COMP0086 : Summative Assessment

University College London

13 November 2024

Table of Contents

1 | Question 1 - Models for Binary Vectors

1.1 Question 1.(a)

We dispose of a data set of N binary images. Each of these images has a finite number of D discrete pixels, each taking a value 0 or 1, but not in between. The Multivariate Gaussian distribution is defined on the domain $x \in \mathbb{R}^k$, and its data type is continuous and unbounded. Compared to our data which is discrete and bounded ($x_d^{(n)} \in \{0, 1\}$), then the Multivariate Gaussian distribution is not suited for the data set of images.

1.2 Question 1.(b)

We assume the images were modelled as independently and identically distributed (**iid**) samples from a D-dimensional multivariate Bernoulli distribution with parameter vector $\mathbf{p} = (p_1, ..., p_D)$:

$$P(\mathbf{x^{(n)}}|\mathbf{p}) = \prod_{d=1}^{D} p_d^{x_d^{(n)}} (1 - p_d)^{1 - x_d^{(n)}} \qquad n \in 1, ..., N \tag{1.1}$$

where $\mathbf{x}$, $\mathbf{p}$ are D-dimensional vectors.

The equation for the likelihood of $\mathbf{p}$ is calculated as:

$$P(\mathbf{x}|\mathbf{p}) = \prod_{n=1}^{N} P(\mathbf{x^{(n)}}|\mathbf{p}) = \prod_{n=1}^{N} \prod_{d=1}^{D} p_d^{x_d^{(n)}} (1 - p_d)^{1 - x_d^{(n)}} \tag{1.2}$$

To obtain the log-likelihood, we take the log of the expression, yielding:

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

$$\mathcal{L} = \sum_{n=1}^{N} \sum_{d=1}^{D} p_d^{x_d^{(n)}} (1 - p_d)^{1 - x_d^{(n)}}$$

$$\mathcal{L} = \sum_{n=1}^{N} \sum_{d=1}^{D} x_d^{(n)} \log p_d + (1 - x_d^{(n)}) \log (1 - p_d) \tag{1.3}$$

To evaluate the Maximum Likelihood, we need to find the maximum of that function:

$$\frac{\partial \mathcal{L}(p)}{\partial p_d} = 0$$

Equivalent to:

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

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

Which is achieved when the numerator is equal to 0:

$$\sum_{n=1}^{N} x_d^{(n)}(1-p_d)-p_d(1-x^{(n)})=0$$

$$\sum_{n=1}^{N}(x_d^{(n)}-p_d)=0$$

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

And since p_d does not depend on N, we obtain the Maximum Likelihood (ML) estimation of p_d, denoted as $\hat{p}_d^{ML}$:

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

$$\hat{p}_d^{ML}=\frac{1}{N}\sum_{n=1}^{N} x^{(n)} \tag{1.4}$$

Generalizing for all values of d since the pixels are assumed to be independent, we get:

$$\hat{\mathbf{p}}^{\mathbf{ML}}=\frac{1}{N}\sum_{n=1}^{N}\mathbf{x}^{(n)} \tag{1.5}$$

1.3 Question 1.(c)

We assume independent Beta priors on the parameters p_d:

$$P(p_d)=\frac{1}{B(\alpha,\beta)}p_d^{\alpha-1}(1-p_d)^{\beta-1} \tag{1.6}$$

And:

$$P(\mathbf{p})=\prod_d P(p_d) \tag{1.7}$$

To find the MAP estimator of p, we firstly use Bayes Rule:

$$P(\mathbf{p}|\mathbf{x})=\frac{P(\mathbf{x}|\mathbf{p})P(\mathbf{p})}{P(\mathbf{x})} \tag{1.8}$$

Combining the expressions, we get:

$$P(\mathbf{p}|\mathbf{x})=\frac{\prod_{n=1}^{N}\prod_{d=1}^{D} p_d^{x_d^{(n)}}(1-p_d)^{1-x_d^{(n)}}\prod_{d=1}^{D} P(p_d)}{P(\mathbf{x})} \tag{1.9}$$

