even rare events are assigned a non-zero likelihood, which might reflect their true probabilities more accurately.

Other advantages of MAP estimate over ML estimate:

- **Preventing Zero Probabilities:** With MLE, this could lead to issues in computational models where calculations depend on probabilities such as log-likelihood computations. MAP estimation ensures that each event has a non-zero probability which can lead to more stable computation.
- **Choice of α & β:** This essentially denotes the prior belief. With $\alpha = \beta = 3$, this indicates each pixel has equal chance of being '0' and '1' before seeing the data. The prior is not biased to either outcome

but it is not completely non-informative.

- **Adaptability to different scenarios:** MAP estimate allows, depending on the situation, adjustment to α and β. If there is reason that pixels are more likely to be 'on', you can choose $\alpha \geq \beta$.

Problem 2: Model Selection

a) $\mathcal{M}_1$

Given this model 1, the probability of a single pixel being either 0 or 1 is $p_d = 0.5$. For a single image $x^(n)$ with D pixels:

- The probability of the first pixel being either 0 or 1 is 0.5
- The probability of the second pixel being either 0 or 1 is 0.5

Therefore, the probability of observing any particular binary image $x^{(n)}$ of D pixels under this model is:

$$P(x^{(n)}|\mathcal{M}_1) = \underbrace{P(x_1|\mathcal{M}_1) \times P(x_2|\mathcal{M}_1) \times \cdots \times P(x_D|\mathcal{M}_1)}_{\text{for } D \text{ pixels}}$$

Using our single pixel probability:

$$P(x^{(n)}|\mathcal{M}_1) = (0.5) \times (0.5) \times \cdots \times (0.5)$$

$$P(x^{(n)}|\mathcal{M}_1) = (0.5^D)^N$$

Using base conversion, and logarithmic properties:

$$(0.5^D)^N = (2^{-1})^{D \times N}$$

$$2^{-D \times N} = 2^{-ND}$$

$$P(D|\mathcal{M}_1) = 2^{-ND}$$

b) $\mathcal{M}_2$

Given this model 2, all D components are generated from Bernoulli distributions with unknown but identical p_d. Let us denote X, where $x^{(n)}$ is a D-dimensional binary vector:

$$X = \{x^{(1)}, x^{(2)}, ..., x^{(n)}\}$$

and $P(D|\mathcal{M}_2)$ as the likelihood of data D given the model $\mathcal{M}_2$. $P(x|p)$ which has a D-dimensional multivariate Bernoulli distribution with parameter vector $p = (p_1, ..., p_D)$ as:

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

Therefore, for all parameters vectors p:

$$P(x^{(n)}|p_d) = p_d^{\Sigma_d x_d^{(n)}}(1-p_d)^{D-\Sigma_d x_d^{(n)}}$$

$$P(D|p_d) = \Pi_{n=1}^{N} P(x^{(n)}|p_d)$$

To find the overall likelihood of data under model $\mathcal{M}_2$, without knowing the specific p_d, we integrate over all possible values of p_d which range from $0 < p_d < 1$, due to the uniform prior.

$$P(D|\mathcal{M}_2) = \Pi_{n=1}N \int_0^1 P(x^{(n)}|p_d)P(p_d)dp_d$$

$$P(D|\mathcal{M}_2) = \int_0^1 p_d^{\Sigma_{n=1}^N \Sigma_d dx_d^{(n)}} (1-p_d)^{ND - \Sigma_{n=1}^N \Sigma_d dx_d^{(n)}} x_1 dp_d$$

$$= \int_0^1 \left(p_d^{\Sigma_d x_d^{(n)}} (1-p_d)^{ND - \Sigma_d x_d^{(n)]}} \right) \times 1 dp_d$$

Rewriting as a Beta function where the beta function is:

$$B(z_1, z_2) = \int_0^1 t^{z_1 - 1} (1-t)^{z_2 - 1} dt$$

$$B(\Sigma_{n=1}^N \Sigma_d x_d^{(n)} + 1, ND - \Sigma_{n=1}^N \Sigma_d x_d^{(n)} + 1) = \int_0^1 p_d^{z_1 - 1} (1-p_d)^{z_2 - 1} dt$$

$$P(D|\mathcal{M}_2) = B(\Sigma_{n=1}^N \Sigma_d x_d^{(n)} + 1, ND - \Sigma_{n=1}^N \Sigma_d x_d^{(n)} + 1)$$

$$= \int_0^1 p_d^{z_1 - 1} (1-p_d)^{z_2 - 1} dt$$

c) $\mathcal{M}_3$

Given $\mathcal{M}_3$ where each component is Bernoulli distributed with separate, unknown p_d, and given a single pixel d, likelihood contribution from all images for that pixel n:

$$P(D_d|p_d) = \Pi_{n=1}^N p_d^{x_d^{(n)}} (1-p_d)^{1-x_d^{(n)}}$$

Marginalizing p_d out, we integrate over its possible range [0,1] with respect to the uniform prior.

$$P(D_d) = \int_0^1 P(D_d|p_d) dp_d$$

$$= \int_0^1 \left(\Pi_{n=1}^N p_d^{x_d^{(n)}} (1-p_d)^{1-x_d^{(n)}} \right) dp_d$$

Thus, as pixels are independent, evidence for entire dataset is product of evidence for each pixel:

$$P(D|\mathcal{M}_3) = \Pi_{d=1}^D P(D_d)$$

$$P(D|\mathcal{M}_3) = \Pi_{d=1}^D p_d^{x_d^{(n)}} (1-p_d)^{1-x_d^{(n)}}$$

Rewriting this as a Beta function:

$$P(D|\mathcal{M}_3) = \Pi_{d=1}^D B\left(\Sigma_{n=1}^N x_d^{(n)} + 1, \Sigma_{n=1}^N (1 - x_d^{(n)}) + 1) \right)$$

In order to calculate the posterior probabilities of each of the 3 models, we have Bayes theorem:

$$P(\mathcal{M}|D) = \frac{P(D|\mathcal{M}) \cdot P(\mathcal{M})}{P(D)}$$

Since we assumed that all models are equally likely a priori:

$$P(\mathcal{M}_1) = P(\mathcal{M}_2) = P(\mathcal{M}_3)$$

With the likelihoods provided above, and the evidence $P(D)$ which is the probability of data under any model, which is a sum of likelihoods of the data under all models, considering priors:

$$P(D) = P(D|\mathcal{M}_1) \cdot P(\mathcal{M}_1) + P(D|\mathcal{M}_2) \cdot P(\mathcal{M}_2) + P(D|\mathcal{M}_3) \cdot P(\mathcal{M}_3)$$

Posterior Probabilities of Model 1,2,3

As priors are equal, P(D) becomes sum of the likelihoods of the 3 models:

$$P(D) = P(D|\mathcal{M}_1) + P(D|\mathcal{M}_2) + P(D|\mathcal{M}_3)$$

To compute posterior probabilities for each model, in log form:

$$logP(\mathcal{M}_i|D) = log\left(P(D|\mathcal{M}_i)\right) - log\left(P(D|\mathcal{M}_1) + P(D|\mathcal{M}_2) + P(D|\mathcal{M}_3)\right)$$

Converting these log probabilities back to posterior probabilities:

$$P(\mathcal{M}_i|D) = e^{log(P(D|\mathcal{M}_i)) - log(P(D|\mathcal{M}_1) + P(D|\mathcal{M}_2) + P(D|\mathcal{M}_3))}$$

Therefore:

Table 1: Posterior Probabilities of $\mathcal{M}_1, \mathcal{M}_2, \mathcal{M}_3$

$\mathcal{M}_i$	**Natural Log Likelihood**	$P(\mathcal{M}_i|D)$
$\mathcal{M}_1$	-4436	$9.143e-255$
$\mathcal{M}_2$	-4283	$1.434e-188$
$\mathcal{M}_3$	-3851	1.0

Problem 3: EM for Binary Data

a) Likelihood of mixture of K multivariate Bernoulli distributions

Probability of observing $x^{(n)}$ under a single Bernoulli component k is:

$$P(x^{(n)}|k) = \Pi_{d=1}^{D} p_{kd}^{x_d^{(n)}} (1 - p_{kd})^{1 - x_d^{(n)}} (4)$$

- where $x_d^{(n)}$ value of d^{th} pixel in n^{th} image.
- p_{kd} is probability that d^{th} pixel takes value 1 under k^{th} Bernoulli component.

Overall likelihood of observing $x^{(n)}$ under the mixture model is weighted sum of likelihoods from each component:

$$P(x^{(n)}|\pi, P) = \Sigma_{k=1}^{K} \pi_k P(x^{(n)}|k),$$

$$P(x^{(n)}|\pi, P) = \Sigma_{k=1}^{K} \pi_k \Pi_{d=1}^{D} p_{kd}^{x_d^{(n)}} (1 - p_{kd})^{1 - x_d^{(n)}} (5)$$

Likelihood of entire dataset $\{x^{(1)}, x^{(2)}, ..., x^{(n)}\}$, under the assumption that images are i.i.d:

$$\mathcal{L}(\pi, P) = \Pi_{n=1}^{N} P(x^{(n)}|\pi, P)$$

$$\mathcal{L}(\pi, P) = \Pi_{n=1}^{N} \Sigma_{k=1}^{K} \pi_k \Pi_{d=1}^{D} p_{kd}^{x_d^{(n)}} (1 - p_{kd})^{1 - x_d^{(n)}}$$

b) R_{nk}

For mixture models with latent variables, let R_{nk} be the posterior probability of latent variable $s^{(n)}$, being in state k. Given observation x:

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

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

From Equation 5:

$$P(x^{(n)}|\pi, P) = \Sigma_{j=1}^{K}\pi_j\Pi_{d=1}^{D}p_{jd}^{x_d^{(n)}}(1-p_{jd})^{1-x_d^{(n)}}$$

and:

$$P(x^{(n)}|k) = \Pi_{d=1}^{D}p_{kd}^{x_d^{(n)}}(1-p_{kd})^{1-x_d^{(n)}}$$

Therefore:

$$R_{nk} = \frac{\pi_k\Pi_{d=1}^{D}p_{kd}^{x_d^{(n)}}(1-p_{kd})^{1-x_d^{(n)}}}{\Sigma_{j=1}^{K}\pi_j\Pi_{d=1}^{D}p_{jd}^{x_d^{(n)}}(1-p_{jd})^{1-x_d^{(n)}}}$$

c) Maximizing Parameters of p_{kd} & π_k

Complete data log likelihood is the log joint log $P(S^{(n)} = k|x^{(n)}, \pi, P)$. Expected log joint with respect to R_{nk} which is found in the E-step is:

$$Q(\pi, P) = \Sigma_{n=1}^{N}\Sigma_{k=1}^{K}R_{nk}logP(x^{(n)}, s^{(n)} = k|\pi, P) \tag{6}$$

From the Equation 6, $logP(x^{(n)}, s^{(n)} = k|\pi, P)$:

$$logP(x^{(n)}, S^{(n)} = k|\pi, P) = logP(x^{(n)}|s^{(n)} = k, P) + logP(s^{(n)} = k|\pi)$$

Substituting Equation 4 into Equation 7:

$$\begin{aligned} logP(x^{(n)}, S^{(n)} = k|\pi, P) &= logP(x^{(n)}|s^{(n)} = k, P) + logP(s^{(n)} = k|\pi) \qquad (7)\\ &= log\left(\Pi_{d=1}^{D}p_{kd}^{x_d^{(n)}}(1-p_{kd})^{1-x_d^{(n)}}\right) + log\pi_k \\ &= \Sigma_{d=1}^{D}x_d^{(n)}logp_{kd} + (1-x_d^{(n)})log(1-p_{kd}) + log\pi_k \end{aligned}$$

Thus:

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

Setting up derivative with respect to p_{kd}:

$$\frac{\delta Q(\pi, P)}{\delta p_{kd}} = \frac{x_d^{(n)}}{p_{kd}} + \left(-\frac{(1-x_d^{(n)})}{1-p_{kd}}\right)$$

$$\frac{\delta Q(\pi, P)}{\delta p_{kd}} = \Sigma_{n=1}^{N}R_{nk}\left(\frac{x_d^{(n)}}{p_{kd}} - \frac{1-x_d^{(n)}}{1-p_{kd}}\right)$$

Setting derivative above to 0:

$$\frac{\delta Q(\pi, P)}{\delta p_{kd}} = \Sigma_{n=1}^{N}R_{nk}\left(\frac{x_d^{(n)}}{p_{kd}} - \frac{1-x_d^{(n)}}{1-p_{kd}}\right) = 0$$

$$\Sigma_{n=1}^{N}R_{nk}\frac{x_d^{(n)}(1-p_{kd}) - p_{kd}(1-x_d^{(n)})}{p_{kd}(1-p_{kd})} = 0$$

$$\Sigma_{n=1}^{N}R_{nk}x_d^{(n)}(1-p_{kd}) - p_{kd}(1-x_d^{(n)}) = 0$$

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

$$\Sigma_{n=1}^{N}R_{nk} - \left(x_d^{(n)}p_{kd} + p_{kd} - p_{kd}x_d^{(n)}\right) = \Sigma_{n=1}^{N}R_{nk} - x_d^{(n)}$$

$$\Sigma_{n=1}^{N}R_{nk}(p_{kd}) = \Sigma_{n=1}^{N}R_{nK}(x_d^{(n)})$$

Therefore, solving for p_{kd}:

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

Setting up derivative with respect to π_k

Considering the constraint:

$$\Sigma_{k=1}^{K}\pi_k = 1 \tag{8}$$

Making use of a Lagrange multiplier λ and define the Lagrangian as:

$$\mathcal{L}(\pi, \lambda) = Q(\pi, P) + \lambda\left(\Sigma_{k=1}^{K}\pi_k - 1\right)$$

Differentiating the Lagrangian with respect to π_k:

$$\frac{\delta\mathcal{L}}{\delta\pi_k} = \Sigma_{n=1}^{N}R_{nk}\frac{1}{\pi_k} + \lambda$$

Setting the differentiated equation above to 0:

$$\Sigma_{n=1}^{N}R_{nk}\frac{1}{\pi_k} + \lambda = 0$$

$$\Sigma_{n=1}^{N}R_{nk} = -\lambda\pi_k$$

Summing over k both sides:

$$\Sigma_{k=1}^{K}\Sigma_{n=1}^{N}R_{nk} = -\lambda\Sigma_{k=1}^{K}\pi_k$$

Now with the constraint 8 in mind:

$$\Sigma_{k=1}^{K}\Sigma_{n=1}^{N}R_{nk} = -\lambda$$

Due to the constraint 8, for all n because the sum of responsibilities over all components for a given data point is 1, so:

$$N = -\lambda; \lambda = -N$$

Plugging into equation for π_k:

$$\Sigma_{n=1}^{N}R_{nk} = N\pi_k$$

$$\pi_k = \frac{1}{N}\Sigma_{n=1}^{N}R_{nk}$$

d) Log Likelihood for different K

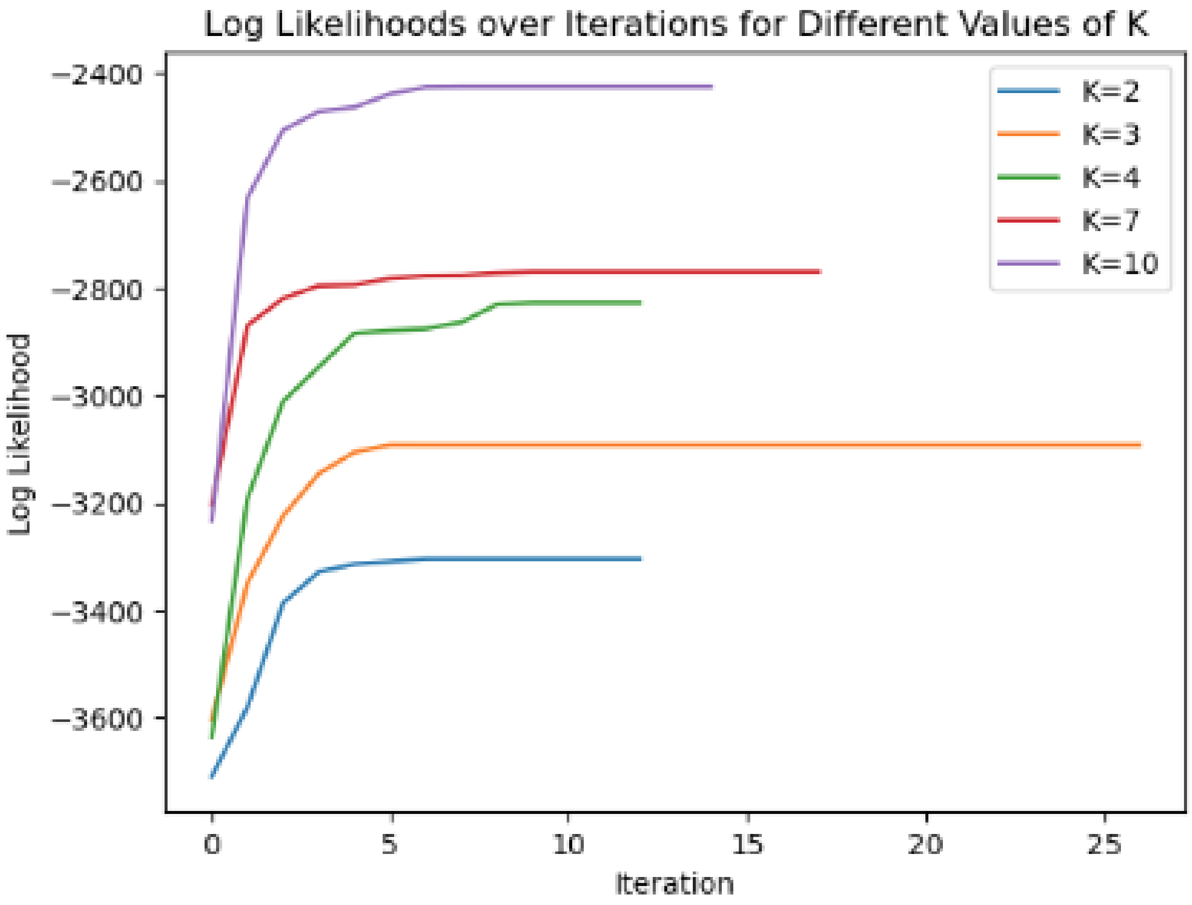


Figure 3: Log Likelihoods over Iterations for Different Values of K

e) Responsibilities, Cluster Means & Probability Vector Images

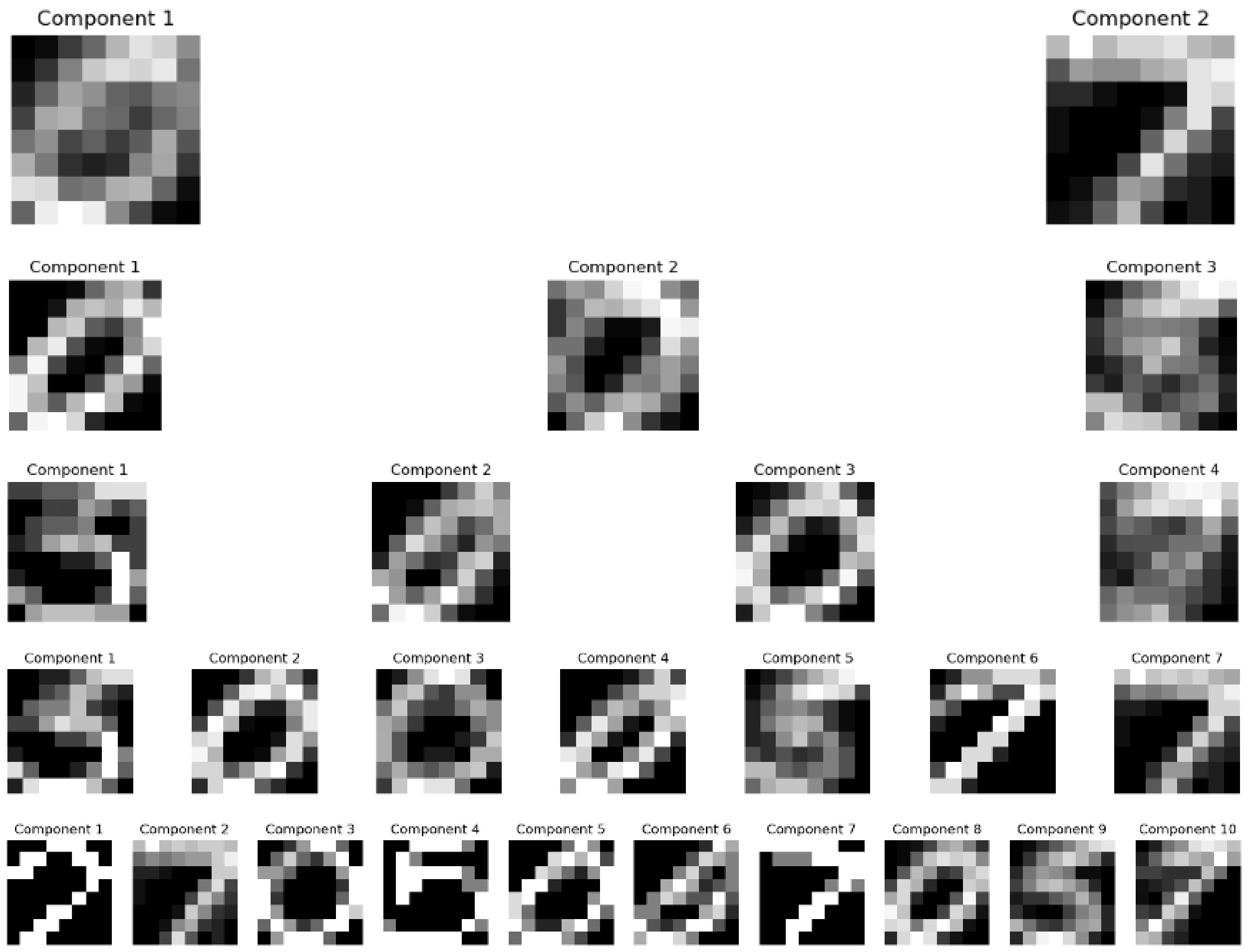
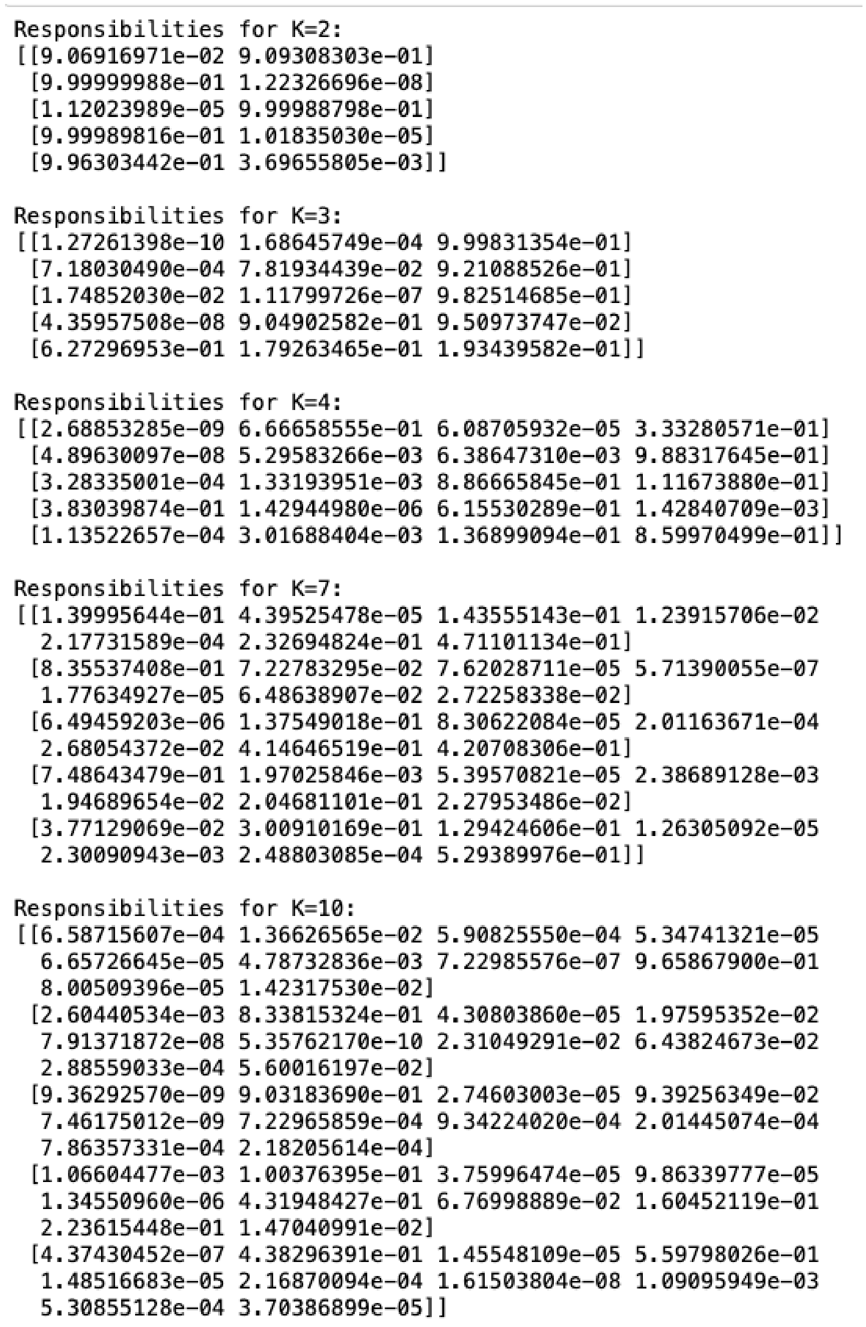


Figure 4: Learned probability vectors as images for $k = \{2, 3, 4, 7, 10\}$, each run 5 times

From the Figure 4 above, for lower K values, the components tend to capture more general patterns of the data, while for higher K values, the components might capture more specific features but might have a greater risk of modeling noise. For k=10, we can see digits 2,7,0 and 5 appearing in the 8 $\times$ 8 images. The algorithm works well, and finds good clusters. Looking at the responsibility of the EM algorithm, the numbers from the responsibilities above imply that as K increases, clarity of cluster assignments might decrease.



```
Responsibilities for K=2:
[[9.06916971e-02 9.09308303e-01]
 [9.99999988e-01 1.22326696e-08]
 [1.12023989e-05 9.99988798e-01]
 [9.99989816e-01 1.01835030e-05]
 [9.96303442e-01 3.69655805e-03]]

Responsibilities for K=3:
[[1.27261398e-10 1.68645749e-04 9.99831354e-01]
 [7.18030490e-04 7.81934439e-02 9.21088526e-01]
 [1.74852030e-02 1.11799726e-07 9.82514685e-01]
 [4.35957508e-08 9.04902582e-01 9.50973747e-02]
 [6.27296953e-01 1.79263465e-01 1.93439582e-01]]

Responsibilities for K=4:
[[2.68853285e-09 6.66658555e-01 6.08705932e-05 3.33280571e-01]
 [4.89630097e-08 5.29583266e-03 6.38647310e-03 9.88317645e-01]
 [3.28335001e-04 1.33193951e-03 8.86665845e-01 1.11673880e-01]
 [3.83039874e-01 1.42944980e-06 6.15530289e-01 1.42840709e-03]
 [1.13522657e-04 3.01688404e-03 1.36899094e-01 8.59970499e-01]]

Responsibilities for K=7:
[[1.39995644e-01 4.39525478e-05 1.43555143e-01 1.23915706e-02
  2.17731589e-04 2.32694824e-01 4.71101134e-01]
 [8.35537408e-01 7.22783295e-02 7.62028711e-05 5.71390055e-07
  1.77634927e-05 6.48638907e-02 2.72258338e-02]
 [6.49459203e-06 1.37549018e-01 8.30622084e-05 2.01163671e-04
  2.68054372e-02 4.14646519e-01 4.20708306e-01]
 [7.48643479e-01 1.97025846e-03 5.39570821e-05 2.38689128e-03
  1.94689654e-02 2.04681101e-01 2.27953486e-02]
 [3.77129069e-02 3.00910169e-01 1.29424606e-01 1.26305092e-05
  2.30090943e-03 2.48803085e-04 5.29389976e-01]]

Responsibilities for K=10:
[[6.58715607e-04 1.36626565e-02 5.90825550e-04 5.34741321e-05
  6.65726645e-05 4.78732836e-03 7.22985576e-07 9.65867900e-01
  8.00509396e-05 1.42317530e-02]
 [2.60440534e-03 8.33815324e-01 4.30803860e-05 1.97595352e-02
  7.91371872e-08 5.35762170e-10 2.31049291e-02 6.43824673e-02
  2.88559033e-04 5.60016197e-02]
 [9.36292570e-09 9.03183690e-01 2.74603003e-05 9.39256349e-02
  7.46175012e-09 7.22965859e-04 9.34224020e-04 2.01445074e-04
  7.86357331e-04 2.18205614e-04]
 [1.06604477e-03 1.00376395e-01 3.75996474e-05 9.86339777e-05
  1.34550960e-06 4.31948427e-01 6.76998889e-02 1.60452119e-01
  2.23615448e-01 1.47040991e-02]
 [4.37430452e-07 4.38296391e-01 1.45548109e-05 5.59798026e-01
  1.48516683e-05 2.16870094e-04 1.61503804e-08 1.09095949e-03
  5.30855128e-04 3.70386899e-05]]
```

Figure 5: Responsibilities π_k for $k = \{2, 3, 4, 7, 10\}$

From the responsibilities figure above, and the cluster means as learned probability vector images in the Appendix, for k=2, model is very certain about cluster assignments as the probabilities in the rows are close to 1 or 0, suggesting a clear division. When k=3, there is still a high degree of certainty. As the K values increase, the responsibilities are more distributed evenly across clusters for some data points or might be assigning data points into too many clusters.However, when looking at the cluster means(Appendix), and the learned probability vector images, it might say otherwise. As the number of K increases as shown in Figure 4, the cluster assignments are much more certain. **Improvements to the Model:**

- Better initialization strategies can lead to more consistent solutions, such as using k-means clustering to initialize the means.

- Tuning the convergence criteria (tolerance, maximum iterations) can ensure the algorithm has converged properly.

f) GZip & Naive Encoding

To express the log-likelihood in bits, we converted the natural log likelihoods to base 2. THe natural log is converted to log base 2 as:

$$log_2(x) = \frac{ln(x)}{ln(2)}$$

In information theory, entropy is typically measured in bits, which are units of information based on binary logarithms (base 2).The notion of entropy is tied to the efficiency of encoding schemes: a lower entropy suggests that less data is required, on average, to encode samples from the distribution. we can interpret the log-likelihood as an average code length in bits required to encode the data if we were to use an optimal code based on the probability distribution estimated by our model.

Table 2: Sizes and Log likelihood of Data

Log-likelihood	-4452
Naive Encoding Length	6400 bits
GZip Compressed Size	5544 bits

From the table, the negative value suggests that we're dealing with the log of a probability less than 1, which is typical in information theory when calculating entropy or expected message length. The naive encoding length is the total number of bits required to store the data without any compression, simply by storing each binary pixel directly, amounting to 6400 bits for the dataset. The compressed size using gzip, which is 5544 bits, indicates that gzip is able to compress the data to a size that is less than the naive encoding, showing the efficacy of gzip's compression algorithm in reducing the size of the data. However, the compressed size is still larger than the magnitude of the log-likelihood in bits, which suggests that gzip, while effective, does not reach the theoretical limit of data compressibility as implied by the model's log-likelihood. The difference between the gzip compression and the theoretical limit can be attributed to the practical constraints and heuristic methods used in gzip, which, unlike the theoretical model, has to work with actual data encoding and cannot achieve the entropy limit of the modeled distribution.

g) Total Encoding Cost

Table 3: Total Encoding Cost with k vs Gzip Compression

Total Encoding Cost: k=2	5071 bits
Total Encoding Cost: k=3	4732 bits
Total Encoding Cost: k=4	4730 bits
Total Encoding Cost: k=7	4027 bits
Total Encoding Cost: k=10	3780 bits
GZip Compressed size	5568 bits

For all values of K, total encoding costs are less than the data size compressed using gzip, which suggests that the model-based encoding is more efficient in terms of space. As K increases from 2 to 7, total encoding cost generally decreases, indicating that models with more components can capture the data structure better, leading to more efficient encoding of data. However, as K becomes too large, the cost starts to diminish which could be a sign of overfitting- where model starts to fit the noise. There seems to be an optimal range for K. The table shows that with an appropriately chosen K, it can offer a better representation than gzip.