The expression is hard to compute, but, since P(x) does not depend on d, we can state the following:

$$P(\mathbf{p}|\mathbf{x}) \propto p_d^{x_d^{(n)}} (1-p_d)^{1-x_d^{(n)}} \tag{1.10}$$

Taking the log of the expression:

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

$$+ \sum_{d=1}^{D} (\log(p_d)(\alpha - 1) + (\beta - 1) \log (1 - p_d)) - D \log B(\alpha, \beta)$$

We need to find the maximum of this expression. Differentiating with respect to p_d, we get:

$$\frac{\partial \log P(\mathbf{p}|\mathbf{x})}{\partial p_d} = \sum_{n=1}^{N} \left(x_d^{(n)} \frac{1}{p_d} + (1 - x_d^{(n)}) \frac{-1}{1 - p_d} \right) + \frac{(\alpha - 1)}{p_d} - \frac{\beta - 1}{1 - p_d} \tag{1.11}$$

Because the Beta function term is independent of d.

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

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

$$\frac{\partial \log P(\mathbf{p}|\mathbf{x})}{\partial p_d} = \frac{\sum_{n=1}^{N} \left(x_d^{(n)} - p_d \right) + \alpha - 1 + p_d(2 - \alpha - \beta)}{p_d(1 - p_d)} \tag{1.12}$$

Making this expression equal to 0:

$$\frac{\partial \log P(\mathbf{p}|\mathbf{x})}{\partial p_d} = 0 \tag{1.13}$$

Equivalent to:

$$\sum_{n=1}^{N} \left(x_d^{(n)} - p_d \right) + \alpha - 1 + p_d(2 - \alpha - \beta) = 0 \tag{1.14}$$

$$\sum_{n=1}^{N} (x_d^{(n)}) - N p_d + \alpha - 1 + p_d(2 - \alpha - \beta) = 0$$

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

$$\hat{p}_d^{MAP} = \frac{\sum_{n=1}^{N} (x_d^{(n)}) + \alpha - 1}{N + \alpha + \beta - 2} \tag{1.15}$$

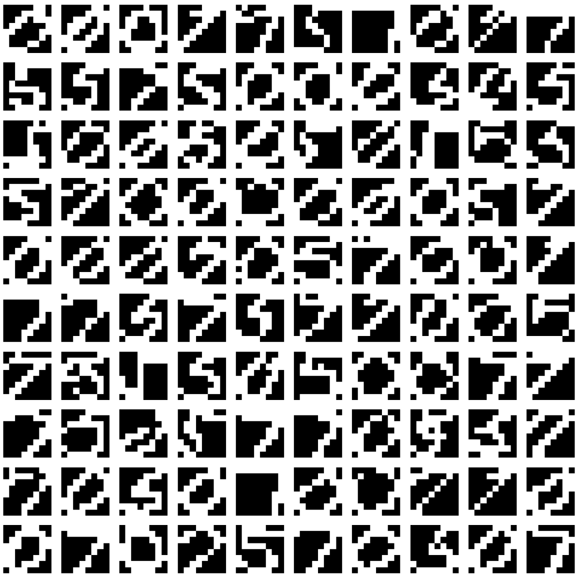


Figure 1.1: Original data from binarydigits.txt, used for training

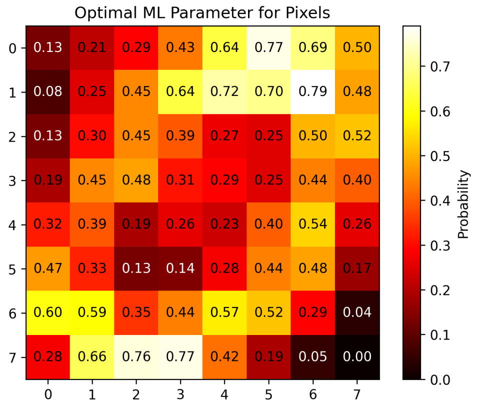
And thus, generalizing for **p**, we obtain the Maximum A Posteriori (MAP) expression of **p**:

$$\hat{\mathbf{p}}^{MAP} = \frac{\sum_{n=1}^{N}(x^{(n)}) + \alpha - 1}{N + \alpha + \beta - 2} \tag{1.16}$$

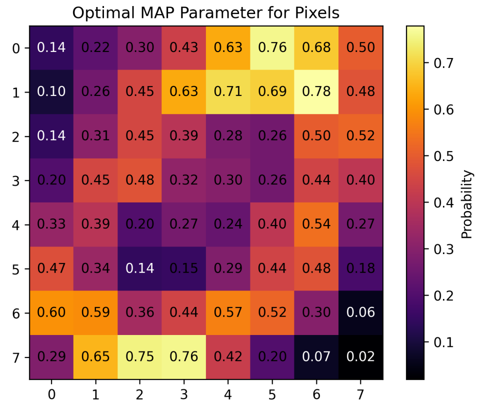
1.4 Question 1.(d)

The original data contains $N = 100$ image, each made of $D = 64$ pixels and stored in an $N \times D$ matrix. Rearranging the pixels, Figure 1.1 shows the original data from the binarydigits.txt file, and displaying them as an 8×8 image, which was displayed by readapting the code from bindigit.py.

After executing the code to learn the ML parameters of the multivariate Bernoulli from the dataset, the obtained ML parameters are shown in Figure 1.2(a). Those parameters are displayed as an 8×8 image. The code for the execution can be found below.



((a)) ML Parameters calculated



((b)) MAP Parameters calculated

Figure 1.2: ML and MAP Parameters

Listing 1: Code - Q1.(d)

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

# Loading the training data
def load_binarydigits(filename='binarydigits.txt'):
    Y = np.loadtxt('binarydigits.txt')
    return Y

# Displaying the original training data
def save_data_binary(Y: np.ndarray, save=False):
    N, _ = Y.shape
    plt.figure(figsize=(5, 5))
    for n in range(N):
        plt.subplot(10, 10, n+1)
        plt.imshow(np.reshape(Y[n, :], (8,8)),
                   interpolation="None",
                   cmap='gray')
        plt.axis('off')
    if save:
            plt.savefig('data_binary.png', format="png", dpi=300,
            bbox_inches="tight")
    plt.show()

# ML Parameter Learning
def ML_learning(Y: np.ndarray, save_ML = False):
    N, D = Y.shape
    p_ML = np.zeros((D, 1), dtype=np.float64)

    for d in range(D):
        p_ML[d] = (1/N)*np.sum(Y[:, d])

    p_ML_image = np.reshape(p_ML, (8,8))
    plt.figure()
    plt.imshow(p_ML_image, cmap="hot", interpolation='nearest')
    plt.colorbar(label='Probability')
    plt.title('Optimal ML Parameter for Pixels')
    for i in range(8):
        for j in range(8):
            if p_ML_image[i,j] > 0.15:
                plt.text(j, i, f"{p_ML_image[i, j]:.2f}", ha='center',
                va='center', color="black")
            else:
                plt.text(j, i, f"{p_ML_image[i, j]:.2f}", ha='center',
                va='center', color="white")
    if save_ML:
        plt.savefig('ML_parameter.png', format="png", dpi=300, bbox_inches="tight")
    plt.show()
```

```python

    return p_ML_image, p_ML

def main():
    Y = load_binarydigits()
    save_data_binary(Y)
    p_ML_image, p_ML = ML_learning(Y)