Problem 4: LGSSMs, EM and SSID

a) Kalman Filter & Kalman Smoother

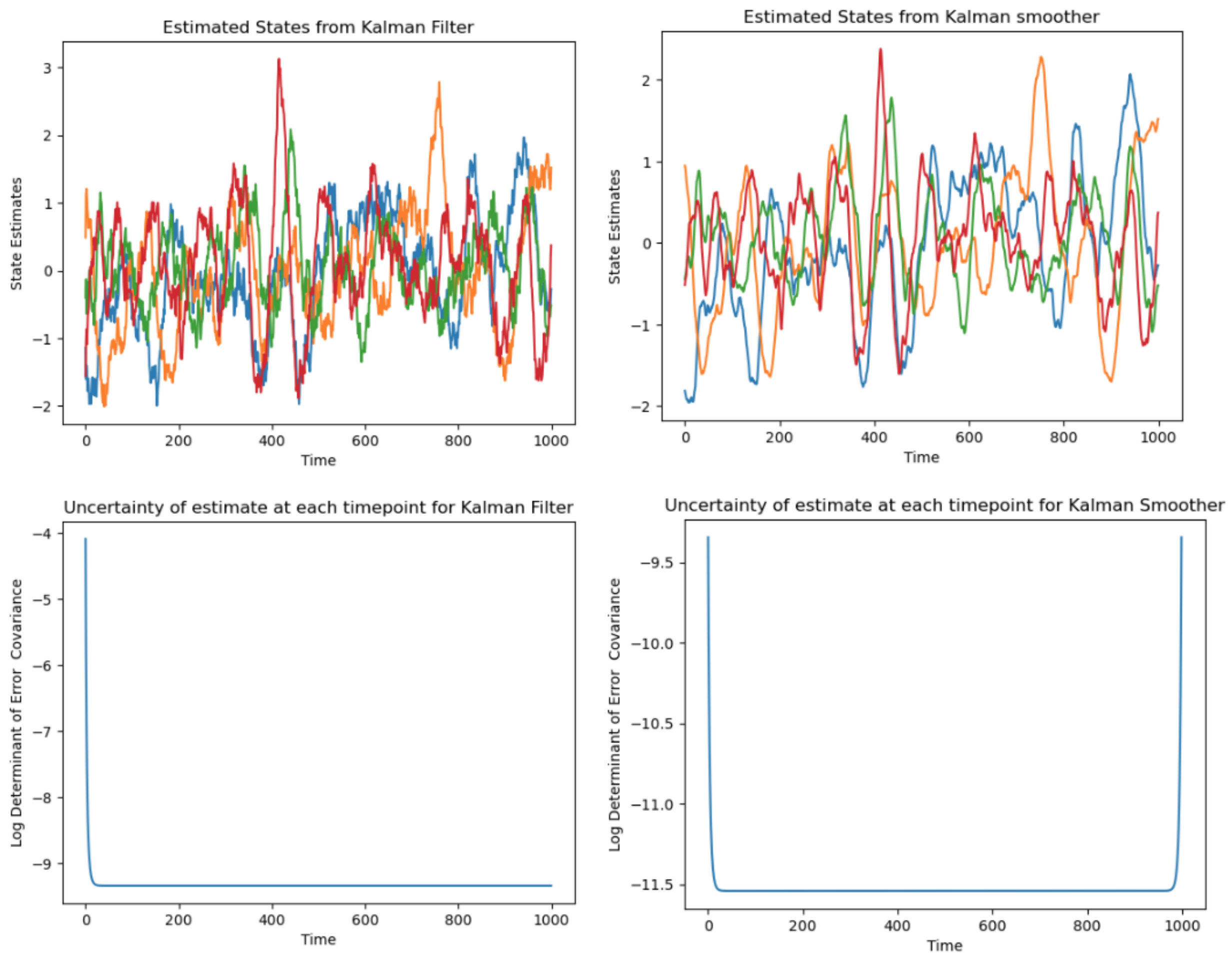


Figure 6: Estimated States from Kalman Filter $y_t|x_{1:t}$ vs Kalman Smoother $y_t|x_{1:T}$ over Time

The top two plots show the estimated states from the Kalman Filter (left) and Kalman Smoother (right) respectively. The Kalman Filter provides real-time estimates as new data comes in, thus it can be more reactive to recent observations, which may lead to more volatility in the state estimates. This can be observed as more fluctuations in the state lines. The Kalman Smoother, on the other hand, adjusts the state estimates by considering both past and future observations, leading to smoother state estimates which are less influenced by recent noise.

The uncertainty in the state estimate at any given time point during smoothing is influenced by observations that occur later in time. Future observations can provide context that is not available at the moment when the Kalman filter makes its predictions, such as revealing dependencies that only become apparent later. The Kalman Smoother, on the other hand, shows a significant reduction in uncertainty (higher log determinant values) for most time points, with a dramatic increase towards the end. The Kalman smoother, as shown, provides a more accurate estimate by considering the whole dataset, which shows why V (uncertainty on the estimate) decreases over time compared to the filter, which utilizing data up to only that current state.

b) EM on ssm_spin.txt training data

Update for Rnew:

M-Step for R:

$$p(x_t|y_t) \propto exp\left(-\frac{1}{2}(x_t - Cy_t)^T R^{-1}(x_t - Cy_t)\right)$$

The goal is to maximize the expected log-likelihood with respect to R, given current estimates of y_t and C.

$$R_{\text{new}} = \arg\max_R \left\langle \sum_t \ln p(x_t|y_t) \right\rangle_q$$

Since the log probability of the Gaussian distribution is give by:

$$\ln p(x_t|y_t) = ln\left((exp(\frac{1}{2}(x_t - Cy_t)^T R^{-1}(x_t - Cy_t))\right) - \frac{1}{2}\ln|2\pi R^{-1}|$$

where $|2\pi R^{-1}|$ denotes the determinant of $2\pi R^{-1}$ which is part of the normalization constant of the Gaussian distribution. The log of an exponential function simplifies to the exponent:

$$\ln\left((exp(\frac{1}{2}(x_t - Cy_t)^T R^{-1}(x_t - Cy_t))\right) = -\frac{1}{2}(x_t - Cy_t)^T R^{-1}(x_t - Cy_t)$$

The normalization constant term becomes:

$$-\frac{1}{2}\ln|2\pi R^{-1}| = -\frac{T}{2}\ln|det(R^{-1})| - constant$$

Combining these results to gain expression for log likelihood for entire data-set:

$$\Sigma_{t=1}^T \ln p(x_t|y_t) = -\frac{T}{2}\ln|det(R^{-1})| - \frac{1}{2}\Sigma_{t=1}^T (x_t - Cy_t)^T R^{-1}(x_t - Cy_t)$$

The quadratic term, when expanded becomes:

$$-\frac{1}{2}\Sigma_{t=1}^T (x_t - Cy_t)^T R^{-1}(x_t - Cy_t) = -\frac{1}{2}\Sigma_{t=1}^T (x_t^T R^{-1}x_t - x_t^T R^{-1}Cy_t - y_t^T C^T R^{-1}x_t + y_t^T C^T R^{-1}Cy_t)$$

Thus, the final arg max argument becomes:

$$\Sigma_{t=1}^T \ln p(x_t|y_t) = -\frac{T}{2}ln|det(R^{-1})| - \frac{1}{2}\Sigma_{t=1}^T \left(x_t^T R^{-1}x_t - x_t^T R^{-1}Cy_t - y_t^T C^T R^{-1}x_t + y_t^T C^T R^{-1}Cy_t\right)$$

Using $\frac{\delta Tr[AB]}{\delta A} = B^T$, and using matrix calculus, for matrix X:

$$\frac{\delta \ln|det(X)|}{\delta X} = (X^{-1})^T \tag{9}$$

$$\frac{\delta A^T X B}{\delta X} = AB^T \tag{10}$$

Differentiating first term of Equation **??** and using Equation 10:

$$\frac{\delta}{\delta R^{-1}}(\Sigma_{t=1}^T lnp(x_t|y_t)) = \frac{T}{2}R^T + \frac{\delta\{\cdot\}}{\delta R^{-1}}\left(-\frac{1}{2}\Sigma_{t=1}^T \left(x_t^T R^{-1}x_t - x_t^T R^{-1}Cy_t - y_t^T C^T R^{-1}x_t + y_t^T C^T R^{-1}Cy_t\right)\right)$$

$$\frac{\delta\{\cdot\}}{\delta R^{-1}}\left(-\frac{1}{2}\Sigma_{t=1}^T \left(x_t^T R^{-1}x_t - x_t^T R^{-1}Cy_t - y_t^T C^T R^{-1}x_t + y_t^T C^T R^{-1}Cy_t\right)\right)$$

$$= -\frac{1}{2}\Sigma_{t=1}^T (x_t x_t^T - x_t y_t^T C^T - Cy_t x_t^T + Cy_t y_t^T C^T)$$

Setting $\frac{\delta}{\delta R^{-1}} = 0$:

$$\frac{T}{2} R^T = \frac{1}{2} \Sigma_{t=1}^T (x_t x_t^T - x_t y_t^T C^T - C y_t x_t^T + C y_t y_t^T C^T)$$

Dividing by T:

$$R_{\text{new}} = \frac{1}{T} \Sigma_{t=1}^T (x_t x_t^T - x_t y_t^T C^T - C y_t x_t^T + C y_t y_t^T C^T) \tag{11}$$

As from lecture notes for C_{new} derivation:

$$C_{new} = \left(\Sigma_t x_t \langle y_t \rangle^T\right) \left(\Sigma_t \langle y_t y_t^T \rangle\right)^{-1}$$

Substituting C_{new} in expression $C y_t y_t^T C^T$:

$$\begin{aligned}
\sum_{t=1}^T C_{\text{new}} y_t y_t^T C_{\text{new}}^T &= C_{\text{new}} \left(\sum_{t=1}^T y_t y_t^T \right) C_{\text{new}}^T \\
&= \left(\sum_{t=1}^T x_t y_t^T \right) \left(\sum_{t=1}^T y_t y_t^T \right)^{-1} \left(\sum_{t=1}^T y_t y_t^T \right) \left(\left(\sum_{t=1}^T y_t y_t^T \right)^{-1} \right)^T \left(\sum_{t=1}^T x_t y_t^T \right)^T \\
&= \left(\sum_{t=1}^T x_t y_t^T \right) \left(\sum_{t=1}^T y_t y_t^T \right)^{-1} \left(\sum_{t=1}^T y_t x_t^T \right) \\
&= C_{\text{new}} \left(\sum_{t=1}^T y_t x_t^T \right)
\end{aligned}$$

where: $C_{\text{new}} = \left(\sum_{t=1}^T x_t y_t^T \right) \left(\sum_{t=1}^T y_t y_t^T \right)^{-1}$. Substituting this into equation 11:

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

Update for Qnew:

M-step for Q:

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

The goal to maximize the expected log likelihood with respect to Q, given estimates of y_t and A:

$$Q_{\text{new}} = \arg\max_Q \left\langle \sum_{t=2}^T \ln p(y_t|y_{t-1}) \right\rangle_q$$

Same as for R_{new}, since log probability of Gaussian distribution is given by:

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

$$ln\left((exp(-\frac{1}{2}(y_t - Ay_{t-1})^T Q^{-1} (y_t - Ay_{t-1})) \right) = -\frac{1}{2}(y_t - Ay_{t-1})^T Q^{-1} (y_t - Ay_{t-1}))$$

The normalization constant term becomes:

$$\frac{T-1}{2} \ln |det(Q^{-1})|$$

Combining these results to gain expression for log ikelihood for entire dataset:

$$\Sigma_{t=2}^T \ln(y_t|y_{t-1}) = \frac{T-1}{2} \ln |det(Q^{-1})| - \frac{1}{2}(y_t - Ay_{t-1})^T Q^{-1} (y_t - Ay_{t-1}))$$

The quadratic term, when expanded becomes:

$$-\frac{1}{2}(y_t - Ay_{t-1})^T Q^{-1}(y_t - Ay_{t-1})) = -\frac{1}{2}\Sigma_{t=2}^T(y_t^T Q^{-1}y_t - y_t^T Q^{-1}Ay_{t-1} - y_{t-1}^T A^T Q^{-1}y_t + y_{t-1}^T A^T Q^{-1}y_{t-1})$$

Using the same Equations 9 & 10:

$$\frac{\delta}{\delta Q^{-1}}(\Sigma_{t=1}^T lnp(y_t|y_{t-1})) = \frac{T-1}{2}Q^T + \frac{\delta\{\cdot\}}{\delta Q^{-1}}\left(-\frac{1}{2}\Sigma_{t=2}^T\left(y_t^T Q^{-1}y_t - y_t^T Q^{-1}Ay_{t-1} - y_{t-1}^T A^T Q^{-1}y_t + y_{t-1}^T A^T Q^{-1}y_{t-1}\right)\right)$$

$$\begin{aligned}\frac{\delta\{\cdot\}}{\delta Q^{-1}}&\left(-\frac{1}{2}\Sigma_{t=2}^T\left(y_t^T Q^{-1}y_t - y_t^T Q^{-1}Ay_{t-1} - y_{t-1}^T A^T Q^{-1}y_t + y_{t-1}^T A^T Q^{-1}y_{t-1}\right)\right)\\ &= -\frac{1}{2}\Sigma_{t=2}^T(y_t y_t^T - y_t y_{t-1}^T A^T - Ay_{t-1}y_t + Ay_{t-1}y_{t-1}^T A^T)\end{aligned}$$

Setting $\frac{\delta}{\delta Q^{-1}}$ =0:

$$\frac{T-1}{2}Q^T = \frac{1}{2}\Sigma_{t=2}^T(y_t y_t^T - y_t y_{t-1}^T A^T - Ay_{t-1}y_t + Ay_{t-1}y_{t-1}^T A^T)$$

Dividing by $T-1$:

$$Q_{new} = \frac{1}{T-1}\Sigma_{t=2}^T\left((y_t y_t^T - y_t y_{t-1}^T A^T - Ay_{t-1}y_t + Ay_{t-1}y_{t-1}^T A^T)\right) \tag{12}$$

As from lecture notes for A_{new} derivation:

$$A_{new} = \left(\Sigma_{t=2}\langle y_t y_{t-1}^T\rangle\right)\left(\Sigma_{t=2}\langle y_t y_{t-1}^T\rangle\right)^{-1}$$

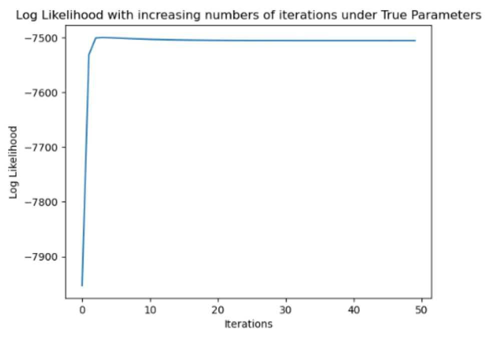
Substituting A_{new} into expression $Ay_{t-1}y_{t-1}^T A^T$:

$$\begin{aligned}\sum_{t=2}^T A_{\text{new}}y_{t-1}y_{t-1}^T A_{\text{new}}^T &= A_{\text{new}}\left(\sum_{t=2}^T y_{t-1}y_{t-1}^T\right)A_{\text{new}}^T\\ &= \left(\sum_{t=2}^T y_t y_{t-1}^T\right)\left(\sum_{t=2}^T y_t y_{t-1}^T\right)^{-1}\left(\sum_{t=2}^T y_{t-1}y_{t-1}^T\right)\left(\left(\sum_{t=1}^T y_t y_{t-1}^T\right)^{-1}\right)^T\left(\sum_{t=2}^T y_t y_{t-1}^T\right)^T\\ &= \left(\sum_{t=2}^T y_{t-1}y_t^T\right)\left(\sum_{t=2}^T y_{t-1}y_{t-1}^T\right)^{-1}\left(\sum_{t=2}^T y_{t-1}y_t^T\right)\\ &= A_{\text{new}}\left(\sum_{t=2}^T y_{t-1}y_t^T\right)\end{aligned}$$

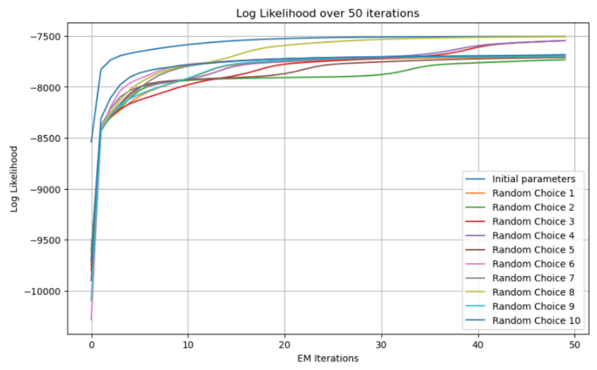
where: $A_{\text{new}} = \left(\sum_{t=2}^T y_{t-1}y_t^T\right)\left(\sum_{t=2}^T y_t y_t^T\right)^{-1}$. Substituting this into equation 12:

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

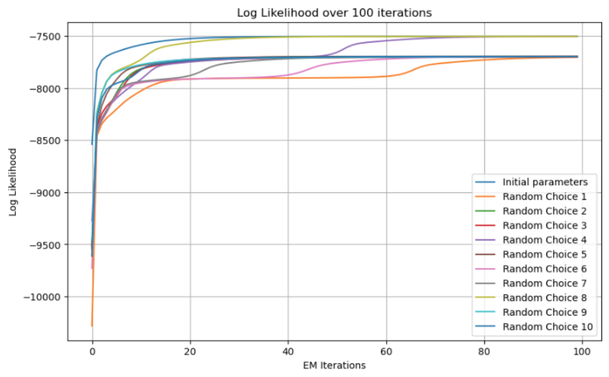
EM on ssm_spin.txt training data



(a) Log-likelihood over 50 iterations under true parameters



(b) Log-likelihood over 50 iterations



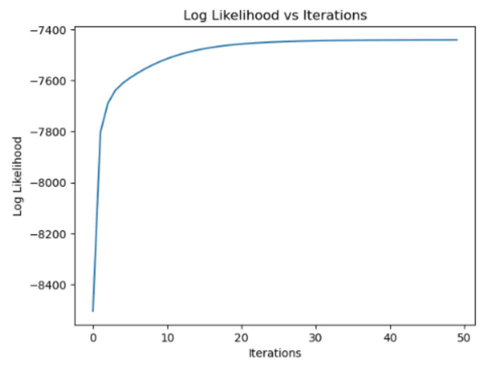
(c) Log-Likelihood over 100 iterations

Figure 7: Log-likelihoods values computed of 50&100 EM iterations for LGSSM with ssm_spins.txt training data

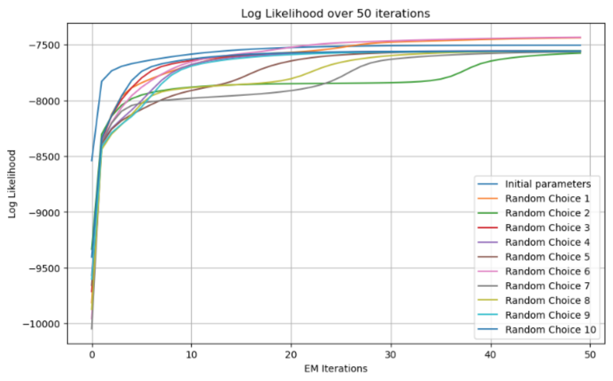
From Figure 6(a), even if the initial parameters are true parameters, the EM algorithm still undergoes a few iterations to ensure that it haws reached a point of convergence. This is due to the iterative refinement and relies on the convergence criteria which is placed in the Appendix Code for this question. The EM algorithm, even with true parameters, it is common for initial fluctuation in log-likelihood as it refines the estimates. Additionally, it could be the stochastic nature of the data. From the first figure above, all the curves exhibit a steep increase with 10 different random initial conditions during the initial iterations. This indicates that EM algorithm makes significant improvements to parameter estimates in the early iterations. Similar trends are seen with the log-likelihoods in 50 iterations and 100 iterations, EM algorithm still converges between 15-30 iterations.

c) EM on ssm_spins_test.txt test data

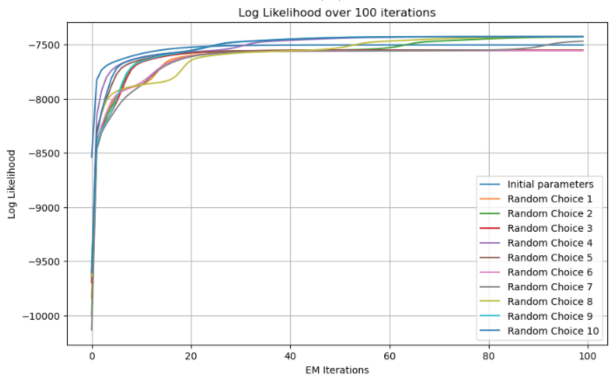
EM on ssm_spins_test.txt test data



(a) Log-likelihood over 50 iterations under true parameters



(b) Log-likelihood over 50 iterations (10 random choices



(c) Log-likelihood over 100 iterations (10 random choices)

Figure 8: Log-likelihoods values computed of 50&100 EM iterations for LGSSM ssm_spins_test.txt test data

In, Figure 8(a), depicting the test data, shows a sharp increase in log-likelihood in the initial iterations, which then levels off, much like the training data. However, the log-likelihood in the test data doesn't reach the same height as in the training data, which is common since models usually fit better to the data they were trained on. Nonetheless, the increase and leveling off indicate that the model, with parameters learned from the training data, generalizes to the test data, capturing its underlying structure. In this Figure, the EM algorithm converges later than when it is run on the training data as there is a less steep curve as compared to Figure 7(a).

In both figures, the log-likelihood values indicate the EM algorithm's effectiveness in parameter estimation. With each iteration, the algorithm refines the estimates of the model parameters to better fit the data, reflected in the increasing log-likelihood values. The plateauing effect illustrates the convergence of the algorithm, where further iterations do not significantly increase the fit of the model to the data, as measured by the log-likelihood. This suggests that the EM algorithm has effectively learned the parameters that best represent the underlying process that generated the data. The behavior in the test data confirms that the learned model is not just memorizing the training data but also capturing general features that can be applied to unseen data.

Problem 5: Decrypting Messages with MCMC

a) ML Estimates Formulae & Probability Estimates

Model given, so that each symbol is independent of the preceding text given only the symbol before:

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

Letting $\hat{S}$ be the set of all symbol and $C = |\hat{S}|$ be the cardinality of $\hat{S}$. This also means k is the number of symbols.
Let $\mathcal{N} : \hat{S} \to \mathcal{N}^+$ be the count of number of occurrences of symbols and pairs in "war-and-peace.txt".
The transition matrix $\psi \in \mathcal{R}^{k \times k}$ satisfies:

$$\begin{aligned} p(s_i = \alpha | s_{i-1} = \beta) &= \psi(\alpha, \beta) \\ &= \frac{\mathcal{N}(\alpha\beta)}{\Sigma_{\beta \in \hat{S}} \mathcal{N}(\beta)} \\ &= \frac{\mathcal{N}(\alpha\beta)}{\mathcal{N}(\beta)} \end{aligned}$$

We can define a transition matrix T, where T_{ij} is probability of state i transitioning to state j. As denoted from above, there are $\hat{S}$ states and we have the following constraints:

$$\Sigma_{j=1}^{\hat{S}} T_{ij} = 1 \tag{13}$$

Therefore,

$$logP(S) = logP(S_1 = t) + \Sigma_{i,j} n_{ij} log(T_{ij})$$

From $\mathcal{N} : \hat{S} \to \mathcal{N}^+$ be the count of number of occurrences of symbols and pairs, n_{ij} os number of times $i \to j$. The Lagrange functions become:

$$\mathcal{L}(T) = logP(S_1 = t) + \Sigma_{i,j} n_{ij} log(T_{ij}) - \Sigma_{i=1}^{S} \left(\lambda_i (\Sigma_{j=1}^{S} T_{ij-1}) \right)$$

Deriving the Lagrange function with respect to T_{ij}:

$$\frac{\delta \mathcal{L}(T)}{\delta T_{ij}} = \frac{n_{ij}}{T_{ij}} - \lambda_i = 0$$

To get T_{ij}:

$$T_{ij} = \frac{n_{ij}}{\lambda_i}$$

Since the constraints 13:

$$\Sigma_{j=1}^{S} \frac{n_{ij}}{\lambda_i} = 1$$

Multiplying by λ_i:

$$\lambda_i = \Sigma_{j=1}^{S} n_{ij}$$

Substituting back into the transition matrix T_{ij}:

$$T_{ij} = \frac{n - ij}{\Sigma_{j=1}^{S} n_{ij}}$$

hence, this gives the transition probability by ML estimation. To solve for the stationary distribution ϕ of the Markov chain, let's denote $\phi = [\phi(u), ...\phi(S)]$ as stationary distribution, and T as the transition matrix with elements T_{ij}. The relationship of ϕ and T can be expanded into a system of linear equations:

$$\begin{aligned} \phi(u)T_{u1} + \phi(v)T_{v1} + \cdots + \phi(S)T_{S1} &= \phi(1) \\ \phi(u)T_{u2} + \phi(v)T_{v2} + \cdots + \phi(S)T_{S2} &= \phi(2) \\ &\vdots \\ \phi(u)T_{uS} + \phi(v)T_{vS} + \cdots + \phi(S)T_{SS} &= \phi(S) \end{aligned}$$

where each equation corresponds to one state of the Markov chain, and coefficients T_{ij} are the probabilities of transitioning from state j to state i. In order to solve ϕ, we need to find a vector that satisfies this system of equations and constraint that probabilities will sum to 1:

$$\phi(1) + \phi(2) + \cdots + \phi(S) = 1$$

In matrix form:

$$\begin{bmatrix}\phi(1) & \phi(2) & \cdots & \phi(S)\end{bmatrix}\begin{bmatrix}T_{11} & T_{12} & \cdots & T_{1S}\\ T_{21} & T_{22} & \cdots & T_{2S}\\ \vdots & \vdots & \ddots & \vdots\\ T_{S1} & T_{S2} & \cdots & T_{SS}\end{bmatrix} = \begin{bmatrix}\phi(1) & \phi(2) & \cdots & \phi(S)\end{bmatrix}$$

This simplifies to:

$$\phi T = \phi$$

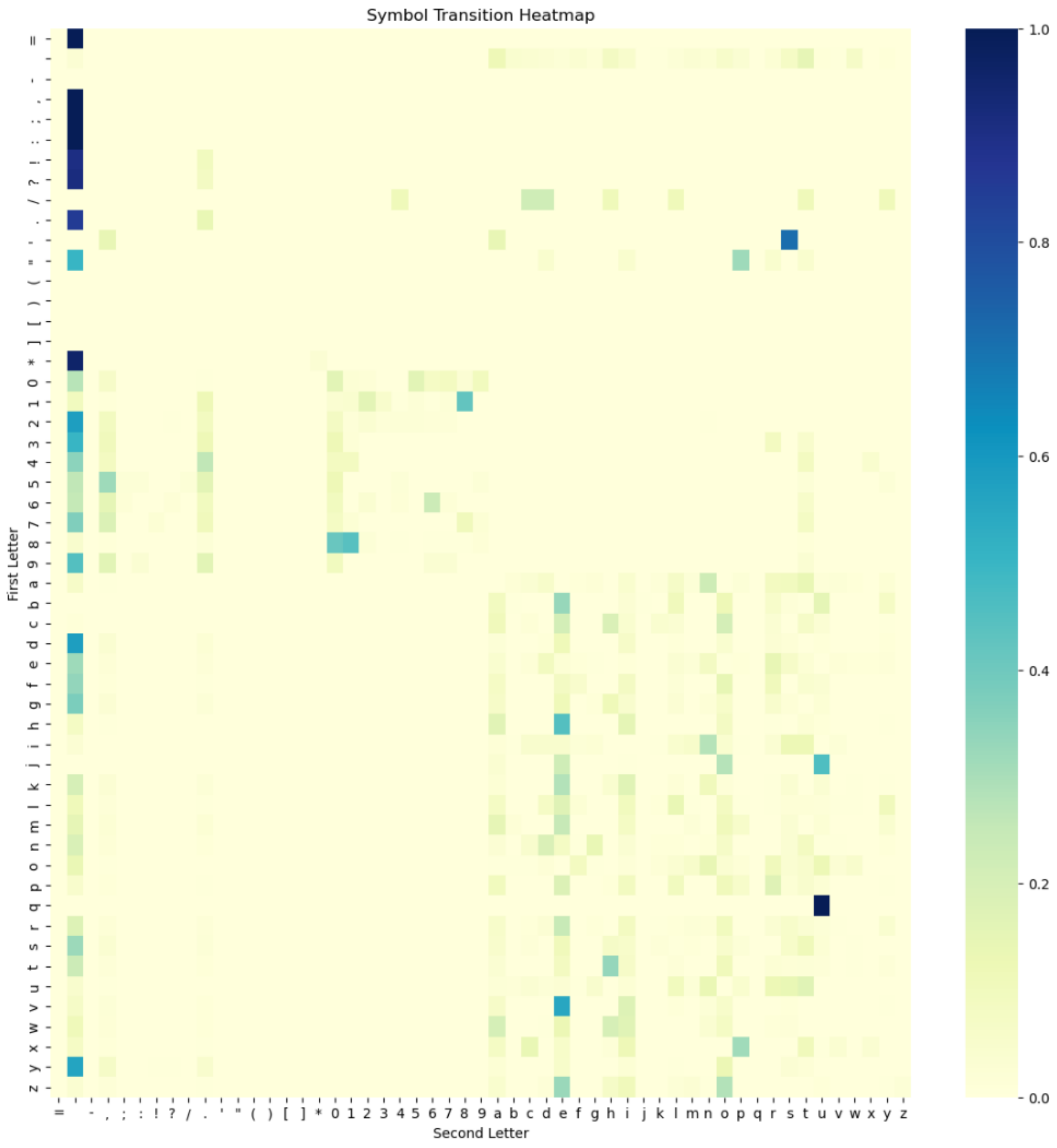


Figure 9: Transition Matrix T: Columns on the x axis represent the current symbol(S_i) and Rows on the y axis represent the S_{i-1}

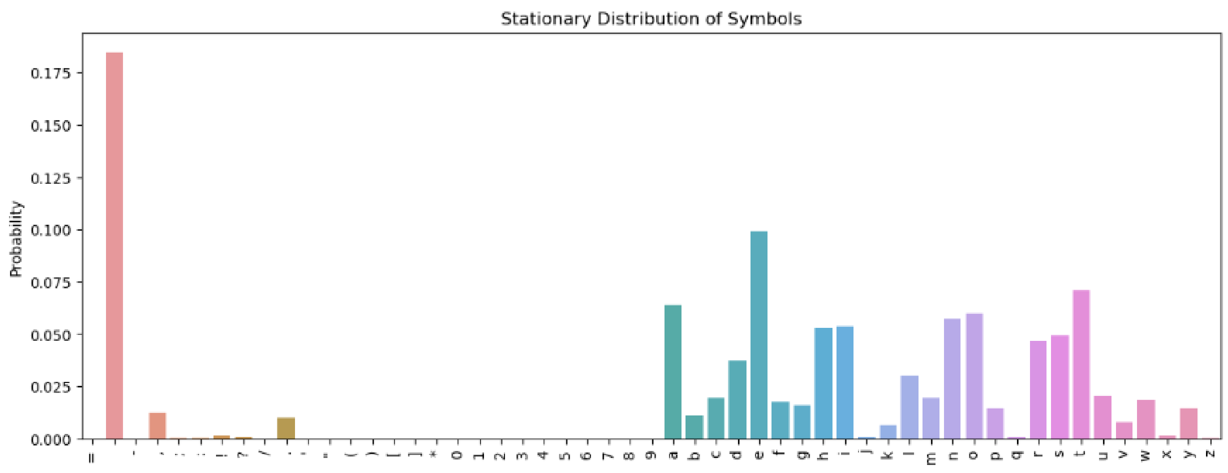


Figure 10: Estimated Stationary Distribution of the symbols

b) Joint Probability given σ

Latent variables $\sigma(s)$ for different symbols s are not independent. This is because each symbol in plain text is mapped to a unique symbol in the encrypted text, such as shown here:

$$\sigma(a) = s$$

This indicates that other symbols in the $\hat{S} = \{s_1, s_2, s_3 .....S_n\}$ cannot be equal to s, except for 'a' which belongs in the $\hat{S}$ dataset.

$$\sigma(s_i) \neq s$$

In this case, let us denote:

$$\{s_1, s_2, s_3 .....S_n\} = decryptedtext$$

$$\{e_1, e_2, s_3 .....e_n\} = encryptedtext$$

$$\sigma = e_n \rightarrow s_n$$

Initial probability is denoted as:

$$P(e_1|\sigma) = \phi(\sigma^{-1}(e_1))$$

where ϕ is stationary distribution of decrypted symbols and σ^{-1} is inverse mapping the encrypted symbol back to decrypted symbol.
Transition probability is denoted as:

$$P(e_i|e_{i-1}, \sigma) = T_{\sigma^{-1}(e_{i-1}\sigma^{-1}(e_i))}$$

Joint probability of encrypted text given permutation σ is:

$$P(e_1, e_2, s_3 .....e_n|\sigma) = (e_1|\sigma)\Pi_{i=2}^{n} P(e_i|e_{i-1}, \sigma)$$

$$P(e_1, e_2, s_3 .....e_n|\sigma) = P(e_1)$$

Given these components, the full joint probability expression for encrypted text given the permutation σ is:

$$P(e_1, e_2, s_3 .....e_n|\sigma) = \phi(\sigma^{-1}(e_1))\Pi_{i=2}^{n} T_{\sigma^{-1}(e_{i-1}\sigma^{-1}(e_i))}$$

c) Acceptance Probability in MH algorithm

As proposal function S is symmetric, the probability of proposing move from permutation σ to σ' is the same as probability of proposing a move from σ' to σ. Therefore, this can be seen as:

$$S(\sigma \to \sigma') = \begin{cases} \frac{1}{\binom{n}{2}} & \text{if } \sigma \leftrightarrow \sigma' \\ 0 & \text{otherwise} \end{cases}$$