# Executing the main function
main()
```

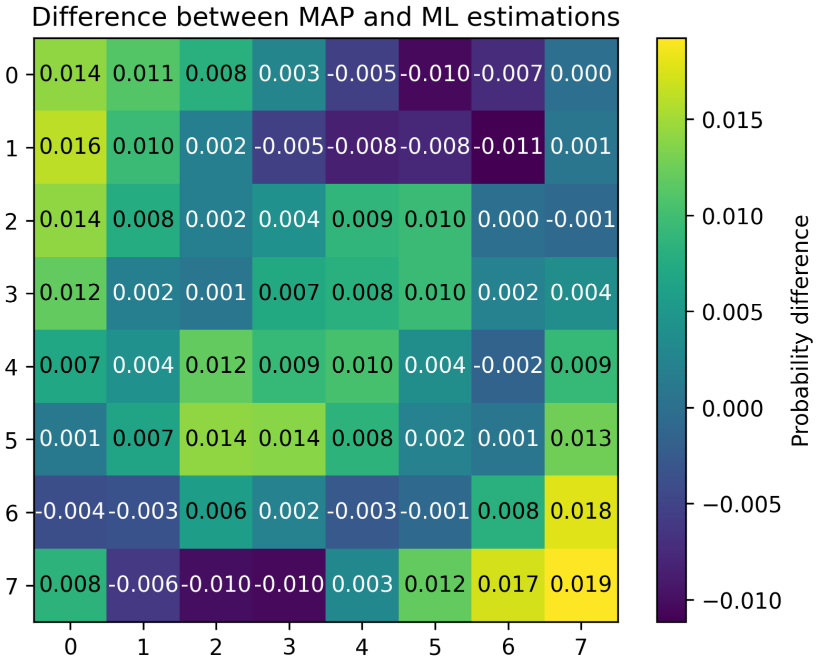


Figure 1.3: Substracting ML parameters to MAP parameters: $\hat{\mathbf{p}}^{MAP} - \hat{\mathbf{p}}^{ML}$

1.5 Question 1.(e)

Modifying the code built for the previous question, we obtain the MAP parameters displayed in Figure 1.2(b), which appear different if we look closely to the displayed numbers on each tile and compare them with that of the ML parameters. For the difference to be more visible, Figure 1.3 shows the difference between MAP and ML parameters, calculated as $\hat{\mathbf{p}}^{MAP} - \hat{\mathbf{p}}^{ML}$[1]. It provides a better visual of the difference between these two estimations.
The new code is displayed below.
The new learned parameters are better than the ML estimate due to the limited amount of data. If we had more data, the MAP estimate would yield the same result as the ML estimate. Having the MAP estimate permits the model to not assume some pixels are 1's and others are 0's by defaults, because it takes into account what it has seen on the prior.

[1] Why the difference, and not the quotient: some entries are 0, the computer will display an error in such case.

2 | Question 2 - Model selection

We would like to find the expressions needed to calculate the relative probabilities of three different models regarding our binary data images.

2.1 Question 2.(a)

For this model (named *Model 1*), we assume all D components are generated from a Bernoulli distribution with identical $p_d = 0.5$. Thus, the probability of data x given $p_d = 0.5$ is:

$$P_1(x^n|\mathbf{p} = [p_d, ..., p_d]^T, p_d = 0.5) = \prod_{i=1}^{D} p_d^{x_i^{(n)}} (1-p_d)^{1-x_i^{(n)}} \tag{2.1}$$

$$= \prod_{d=i}^{D} 0.5^{x_i^{(n)}} (1-0.5)^{1-x_i^{(n)}}$$

$$= \prod_{i=1}^{D} 0.5^{x_i^{(n)}+1-x_i^{(n)}}$$

$$= \prod_{i=1}^{D} 0.5$$

$$P_1(x^{(n)}|\mathbf{p} = [p_d, ..., p_d]^T, p_d = 0.5) = 0.5^D \tag{2.2}$$

Thus, for the entire dataset, since all images are considered to be independent and identically sampled (iid):

$$P_1(\mathbf{x}|\mathbf{p} = [p_d, ..., p_d]^T, p_d = 0.5) = \prod_{n=1}^{N} P_1(x^{(n)}|\mathbf{p} = [p_d, ..., p_d]^T, p_d = 0.5) \tag{2.3}$$

$$= \prod_{n=1}^{N} 0.5^D$$

$$P_1(\mathbf{x}|\mathbf{p} = [p_d, ..., p_d]^T, p_d = 0.5) = 0.5^{N\times D} \tag{2.4}$$

For each model, the likelihood of the data $\mathbf{x}$ given the model i is given by:

$$P(\mathbf{x}|\text{Model } i) = \int_0^1 P_i(\mathbf{x}|\mathbf{p})P(\mathbf{p})dp \tag{2.5}$$

Where $P(\mathbf{p})$ is known to be uniform for all unknown probabilities. In the case of Model 1, we know the prior probability and it is equal to 1 (no probability of having something else than $p_d = 0.5$). Therefore:

$$P(\mathbf{x}|\text{Model } 1) = \int_0^1 P_1(\mathbf{x}|\mathbf{p})d\mathbf{p} \tag{2.6}$$

$$= \int_0^1 0.5^{N\times D} d\mathbf{p}$$

$$= 0.5^{N\times D} \int_0^1 d\mathbf{p}$$

$$P(\mathbf{x}|\text{Model 1}) = 0.5^{N\times D} \tag{2.7}$$

2.2 Question 2.(b)

For this model (named *Model 2*), we assume all D components are generated from Bernoulli distributions with unknown but identical p_d. The probability $P_2(x^{(n)}|\mathbf{p} = [p_d, ..., p_d]^T)$ is:

$$P_2(x^{(n)}|\mathbf{p}) = \prod_{i=1}^{D} p_d^{x_i^{(n)}} (1-p_d)^{1-x_i^{(n)}} \tag{2.8}$$

Leading to:

$$P_2(x^{(n)}|\mathbf{p}) = p_d^{\sum_{i=1}^{D} x_i^{(n)}} (1-p_d)^{\sum_{d=1}^{D}(1-x_i^{(n)})} \tag{2.9}$$

N.B. notice we changed the indice of the product, to avoid confusion.
For the entire image dataset, we obtain:

$$P_2(\mathbf{x}|\mathbf{p}) = \prod_{n=1}^{N} P_2(x^{(n)}|\mathbf{p}) \tag{2.10}$$

Which we calculated before:

$$= \prod_{n=1}^{N} p_d^{\sum_{i=1}^{D} x_i^{(n)}} (1-p_d)^{\sum_{d=1}^{D}(1-x_i^{(n)})}$$

$$= p_d^{\sum_{n=1}^{N}\sum_{i=1}^{D} x_i^{(n)}} (1-p_d)^{\sum_{n=1}^{N}\sum_{d=1}^{D}(1-x_i^{(n)})}$$

Thus, as for Model 1, to obtain the likelihood of Model 2:

$$P(\mathbf{x}|\text{Model 2}) = \int_0^1 P_2(\mathbf{x}|\mathbf{p})P(\mathbf{p})d\mathbf{p} \tag{2.11}$$

And since we assumed a uniform prior because p_d is unknown, therefore it becomes:

$$P(\mathbf{x}|\text{Model 2}) = \int_0^1 P_2(\mathbf{x}|\mathbf{p})dp_d \tag{2.12}$$

$$= \int_0^1 p_d^{\sum_{n=1}^{N}\sum_{i=1}^{D} x_i^{(n)}} (1-p_d)^{\sum_{n=1}^{N}\sum_{d=1}^{D}(1-x_i^{(n)})} dp_d$$

And since integrating a Binomial function (parameters over the interval $[0,1]$ is the same as evaluating the Beta function:

$$B(\alpha,\beta) = \int_0^1 p^{\alpha-1}(1-p)^{\beta-1}dp$$

Then, by identification:

$$P(\mathbf{x}|\text{Model 2}) = B\left(1 + \sum_{n=1}^{N}\sum_{i=1}^{D} x_i^{(n)}, 1 + \sum_{n=1}^{N}\sum_{i=1}^{D}(1-x_i^{(n)})\right)$$

Moreover, the Beta function, if its parameters are integers, can be evaluated as:

$$Beta(i,j) = \frac{(i-1)!(j-1)!}{(i+j-1)!}$$

Since $x_i^{(n)}$'s only take 0's or 1's as values, we can deduce their sums will be integers and therefore:

$$P(\mathbf{x}|\text{Model 2}) = \frac{\left(\sum_{n=1}^{N}\sum_{i=1}^{D} x_i^{(n)}\right)!\left(\sum_{n=1}^{N}\sum_{i=1}^{D}(1-x_i^{(n)})\right)!}{\left(\sum_{n=1}^{N}\sum_{i=1}^{D} x_i^{(n)} + \sum_{n=1}^{N}\sum_{i=1}^{D}(1-x_i^{(n)})+1\right)!} \tag{2.13}$$

Finally, we can rapidly develop the second double sum to obtain:

$$P(\mathbf{x}|\text{Model 2}) = \frac{\left(\sum_{n=1}^{N}\sum_{i=1}^{D} x_i^{(n)}\right)!\left(N\times D + \sum_{n=1}^{N}\sum_{i=1}^{D}(-x_i^{(n)})\right)!}{\left(\sum_{n=1}^{N}\sum_{i=1}^{D} x_i^{(n)} + N\times D + \sum_{n=1}^{N}\sum_{i=1}^{D}(-x_i^{(n)})+1\right)!}$$