Given a current state σ, algorithm proposes a new state σ' by swapping two symbols at random Likelihood $\phi(\sigma)$ of a permutation σ is probability of observing the encrypted sequence under that key which is written as:

$$\phi(\sigma) = P(e_1|\sigma)\Pi_{i=2}^{n} P(e_i|e_{i-1,\sigma}) \tag{14}$$

As $S(\sigma \to \sigma') \leftrightarrow S(\sigma' \to \sigma)$:

$$= P(s_1)\Pi_{i=2}^{n} P(s_i|s_{i-1})$$

Using Equation 14:

$$\frac{\phi(\sigma')S(\sigma' \to \sigma)}{\phi(\sigma)S(\sigma \to \sigma')} = \frac{P(\sigma'(e_1))}{P(\sigma(e_1))}\Pi_{i=2}^{N} \frac{P(\sigma'(e_i)|\sigma'(e_{i-1}))}{P(\sigma(e_i)|\sigma(e_{i-1}))}$$

$$= \frac{P(\sigma'(e_1))}{P(\sigma(e_1))}\Pi_{i=2}^{N} \frac{P(e_i|e_{i-1}, \sigma')}{P(P(e_i|e_{i-1}, \sigma)}$$

From this above, we can denote it as:

$$= \frac{\phi(\sigma')S(\sigma' \to \sigma)}{\phi(\sigma)S(\sigma \to \sigma')}$$

Therefore, the acceptance probability is:

$$\alpha = min\{1, \frac{\phi(\sigma')S(\sigma' \to \sigma)}{\phi(\sigma)S(\sigma \to \sigma')}\}$$

As the proposal probability is symmetric, we can cancel the $S(\sigma \to \sigma') \leftrightarrow S(\sigma' \to \sigma)$:

$$\alpha = min\{1, \frac{\phi(\sigma')}{\phi(\sigma)}\}$$

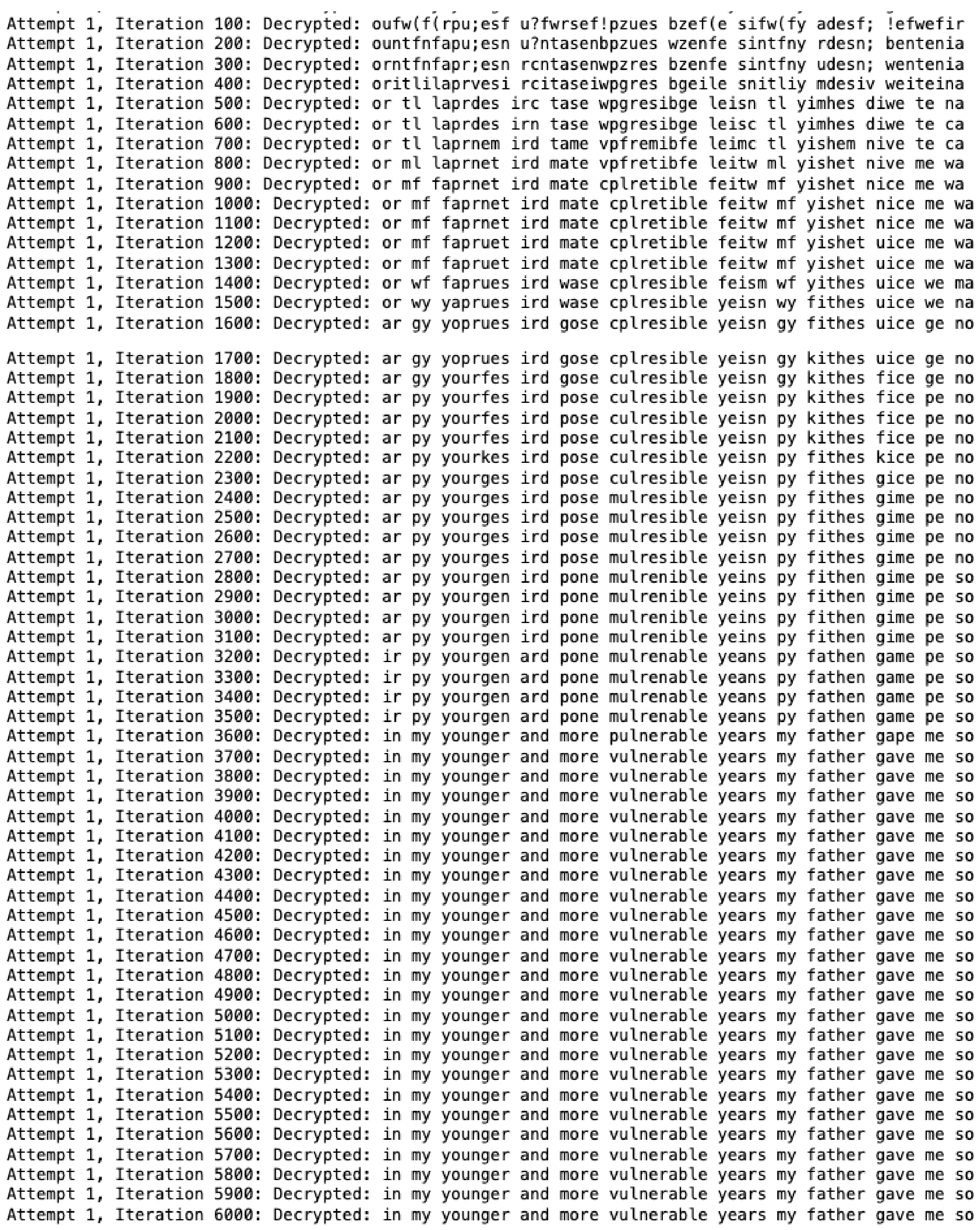
d) MH algorithm

For Metropolis Hastings sampler, here I have given the pseudocode to what I coded in order for the decryption to occur. This summarizes the code I have put in Appendix: Question 5 Code.

Algorithm 1 Metropolis-Hastings for Decryption

```
1: Input: Encrypted message, reference text 'War and Peace', symbols file
2: Output: Decrypted message
3: procedure Initialize
4:     Load encrypted message and symbols
5:     Analyze frequencies in message and reference text
6:     Map common symbols between them
7:     Prepare transition matrix $T_{\text{norm}}$ from reference text
8: procedure MetropolisHastings
9:     Start with initial permutation from frequency analysis
10:    for each iteration do
11:        Propose new permutation by swapping symbols
12:        Calculate and compare log-likelihoods
13:        Accept new permutation if likelihood is higher
14:        if iteration is a multiple of 100 then
15:            Preview decryption
16:        Record best score and permutation
17: procedure Output
18:    Decrypt message using best permutation
19:    Display decrypted message
```

The first run of my code didn't converge until 15200 iterations. Therefore, to make MH sampler more efficient, we used the transition matrix to see how letters are commonly arranged in the language. The algorithm now has a map as a guide that tells it which symbol in the encrypted message corresponds to the alphabet. It then tries many different combinations, making small changes each time (like swapping two letters around), to find the best match. It scores each attempt by how much it resembles the expected language structure and keeps track of the highest score. Furthermore, we used multiple initialization attempts, each time tracking the highest likelihood score achieved. The permutation with the overall highest score across these attempts was considered the best solution, yielding the most sensible decryption. This multi-initialization attempt was to aid in the potential problem of converging to local optima, instead directing the search toward the global optimum in the space of possible decryption. Both the original attempt code and the "smart initialization" code are placed in the Appendix below (Appendix: Question 5)



```
Attempt 1, Iteration 100: Decrypted: oufw(f(rpu;esf u?fwrsef!pzues bzef(e sifw(fy adesf; !efwefir
Attempt 1, Iteration 200: Decrypted: ountfnfapu;esn u?ntasenbpzues wzenfe sintfny rdesn; bentenia
Attempt 1, Iteration 300: Decrypted: orntfnfapr;esn rcntasenwpzres bzenfe sintfny udesn; wentenia
Attempt 1, Iteration 400: Decrypted: oritlilaprvesi rcitaseiwpgres bgeile snitliy mdesiv weiteina
Attempt 1, Iteration 500: Decrypted: or tl laprdes irc tase wpgresibge leisn tl yimhes diwe te na
Attempt 1, Iteration 600: Decrypted: or tl laprdes irn tase wpgresibge leisc tl yimhes diwe te ca
Attempt 1, Iteration 700: Decrypted: or tl laprnem ird tame vpfremibfe leimc tl yishem nive te ca
Attempt 1, Iteration 800: Decrypted: or ml laprnet ird mate vpfretibfe leitw ml yishet nive me wa
Attempt 1, Iteration 900: Decrypted: or mf faprnet ird mate cplretible feitw mf yishet nice me wa
Attempt 1, Iteration 1000: Decrypted: or mf faprnet ird mate cplretible feitw mf yishet nice me wa
Attempt 1, Iteration 1100: Decrypted: or mf faprnet ird mate cplretible feitw mf yishet nice me wa
Attempt 1, Iteration 1200: Decrypted: or mf fapruet ird mate cplretible feitw mf yishet uice me wa
Attempt 1, Iteration 1300: Decrypted: or mf fapruet ird mate cplretible feitw mf yishet uice me wa
Attempt 1, Iteration 1400: Decrypted: or wf faprues ird wase cplresible feism wf yithes uice we ma
Attempt 1, Iteration 1500: Decrypted: or wy yaprues ird wase cplresible yeisn wy fithes uice we na
Attempt 1, Iteration 1600: Decrypted: ar gy yoprues ird gose cplresible yeisn gy fithes uice ge no

Attempt 1, Iteration 1700: Decrypted: ar gy yoprues ird gose cplresible yeisn gy kithes uice ge no
Attempt 1, Iteration 1800: Decrypted: ar gy yourfes ird gose culresible yeisn gy kithes fice ge no
Attempt 1, Iteration 1900: Decrypted: ar py yourfes ird pose culresible yeisn py kithes fice pe no
Attempt 1, Iteration 2000: Decrypted: ar py yourfes ird pose culresible yeisn py kithes fice pe no
Attempt 1, Iteration 2100: Decrypted: ar py yourfes ird pose culresible yeisn py kithes fice pe no
Attempt 1, Iteration 2200: Decrypted: ar py yourkes ird pose culresible yeisn py fithes kice pe no
Attempt 1, Iteration 2300: Decrypted: ar py yourges ird pose culresible yeisn py fithes gice pe no
Attempt 1, Iteration 2400: Decrypted: ar py yourges ird pose mulresible yeisn py fithes gime pe no
Attempt 1, Iteration 2500: Decrypted: ar py yourges ird pose mulresible yeisn py fithes gime pe no
Attempt 1, Iteration 2600: Decrypted: ar py yourges ird pose mulresible yeisn py fithes gime pe no
Attempt 1, Iteration 2700: Decrypted: ar py yourges ird pose mulresible yeisn py fithes gime pe no
Attempt 1, Iteration 2800: Decrypted: ar py yourgen ird pone mulrenible yeins py fithen gime pe so
Attempt 1, Iteration 2900: Decrypted: ar py yourgen ird pone mulrenible yeins py fithen gime pe so
Attempt 1, Iteration 3000: Decrypted: ar py yourgen ird pone mulrenible yeins py fithen gime pe so
Attempt 1, Iteration 3100: Decrypted: ar py yourgen ird pone mulrenible yeins py fithen gime pe so
Attempt 1, Iteration 3200: Decrypted: ir py yourgen ard pone mulrenable yeans py fathen game pe so
Attempt 1, Iteration 3300: Decrypted: ir py yourgen ard pone mulrenable yeans py fathen game pe so
Attempt 1, Iteration 3400: Decrypted: ir py yourgen ard pone mulrenable yeans py fathen game pe so
Attempt 1, Iteration 3500: Decrypted: ir py yourgen ard pone mulrenable yeans py fathen game pe so
Attempt 1, Iteration 3600: Decrypted: in my younger and more pulnerable years my father gape me so
Attempt 1, Iteration 3700: Decrypted: in my younger and more vulnerable years my father gave me so
Attempt 1, Iteration 3800: Decrypted: in my younger and more vulnerable years my father gave me so
Attempt 1, Iteration 3900: Decrypted: in my younger and more vulnerable years my father gave me so
Attempt 1, Iteration 4000: Decrypted: in my younger and more vulnerable years my father gave me so
Attempt 1, Iteration 4100: Decrypted: in my younger and more vulnerable years my father gave me so
Attempt 1, Iteration 4200: Decrypted: in my younger and more vulnerable years my father gave me so
Attempt 1, Iteration 4300: Decrypted: in my younger and more vulnerable years my father gave me so
Attempt 1, Iteration 4400: Decrypted: in my younger and more vulnerable years my father gave me so
Attempt 1, Iteration 4500: Decrypted: in my younger and more vulnerable years my father gave me so
Attempt 1, Iteration 4600: Decrypted: in my younger and more vulnerable years my father gave me so
Attempt 1, Iteration 4700: Decrypted: in my younger and more vulnerable years my father gave me so
Attempt 1, Iteration 4800: Decrypted: in my younger and more vulnerable years my father gave me so
Attempt 1, Iteration 4900: Decrypted: in my younger and more vulnerable years my father gave me so
Attempt 1, Iteration 5000: Decrypted: in my younger and more vulnerable years my father gave me so
Attempt 1, Iteration 5100: Decrypted: in my younger and more vulnerable years my father gave me so
Attempt 1, Iteration 5200: Decrypted: in my younger and more vulnerable years my father gave me so
Attempt 1, Iteration 5300: Decrypted: in my younger and more vulnerable years my father gave me so
Attempt 1, Iteration 5400: Decrypted: in my younger and more vulnerable years my father gave me so
Attempt 1, Iteration 5500: Decrypted: in my younger and more vulnerable years my father gave me so
Attempt 1, Iteration 5600: Decrypted: in my younger and more vulnerable years my father gave me so
Attempt 1, Iteration 5700: Decrypted: in my younger and more vulnerable years my father gave me so
Attempt 1, Iteration 5800: Decrypted: in my younger and more vulnerable years my father gave me so
Attempt 1, Iteration 5900: Decrypted: in my younger and more vulnerable years my father gave me so
Attempt 1, Iteration 6000: Decrypted: in my younger and more vulnerable years my father gave me so
```

Figure 11: 6000 iterations for decrypting text ending with Metropolis Hastings algorithm; **Best Decryption:** in my younger and more vulnerable years my father gave me so

e) Ergodicity

$\psi(\alpha, \beta)$ does affect the ergodicity of the chain. When some transition probabilities are 0, this indicates that some states cannot be reached from others. This does not necessarily violate ergodicity, as long as there is an indirect path between every pair of states and the chain can still visit all states with non-zero probability in the long run. The proof for ergodicity, in the context of this question, involves several steps. Given the structure of our Markov chain and transition probabilities, we need to prove that the Markov chain is both irreducible and aperiodic.

Irreducibility Proof

- Let Σ be finite set of all symbols
- Let S be the state space of Markov chain, where each state $s \in S$ corresponds to permutation of symbols in Σ
- Let T be transition matrix obtained by MLE, where T_{ij} is probability of transition from state $i \to j$

We must show for every pair of states $\alpha, \beta \in S$, there exists a non-zero probability path from α to β. This can be a sequence of intermediate states.
Given any two permutations α and β, through series of swaps, each is a valid transition in the Markov chain.
:

$$\Sigma = |n|$$

where n is a finite number of symbols, the number of required swaps is finite. Such as in War and Peace text, transition probabilities T_{ij} are derived from observed frequencies of symbol pairs with smoothing terms added to ensure all $T_{ij} \geq 0$. Through MH algorithm, the probability of proposing any swap is positive. We construct a sequence of states from α to β:

$$\alpha_1, \alpha_2, \alpha_3, ..., \alpha_k$$

Thus, probability of any permutation through a finite number of swaps is non-zero.

$$P(\alpha \to \beta) = P(\alpha \to \alpha_1)P(\alpha_1 \to \alpha_2).....P(\alpha_{k-1} \to \beta) \geq 0$$

Thus, Markov chain is irreducible.

Aperiodicity Proof

A state $\alpha \in S$ is aperiodic if greatest common divisor GCD of all possible return times to α is 1. To prove aperiodicity, we need to show that for any state α, there is a positive probability of returning to α, there is a positive probability of returning to α at times that are not multiples for some $k > 1$.
Define $d(\alpha)$ as period of state α which is the greatest common dividor of all n such that $T^n_{\alpha\alpha} > 0$. In MH algorithm, as proposal is symmetric, to propose a swap back to original α, it is possible to have:

$$T_{\alpha\alpha} > 0$$

Hence, every state α is aperiodic. Hence:

- Irreducibility: $\forall \alpha, \beta \in S, \exists P : P(\alpha \to \beta) > 0$
- Aperiodicity: $\forall \ alpha \in S, \gcd\{n > 0 | T^n_{\alpha\alpha} > 0\} = 1$

f) Different Approaches to decoding

Symbol Probabilities alone:

This is not sufficient. It does not account for context or order of symbols. In English, pairs of letters occur with high frequency which this model cannot exploit.

2nd Order Markov Chain:

In the 1st order Markov chain, we have a transition matrix P where P_{ij} is probability of transitioning from $i \to j$:

$$P = [P_{ij}]; i, j \in \{1, 2, ...n\}$$

Matrix dimensions for n states are $n \times n$. For a 2nd order markov chain:

$$P = [P_{ijk}]; i, j, k \in \{1, 2, ...n\}$$

Transition probabilities for 2nd order:

$$P(S_t | S_{t-1} = s_j, S_{t-2} = s_i) = P_{ijk}$$

Therefore:

$$Number of parameters = n^3$$

This becomes a tensor representation:

$$P \in \mathcal{R}^{n \times n \times n}$$

For 2nd order MC, this does work, but this complexity can be prohibitive for large symbol sets or when computational resources are limited.

Encryption Scheme with Non-unique Mappings:

Two symbols mapped to same encrypted value, this does not work. Let's denote:

- Σ as the set of all plaintext symbols.
- E as the set of all encrypted symbols.
- $\sigma : \Sigma \to E$ as the encryption function.
- $\sigma^{-1} : E \to 2^{\Sigma}$ as the decryption function, where 2^{Σ} denotes the power set of Σ due to non-unique mappings.

For non-unique mappings:

$$\exists s_1, s_2 \in \Sigma, s_1 \neq s_2 : \sigma(s_1) = \sigma(s_2) = e$$

For decryption with ambiguity:

$$\sigma^{-1}(e) = \{s_1, s_2\}$$

For likelihood of a decryption:

$$P(S = s \mid E = e)$$

If σ^{-1} is not a function but a relation due to non-unique mappings, the likelihood must consider all mappings:

$$P(S = s_1 \mid E = e) = P(S = s_2 \mid E = e)$$

This results in a set of possible plaintexts for each encrypted symbol:

$$\forall e \in E, \sigma^{-1}(e) \text{ is not a singleton}$$

The decryption process must then consider all combinations:

$$P(S_1 S_2 \dots S_n \mid E_1 E_2 \dots E_n) = \sum_{(s'_1, s'_2, \dots, s'_n) \in \sigma^{-1}(E_1) \times \sigma^{-1}(E_2) \times \dots \times \sigma^{-1}(E_n)} P(S_1 = s'_1, S_2 = s'_2, \dots, S_n = s'_n)$$

This sum will run over all combinations of possible symbols to each encrypted symbol, leading to endless combinations as number of symbols increases.

Chinese Language:

This would not work with Chinese. There are too many symbols where > 10000 and computational power is limited. Let Σ represent the set of Chinese characters with cardinality $|\Sigma| = N$.

State space size for substitution cipher:

$$|S| = N!$$

Mapping space:

$$|M| = (N-1)^N$$

First-order transition matrix size:

$$|T| = N^2$$

Second-order transition tensor size:

$$|T_2| = N^3$$

Iterations for convergence (minimum):

$$\text{Iter}_{\text{min}} \propto |S| = N!$$

Corpus size for accurate transition probabilities estimation:

$$\text{Corpus}_{\text{size}} \propto N^2 \text{ (first-order)}$$

$$\text{Corpus}_{\text{size}} \propto N^3 \text{ (second-order)}$$

Computation time - 1 iteration:

$$O(N^2) \text{ (matrix operations)}$$

Total computation time:

$$O(N^2 \cdot N!)$$

Given factorial growth of state space, and cubic growth of transition tensor, computational resources are insufficient for this large N.

Problem 6: Implementing Gibbs Sampling for LDA

a) ToyExample.data

The code for the graphs below are as attached in Appendix: Question 6. In the python code file gibbs_sampler.py, there were 6 todos that had to be implemented in order to achieve the graphs in part 6a). The functions that needed to be implemented are as documented below.

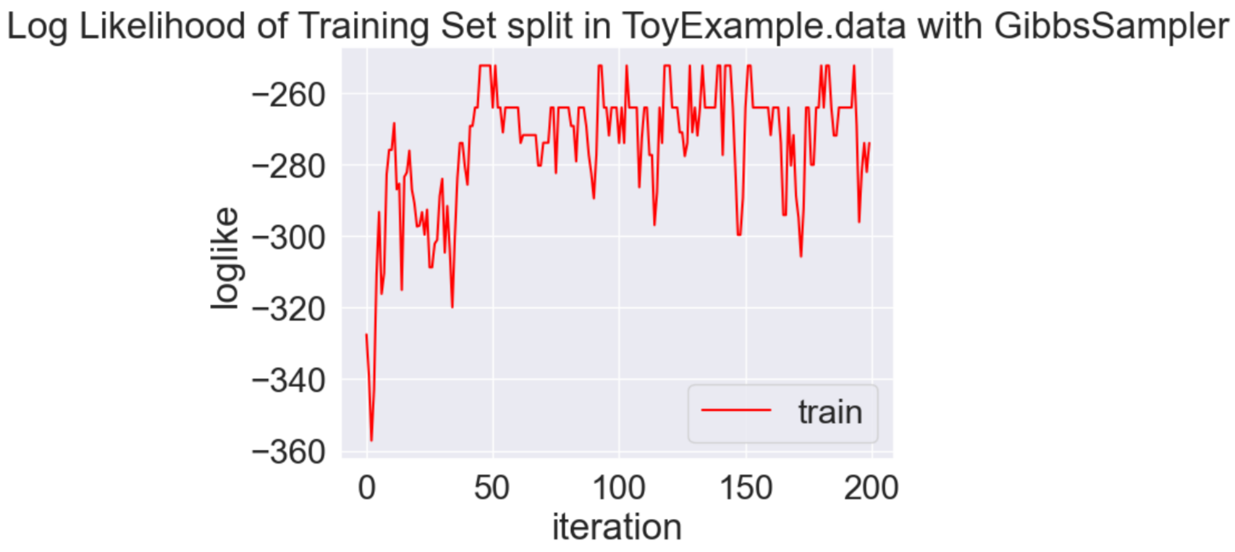


Figure 12: Standard Gibbs Sampler implemented on toyexample.data training set

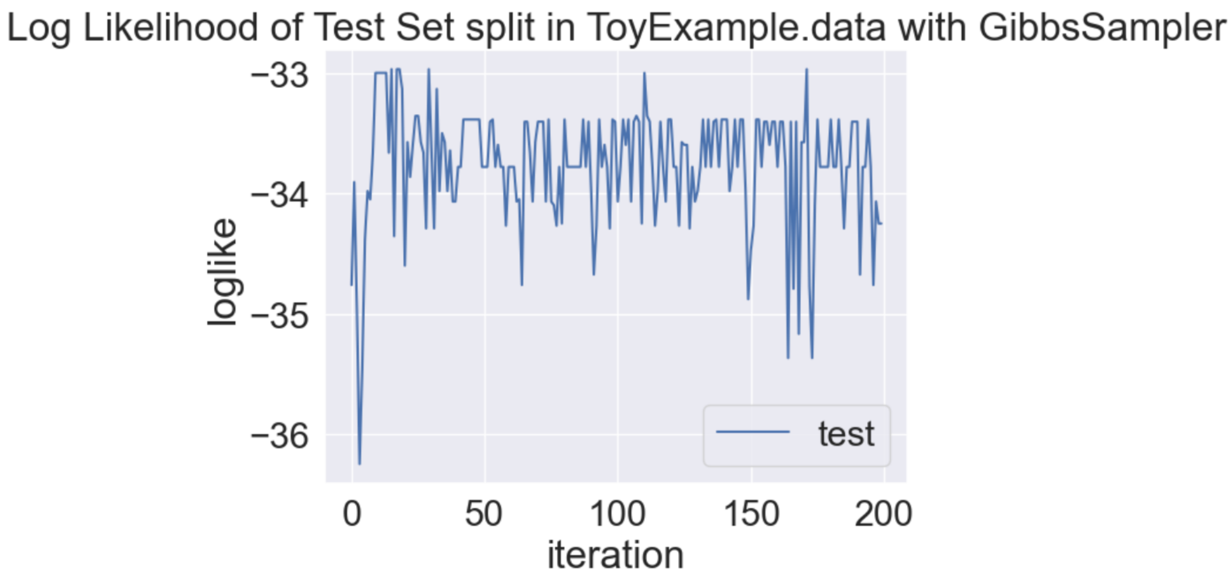


Figure 13: Standard Gibbs Sampler implemented on toyexample.data test set

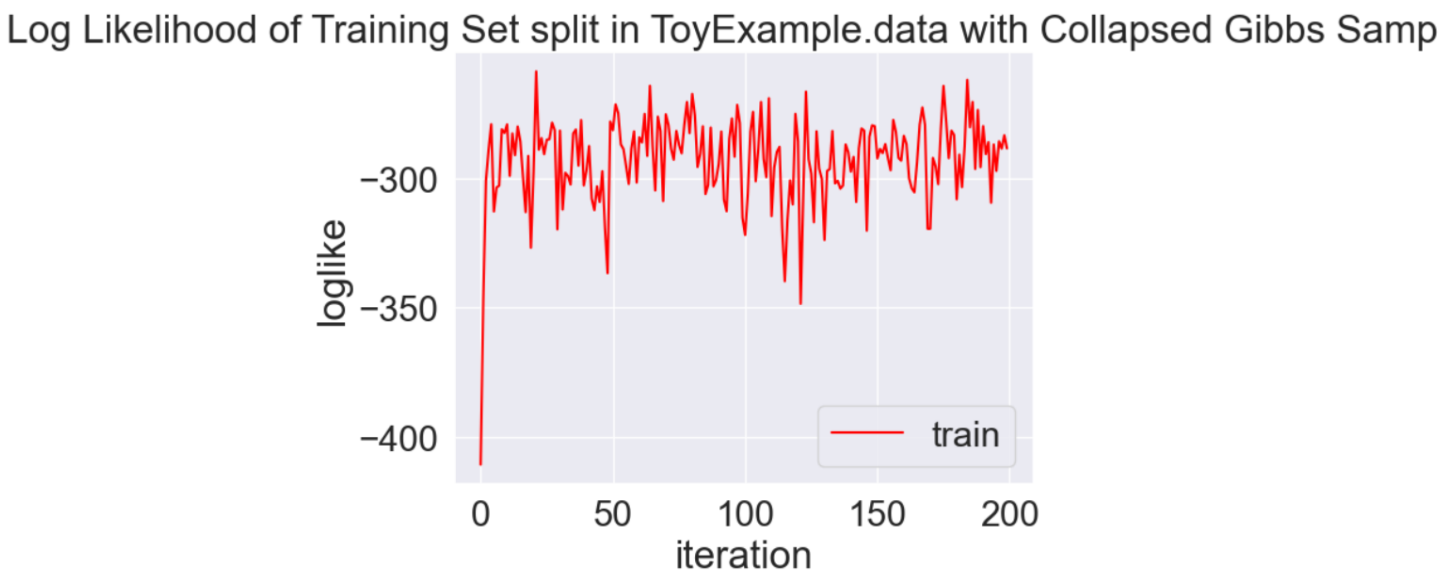


Figure 14: Collapsed Gibbs Sampler implemented on toyexample.data training set

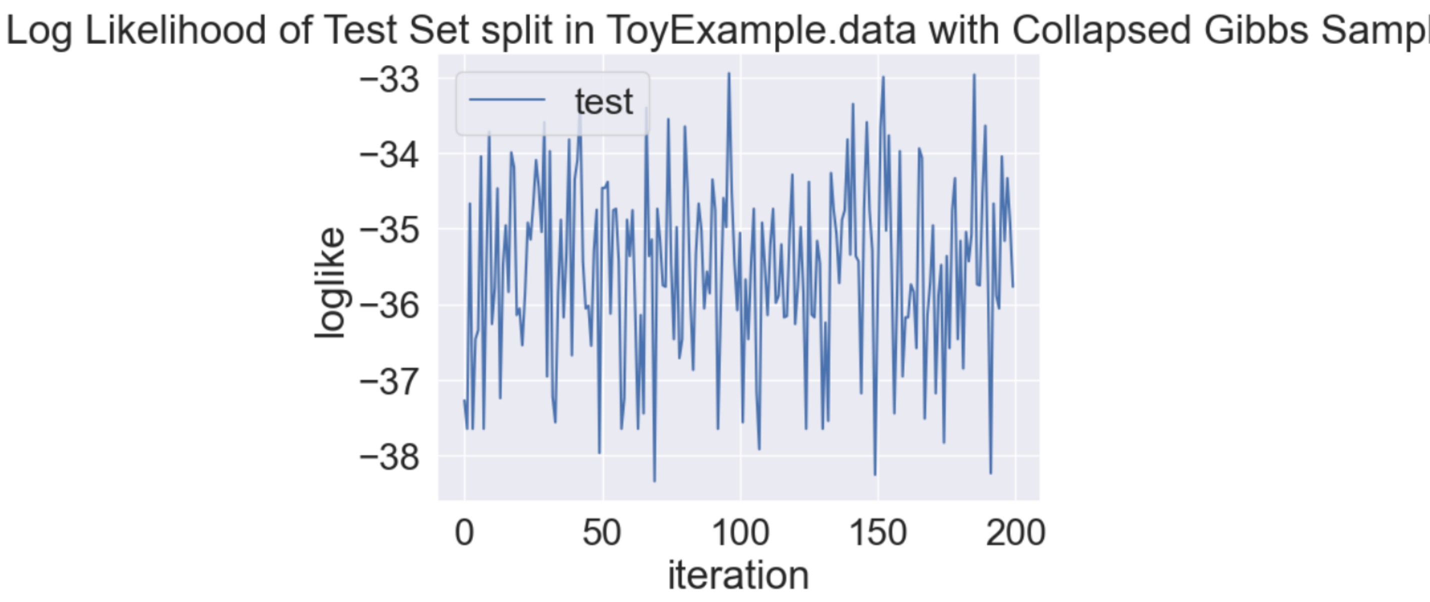
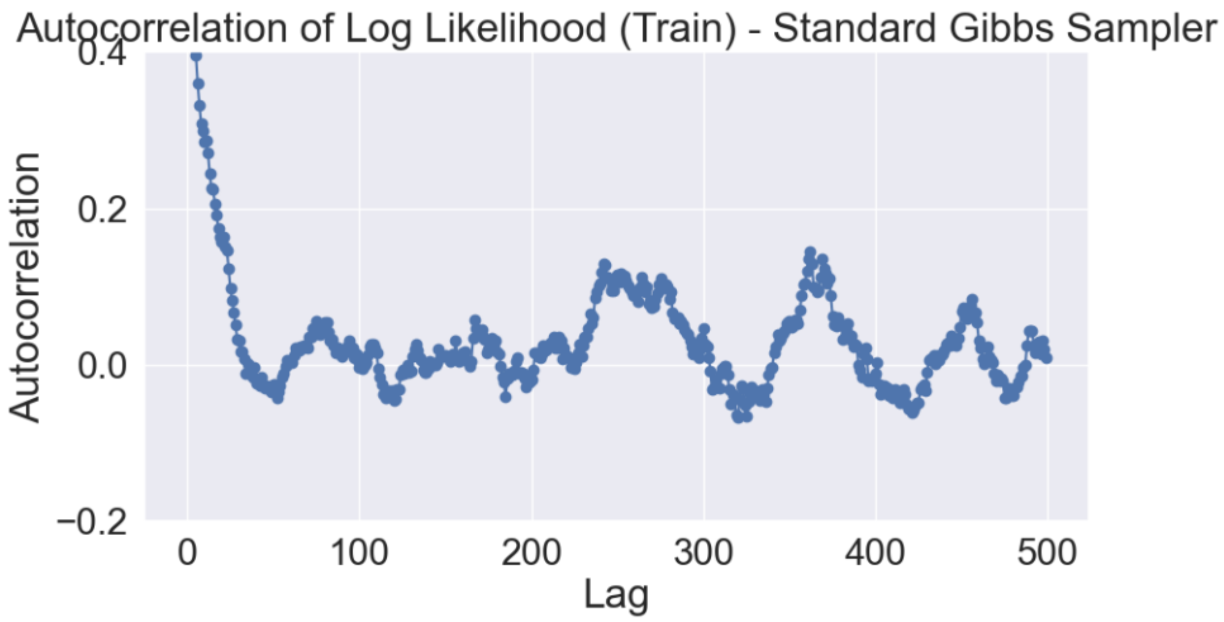


Figure 15: Collapsed Gibbs Sampler implemented on toyexample.data test set

From Figure 11 and 12 above, roughly around 20-30 iterations, the log likelihood of the training and test data with standard Gibbs sampling reaches around -270 and -33 respectively before they fluctuate repeated for 180 iterations. In Figure 13, with Collapsed Gibbs Sampling on the training data, it increased rapidly until around 15-20 iterations, and fluctuates around that value at -300 approximately for 180 iterations. In Figure 14, it was fluctuating repeatedly for the 200 iterations.

b) Autocorrelation



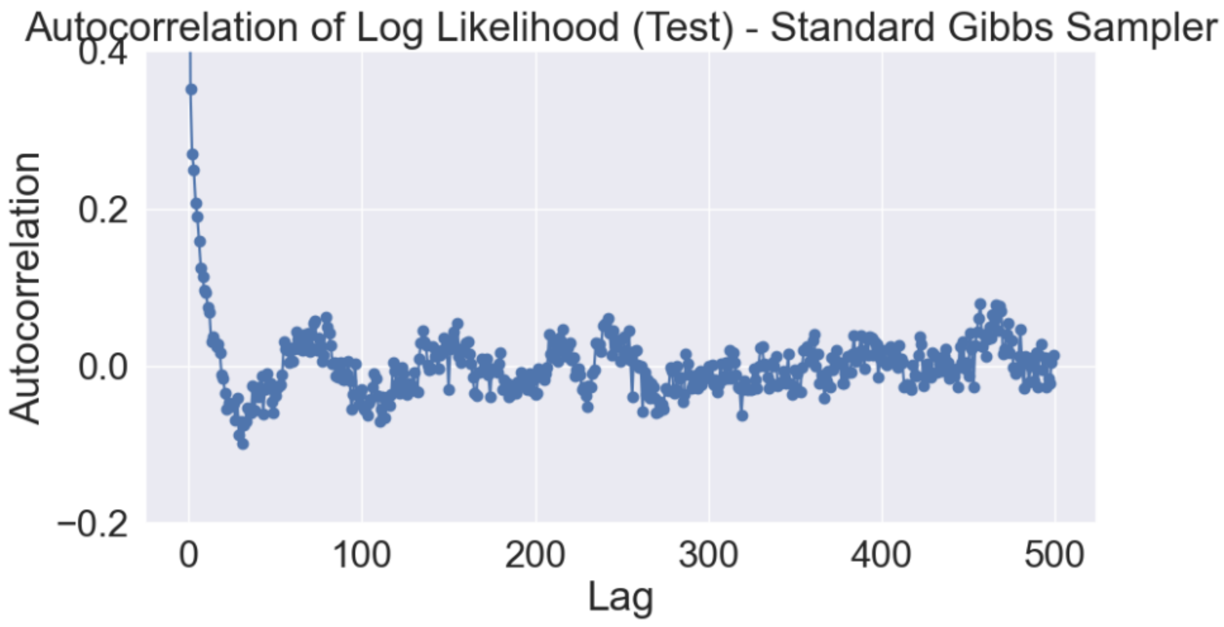
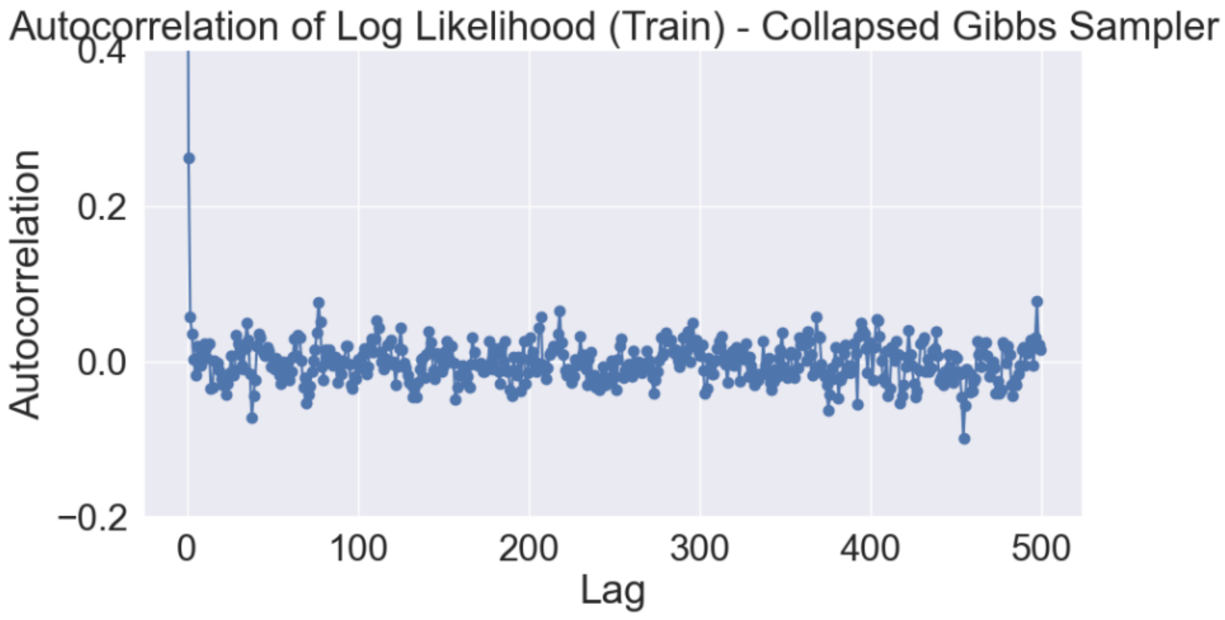


Figure 16



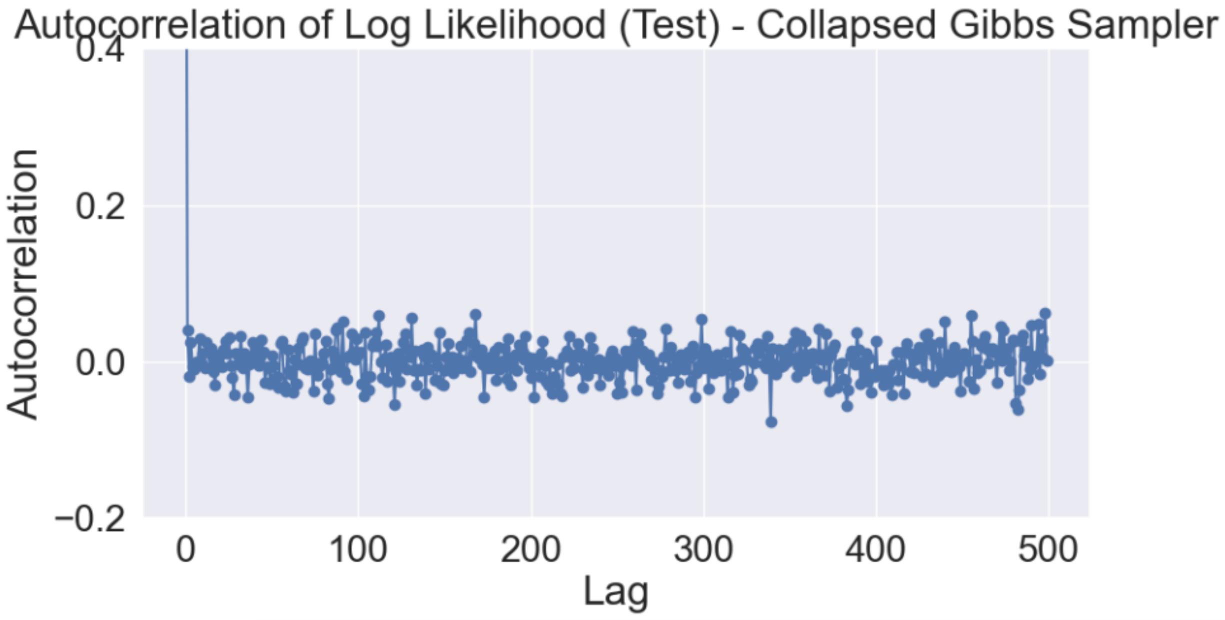


Figure 17

In this question, I increased the number of iterations for Standard and Collapsed Gibbs Sampling to 2000 iterations to reduce noise in the autocorrelation plots as shown below. Intuitively, we discarded 30 burnin iterations and plotted the autocorrelations as shown above. From the autocorrelation plots, the samples are less correlated and have reached point of fluctuations where the Markov chain provides representative set of samples from posterior distribution. From Figure 17 for Collapsed Gibbs Sampler, autocorrelation drops sharply and fluctuates around 0 quickly suggesting burnin period is short. For Figure 16, in Standard Gibbs Sampler, burnin period is alot longer than Collapsed. It takes an estimate of more than 10 iterations to start fluctuating around zero. About 20-40 samples will be required for the Gibbs sampler until we have a representative set of samples from the posterior. This is because autocorrelation drops below 0.1 within the first 50 lags of both Gibbs samplers and for the Collapsed Gibbs Sampler, this is even lower from the start, which means that it requires a shorter burn-in. However, to further improve this to obtain a representative set from the posterior, I could have sampled every 10 iterations to reduce autocorrelation further.

c) Gibbs Sampler Convergence

The Collapsed Gibbs Sampler converges faster. Comparing Figure 16 and 17, Collapsed Gibbs Sampler's training and test autocorrelation plots show very low initial values that stabilize and fluctuate around 0, indicating a faster convergence as compared to the Standard Gibbs Sampler. This could also be because Collapsed Gibbs Sampler integrates out one of more variables (topic assignments), leading to reduced state space. The resulting Markov chain demonstrates higher convergence at early lags, suggesting each iteration provides a more independent sample from posterior distribution.

d) Varying parameters

Varying α

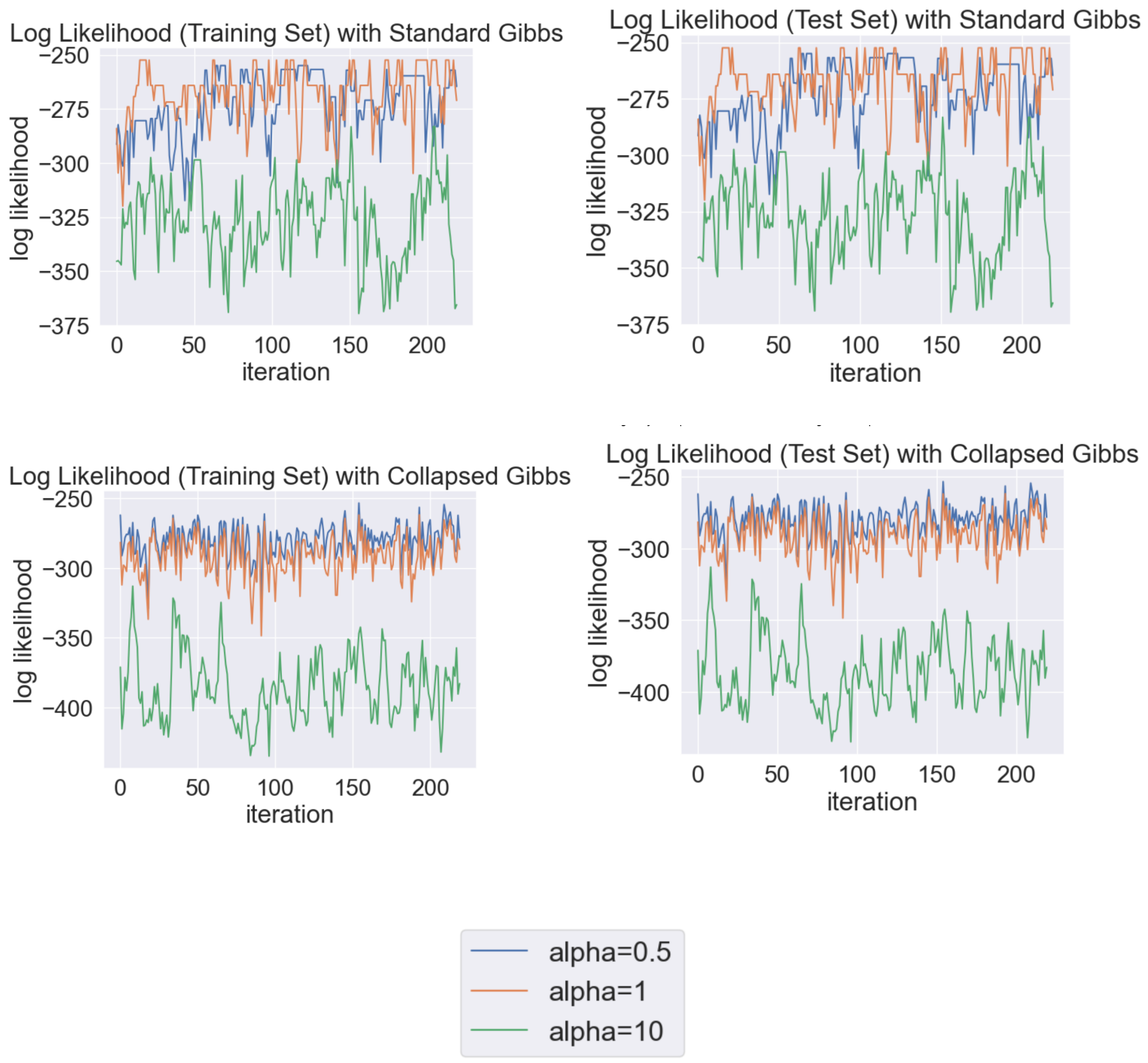


Figure 19: Log-likelihoods of training and test data with standard Gibbs Sampler and Collapsed Gibbs Sampler, with varying alpha values

For test set, from the above figures, show mixed results. Standard Gibbs Sampler shows better performance with $\alpha = 1$ while Collapsed Gibbs has a less clear pattern. The model $\alpha = 1$ seems to generalize better in the standard Gibbs. Noting that higher log likelihood on test set indicates better generalization and predictive performance, means that our values at lower alpha achieve better performance. A higher alpha suggest that each document is likely to contain a mixture of most topics. A lower alpha value indicates a document is more likely to be represented by fewer topics, which results in the fluctuations in log likelihood.

Varying β

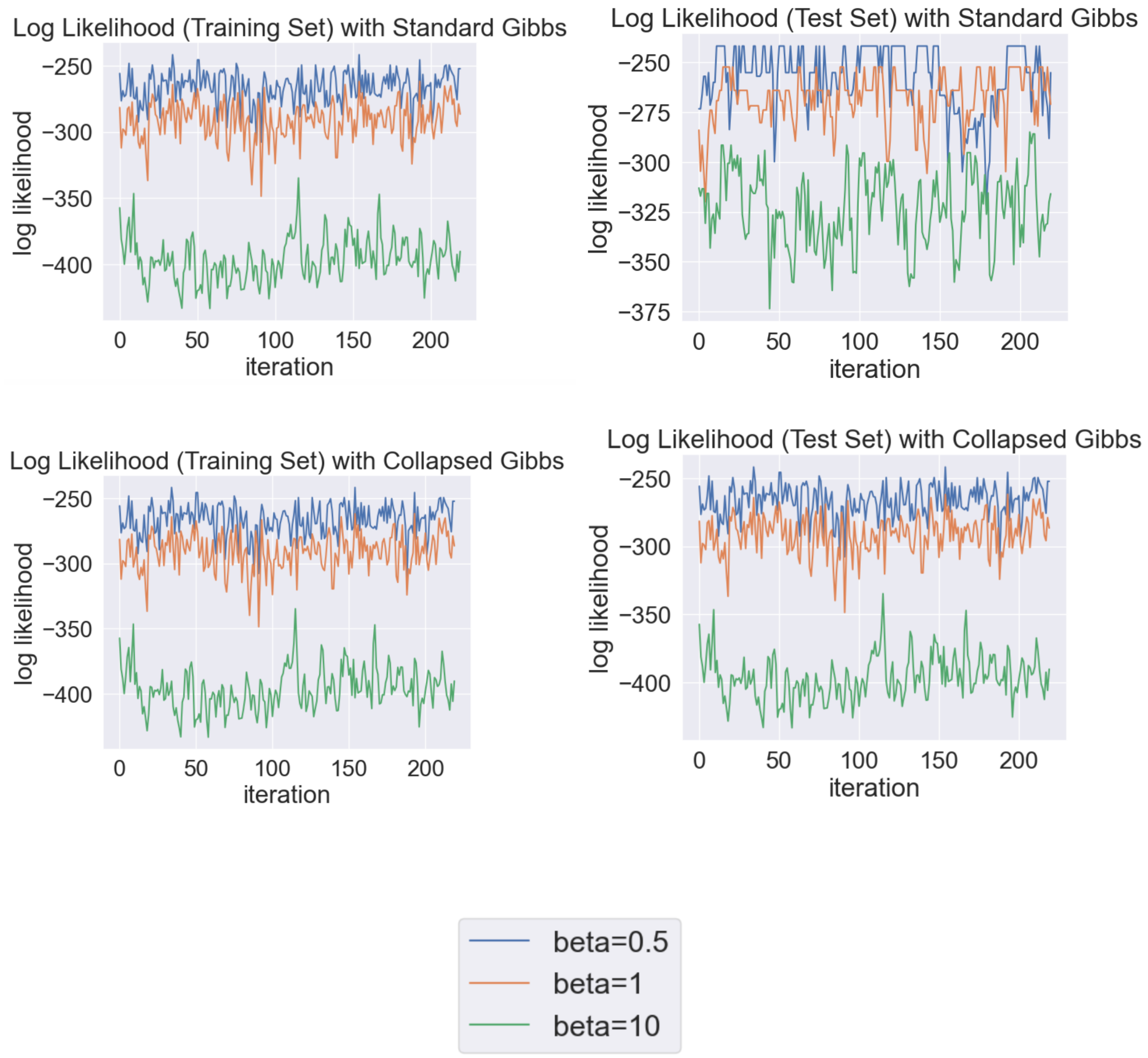


Figure 21: Log-likelihoods of training and test data with standard Gibbs Sampler and Collapsed Gibbs Sampler, with varying beta values

Similarly to α, β behaves the same way on the posterior probability and predictive performance. The beta parameter influences the word distributions within topics. A lower beta promotes sparser word distributions, implying that a few words are strongly associated with each topic. Lower beta results in more fluctuations in log likelihood, indicating model is sensitive to changes in topic-word assembly. The plots suggest that beta=1 might offer a reasonable trade-off between these extremes, as indicated by relatively higher log likelihoods for both training and test sets with the Standard Gibbs Sampler.

Varying K/Number of Topics

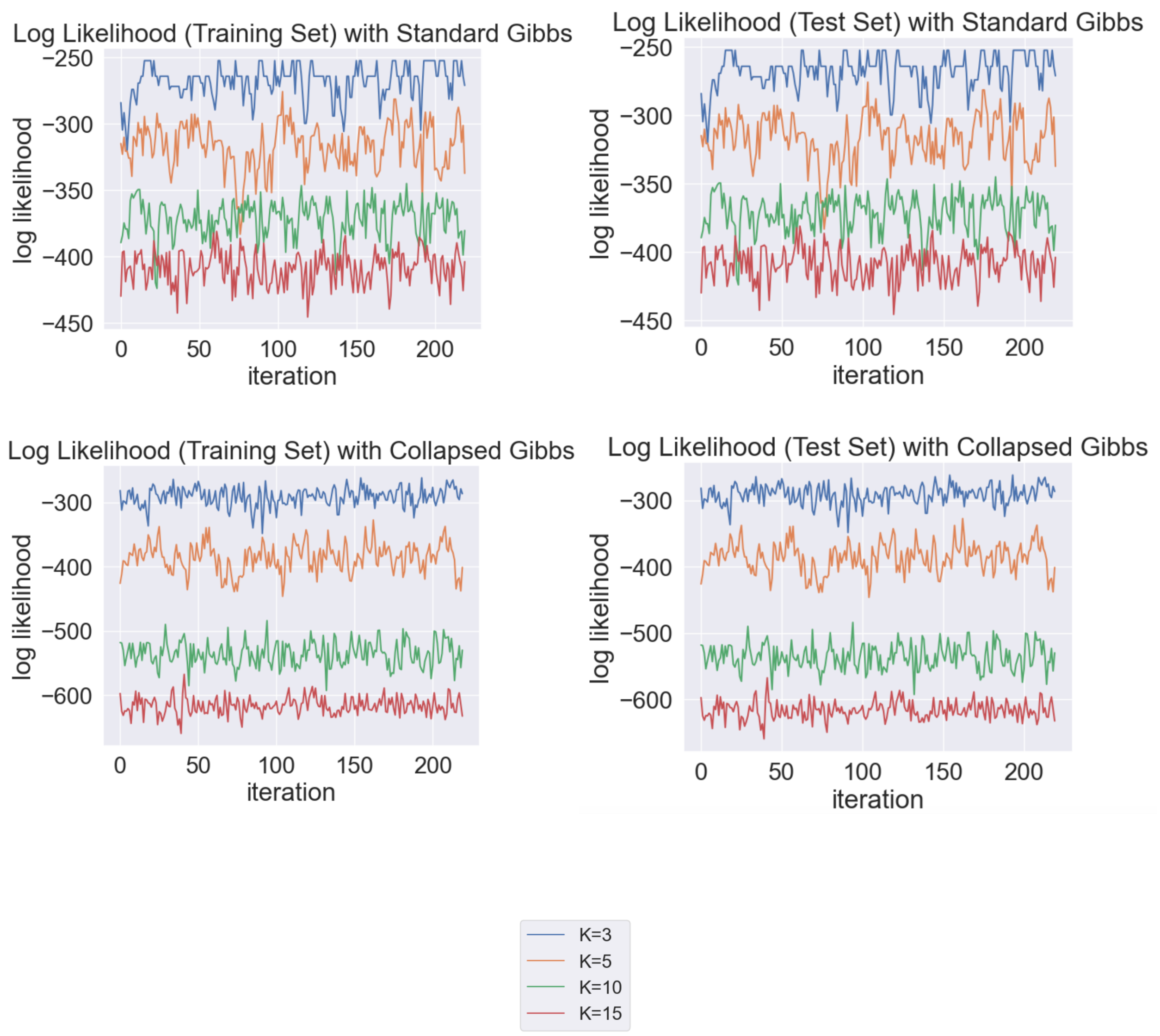


Figure 23: Log-likelihoods of training and test data with standard Gibbs Sampler and Collapsed Gibbs Sampler, with varying K (n_topics)values

In this figure above, where for higher values of K, the model captures more restrictive patterns in the data, which is indicated in the lower log-likelihood values for higher K on training set. In training set, Standard Gibbs and Collapsed Gibbs samplers show higher log likelihood for lower K values. This shows maybe a small K is sufficient to capture the themes of training data. This is the same for the test set. Optimal number of topics is usually a balance with generalization and not overfitting. Plots suggest that for this dataset, lower K values are better in predictive performance.

Problem 7: Optimization

a) Lagrange Multiplier Method & Local Extrema

In order to find the local extrema of the function $f(x, y) = x + 2y$ subject to constraints $y^2 + xy = 1$. **Lagrange Multiplier Method:** We are seeking for points where the gradient of $f(x, y)$ is parallel to gradient of constraint $g(x, y) = y^2 + xy = 1$. We define the Lagrangian as:

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

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

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

Gradient of $\mathcal{L}$ with respect to x, y and λ is denoted as:

$$\nabla\mathcal{L}(x, y, \lambda) = 0$$

This gives us a system of equations of partial derivatives:

$$\frac{\delta\mathcal{L}}{\delta x} = 1 - \lambda y \tag{15}$$

$$\frac{\delta\mathcal{L}}{\delta y} = 2 - \lambda(2y + x) \tag{16}$$

$$\frac{\delta\mathcal{L}}{\delta\lambda} = y^2 + xy - 1 \tag{17}$$

From Equation 15 and equating partial derivatives to 0:

$$\lambda y = 1$$

$$\lambda = \frac{1}{y} assuming y \neq 0$$

Substituting this λ into Equation 16:

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

$$\frac{x}{y} = 0$$

Given that $y \neq 0$, this implies that x=0. Substituting x=0 into constraint equation 17:

$$y^2 + (0)y - 1 = 0$$

$$y^2 - 1 = 0$$

$$y = \pm 1$$

Thus, function values local extrema occur are $(0, 1)$ and $(0, -1)$. To determine whether the points are maxima, minima or saddle points, we use the bordered Hessian matrix for the constrained optimization problem. Given the Lagrangian:

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

Calculating the partial derivatives with respect to x, y and λ:

$$\frac{\delta^2\mathcal{L}}{\delta x^2} = 0$$

$$\frac{\delta^2\mathcal{L}}{\delta y^2} = -2\lambda$$

$$\frac{\delta^2 \mathcal{L}}{\delta x \delta y} = \frac{\delta^2 \mathcal{L}}{\delta y \delta x} = -\lambda$$

The constraint function is $g(x, y) = y^2 + xy - 1$, so the partial derivatives are:

$$\frac{\delta g}{\delta x} = y$$

$$\frac{\delta g}{\delta y} = 2y + x$$

The bordered Hessian H for a problem with 2 variables and 1 constraint is a 3×3 matrix:

$$\begin{bmatrix} 0 & \frac{\partial g}{\partial x} & \frac{\partial g}{\partial y} \\ \frac{\partial g}{\partial x} & \frac{\partial^2 \mathcal{L}}{\partial x^2} & \frac{\partial^2 \mathcal{L}}{\partial x \partial y} \\ \frac{\partial g}{\partial y} & \frac{\partial^2 \mathcal{L}}{\partial y \partial x} & \frac{\partial^2 \mathcal{L}}{\partial y^2} \end{bmatrix}$$

We'll evaluate this matrix at the points $(0, -1)$ and $(0, 1)$ and check the determinant of the bordered Hessian:
At the point $(0, -1)$:

$$H = \begin{bmatrix} 0 & -1 & -2 \\ -1 & 0 & -\lambda \\ -2 & -\lambda & -2\lambda \end{bmatrix}$$

Remember, $\lambda = \frac{1}{y}$, so at $y = -1$, $\lambda = -1$. Substituting this in:

$$H = \begin{bmatrix} 0 & -1 & -2 \\ -1 & 0 & 1 \\ -2 & 1 & 2 \end{bmatrix}$$

At the point $(0, 1)$: Similarly, $\lambda = 1$ for $y = 1$, so:

$$H = \begin{bmatrix} 0 & 1 & 2 \\ 1 & 0 & -1 \\ 2 & -1 & -2 \end{bmatrix}$$

Computing the determinant of these matrices to determine the nature of points:
For $(0, 1)$:

$$\det(H) = 0(0 - 1(-1)) - (-1)(-1(2) - 1(-2)) + (-2)(-1(-2) - 1(0))$$

$$\det(H) = 0 - 2 + 4 = 2$$

For $(0, -1)$:

$$\det(H) = 0(0 - 1(-1)) - (1)(1(2) - (-1)(-2)) + (2)(1(-2) - (-1)(0))$$

$$\det(H) = 0 - 2 - 4 = -6$$

Based on this, the function values point (0,1) is a maximum and (0,-1) is a minimum. Plugging these values into $f(x, y)$:

$$f(0, -1) = -2$$

$$f(0, 1) = 2$$

Therefore, the local extrema occurs at 2, and -2.

b) Newton-Raphson Method

Using Newton-Raphson method to compute $ln(a)$ for a given $a \in \mathbb{R}^+$, we need to find the root of a function where $f(x,a)$ is equivalent to solving x in $e^x = a$. We know that $e^x = a$ should be satisfied for $x = ln(a)$. Therefore,

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

From the above function derived:

$$f'(x) = \frac{\delta}{\delta x}(e^x - a) = e^x$$

The Newton's Method Update Equation, iteratively finds better approximation to the roots of a real-valued function. The general form of Newton's method is:

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

Now substituting $f(x,a)$ and $f'(x)$ into Newton's method update formula:

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

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

Problem 8: Eigenvalues as solutions of optimization problems

a) $\underset{x\in\mathbb{R}^n}{sup}\, R_A(x)$

Given a symmetric $n \times n$ matrix A, we define $R_A(x)$:

$$q_A(x) := x^T Ax \tag{18}$$

$$R_A(x) := \frac{x^T Ax}{x^T x} = \frac{q_A(x)}{||x||^2} \tag{19}$$

To prove:

$$\exists x* \in \mathbb{R}^n$$

such that $R_A(x^*) = \underset{x\in\mathbb{R}^n}{sup}\, R_A(x)$ where x^* is the point where $R_A(x)$ attains its supremum in $\mathbb{R}^n$.
Consider the unit sphere, which is compact:

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

$$R_A(x^*) = \frac{q_A(x^*)}{||x^*||^2} = q_A(x*) = M$$

as $||x^*|| = 1$ For any $y \in \mathbb{R}^n$, $y \neq 0$, let $z = \frac{y}{||y||}$. Then $z \in S^{n-1}$ and:

$$R_a(y) = \frac{q_A(y)}{||y||^2} = \frac{q_A(z)}{||z||^2} = R_A(z) \leq M$$

Therefore, $R_A(x^*) \geq R_A(y)$ for all $y \in \mathbb{R}^n, y \neq 0$. x^* is point where $R_A(x)$ attains supremum. Now, to show that $q_A(x)$ is a continuous function at all points $x\ in \mathbb{R}^n$: Let $x, x_n \in \mathbb{R}^n$:

$$q_A(x) - q_A(x_n) = x^T Ax - x_n^T Ax_n.$$

Using linearity and symmetry of A, we expand:

$$q_A(x) - q_A(x_n) = (x - x_n)^T Ax + x_n^T A(x - x_n).$$

Applying the Cauchy-Schwarz inequality:

$$|q_A(x) - q_A(x_n)| \leq |(x - x_n)^T Ax| + |x_n^T A(x - x_n)|.$$

As norm is sub-multiplicative,

$$|q_A(x) - q_A(x_n)| \leq \|x - x_n\| \|Ax\| + \|x_n\| \|A(x - x_n)\|.$$

Let $M_1 = \|Ax\|$ and $M_2 = \|x_n\| \|A\|$:

$$|q_A(x) - q_A(x_n)| \leq \|x - x_n\| M_1 + \|x - x_n\| M_2.$$

$$|q_A(x) - q_A(x_n)| \leq (\|x - x_n\|)(M_1 + M_2).$$

Given $\epsilon > 0, \exists \delta > 0$ such that $\|x - x_n\| < \delta$ implies $|q_A(x) - q_A(x_n)| < \epsilon$. Since $q_A(x) - q_A(x_n)$ can be made arbitrarily small by taking x sufficiently close to x_n, $q_A(x)$ is continuous at x_0. Since x_n is arbitrary, $q_A(x)$ is continuous everywhere in $\mathbb{R}^n$.

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

Given the eigenvalues of A as $\lambda_1 \geq \lambda_2 \geq ... \geq \lambda_n$ and corresponding eigenvectors, $\{\mathcal{E}_1, ..., \mathcal{E}_n\}$ which forms an ONB, we need to prove that for any vector $x \in \mathbb{R}^n$, Rayleigh quotient $R_A(x) \leq \lambda_1$:
With x in terms of ONB of eigenvectors:

$$x = \Sigma_{i=1}^n (\mathcal{E}_i^T x)\mathcal{E}_i \tag{20}$$

Using Equation 18:

$$q_A(x) = \left(\Sigma_{i=1}^n (\mathcal{E}_i^T x)\mathcal{E}_i\right)^T A \left(\Sigma_{j=1}^n (\mathcal{E}_j^T x)\mathcal{E}_j\right)$$

Expanding using inner product linearity and eigenvalue equation:

$$A\mathcal{E}_j = \lambda_j \mathcal{E}_j$$

$$q_A(x) = \Sigma_{i=1}^n \Sigma_{j=1}^n (\mathcal{E}_i^T x)(\mathcal{E}_j^T x)\mathcal{E}_i^T \lambda_j \mathcal{E}_j$$

With ONB:

$$\mathcal{E}_i^T \lambda_j = \delta_{ij}$$

$$q_A(x) = \Sigma_{i=1}^n (\mathcal{E}_i^T x)^2 \lambda_i$$

Now, we compute norm squared x:

$$||x||^2 = \left(\Sigma_{i=1}^n (\mathcal{E}_i^T x)\mathcal{E}_i\right)^T \left(\Sigma_{j=1}^n (\mathcal{E}_j^T x)\mathcal{E}_j\right)$$

$$= (\Sigma_{i=1}^n (\mathcal{E}_i^T x)^2$$

Substituting this into the Rayleigh Quotient formula 19:

$$R_A(x) = \frac{\Sigma_{i=1}^n (\mathcal{E}_i^T x)^2 \lambda_i}{\Sigma_{i=1}^n (\mathcal{E}_i^T x)^2}$$

As λ_1 is the largest eigenvalue for all i, $\lambda_i \leq \lambda_1$:

$$R_A(x) \leq \frac{\Sigma_{i=1}^n (\mathcal{E}_i^T x)^2 \lambda_i}{\Sigma_{i=1}^n (\mathcal{E}_i^T x)^2} = \lambda_1$$

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

c) $R_A(x) < \lambda_1$

Given eigenvectors $\mathcal{E}_1, ..., \mathcal{E}_k$ corresponding to eigenvalue λ_1, we need to show that for any $x \in \mathbb{R}^n$ is not in the span of $\{\mathcal{E}_1, ..., \mathcal{E}_k\}$, Rayleigh quotient $R_A(x) < \lambda_1$.
Given Equation for x 20and Rayleigh quotient for a vector is $R_A(x)$ 19, for x not in span of $\{\mathcal{E}_1, ..., \mathcal{E}_k\}$, there exists some $\mathcal{E}_j$ with $j > k$ such that $(\mathcal{E}_j^T x) \neq 0$:

$$R_A(x) = \frac{\Sigma_{i=1}^n \lambda_i (\mathcal{E}_i^T x)^2}{(\mathcal{E}_i^T x)^2}$$

As λ_1 is largest eigenvalue for $i > k$, $\lambda_i < \lambda_1$:

$$R_A(x) = \frac{\Sigma_{i=1}^k \lambda_1 (\mathcal{E}_i^T x)^2 + \Sigma_{i=k+1}^n \lambda_i (\mathcal{E}_i^T x)^2}{(\Sigma_{i=1}^n (\mathcal{E}_i^T x)^2}$$

Focusing on the numerator of the equation:

$$\Sigma_{i=1}^k \lambda_1 (\mathcal{E}_i^T x)^2 = \Sigma_{i=k+1}^n \lambda_i (\mathcal{E}_i^T x)^2$$

Since for $\lambda_i < \lambda_1$, the difference is positive:

$$\Sigma_{k+1}^n (\lambda_1 - \lambda_i)(\mathcal{E}_i^T x)^2 > 0$$

With the denominator of $R_A(x)$ still the same, the Rayleigh quotient is less than 1 because numerator is less than the denominator due to contribution from $i > k$. Thus, for x not in span of $\mathcal{E}_1, ..., \mathcal{E}_k$, $R_A(x) < \lambda_1$.