Leading to the final expression:

$$P(\mathbf{x}|\text{Model 2}) = \frac{\left(\sum_{n=1}^{N}\sum_{i=1}^{D} x_i^{(n)}\right)!\left(N\times D - \sum_{n=1}^{N}\sum_{i=1}^{D}(x_i^{(n)})\right)!}{(N\times D+1)!} \tag{2.14}$$

A note on implementation After implementing and testing the model's probability, the difficulty of implementing large factorial functions (displaying errors due to too intensive computation) led to finally implement them using the log of Beta function formula. This have been done using *betaln()* function from *scipy.special* library.

2.3 Question 2.(c)

We now consider a model (named *Model 3*) with each component being Bernoulli distributed, with a separate and unknown p_d. We therefore have:

$$P_3(x^{(n)}|\mathbf{p}) = \prod_{d=1}^{D} p_d^{x_d^{(n)}}(1-p_d)^{(1-x_d^{(n)})} \tag{2.15}$$

$$P_3(\mathbf{x}|\mathbf{p}) = \prod_{n=1}^{N}\prod_{d=1}^{D} p_d^{x_d^{(n)}}(1-p_d)^{(1-x_d^{(n)})} \tag{2.16}$$

For Model 3's likelihood, it is obtained similarly to that of Model 2:

$$P(\mathbf{x}|\text{Model 3}) = \int_0^1 P_3(\mathbf{x}|\mathbf{p})P(\mathbf{p})d\mathbf{p} \tag{2.17}$$

But since each component has a separate unknown p_d, still with uniform prior, we get:

$$P(\mathbf{x}|\text{Model 3}) = \int_0^1 ... \int_0^1 P_3(\mathbf{x}|\mathbf{p})dp_1...dp_D \tag{2.18}$$

Equivalent to:

$$P(\mathbf{x}|\text{Model 3}) = \int_0^1 ... \int_0^1 \prod_{n=1}^{N} \prod_{d=1}^{D} p_d^{x_d^{(n)}} (1-p_d)^{(1-x_d^{(n)})} dp_1...dp_D$$

Which can be separated into (since all p_d are independent and separate):

$$P(\mathbf{x}|\text{Model 3}) = \prod_{d=1}^{D} \left(\int_0^1 \prod_{n=1}^{N} p_d^{x_d^{(n)}} (1-p_d)^{(1-x_d^{(n)})} dp_d \right)$$

$$= \prod_{d=1}^{D} \left(\int_0^1 p_d^{\sum_{n=1}^{N} x_d^{(n)}} (1-p_d)^{\sum_{n=1}^{N} (1-x_d^{(n)})} dp_d \right)$$

Leading to:

$$P(\mathbf{x}|\text{Model 3}) = \prod_{d=1}^{D} B\left(1 + \sum_{n=1}^{N} x_d^{(n)}, 1 + \sum_{n=1}^{N} (1-x_d^{(n)})\right)$$

$$P(\mathbf{x}|\text{Model 3}) = \prod_{d=1}^{D} \frac{(\sum_{n=1}^{N} x_d^{(n)})!(\sum_{n=1}^{N}(1-x_d^{(n)}))!}{(\sum_{n=1}^{N} x_d^{(n)} + \sum_{n=1}^{N}(1-x_d^{(n)}) + 1)!} \tag{2.19}$$

And, after manipulations, leads us to the final expression:

$$P(\mathbf{x}|\text{Model 3}) = \prod_{d=1}^{D} \frac{(\sum_{n=1}^{N} x_d^{(n)})!(\sum_{n=1}^{N}(1-x_d^{(n)}))!}{(N+1)!}$$

$$P(\mathbf{x}|\text{Model 3}) = \frac{1}{D(N+1)!} \times \prod_{d=1}^{D} \left(\sum_{n=1}^{N} x_d^{(n)}\right)! \left(\sum_{n=1}^{N}(1-x_d^{(n)})\right)! \tag{2.20}$$

A note on implementation Again, similarly to Model 2's implementation, the difficulty of implementing large factorial functions for Model 3 led to finally implement them using the log of Beta function formula. This have been done using *betaln()* function from *scipy.special* library (once again).