Appendix

Images & Graphs

1) Learned Probability Vector Images at each run for each component k={2,3,4,7,10}

The code runs EM algorithm for the specified number of mixture components K and then visualizes each component as an 8×8 grayscale image after each run.

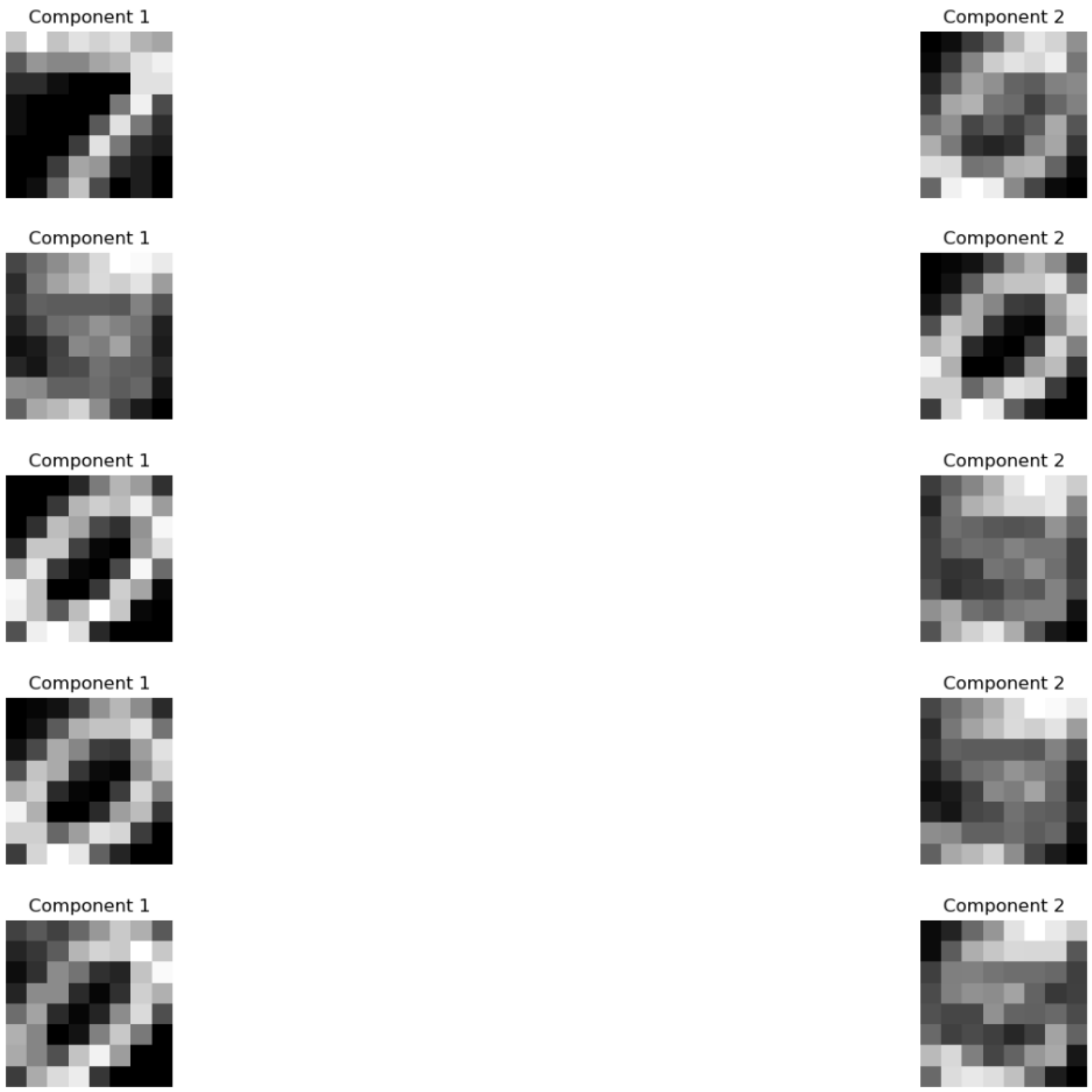


Figure 24: Learned probability vectors as images for $k = 2$, each run 5 times

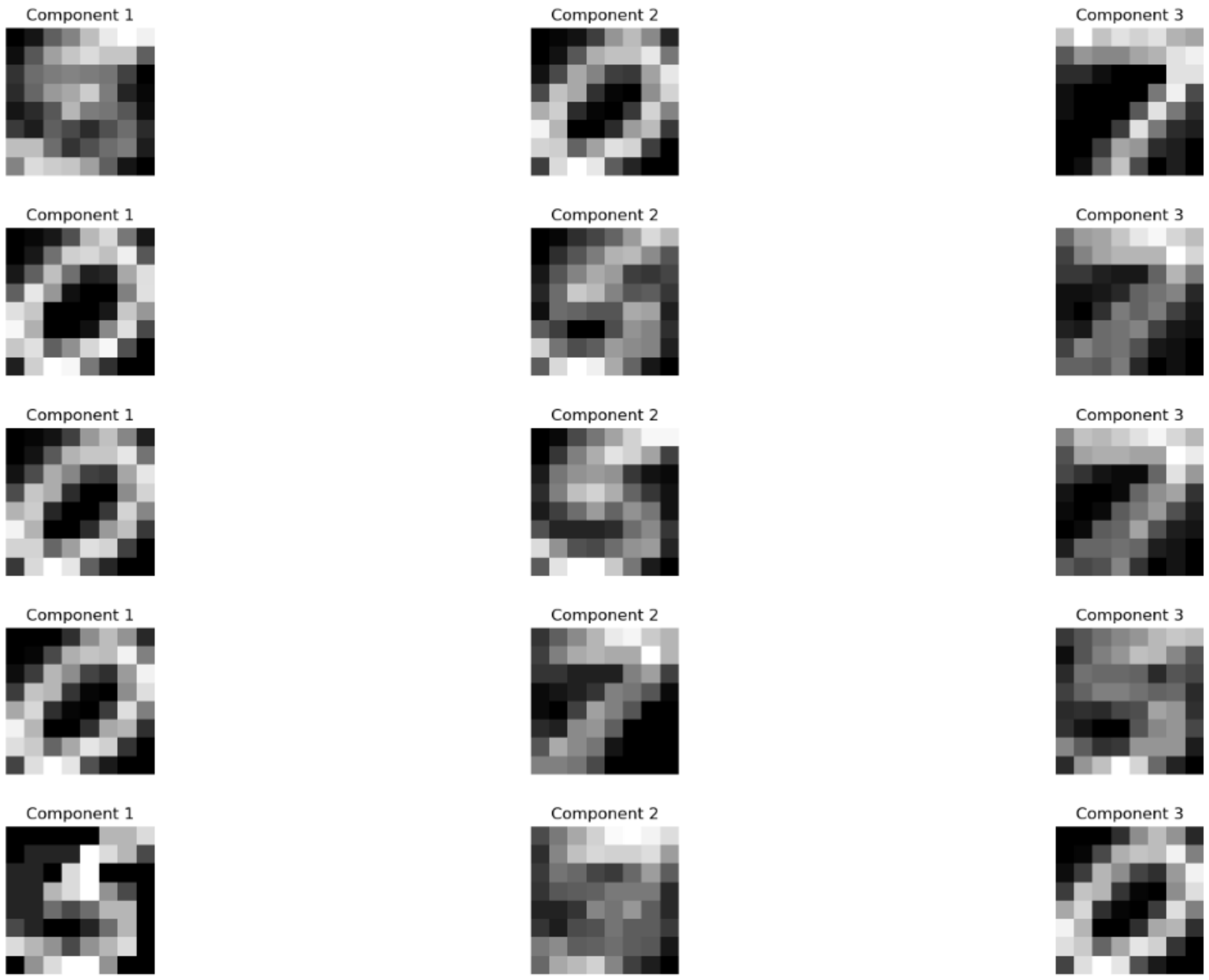


Figure 25: Learned probability vectors as images for $k = 3$, each run 5 times

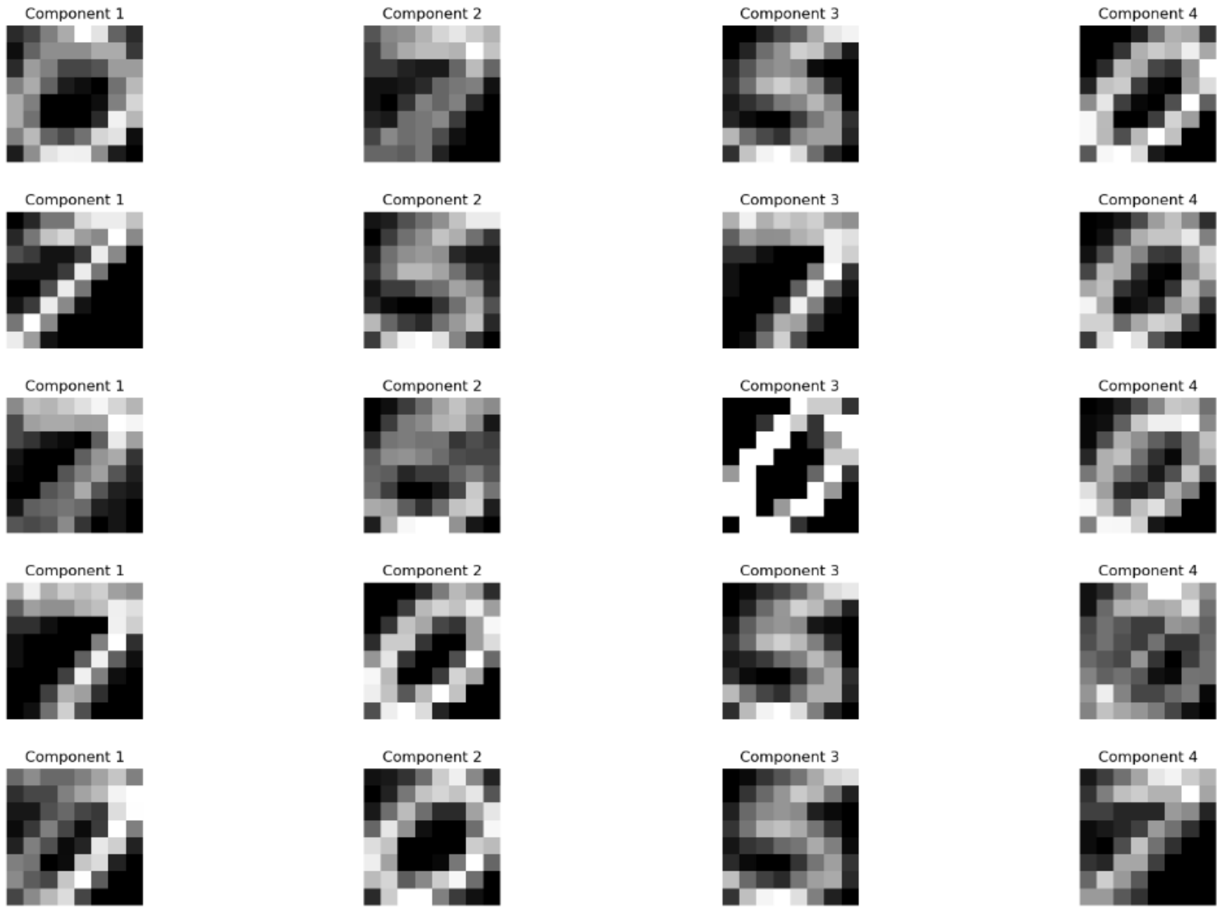


Figure 26: Learned probability vectors as images for $k = 4$, each run 5 times

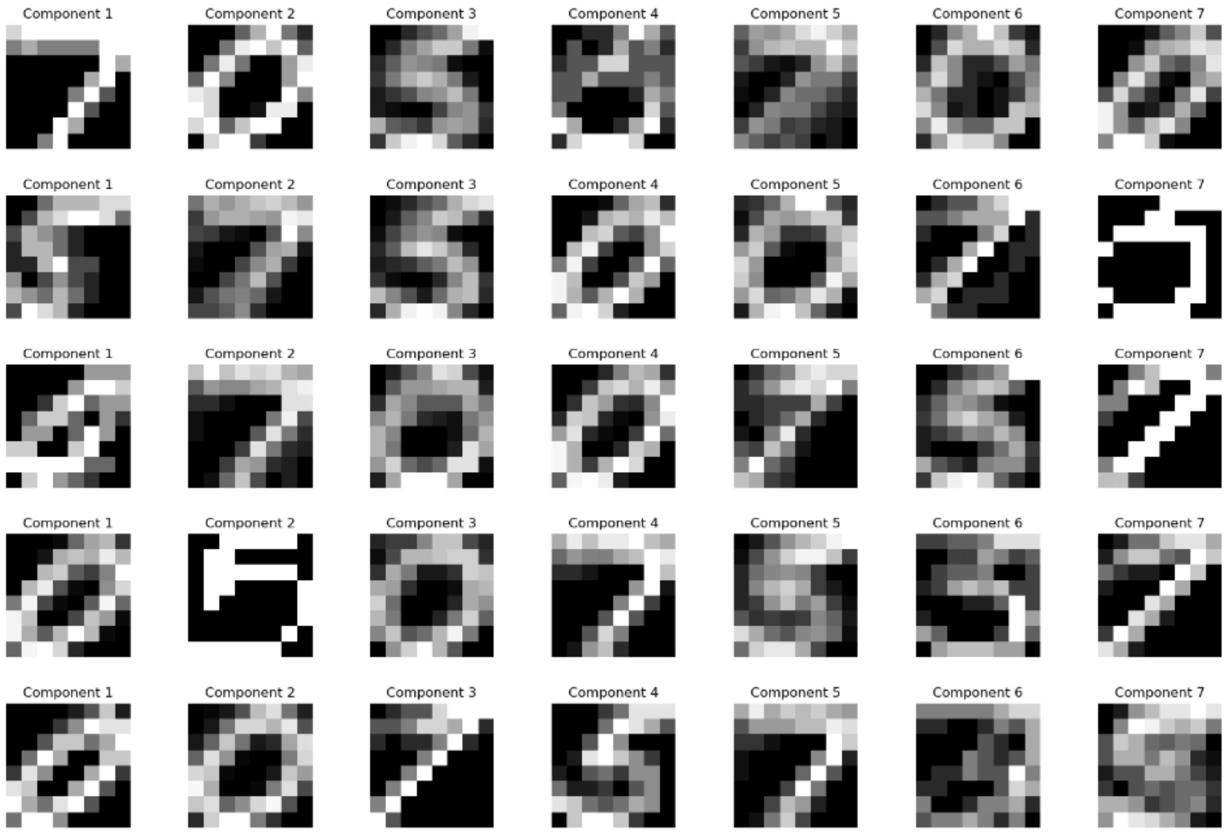


Figure 27: Learned probability vectors as images for $k = 7$, each run 5 times

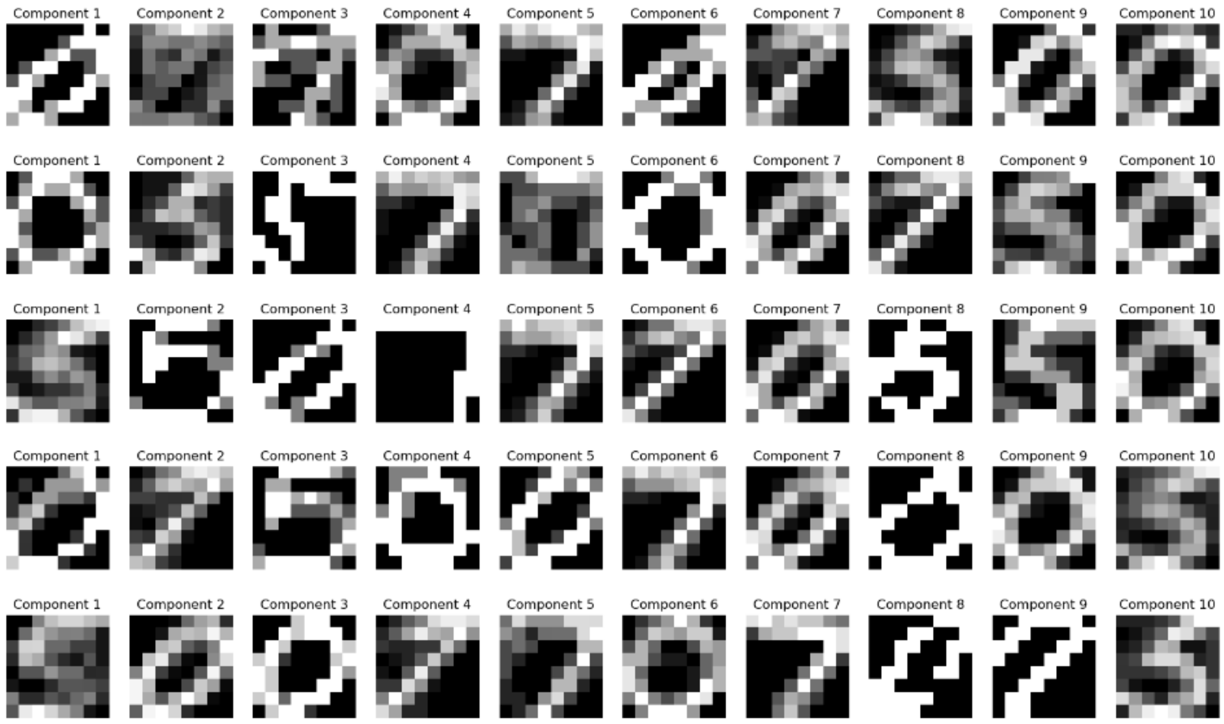


Figure 28: Learned probability vectors as images for $k = 10$, each run 5 times

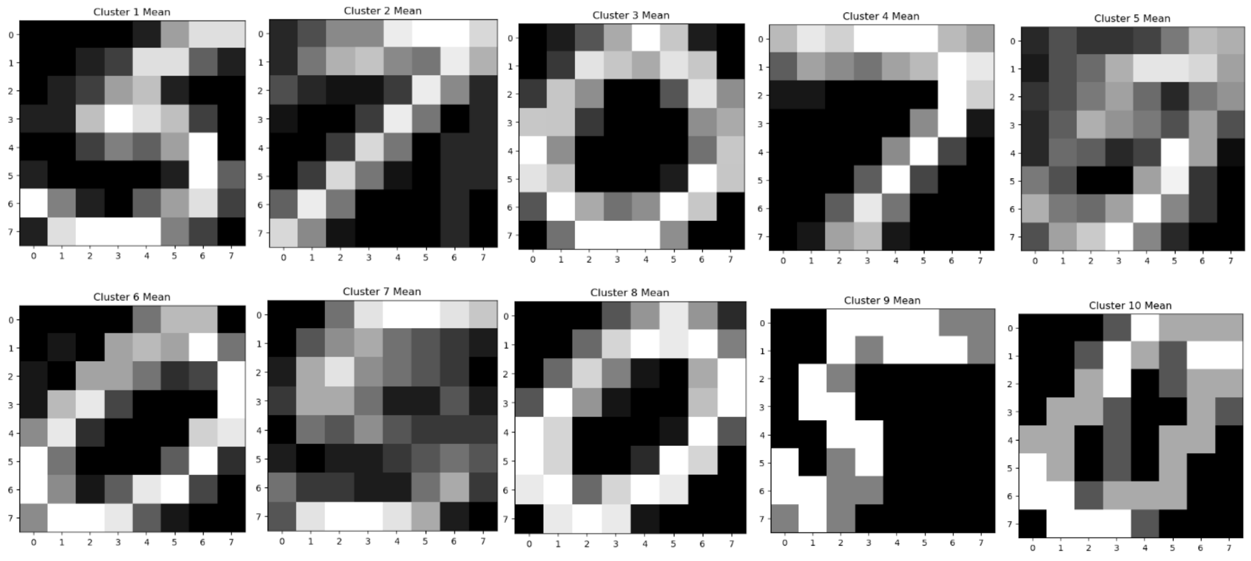
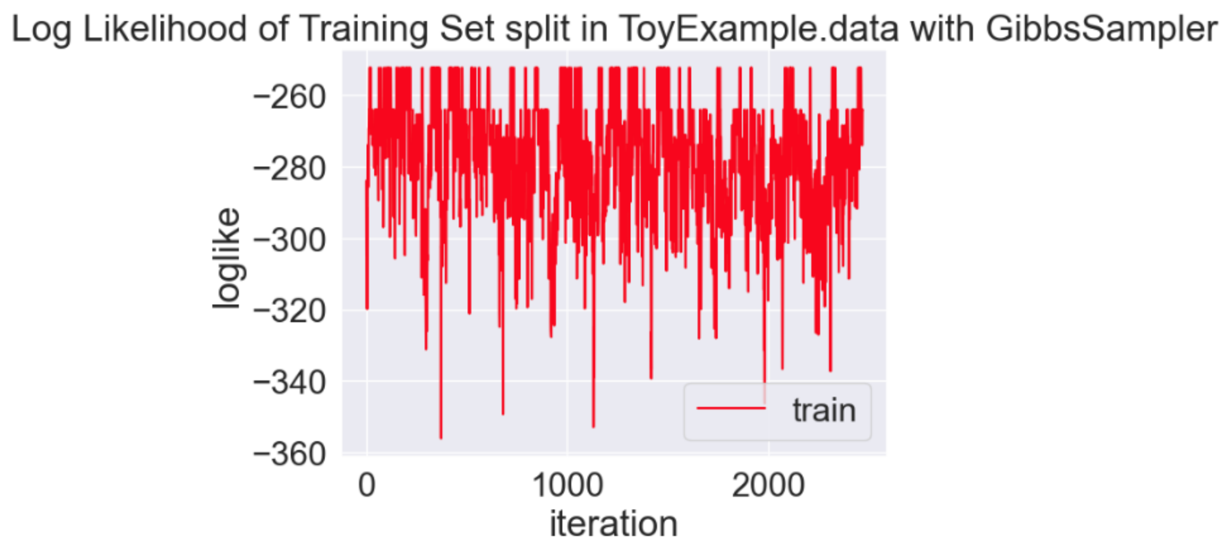


Figure 29: Cluster Mean for k={2,3,4,7,10}

2) Running toyexample.data with larger number of iterations $n\_iter = 2000$



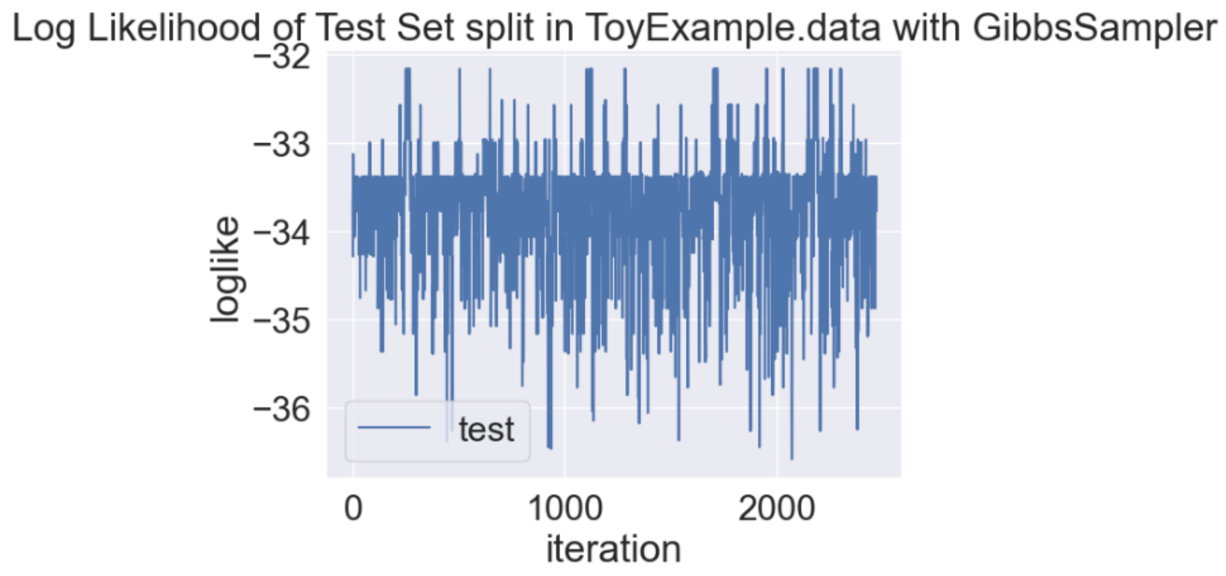
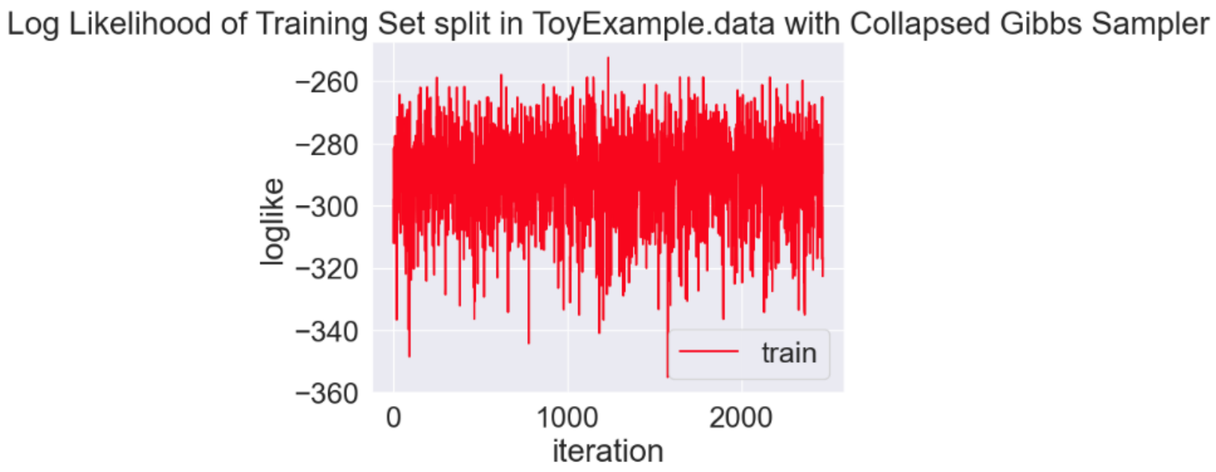


Figure 30: Training and test set from ToyExample.data Log-likelihood over 2000 iterations with Standard Gibbs Sampler to reduce noise



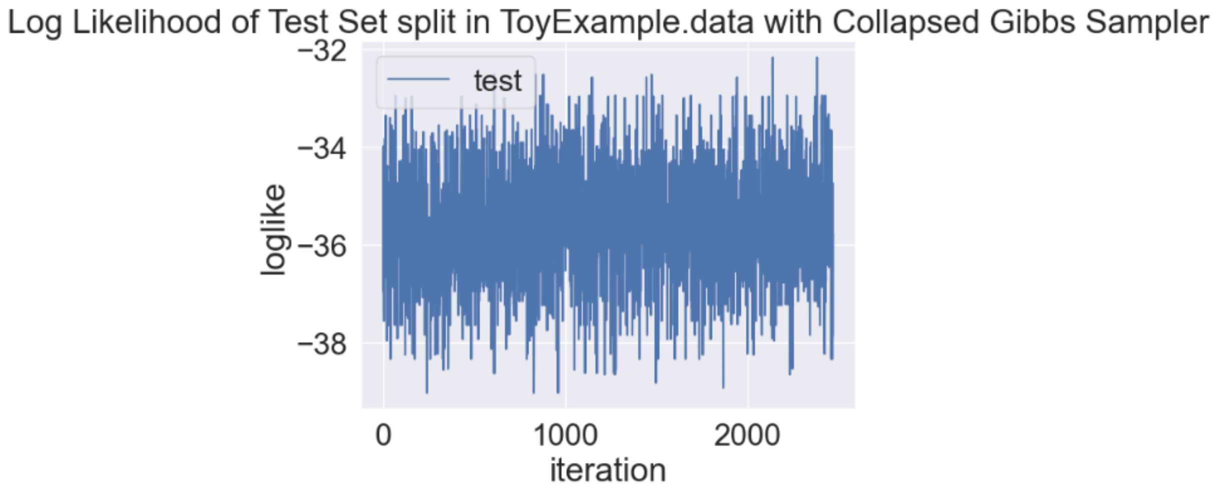


Figure 31: Training and test set from ToyExample.data Log-likelihood over 2000 iterations with Collapsed Gibbs Sampler to reduce noise

Code Execution

All code is implemented in Python through Jupyter Notebook.

Question 1: Code

```
In [ ]: #1d)
```

```
In [41]: def main():
             # load the data set
             Y = np.loadtxt('binarydigits.txt')
             N, D = Y.shape

             # Compute the ML parameters for the Bernoulli distribution
             # The ML estimate for the probability p_d of pixel d being 1 is the proportion of images
             # where pixel d is 1.
             p_ml = np.mean(Y, axis=0)

             ml_image = np.reshape(p_ml, (8, 8))

             #8x8 image
             fig, ax = plt.subplots()
             cax = ax.matshow(ml_image, interpolation="None", cmap='gray')
             fig.colorbar(cax, label="Probability")

             # denote probability in teh pixel
             for i in range(ml_image.shape[0]):
                 for j in range(ml_image.shape[1]):
                     ax.text(j, i, f'{ml_image[i, j]:.2f}', va='center', ha='center', color='red')

             plt.title("ML Estimates as 8x8 Image")
             plt.axis('off')
             plt.show()

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

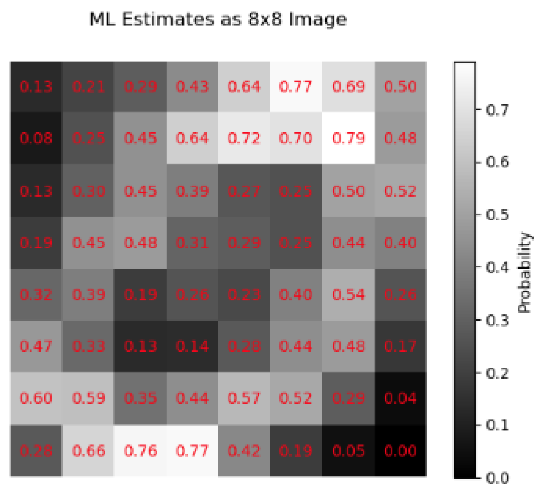


Figure 32: Learned probability vectors as images for $k = \{2, 3, 4, 7, 10\}$, each run 5 times

```
In [ ]: #1e)
```

```
In [3]: def compute_map_parameters(Y, alpha, beta):
            # MAP estimate
            N, D = Y.shape
            sum_pixels = np.sum(Y, axis=0)
            map_probabilities = (alpha-1 + sum_pixels) / (N + alpha + beta - 2)
            return map_probabilities

        def main():
            # load the data set
            Y = np.loadtxt('binarydigits.txt')
            N, D = Y.shape

            # Compute the MAP parameters for the Bernoulli distribution
            alpha, beta = 3, 3

            map_parameters= compute_map_parameters(Y, alpha, beta)

            # Reshape the MAP parameters for display
            map_image = np.reshape(map_parameters, (8, 8))

            # Display the MAP parameters as an 8x8 image
            fig, ax = plt.subplots()
            cax = ax.matshow(map_image, interpolation="None", cmap='gray')

            # Add colorbar to show the probability scale
            fig.colorbar(cax, label="Probability")

            # Loop over data dimensions and create text annotations.
            for i in range(map_image.shape[0]):
                for j in range(map_image.shape[1]):
                    ax.text(j, i, f'{map_image[i, j]:.2f}', va='center', ha='center', color='red')

            plt.title("MAP Estimates as 8x8 Image")
            plt.axis('off')
            plt.show()


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

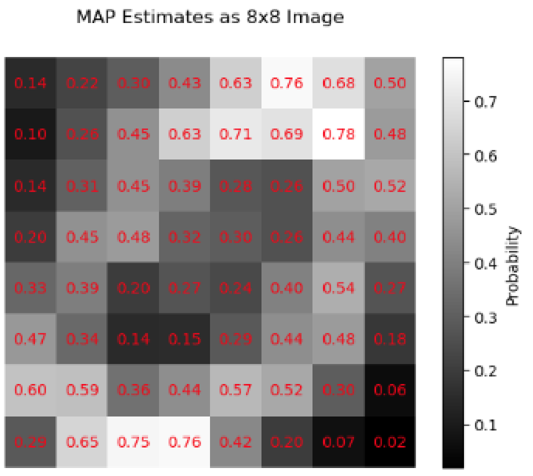


Figure 33: Learned probability vectors as images for $k = \{2, 3, 4, 7, 10\}$, each run 5 times

Question 2: Code

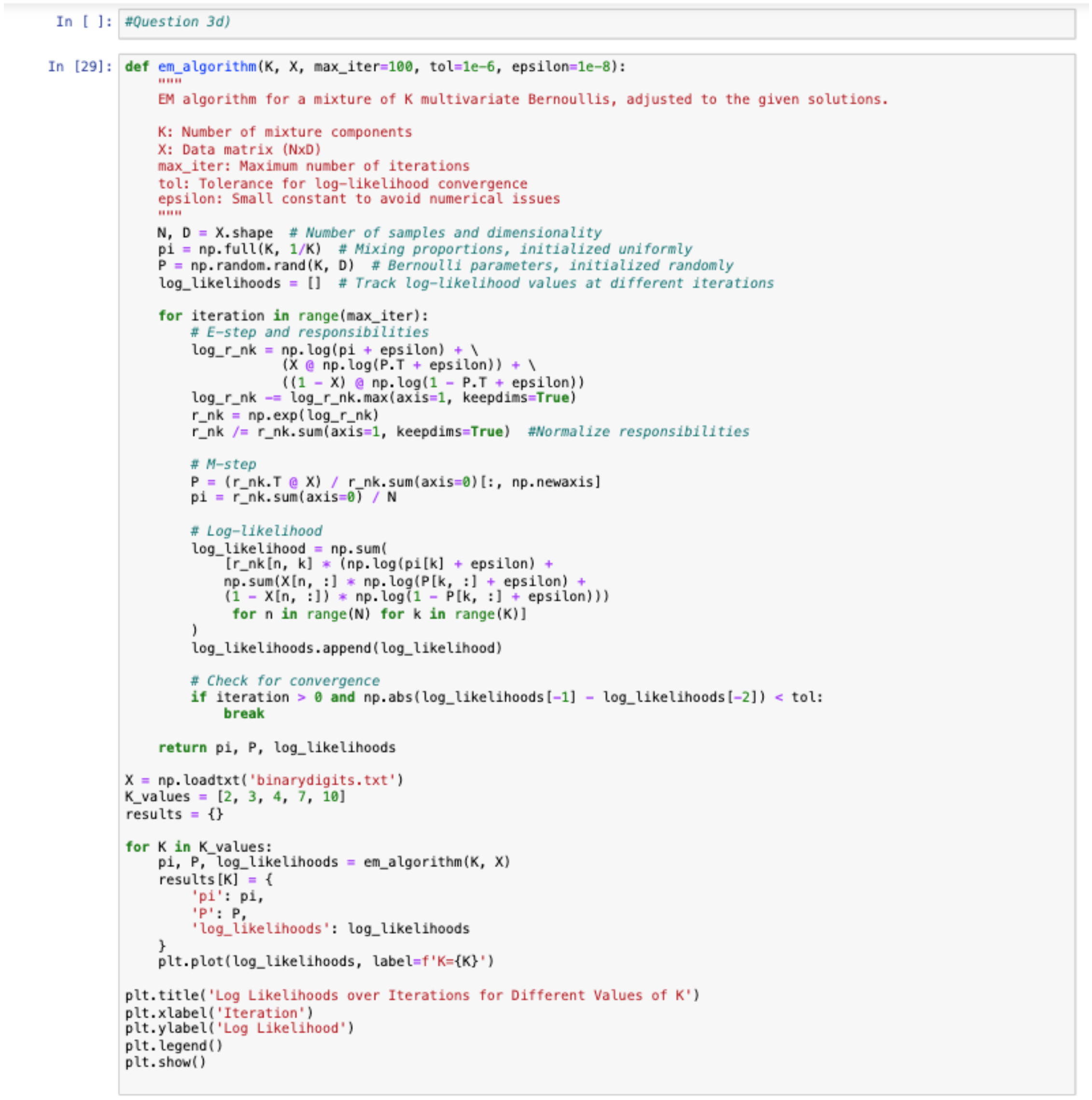
```
In [ ]: #2a,b,c calculating the posterior probabilities of each of the three models generated the data in binarydigits.txt
```

```
In [98]: from scipy.special import betaln
         def log_model_a_likelihood(N, D):
             #log likelihood is simply ND * log(0.5).
             return N * D * np.log(0.5)

         def log_model_b_likelihood(Y):
             # integrate over the possible values of p_d, which are all the same for each pixel.
             N, D = Y.shape
             sum_pixels = np.sum(Y)
             log_likelihood = betaln(sum_pixels + 1, N*D - sum_pixels + 1) - D * betaln(1, 1)
             return log_likelihood

         def log_model_c_likelihood(Y):
             # calculate the sum of log probabilities for individual pixels.
             N, D = Y.shape
             sum_pixels = np.sum(Y, axis=0)
             log_likelihoods = [betaln(sum_pixels[d] + 1, N - sum_pixels[d] + 1) - betaln(1, 1) for d in range(D)]
             return np.sum(log_likelihoods)

         # Calculate the size of the dataset
         N, D = Y.shape

         # Calculate the log likelihood of each model
         log_likelihood_a = log_model_a_likelihood(N, D)
         log_likelihood_b = log_model_b_likelihood(Y)
         log_likelihood_c = log_model_c_likelihood(Y)

         log_likelihood_a, log_likelihood_b, log_likelihood_c

Out[98]: (-4436.14195558365, -4283.721342577359, -3851.1957439211315)
```

```
In [99]: def calculate_posterior_probabilities(Y):
             N, D = Y.shape
             # Calculate the log likelihood of each model.
             log_likelihood_a = log_model_a_likelihood(N, D)
             log_likelihood_b = log_model_b_likelihood(Y)
             log_likelihood_c = log_model_c_likelihood(Y)

             # Compute the log of the sum of exponentiated log likelihoods for normalization.
             log_total_likelihood = np.logaddexp(log_likelihood_a, np.logaddexp(log_likelihood_b, log_likelihood_c))

             # Convert log likelihoods to probabilities.
             probability_a = np.exp(log_likelihood_a - log_total_likelihood)
             probability_b = np.exp(log_likelihood_b - log_total_likelihood)
             probability_c = np.exp(log_likelihood_c - log_total_likelihood)

             return probability_a, probability_b, probability_c

         Y = load_data()
         posterior_probabilities = calculate_posterior_probabilities(Y)
         print(f"Posterior probabilities:\nModel 1: {posterior_probabilities[0]}\nModel 2: {posterior_probabilities[1]}
                \nModel 3: {posterior_probabilities[2]}")

         Posterior probabilities:
         Model 1: 9.142986210361563e-255
         Model 2: 1.4339011785434019e-188
         Model 3: 1.0
```

Figure 34: Learned probability vectors as images for $k = \{2, 3, 4, 7, 10\}$, each run 5 times

Question 3: Code



```
In [ ]: #Question 3d)

In [29]: def em_algorithm(K, X, max_iter=100, tol=1e-6, epsilon=1e-8):
             """
             EM algorithm for a mixture of K multivariate Bernoullis, adjusted to the given solutions.

             K: Number of mixture components
             X: Data matrix (NxD)
             max_iter: Maximum number of iterations
             tol: Tolerance for log-likelihood convergence
             epsilon: Small constant to avoid numerical issues
             """
             N, D = X.shape  # Number of samples and dimensionality
             pi = np.full(K, 1/K)  # Mixing proportions, initialized uniformly
             P = np.random.rand(K, D)  # Bernoulli parameters, initialized randomly
             log_likelihoods = []  # Track log-likelihood values at different iterations

             for iteration in range(max_iter):
                 # E-step and responsibilities
                 log_r_nk = np.log(pi + epsilon) + \
                            (X @ np.log(P.T + epsilon)) + \
                            ((1 - X) @ np.log(1 - P.T + epsilon))
                 log_r_nk -= log_r_nk.max(axis=1, keepdims=True)
                 r_nk = np.exp(log_r_nk)
                 r_nk /= r_nk.sum(axis=1, keepdims=True)  #Normalize responsibilities

                 # M-step
                 P = (r_nk.T @ X) / r_nk.sum(axis=0)[:, np.newaxis]
                 pi = r_nk.sum(axis=0) / N

                 # Log-likelihood
                 log_likelihood = np.sum(
                     [r_nk[n, k] * (np.log(pi[k] + epsilon) +
                     np.sum(X[n, :] * np.log(P[k, :] + epsilon) +
                     (1 - X[n, :]) * np.log(1 - P[k, :] + epsilon)))
                      for n in range(N) for k in range(K)]
                 )
                 log_likelihoods.append(log_likelihood)

                 # Check for convergence
                 if iteration > 0 and np.abs(log_likelihoods[-1] - log_likelihoods[-2]) < tol:
                     break

             return pi, P, log_likelihoods

         X = np.loadtxt('binarydigits.txt')
         K_values = [2, 3, 4, 7, 10]
         results = {}

         for K in K_values:
             pi, P, log_likelihoods = em_algorithm(K, X)
             results[K] = {
                 'pi': pi,
                 'P': P,
                 'log_likelihoods': log_likelihoods
             }
             plt.plot(log_likelihoods, label=f'K={K}')

         plt.title('Log Likelihoods over Iterations for Different Values of K')
         plt.xlabel('Iteration')
         plt.ylabel('Log Likelihood')
         plt.legend()
         plt.show()
```

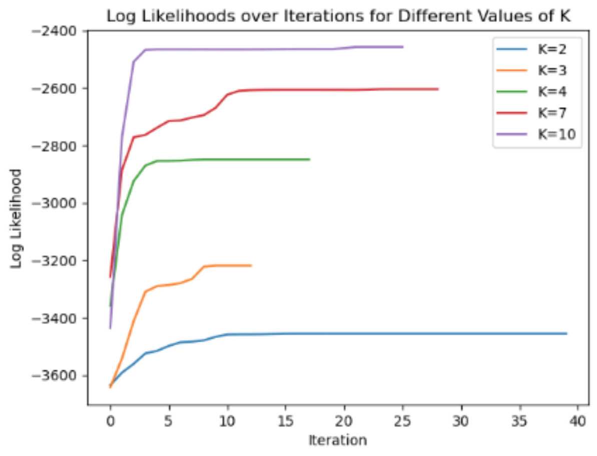


Figure 35: Learned probability vectors as images for $k = \{2, 3, 4, 7, 10\}$, each run 5 times

```
In [ ]: #3e)
```

```
In [10]: # Define the K values and the number of times to run the algorithm
K_values = [2, 3, 4, 7, 10]
num_runs = 5  # Number of runs with different initial conditions

# Store the results for analysis
results = {K: [] for K in K_values}

# Run the EM algorithm multiple times for each K and store the results
for K in K_values:
    for run in range(num_runs):
        pi, P, log_likelihoods = em_algorithm(K, X)
        results[K].append((pi, P, log_likelihoods))
        print(f'Run {run+1} for K={K} completed.')

# Example code to visualize one set of learned probability vectors as images
for K in K_values:
    pi, P, log_likelihoods = results[K][0]  # Take the first run for example
    fig, axs = plt.subplots(1, K, figsize=(20, 2))
    for k in range(K):
        axs[k].imshow(P[k].reshape(8, 8), cmap='gray')  # Replace with actual dimensions
        axs[k].axis('off')
        axs[k].set_title(f'Component {k+1}')
    plt.show()
```

```
Run 1 for K=2 completed.
Run 2 for K=2 completed.
Run 3 for K=2 completed.
Run 4 for K=2 completed.
Run 5 for K=2 completed.
Run 1 for K=3 completed.
Run 2 for K=3 completed.
Run 3 for K=3 completed.
Run 4 for K=3 completed.
Run 5 for K=3 completed.
Run 1 for K=4 completed.
Run 2 for K=4 completed.
Run 3 for K=4 completed.
Run 4 for K=4 completed.
Run 5 for K=4 completed.
Run 1 for K=7 completed.
Run 2 for K=7 completed.
Run 3 for K=7 completed.
Run 4 for K=7 completed.
Run 5 for K=7 completed.
Run 1 for K=10 completed.
Run 2 for K=10 completed.
Run 3 for K=10 completed.
Run 4 for K=10 completed.
Run 5 for K=10 completed.
```

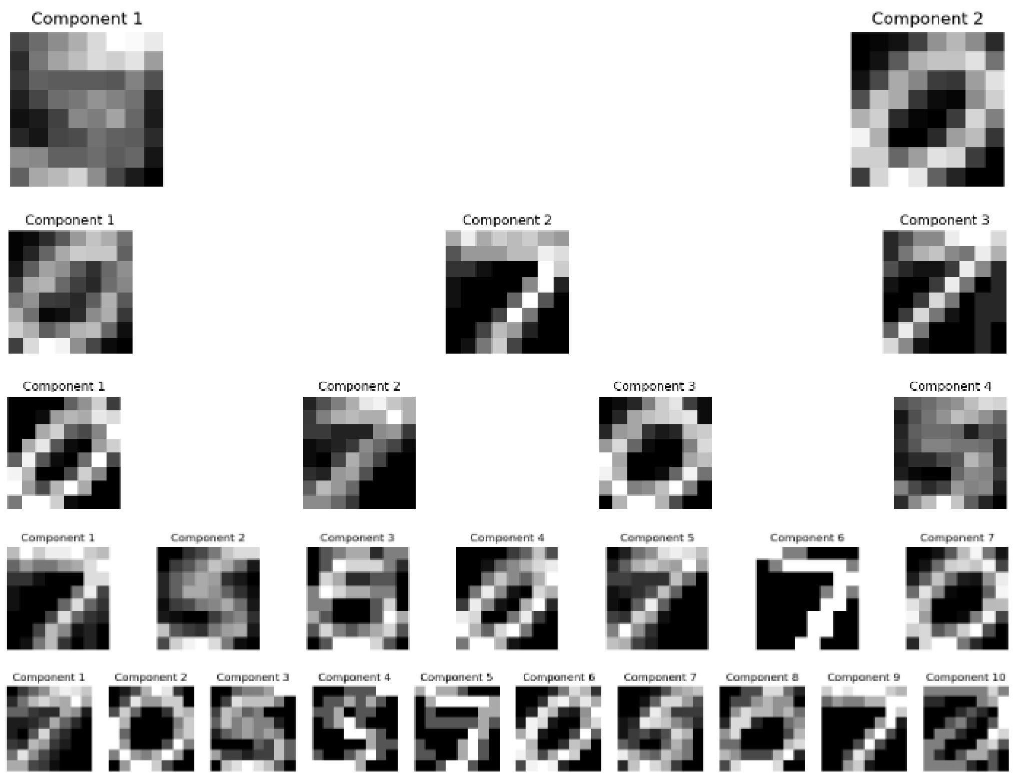
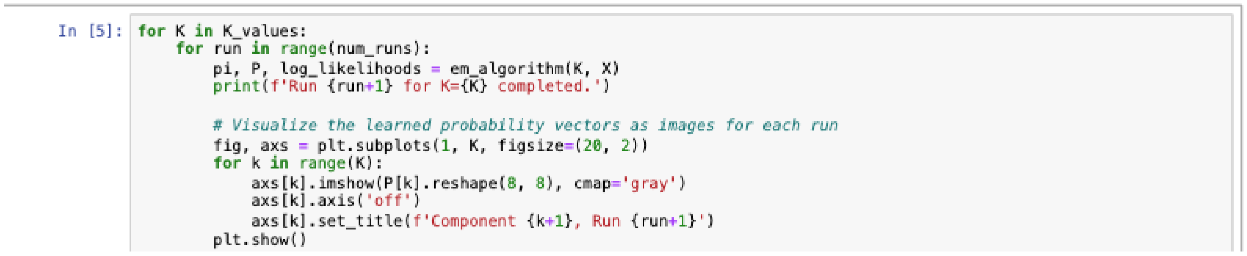
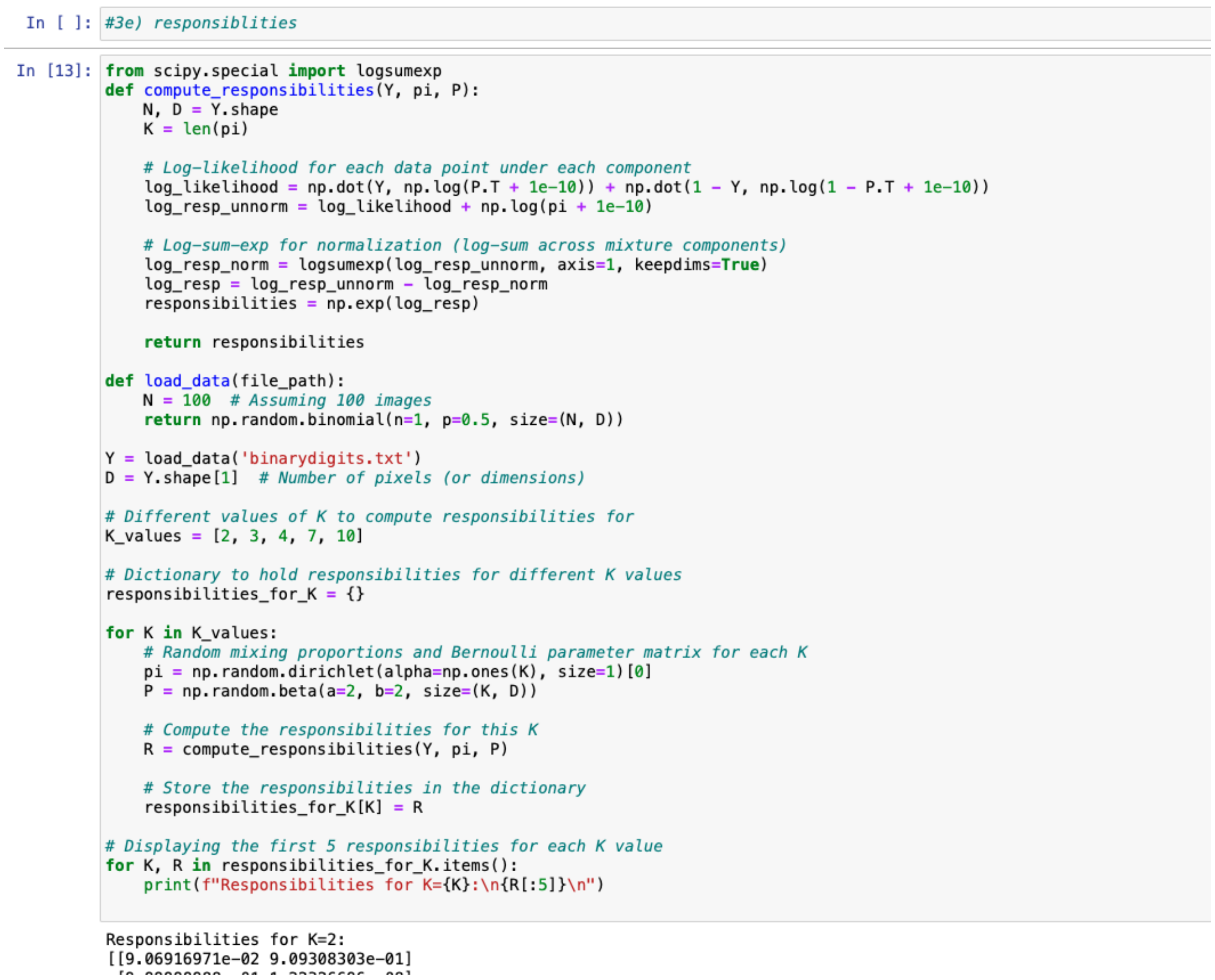


Figure 36: Learned probability vectors as images for $k = \{2, 3, 4, 7, 10\}$, each run 5 times



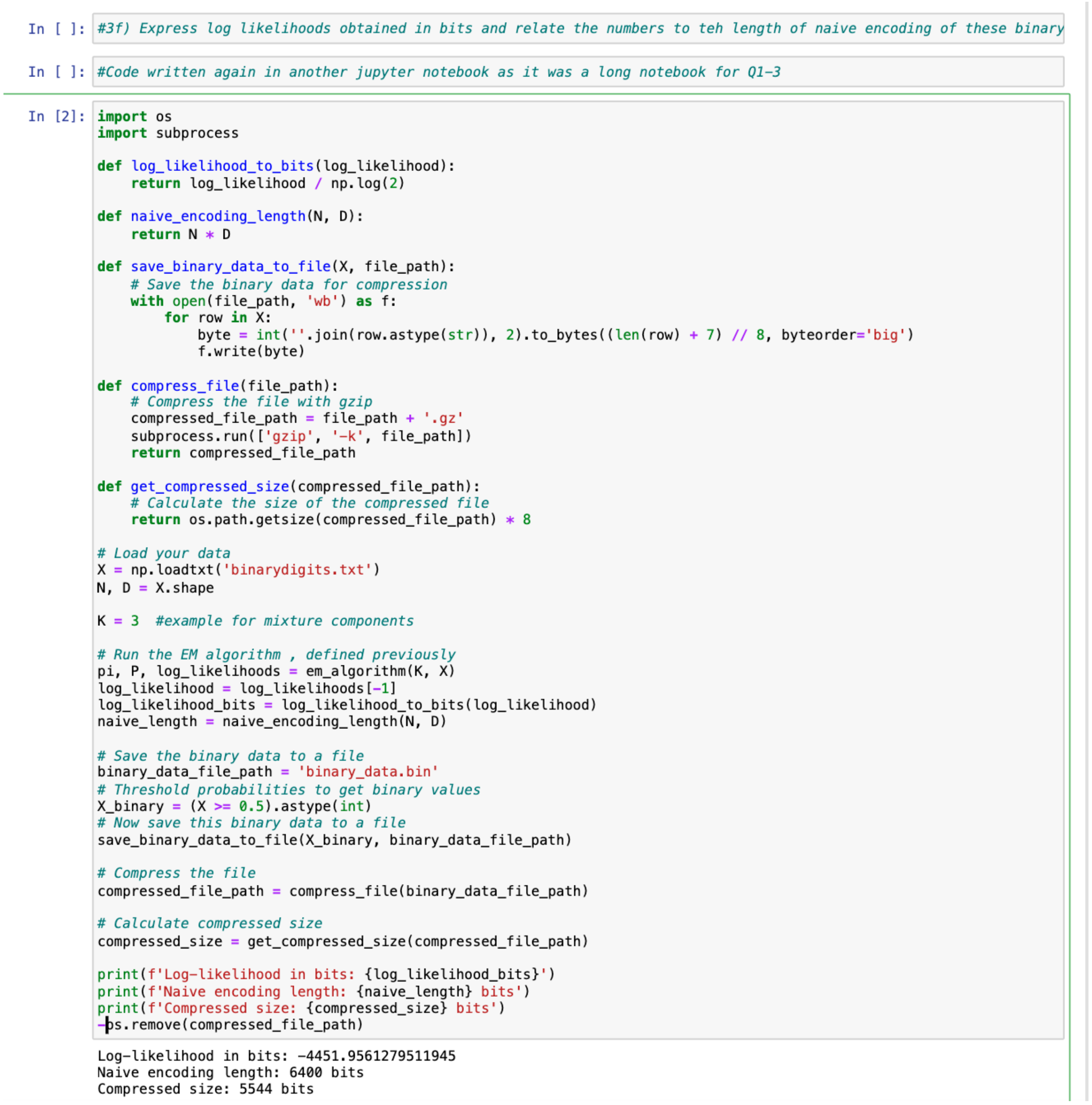
```
In [5]: for K in K_values:
            for run in range(num_runs):
                pi, P, log_likelihoods = em_algorithm(K, X)
                print(f'Run {run+1} for K={K} completed.')

                # Visualize the learned probability vectors as images for each run
                fig, axs = plt.subplots(1, K, figsize=(20, 2))
                for k in range(K):
                    axs[k].imshow(P[k].reshape(8, 8), cmap='gray')
                    axs[k].axis('off')
                    axs[k].set_title(f'Component {k+1}, Run {run+1}')
                plt.show()
```

Figure 37: Code written for Learned Probability Vector Images at each run for each component k={2,3,4,7,10}



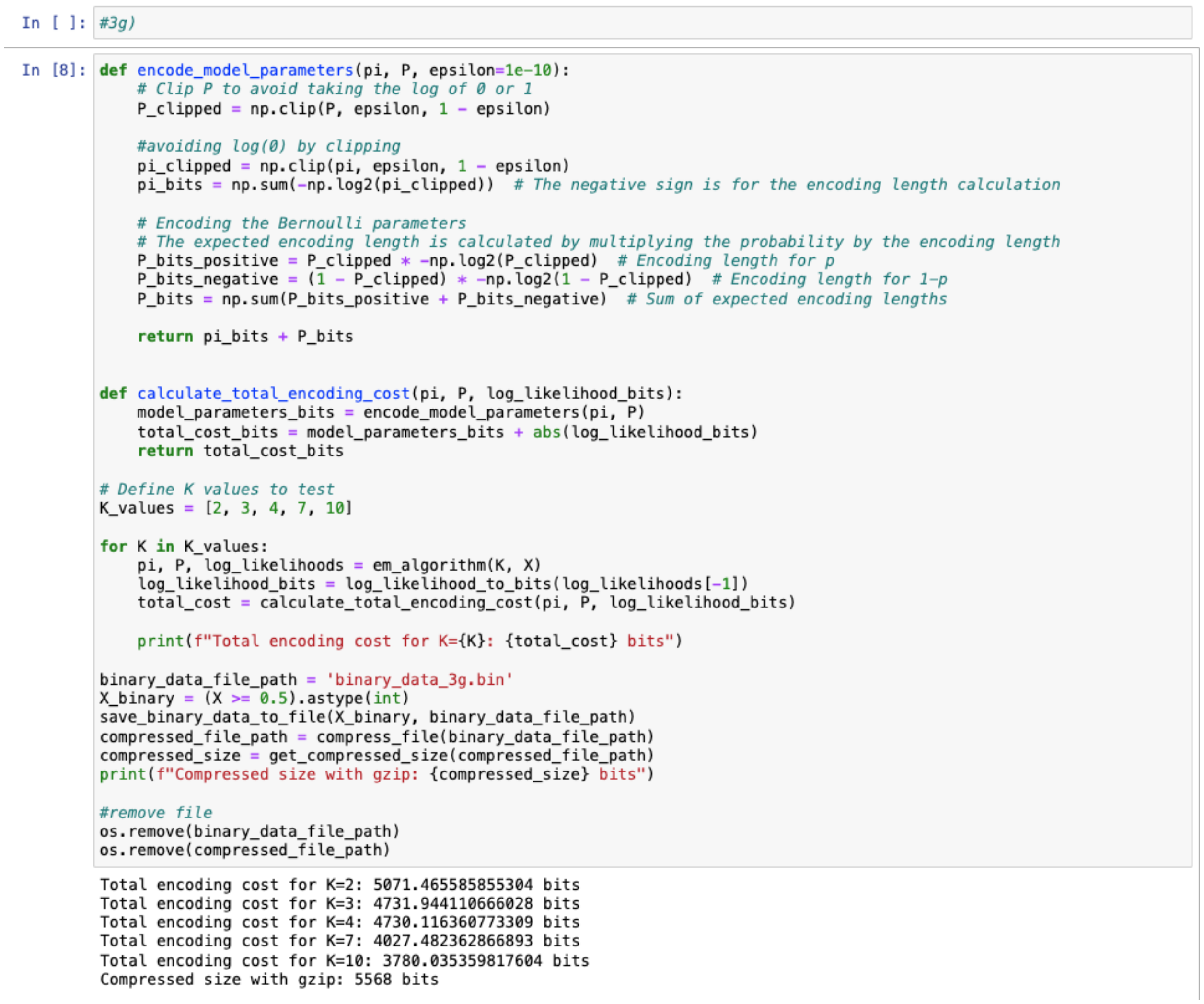
```
In [ ]: #3e) responsiblities

In [13]: from scipy.special import logsumexp
         def compute_responsibilities(Y, pi, P):
             N, D = Y.shape
             K = len(pi)

             # Log-likelihood for each data point under each component
             log_likelihood = np.dot(Y, np.log(P.T + 1e-10)) + np.dot(1 - Y, np.log(1 - P.T + 1e-10))
             log_resp_unnorm = log_likelihood + np.log(pi + 1e-10)

             # Log-sum-exp for normalization (log-sum across mixture components)
             log_resp_norm = logsumexp(log_resp_unnorm, axis=1, keepdims=True)
             log_resp = log_resp_unnorm - log_resp_norm
             responsibilities = np.exp(log_resp)

             return responsibilities

         def load_data(file_path):
             N = 100  # Assuming 100 images
             return np.random.binomial(n=1, p=0.5, size=(N, D))

         Y = load_data('binarydigits.txt')
         D = Y.shape[1]  # Number of pixels (or dimensions)

         # Different values of K to compute responsibilities for
         K_values = [2, 3, 4, 7, 10]

         # Dictionary to hold responsibilities for different K values
         responsibilities_for_K = {}

         for K in K_values:
             # Random mixing proportions and Bernoulli parameter matrix for each K
             pi = np.random.dirichlet(alpha=np.ones(K), size=1)[0]
             P = np.random.beta(a=2, b=2, size=(K, D))

             # Compute the responsibilities for this K
             R = compute_responsibilities(Y, pi, P)

             # Store the responsibilities in the dictionary
             responsibilities_for_K[K] = R

         # Displaying the first 5 responsibilities for each K value
         for K, R in responsibilities_for_K.items():
             print(f"Responsibilities for K={K}:\n{R[:5]}\n")

         Responsibilities for K=2:
         [[9.06916971e-02 9.09308303e-01]
```

Figure 38: Code written computing responsibilities π_k where k={2,3,4,7,10}



```
In [ ]: #3f) Express log likelihoods obtained in bits and relate the numbers to teh length of naive encoding of these binary

In [ ]: #Code written again in another jupyter notebook as it was a long notebook for Q1-3

In [2]: import os
        import subprocess

        def log_likelihood_to_bits(log_likelihood):
            return log_likelihood / np.log(2)

        def naive_encoding_length(N, D):
            return N * D

        def save_binary_data_to_file(X, file_path):
            # Save the binary data for compression
            with open(file_path, 'wb') as f:
                for row in X:
                    byte = int(''.join(row.astype(str)), 2).to_bytes((len(row) + 7) // 8, byteorder='big')
                    f.write(byte)

        def compress_file(file_path):
            # Compress the file with gzip
            compressed_file_path = file_path + '.gz'
            subprocess.run(['gzip', '-k', file_path])
            return compressed_file_path

        def get_compressed_size(compressed_file_path):
            # Calculate the size of the compressed file
            return os.path.getsize(compressed_file_path) * 8

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

        K = 3  #example for mixture components

        # Run the EM algorithm , defined previously
        pi, P, log_likelihoods = em_algorithm(K, X)
        log_likelihood = log_likelihoods[-1]
        log_likelihood_bits = log_likelihood_to_bits(log_likelihood)
        naive_length = naive_encoding_length(N, D)

        # Save the binary data to a file
        binary_data_file_path = 'binary_data.bin'
        # Threshold probabilities to get binary values
        X_binary = (X >= 0.5).astype(int)
        # Now save this binary data to a file
        save_binary_data_to_file(X_binary, binary_data_file_path)

        # Compress the file
        compressed_file_path = compress_file(binary_data_file_path)

        # Calculate compressed size
        compressed_size = get_compressed_size(compressed_file_path)

        print(f'Log-likelihood in bits: {log_likelihood_bits}')
        print(f'Naive encoding length: {naive_length} bits')
        print(f'Compressed size: {compressed_size} bits')
        os.remove(compressed_file_path)

        Log-likelihood in bits: -4451.9561279511945
        Naive encoding length: 6400 bits
        Compressed size: 5544 bits
```

Figure 39: Code written computing size of gzip compressed file and naive encoding length file and log-likelihood



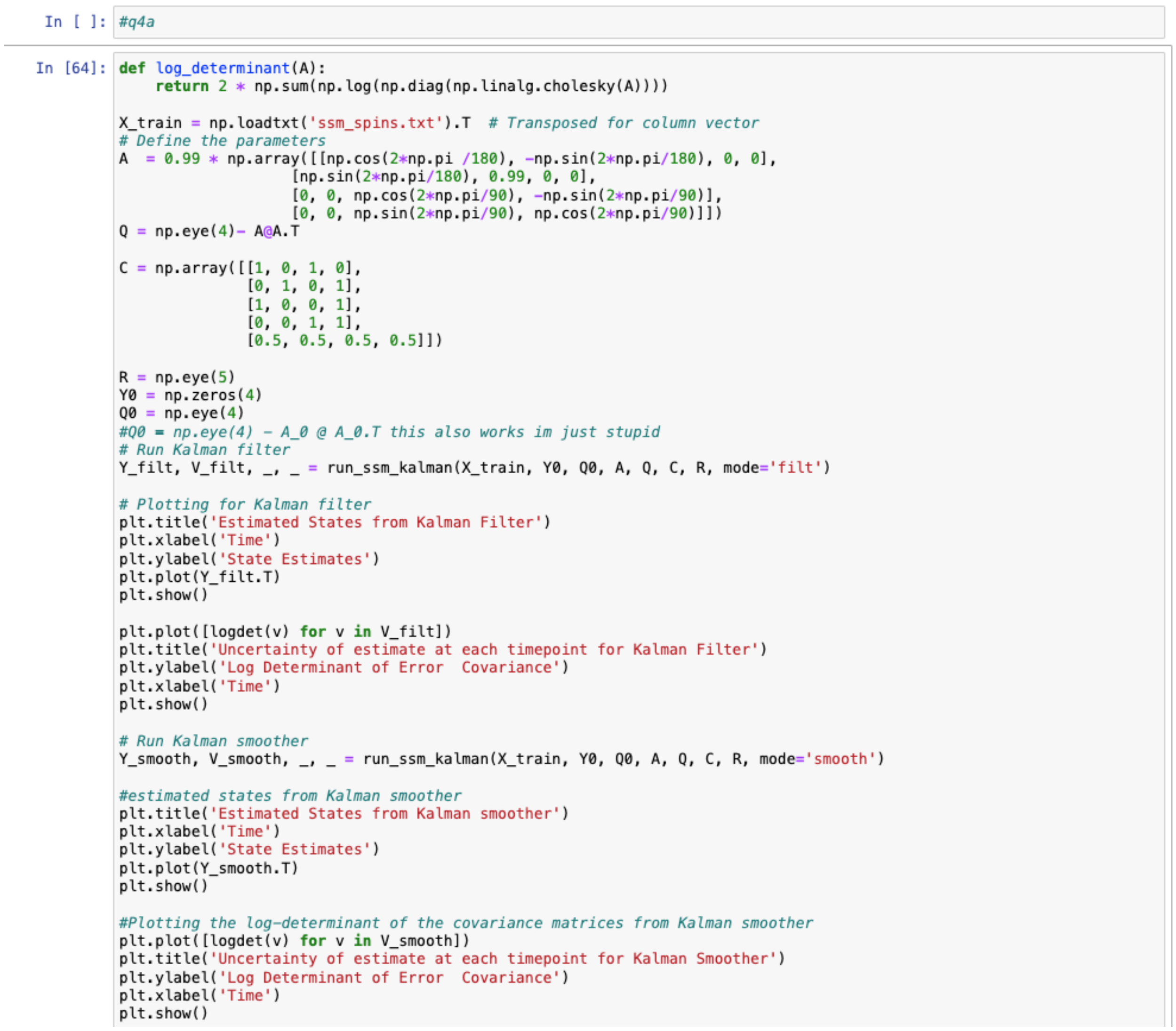
```
In [ ]: #3g)
```

```
In [8]: def encode_model_parameters(pi, P, epsilon=1e-10):
            # Clip P to avoid taking the log of 0 or 1
            P_clipped = np.clip(P, epsilon, 1 - epsilon)

            #avoiding log(0) by clipping
            pi_clipped = np.clip(pi, epsilon, 1 - epsilon)
            pi_bits = np.sum(-np.log2(pi_clipped))  # The negative sign is for the encoding length calculation

            # Encoding the Bernoulli parameters
            # The expected encoding length is calculated by multiplying the probability by the encoding length
            P_bits_positive = P_clipped * -np.log2(P_clipped)  # Encoding length for p
            P_bits_negative = (1 - P_clipped) * -np.log2(1 - P_clipped)  # Encoding length for 1-p
            P_bits = np.sum(P_bits_positive + P_bits_negative)  # Sum of expected encoding lengths

            return pi_bits + P_bits


        def calculate_total_encoding_cost(pi, P, log_likelihood_bits):
            model_parameters_bits = encode_model_parameters(pi, P)
            total_cost_bits = model_parameters_bits + abs(log_likelihood_bits)
            return total_cost_bits

        # Define K values to test
        K_values = [2, 3, 4, 7, 10]

        for K in K_values:
            pi, P, log_likelihoods = em_algorithm(K, X)
            log_likelihood_bits = log_likelihood_to_bits(log_likelihoods[-1])
            total_cost = calculate_total_encoding_cost(pi, P, log_likelihood_bits)

            print(f"Total encoding cost for K={K}: {total_cost} bits")

        binary_data_file_path = 'binary_data_3g.bin'
        X_binary = (X >= 0.5).astype(int)
        save_binary_data_to_file(X_binary, binary_data_file_path)
        compressed_file_path = compress_file(binary_data_file_path)
        compressed_size = get_compressed_size(compressed_file_path)
        print(f"Compressed size with gzip: {compressed_size} bits")

        #remove file
        os.remove(binary_data_file_path)
        os.remove(compressed_file_path)
```

```
Total encoding cost for K=2: 5071.465585855304 bits
Total encoding cost for K=3: 4731.944110666028 bits
Total encoding cost for K=4: 4730.116360773309 bits
Total encoding cost for K=7: 4027.482362866893 bits
Total encoding cost for K=10: 3780.035359817604 bits
Compressed size with gzip: 5568 bits
```

Figure 40: Code written ccomputing total encoding costs with increasing k components where k={2,3,4,7,10}

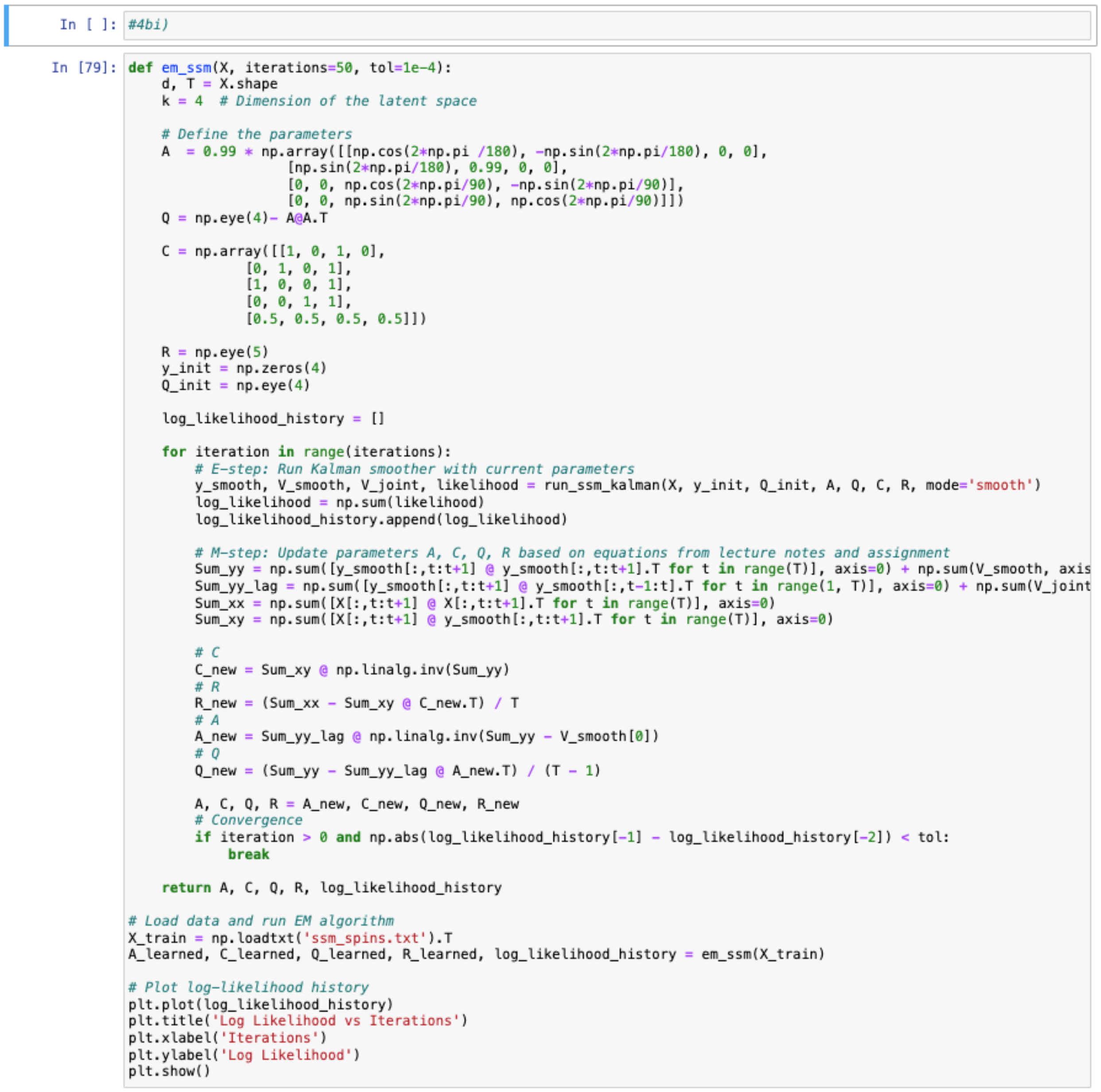
Question 4: Code



```
In [ ]: #q4a
```

```
In [64]: def log_determinant(A):
             return 2 * np.sum(np.log(np.diag(np.linalg.cholesky(A))))

         X_train = np.loadtxt('ssm_spins.txt').T  # Transposed for column vector
         # Define the parameters
         A  = 0.99 * np.array([[np.cos(2*np.pi /180), -np.sin(2*np.pi/180), 0, 0],
                               [np.sin(2*np.pi/180), 0.99, 0, 0],
                               [0, 0, np.cos(2*np.pi/90), -np.sin(2*np.pi/90)],
                               [0, 0, np.sin(2*np.pi/90), np.cos(2*np.pi/90)]])
         Q = np.eye(4)- A@A.T

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

         R = np.eye(5)
         Y0 = np.zeros(4)
         Q0 = np.eye(4)
         #Q0 = np.eye(4) - A_0 @ A_0.T this also works im just stupid
         # Run Kalman filter
         Y_filt, V_filt, _, _ = run_ssm_kalman(X_train, Y0, Q0, A, Q, C, R, mode='filt')

         # Plotting for Kalman filter
         plt.title('Estimated States from Kalman Filter')
         plt.xlabel('Time')
         plt.ylabel('State Estimates')
         plt.plot(Y_filt.T)
         plt.show()

         plt.plot([logdet(v) for v in V_filt])
         plt.title('Uncertainty of estimate at each timepoint for Kalman Filter')
         plt.ylabel('Log Determinant of Error  Covariance')
         plt.xlabel('Time')
         plt.show()

         # Run Kalman smoother
         Y_smooth, V_smooth, _, _ = run_ssm_kalman(X_train, Y0, Q0, A, Q, C, R, mode='smooth')

         #estimated states from Kalman smoother
         plt.title('Estimated States from Kalman smoother')
         plt.xlabel('Time')
         plt.ylabel('State Estimates')
         plt.plot(Y_smooth.T)
         plt.show()

         #Plotting the log-determinant of the covariance matrices from Kalman smoother
         plt.plot([logdet(v) for v in V_smooth])
         plt.title('Uncertainty of estimate at each timepoint for Kalman Smoother')
         plt.ylabel('Log Determinant of Error  Covariance')
         plt.xlabel('Time')
         plt.show()
```

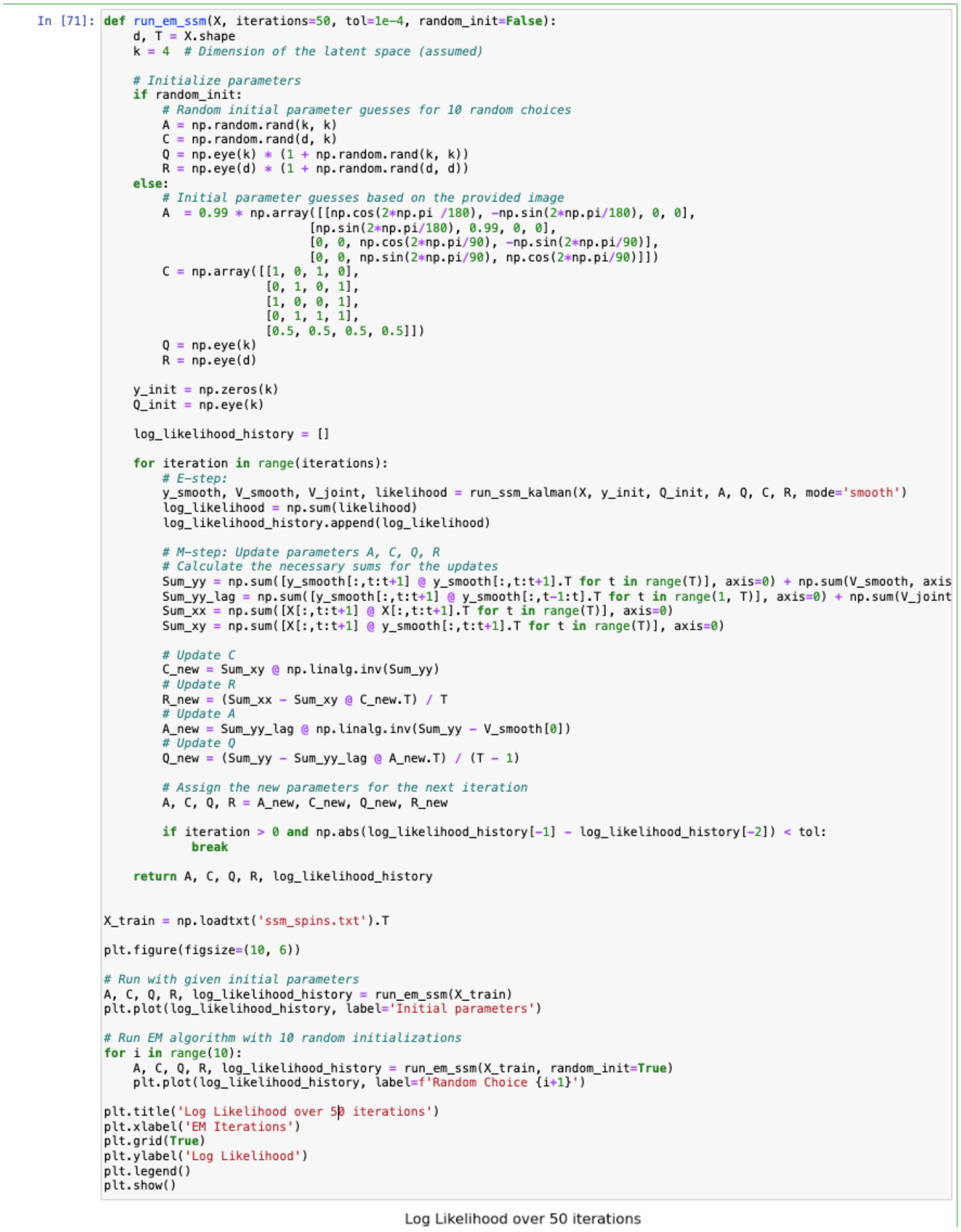
Figure 41: Code written plotting the Kalman Filtering and Smoothing plots in Q4a



```
In [ ]: #4bi)
```

```
In [79]: def em_ssm(X, iterations=50, tol=1e-4):
             d, T = X.shape
             k = 4  # Dimension of the latent space

             # Define the parameters
             A  = 0.99 * np.array([[np.cos(2*np.pi /180), -np.sin(2*np.pi/180), 0, 0],
                             [np.sin(2*np.pi/180), 0.99, 0, 0],
                             [0, 0, np.cos(2*np.pi/90), -np.sin(2*np.pi/90)],
                             [0, 0, np.sin(2*np.pi/90), np.cos(2*np.pi/90)]])
             Q = np.eye(4)- A@A.T

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

             R = np.eye(5)
             y_init = np.zeros(4)
             Q_init = np.eye(4)

             log_likelihood_history = []

             for iteration in range(iterations):
                 # E-step: Run Kalman smoother with current parameters
                 y_smooth, V_smooth, V_joint, likelihood = run_ssm_kalman(X, y_init, Q_init, A, Q, C, R, mode='smooth')
                 log_likelihood = np.sum(likelihood)
                 log_likelihood_history.append(log_likelihood)

                 # M-step: Update parameters A, C, Q, R based on equations from lecture notes and assignment
                 Sum_yy = np.sum([y_smooth[:,t:t+1] @ y_smooth[:,t:t+1].T for t in range(T)], axis=0) + np.sum(V_smooth, axis
                 Sum_yy_lag = np.sum([y_smooth[:,t:t+1] @ y_smooth[:,t-1:t].T for t in range(1, T)], axis=0) + np.sum(V_joint
                 Sum_xx = np.sum([X[:,t:t+1] @ X[:,t:t+1].T for t in range(T)], axis=0)
                 Sum_xy = np.sum([X[:,t:t+1] @ y_smooth[:,t:t+1].T for t in range(T)], axis=0)

                 # C
                 C_new = Sum_xy @ np.linalg.inv(Sum_yy)
                 # R
                 R_new = (Sum_xx - Sum_xy @ C_new.T) / T
                 # A
                 A_new = Sum_yy_lag @ np.linalg.inv(Sum_yy - V_smooth[0])
                 # Q
                 Q_new = (Sum_yy - Sum_yy_lag @ A_new.T) / (T - 1)

                 A, C, Q, R = A_new, C_new, Q_new, R_new
                 # Convergence
                 if iteration > 0 and np.abs(log_likelihood_history[-1] - log_likelihood_history[-2]) < tol:
                     break

             return A, C, Q, R, log_likelihood_history

         # Load data and run EM algorithm
         X_train = np.loadtxt('ssm_spins.txt').T
         A_learned, C_learned, Q_learned, R_learned, log_likelihood_history = em_ssm(X_train)

         # Plot log-likelihood history
         plt.plot(log_likelihood_history)
         plt.title('Log Likelihood vs Iterations')
         plt.xlabel('Iterations')
         plt.ylabel('Log Likelihood')
         plt.show()
```

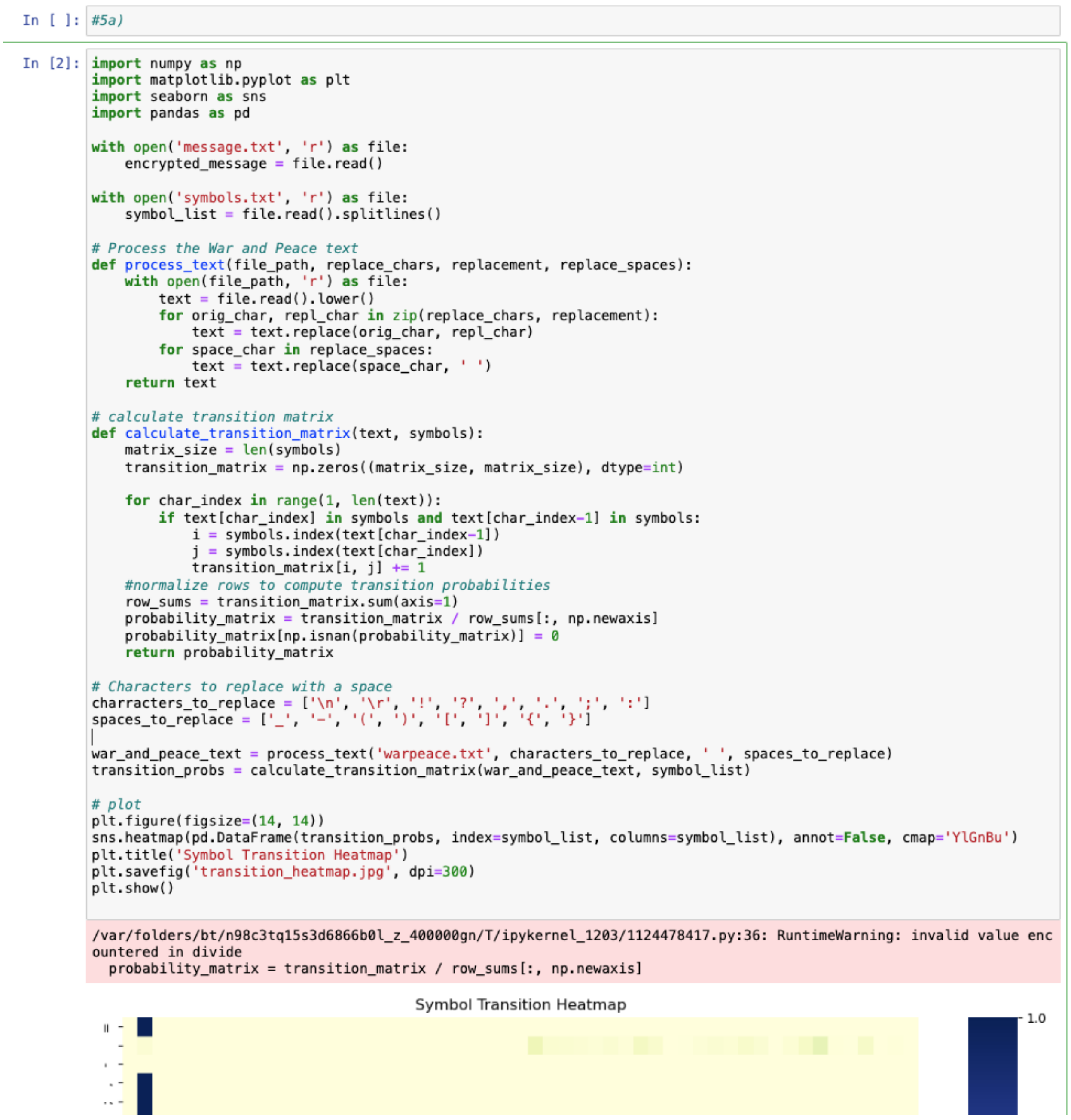
Figure 42: Code written computing loglikelihood under true parameters, with EM algorithm for 50 iterations, adapted by changing X_train with X_test (code omitted as it was just changing X_train attribute to X_test)



```
In [71]: def run_em_ssm(X, iterations=50, tol=1e-4, random_init=False):
             d, T = X.shape
             k = 4  # Dimension of the latent space (assumed)

             # Initialize parameters
             if random_init:
                 # Random initial parameter guesses for 10 random choices
                 A = np.random.rand(k, k)
                 C = np.random.rand(d, k)
                 Q = np.eye(k) * (1 + np.random.rand(k, k))
                 R = np.eye(d) * (1 + np.random.rand(d, d))
             else:
                 # Initial parameter guesses based on the provided image
                 A  = 0.99 * np.array([[np.cos(2*np.pi /180), -np.sin(2*np.pi/180), 0, 0],
                                  [np.sin(2*np.pi/180), 0.99, 0, 0],
                                  [0, 0, np.cos(2*np.pi/90), -np.sin(2*np.pi/90)],
                                  [0, 0, np.sin(2*np.pi/90), np.cos(2*np.pi/90)]])
                 C = np.array([[1, 0, 1, 0],
                               [0, 1, 0, 1],
                               [1, 0, 0, 1],
                               [0, 1, 1, 1],
                               [0.5, 0.5, 0.5, 0.5]])
                 Q = np.eye(k)
                 R = np.eye(d)

             y_init = np.zeros(k)
             Q_init = np.eye(k)

             log_likelihood_history = []

             for iteration in range(iterations):
                 # E-step:
                 y_smooth, V_smooth, V_joint, likelihood = run_ssm_kalman(X, y_init, Q_init, A, Q, C, R, mode='smooth')
                 log_likelihood = np.sum(likelihood)
                 log_likelihood_history.append(log_likelihood)

                 # M-step: Update parameters A, C, Q, R
                 # Calculate the necessary sums for the updates
                 Sum_yy = np.sum([y_smooth[:,t:t+1] @ y_smooth[:,t:t+1].T for t in range(T)], axis=0) + np.sum(V_smooth, axis
                 Sum_yy_lag = np.sum([y_smooth[:,t:t+1] @ y_smooth[:,t-1:t].T for t in range(1, T)], axis=0) + np.sum(V_joint
                 Sum_xx = np.sum([X[:,t:t+1] @ X[:,t:t+1].T for t in range(T)], axis=0)
                 Sum_xy = np.sum([X[:,t:t+1] @ y_smooth[:,t:t+1].T for t in range(T)], axis=0)

                 # Update C
                 C_new = Sum_xy @ np.linalg.inv(Sum_yy)
                 # Update R
                 R_new = (Sum_xx - Sum_xy @ C_new.T) / T
                 # Update A
                 A_new = Sum_yy_lag @ np.linalg.inv(Sum_yy - V_smooth[0])
                 # Update Q
                 Q_new = (Sum_yy - Sum_yy_lag @ A_new.T) / (T - 1)

                 # Assign the new parameters for the next iteration
                 A, C, Q, R = A_new, C_new, Q_new, R_new

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

             return A, C, Q, R, log_likelihood_history


         X_train = np.loadtxt('ssm_spins.txt').T

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

         # Run with given initial parameters
         A, C, Q, R, log_likelihood_history = run_em_ssm(X_train)
         plt.plot(log_likelihood_history, label='Initial parameters')

         # Run EM algorithm with 10 random initializations
         for i in range(10):
             A, C, Q, R, log_likelihood_history = run_em_ssm(X_train, random_init=True)
             plt.plot(log_likelihood_history, label=f'Random Choice {i+1}')

         plt.title('Log Likelihood over 50 iterations')
         plt.xlabel('EM Iterations')
         plt.grid(True)
         plt.ylabel('Log Likelihood')
         plt.legend()
         plt.show()
```

Figure 43: Code written computing loglikelihood under with 10 random choice generations for 50 and 100 iterations, number of iterations manually changed in the code, and also adapted by changing X_train with X_test (code omitted as it was just changing X_train attribute to X_test)

Question 5: Code



```
In [ ]: #5a)
```

```
In [2]: import numpy as np
        import matplotlib.pyplot as plt
        import seaborn as sns
        import pandas as pd

        with open('message.txt', 'r') as file:
            encrypted_message = file.read()

        with open('symbols.txt', 'r') as file:
            symbol_list = file.read().splitlines()

        # Process the War and Peace text
        def process_text(file_path, replace_chars, replacement, replace_spaces):
            with open(file_path, 'r') as file:
                text = file.read().lower()
                for orig_char, repl_char in zip(replace_chars, replacement):
                    text = text.replace(orig_char, repl_char)
                for space_char in replace_spaces:
                    text = text.replace(space_char, ' ')
            return text

        # calculate transition matrix
        def calculate_transition_matrix(text, symbols):
            matrix_size = len(symbols)
            transition_matrix = np.zeros((matrix_size, matrix_size), dtype=int)

            for char_index in range(1, len(text)):
                if text[char_index] in symbols and text[char_index-1] in symbols:
                    i = symbols.index(text[char_index-1])
                    j = symbols.index(text[char_index])
                    transition_matrix[i, j] += 1
            #normalize rows to compute transition probabilities
            row_sums = transition_matrix.sum(axis=1)
            probability_matrix = transition_matrix / row_sums[:, np.newaxis]
            probability_matrix[np.isnan(probability_matrix)] = 0
            return probability_matrix

        # Characters to replace with a space
        charracters_to_replace = ['\n', '\r', '!', '?', ',', '.', ';', ':']
        spaces_to_replace = ['_', '-', '(', ')', '[', ']', '{', '}']

        war_and_peace_text = process_text('warpeace.txt', characters_to_replace, ' ', spaces_to_replace)
        transition_probs = calculate_transition_matrix(war_and_peace_text, symbol_list)

        # plot
        plt.figure(figsize=(14, 14))
        sns.heatmap(pd.DataFrame(transition_probs, index=symbol_list, columns=symbol_list), annot=False, cmap='YlGnBu')
        plt.title('Symbol Transition Heatmap')
        plt.savefig('transition_heatmap.jpg', dpi=300)
        plt.show()
```

```
/var/folders/bt/n98c3tq15s3d6866b0l_z_400000gn/T/ipykernel_1203/1124478417.py:36: RuntimeWarning: invalid value enc
ountered in divide
  probability_matrix = transition_matrix / row_sums[:, np.newaxis]
```

Figure 44: Code written plotting the Symbol Transition Map in Q5a

```
In [ ]: #estimate stationary distribution
```

```
In [3]: # Calculate the stationary distribution
        eigenvalues, eigenvectors = np.linalg.eig(transition_probs.T)  # Note the transpose of the transition matrix
        stationary_distribution = np.abs(eigenvectors[:, np.isclose(eigenvalues, 1)].flatten().real)
        stationary_distribution /= stationary_distribution.sum()  # Normalize to sum to 1

        # Visualize the stationary distribution
        plt.figure(figsize=(14, 5))
        sns.barplot(x=symbol_list, y=stationary_distribution)
        plt.title('Stationary Distribution of Symbols')
        plt.ylabel('Probability')
        plt.xticks(rotation=90)  # Rotate the x labels for better readability
        plt.savefig('stationary_distribution.jpg', dpi=300, bbox_inches='tight')
        plt.show()
```

```
/Users/anabelyong/miniconda3/envs/sl/lib/python3.11/site-packages/seaborn/_oldcore.py:1498: FutureWarning: is_categ
orical_dtype is deprecated and will be removed in a future version. Use isinstance(dtype, CategoricalDtype) instead
  if pd.api.types.is_categorical_dtype(vector):
/Users/anabelyong/miniconda3/envs/sl/lib/python3.11/site-packages/seaborn/_oldcore.py:1498: FutureWarning: is_categ
orical_dtype is deprecated and will be removed in a future version. Use isinstance(dtype, CategoricalDtype) instead
  if pd.api.types.is_categorical_dtype(vector):
/Users/anabelyong/miniconda3/envs/sl/lib/python3.11/site-packages/seaborn/_oldcore.py:1765: FutureWarning: unique w
ith argument that is not not a Series, Index, ExtensionArray, or np.ndarray is deprecated and will raise in a futur
e version.
  order = pd.unique(vector)
/Users/anabelyong/miniconda3/envs/sl/lib/python3.11/site-packages/seaborn/_oldcore.py:1498: FutureWarning: is_categ
orical_dtype is deprecated and will be removed in a future version. Use isinstance(dtype, CategoricalDtype) instead
  if pd.api.types.is_categorical_dtype(vector):
```

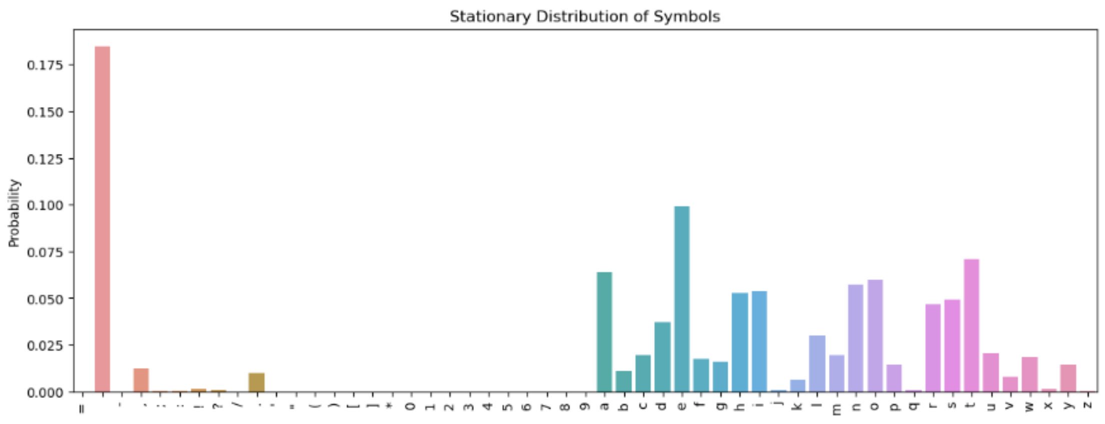
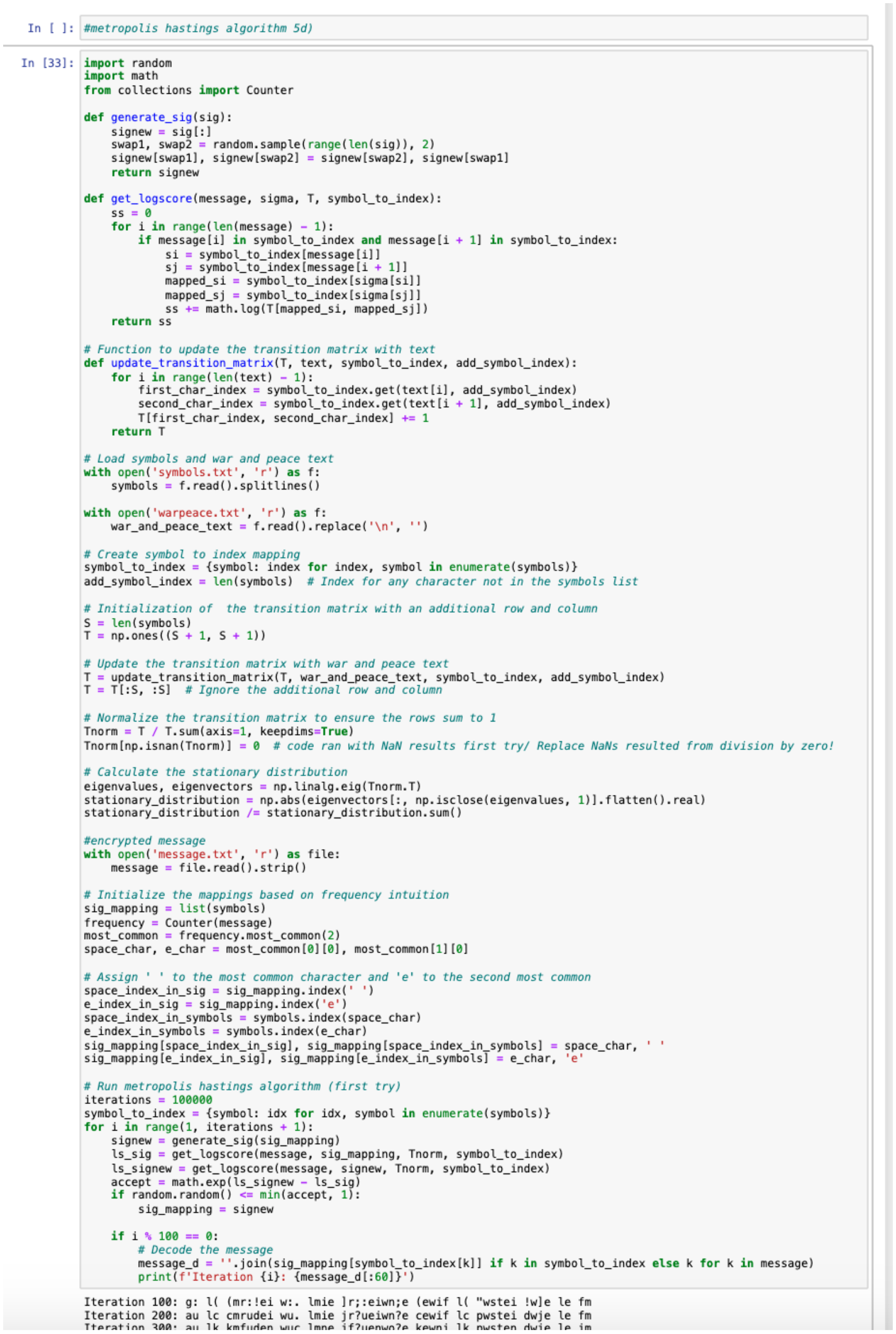


Figure 45: Code written computing stationary distribution for all the symbols for 5a) from mathematical derivation in assignment 5a)

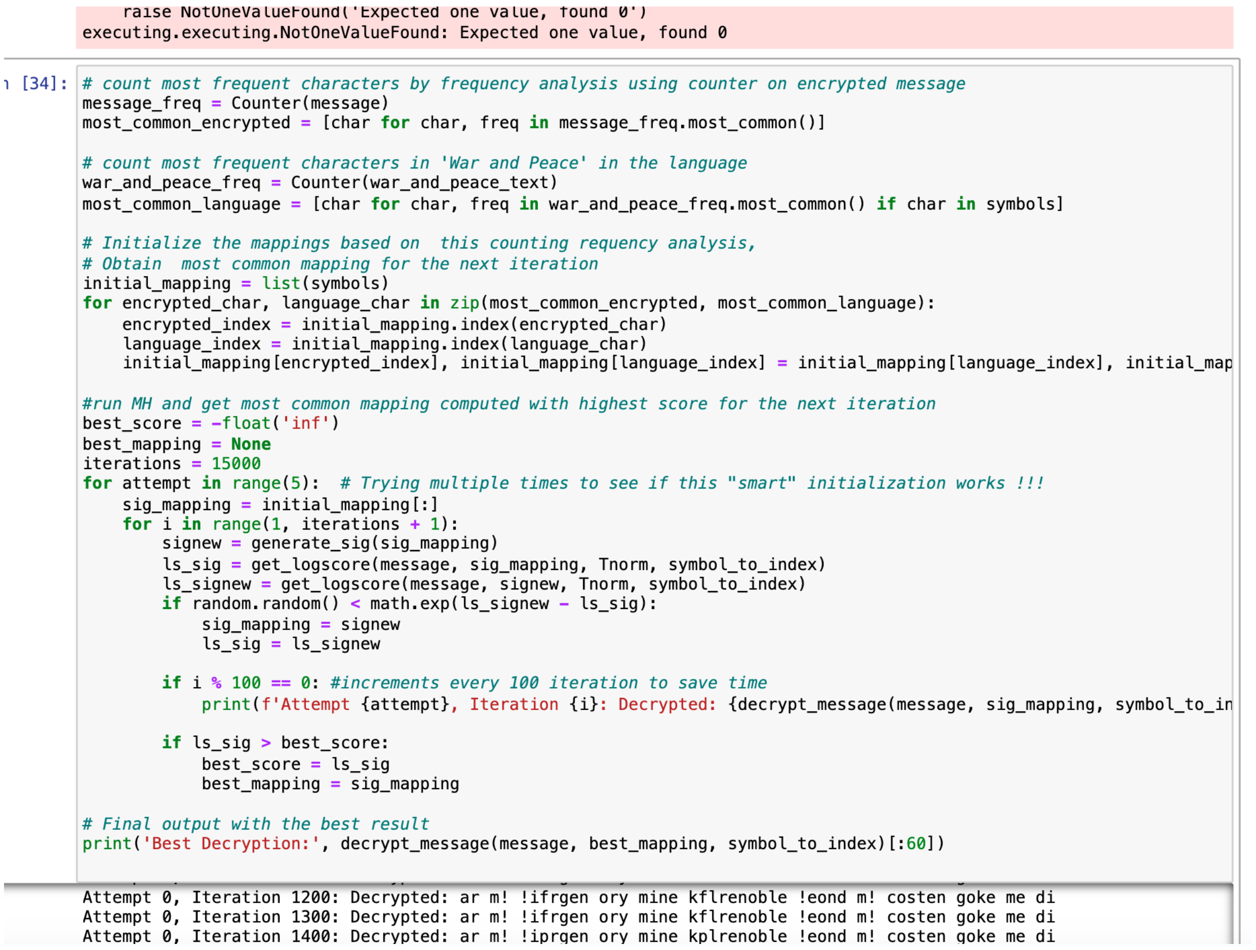


```
In [ ]: #metropolis hastings algorithm 5d)
```

```
In [33]: import random
import math
from collections import Counter

def generate_sig(sig):
    signew = sig[:]
    swap1, swap2 = random.sample(range(len(sig)), 2)
    signew[swap1], signew[swap2] = signew[swap2], signew[swap1]
    return signew

def get_logscore(message, sigma, T, symbol_to_index):
    ss = 0
    for i in range(len(message) - 1):
        if message[i] in symbol_to_index and message[i + 1] in symbol_to_index:
            si = symbol_to_index[message[i]]
            sj = symbol_to_index[message[i + 1]]
            mapped_si = symbol_to_index[sigma[si]]
            mapped_sj = symbol_to_index[sigma[sj]]
            ss += math.log(T[mapped_si, mapped_sj])
    return ss

# Function to update the transition matrix with text
def update_transition_matrix(T, text, symbol_to_index, add_symbol_index):
    for i in range(len(text) - 1):
        first_char_index = symbol_to_index.get(text[i], add_symbol_index)
        second_char_index = symbol_to_index.get(text[i + 1], add_symbol_index)
        T[first_char_index, second_char_index] += 1
    return T

# Load symbols and war and peace text
with open('symbols.txt', 'r') as f:
    symbols = f.read().splitlines()

with open('warpeace.txt', 'r') as f:
    war_and_peace_text = f.read().replace('\n', '')

# Create symbol to index mapping
symbol_to_index = {symbol: index for index, symbol in enumerate(symbols)}
add_symbol_index = len(symbols)  # Index for any character not in the symbols list

# Initialization of  the transition matrix with an additional row and column
S = len(symbols)
T = np.ones((S + 1, S + 1))

# Update the transition matrix with war and peace text
T = update_transition_matrix(T, war_and_peace_text, symbol_to_index, add_symbol_index)
T = T[:S, :S]  # Ignore the additional row and column

# Normalize the transition matrix to ensure the rows sum to 1
Tnorm = T / T.sum(axis=1, keepdims=True)
Tnorm[np.isnan(Tnorm)] = 0  # code ran with NaN results first try/ Replace NaNs resulted from division by zero!

# Calculate the stationary distribution
eigenvalues, eigenvectors = np.linalg.eig(Tnorm.T)
stationary_distribution = np.abs(eigenvectors[:, np.isclose(eigenvalues, 1)].flatten().real)
stationary_distribution /= stationary_distribution.sum()

#encrypted message
with open('message.txt', 'r') as file:
    message = file.read().strip()

# Initialize the mappings based on frequency intuition
sig_mapping = list(symbols)
frequency = Counter(message)
most_common = frequency.most_common(2)
space_char, e_char = most_common[0][0], most_common[1][0]

# Assign ' ' to the most common character and 'e' to the second most common
space_index_in_sig = sig_mapping.index(' ')
e_index_in_sig = sig_mapping.index('e')
space_index_in_symbols = symbols.index(space_char)
e_index_in_symbols = symbols.index(e_char)
sig_mapping[space_index_in_sig], sig_mapping[space_index_in_symbols] = space_char, ' '
sig_mapping[e_index_in_sig], sig_mapping[e_index_in_symbols] = e_char, 'e'

# Run metropolis hastings algorithm (first try)
iterations = 100000
symbol_to_index = {symbol: idx for idx, symbol in enumerate(symbols)}
for i in range(1, iterations + 1):
    signew = generate_sig(sig_mapping)
    ls_sig = get_logscore(message, sig_mapping, Tnorm, symbol_to_index)
    ls_signew = get_logscore(message, signew, Tnorm, symbol_to_index)
    accept = math.exp(ls_signew - ls_sig)
    if random.random() <= min(accept, 1):
        sig_mapping = signew

    if i % 100 == 0:
        # Decode the message
        message_d = ''.join(sig_mapping[symbol_to_index[k]] if k in symbol_to_index else k for k in message)
        print(f'Iteration {i}: {message_d[:60]}')
```

```
Iteration 100: g: l( (mr:!ei w:. lmie ]r;:eiwn;e (ewif l( "wstei !w]e le fm
Iteration 200: au lc cmrudei wu. lmie jr?ueiwn?e cewif lc pwstei dwje le fm
Iteration 300: au lk kmfuden wuc lmne jf?uenwo?e kewni lk pwsten dwje le im
```

Figure 46: Code written for Metropolis Hastings algorithm implementation



```
    raise NotOneValueFound('Expected one value, found 0')
executing.executing.NotOneValueFound: Expected one value, found 0
```

```
n [34]: # count most frequent characters by frequency analysis using counter on encrypted message
message_freq = Counter(message)
most_common_encrypted = [char for char, freq in message_freq.most_common()]

# count most frequent characters in 'War and Peace' in the language
war_and_peace_freq = Counter(war_and_peace_text)
most_common_language = [char for char, freq in war_and_peace_freq.most_common() if char in symbols]

# Initialize the mappings based on  this counting requency analysis,
# Obtain  most common mapping for the next iteration
initial_mapping = list(symbols)
for encrypted_char, language_char in zip(most_common_encrypted, most_common_language):
    encrypted_index = initial_mapping.index(encrypted_char)
    language_index = initial_mapping.index(language_char)
    initial_mapping[encrypted_index], initial_mapping[language_index] = initial_mapping[language_index], initial_map

#run MH and get most common mapping computed with highest score for the next iteration
best_score = -float('inf')
best_mapping = None
iterations = 15000
for attempt in range(5):  # Trying multiple times to see if this "smart" initialization works !!!
    sig_mapping = initial_mapping[:]
    for i in range(1, iterations + 1):
        signew = generate_sig(sig_mapping)
        ls_sig = get_logscore(message, sig_mapping, Tnorm, symbol_to_index)
        ls_signew = get_logscore(message, signew, Tnorm, symbol_to_index)
        if random.random() < math.exp(ls_signew - ls_sig):
            sig_mapping = signew
            ls_sig = ls_signew

        if i % 100 == 0: #increments every 100 iteration to save time
            print(f'Attempt {attempt}, Iteration {i}: Decrypted: {decrypt_message(message, sig_mapping, symbol_to_in

        if ls_sig > best_score:
            best_score = ls_sig
            best_mapping = sig_mapping

# Final output with the best result
print('Best Decryption:', decrypt_message(message, best_mapping, symbol_to_index)[:60])
```

```
Attempt 0, Iteration 1200: Decrypted: ar m! !ifrgen ory mine kflrenoble !eond m! costen goke me di
Attempt 0, Iteration 1300: Decrypted: ar m! !ifrgen ory mine kflrenoble !eond m! costen goke me di
Attempt 0, Iteration 1400: Decrypted: ar m! !iprgen ory mine kplrenoble !eond m! costen goke me di
```

Figure 47: Code written for Metropolis Hastings algorithm implementation- smart initialization

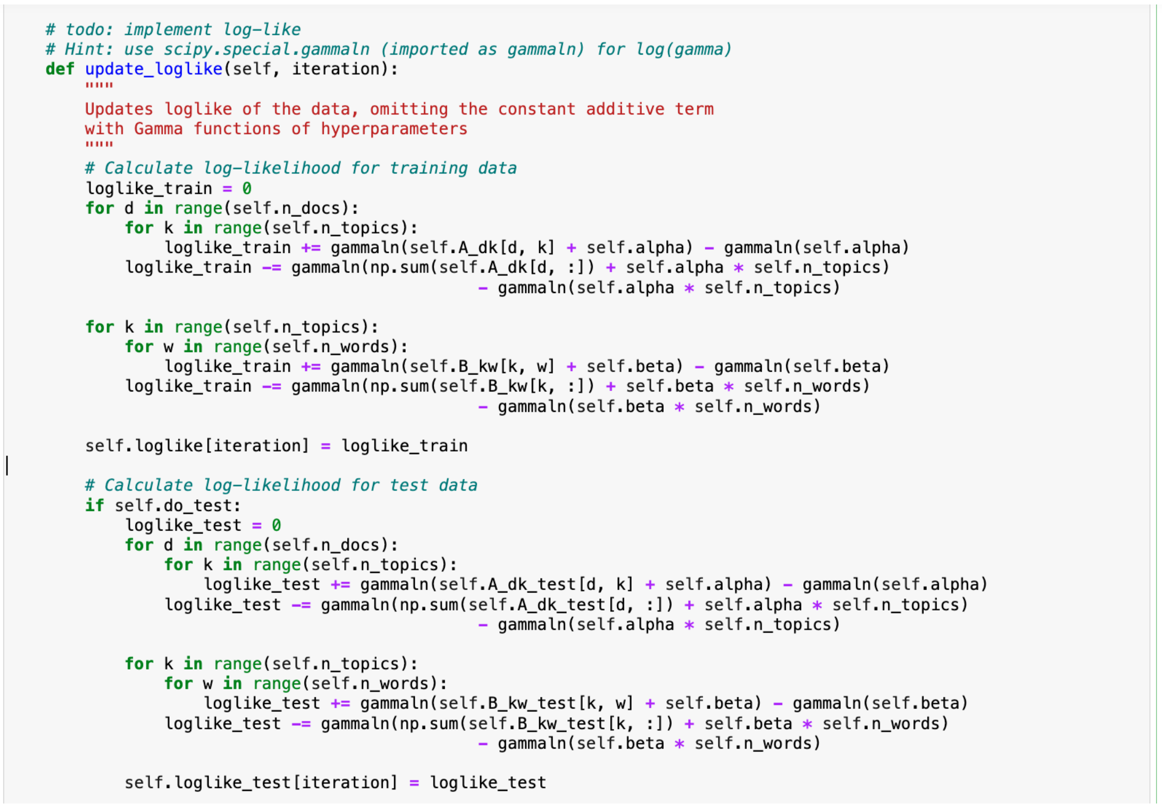
Question 6: Code

```
# todo: sample a topic for each (doc, word) and update A_dk, B_kw correspondingly
def sample_counts(self):
    """
    For each document and each word, samples from z_id|x_id, theta, phi
    and adds the results to the counts A_dk and B_kw
    """
    self.A_dk.fill(0)
    self.B_kw.fill(0)
    if self.do_test:
        self.A_dk_test.fill(0)
        self.B_kw_test.fill(0)

    for d in range(self.n_docs):
        for w in range(self.n_words):
            # Sample topics for each word in each document
            word_count = self.docs_words[d, w]
            if word_count > 0:
                topic_distribution = self.topic_doc_words_distr[:, d, w]
                sampled_topic = self.rand_gen.choice(self.n_topics, p=topic_distribution)
                self.A_dk[d, sampled_topic] += word_count
                self.B_kw[sampled_topic, w] += word_count

            if self.do_test:
                # Repeat sampling for test data
                word_count_test = self.docs_words_test[d, w]
                if word_count_test > 0:
                    sampled_topic_test = self.rand_gen.choice(self.n_topics, p=topic_distribution)
                    self.A_dk_test[d, sampled_topic_test] += word_count_test
                    self.B_kw_test[sampled_topic_test, w] += word_count_test
```

Figure 48: Code written for First Todo: **Sample a topic for each (doc, word) and update A_dk, B_kw correspondingly** in Standard Gibbs Sampling Python Code

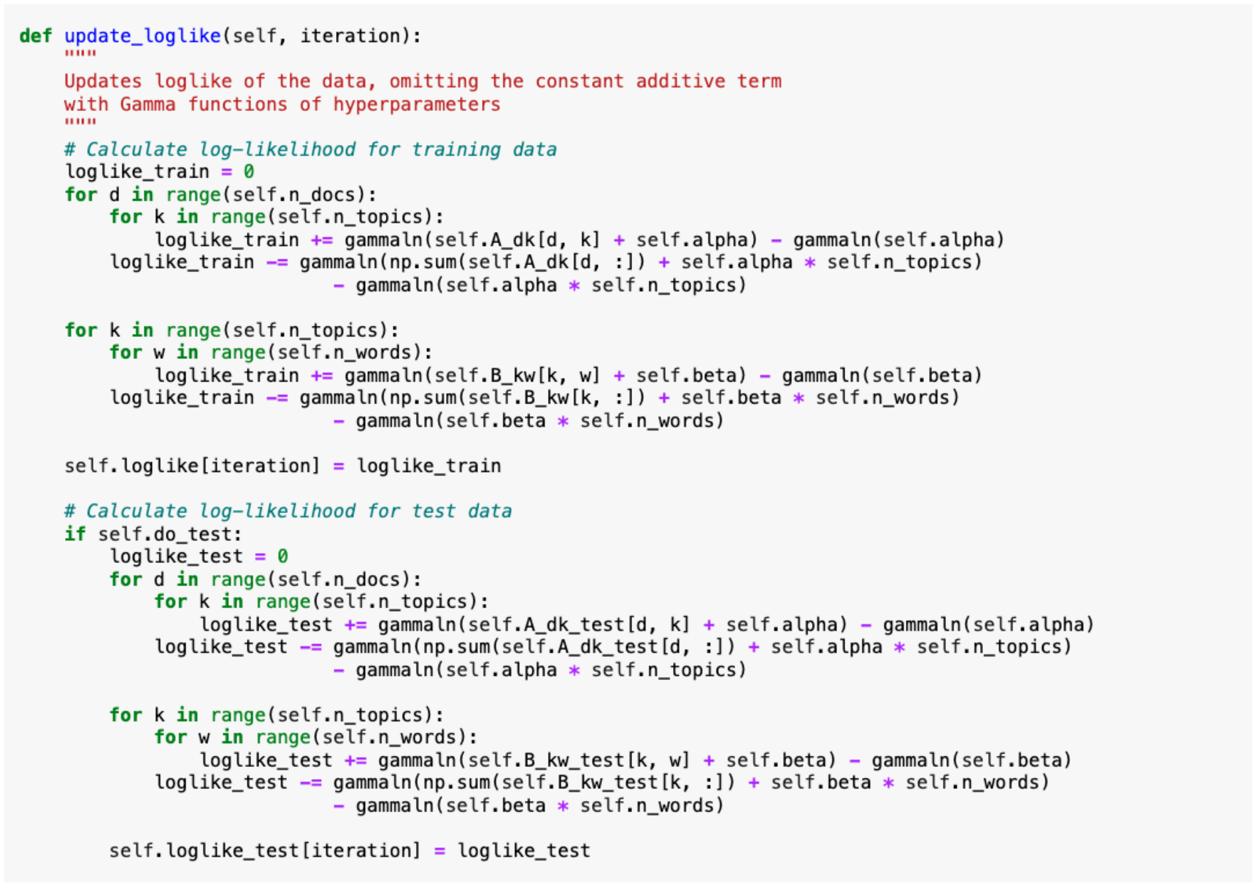
```
# todo: sample everything from self.rang_gen to control the random seed (works as numpy.random)
# todo: sample theta and phi
def update_params(self):
    """
    Samples theta and phi, then computes the distribution of
    z_id and samples counts A_dk, B_kw from it
    """
    # Sample new theta values
    for d in range(self.n_docs):
        self.theta[d, :] = self.rand_gen.dirichlet(self.A_dk[d, :] + self.alpha)

    # Sample new phi values
    for k in range(self.n_topics):
        self.phi[k, :] = self.rand_gen.dirichlet(self.B_kw[k, :] + self.beta)

    self.update_topic_doc_words()
    self.sample_counts()
```

Figure 49: Code written for for Second Todo: **Sample everything from self.rang_gen to control the random seed (works as numpy.random). Third Todo: sample theta and phi** in Standard Gibbs Sampling Python Code



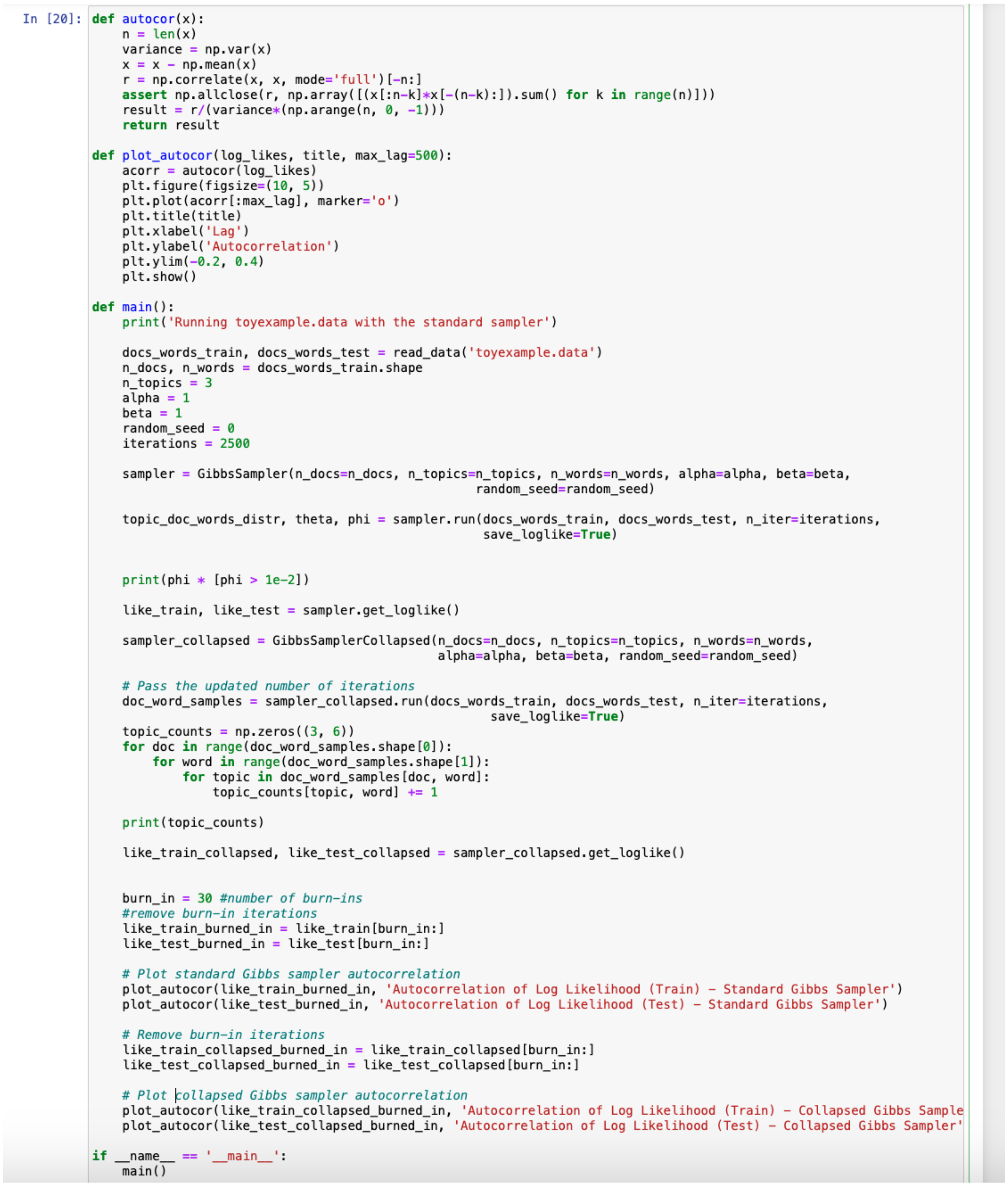
```python
    # todo: implement log-like
    # Hint: use scipy.special.gammaln (imported as gammaln) for log(gamma)
    def update_loglike(self, iteration):
        """
        Updates loglike of the data, omitting the constant additive term
        with Gamma functions of hyperparameters
        """
        # Calculate log-likelihood for training data
        loglike_train = 0
        for d in range(self.n_docs):
            for k in range(self.n_topics):
                loglike_train += gammaln(self.A_dk[d, k] + self.alpha) - gammaln(self.alpha)
            loglike_train -= gammaln(np.sum(self.A_dk[d, :]) + self.alpha * self.n_topics)
                                    - gammaln(self.alpha * self.n_topics)

        for k in range(self.n_topics):
            for w in range(self.n_words):
                loglike_train += gammaln(self.B_kw[k, w] + self.beta) - gammaln(self.beta)
            loglike_train -= gammaln(np.sum(self.B_kw[k, :]) + self.beta * self.n_words)
                                    - gammaln(self.beta * self.n_words)

        self.loglike[iteration] = loglike_train

        # Calculate log-likelihood for test data
        if self.do_test:
            loglike_test = 0
            for d in range(self.n_docs):
                for k in range(self.n_topics):
                    loglike_test += gammaln(self.A_dk_test[d, k] + self.alpha) - gammaln(self.alpha)
                loglike_test -= gammaln(np.sum(self.A_dk_test[d, :]) + self.alpha * self.n_topics)
                                    - gammaln(self.alpha * self.n_topics)

            for k in range(self.n_topics):
                for w in range(self.n_words):
                    loglike_test += gammaln(self.B_kw_test[k, w] + self.beta) - gammaln(self.beta)
                loglike_test -= gammaln(np.sum(self.B_kw_test[k, :]) + self.beta * self.n_words)
                                    - gammaln(self.beta * self.n_words)

            self.loglike_test[iteration] = loglike_test
```

Figure 50: Code written for for 4th Todo: **Implement Log-like** in Standard Gibbs Sampling Python Code



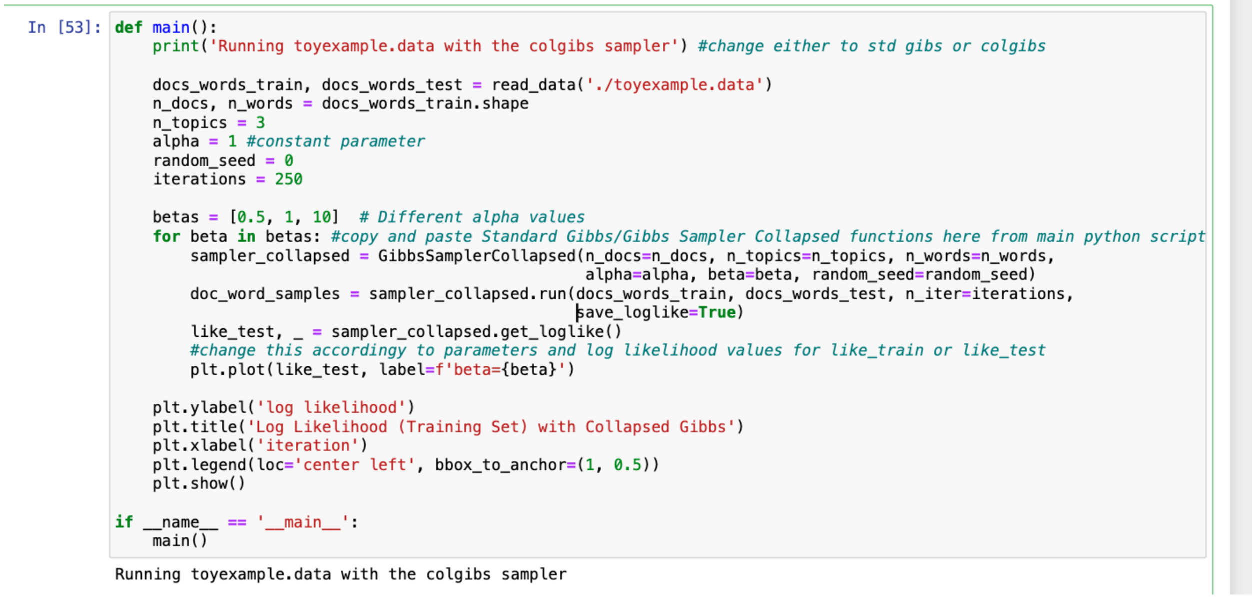
```python
def update_loglike(self, iteration):
    """
    Updates loglike of the data, omitting the constant additive term
    with Gamma functions of hyperparameters
    """
    # Calculate log-likelihood for training data
    loglike_train = 0
    for d in range(self.n_docs):
        for k in range(self.n_topics):
            loglike_train += gammaln(self.A_dk[d, k] + self.alpha) - gammaln(self.alpha)
        loglike_train -= gammaln(np.sum(self.A_dk[d, :]) + self.alpha * self.n_topics)
                         - gammaln(self.alpha * self.n_topics)

    for k in range(self.n_topics):
        for w in range(self.n_words):
            loglike_train += gammaln(self.B_kw[k, w] + self.beta) - gammaln(self.beta)
        loglike_train -= gammaln(np.sum(self.B_kw[k, :]) + self.beta * self.n_words)
                         - gammaln(self.beta * self.n_words)

    self.loglike[iteration] = loglike_train

    # Calculate log-likelihood for test data
    if self.do_test:
        loglike_test = 0
        for d in range(self.n_docs):
            for k in range(self.n_topics):
                loglike_test += gammaln(self.A_dk_test[d, k] + self.alpha) - gammaln(self.alpha)
            loglike_test -= gammaln(np.sum(self.A_dk_test[d, :]) + self.alpha * self.n_topics)
                         - gammaln(self.alpha * self.n_topics)

        for k in range(self.n_topics):
            for w in range(self.n_words):
                loglike_test += gammaln(self.B_kw_test[k, w] + self.beta) - gammaln(self.beta)
            loglike_test -= gammaln(np.sum(self.B_kw_test[k, :]) + self.beta * self.n_words)
                         - gammaln(self.beta * self.n_words)

        self.loglike_test[iteration] = loglike_test
```

Figure 51: Code written for for 4th Todo: **Implement Log-like** in Collapsed Gibbs Sampling Python Code



```
In [20]: def autocor(x):
             n = len(x)
             variance = np.var(x)
             x = x - np.mean(x)
             r = np.correlate(x, x, mode='full')[-n:]
             assert np.allclose(r, np.array([(x[:n-k]*x[-(n-k):]).sum() for k in range(n)]))
             result = r/(variance*(np.arange(n, 0, -1)))
             return result

         def plot_autocor(log_likes, title, max_lag=500):
             acorr = autocor(log_likes)
             plt.figure(figsize=(10, 5))
             plt.plot(acorr[:max_lag], marker='o')
             plt.title(title)
             plt.xlabel('Lag')
             plt.ylabel('Autocorrelation')
             plt.ylim(-0.2, 0.4)
             plt.show()

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

             docs_words_train, docs_words_test = read_data('toyexample.data')
             n_docs, n_words = docs_words_train.shape
             n_topics = 3
             alpha = 1
             beta = 1
             random_seed = 0
             iterations = 2500

             sampler = GibbsSampler(n_docs=n_docs, n_topics=n_topics, n_words=n_words, alpha=alpha, beta=beta,
                                                       random_seed=random_seed)

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


             print(phi * [phi > 1e-2])

             like_train, like_test = sampler.get_loglike()

             sampler_collapsed = GibbsSamplerCollapsed(n_docs=n_docs, n_topics=n_topics, n_words=n_words,
                                                  alpha=alpha, beta=beta, random_seed=random_seed)

             # Pass the updated number of iterations
             doc_word_samples = sampler_collapsed.run(docs_words_train, docs_words_test, n_iter=iterations,
                                                       save_loglike=True)
             topic_counts = np.zeros((3, 6))
             for doc in range(doc_word_samples.shape[0]):
                 for word in range(doc_word_samples.shape[1]):
                     for topic in doc_word_samples[doc, word]:
                         topic_counts[topic, word] += 1

             print(topic_counts)

             like_train_collapsed, like_test_collapsed = sampler_collapsed.get_loglike()


             burn_in = 30 #number of burn-ins
             #remove burn-in iterations
             like_train_burned_in = like_train[burn_in:]
             like_test_burned_in = like_test[burn_in:]

             # Plot standard Gibbs sampler autocorrelation
             plot_autocor(like_train_burned_in, 'Autocorrelation of Log Likelihood (Train) - Standard Gibbs Sampler')
             plot_autocor(like_test_burned_in, 'Autocorrelation of Log Likelihood (Test) - Standard Gibbs Sampler')

             # Remove burn-in iterations
             like_train_collapsed_burned_in = like_train_collapsed[burn_in:]
             like_test_collapsed_burned_in = like_test_collapsed[burn_in:]

             # Plot collapsed Gibbs sampler autocorrelation
             plot_autocor(like_train_collapsed_burned_in, 'Autocorrelation of Log Likelihood (Train) - Collapsed Gibbs Sample
             plot_autocor(like_test_collapsed_burned_in, 'Autocorrelation of Log Likelihood (Test) - Collapsed Gibbs Sampler'

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

Figure 52: Code written for 6b: Autocorrelation plots



```
In [53]: def main():
             print('Running toyexample.data with the colgibs sampler') #change either to std gibs or colgibs

             docs_words_train, docs_words_test = read_data('./toyexample.data')
             n_docs, n_words = docs_words_train.shape
             n_topics = 3
             alpha = 1 #constant parameter
             random_seed = 0
             iterations = 250

             betas = [0.5, 1, 10]  # Different alpha values
             for beta in betas: #copy and paste Standard Gibbs/Gibbs Sampler Collapsed functions here from main python script
                 sampler_collapsed = GibbsSamplerCollapsed(n_docs=n_docs, n_topics=n_topics, n_words=n_words,
                                                           alpha=alpha, beta=beta, random_seed=random_seed)
                 doc_word_samples = sampler_collapsed.run(docs_words_train, docs_words_test, n_iter=iterations,
                                                          save_loglike=True)
                 like_test, _ = sampler_collapsed.get_loglike()
                 #change this accordingy to parameters and log likelihood values for like_train or like_test
                 plt.plot(like_test, label=f'beta={beta}')

             plt.ylabel('log likelihood')
             plt.title('Log Likelihood (Training Set) with Collapsed Gibbs')
             plt.xlabel('iteration')
             plt.legend(loc='center left', bbox_to_anchor=(1, 0.5))
             plt.show()

         if __name__ == '__main__':
             main()

         Running toyexample.data with the colgibs sampler
```

Figure 53: Code written for 6d: Code for modifying different parameters α, β, K and plotting log-likelihood