2.4 Question 2. Answer

We assume all models 1, 2 and 3 are equally likely *a priori*, and their prior distributions to be uniform for any unknown probabilities. We would like to find the posterior probabilities of each of the three models.
From Bayes Rule, $i \in 1, 2, 3$:

$$P(\text{Model } i|\mathbf{x}) = \frac{P(\mathbf{x}|\text{Model } i)P(\text{Model } i)}{P(\mathbf{x})} \tag{2.21}$$

Since all three models are equally likely *a priori*:

$$P(\text{Model 1}) = P(\text{Model 2}) = P(\text{Model 3}) = \frac{1}{3}$$

And using:

$$P(\mathbf{x}) = P(\mathbf{x}, \text{Model 1}) + P(\mathbf{x}, \text{Model 2}) + P(\mathbf{x}, \text{Model 3})$$

$$P(\mathbf{x}) = P(\mathbf{x}|\text{Model 1})P(\text{Model 1}) + P(\mathbf{x}|\text{Model 2})P(\text{Model 2}) + P(\mathbf{x}|\text{Model 3})P(\text{Model 3})$$

Leading to:

$$P(\mathbf{x}) = \frac{1}{3}\left(P(\mathbf{x}|\text{Model 1}) + P(\mathbf{x}|\text{Model 2}) + P(\mathbf{x}|\text{Model 3})\right) \tag{2.22}$$

We therefore get for Model 1:

$$P(\text{Model 1}|\mathbf{x}) = \frac{P(\mathbf{x}|\text{Model 1})P(\text{Model 1})}{P(\mathbf{x})}$$

Simplifying to (due to equally likely models):

$$P(\text{Model 1}|\mathbf{x}) = \frac{P(\mathbf{x}|\text{Model 1}) \times \frac{1}{3}}{\frac{1}{3}\left(P(\mathbf{x}|\text{Model 1}) + P(\mathbf{x}|\text{Model 2}) + P(\mathbf{x}|\text{Model 3})\right)}$$

Permits us to obtain the **MAP equations for Model 1**:

$$P(\text{Model 1}|\mathbf{x}) = \frac{P(\mathbf{x}|\text{Model 1})}{P(\mathbf{x}|\text{Model 1}) + P(\mathbf{x}|\text{Model 2}) + P(\mathbf{x}|\text{Model 3})} \tag{2.23}$$

Similarly for models 2 and 3:

$$P(\text{Model 2}|\mathbf{x}) = \frac{P(\mathbf{x}|\text{Model 3})}{P(\mathbf{x}|\text{Model 1}) + P(\mathbf{x}|\text{Model 2}) + P(\mathbf{x}|\text{Model 3})} \tag{2.24}$$

And

$$P(\text{Model 3}|\mathbf{x}) = \frac{P(\mathbf{x}|\text{Model 3})}{P(\mathbf{x}|\text{Model 1}) + P(\mathbf{x}|\text{Model 2}) + P(\mathbf{x}|\text{Model 3})} \tag{2.25}$$

Log of these probabilities To be easily implemented, the function *betaln()* was the simplest option. However, it is not explicit in the formulas above how they relate. Therefore, for each model i, the probability can be expressed:

$$\log P(\text{Model } i|\mathbf{x}) = \log P(\mathbf{x}|\text{Model } i) - \log\left(P(\mathbf{x}|\text{Model 1}) + P(\mathbf{x}|\text{Model 2}) + P(\mathbf{x}|\text{Model 3})\right)$$

$$\log P(\text{Model } i|\mathbf{x}) = \log P(\mathbf{x}|\text{Model } i) - \log\left(\exp\{\log P(\mathbf{x}|\text{Model 1})\} + \exp\{P(\mathbf{x}|\text{Model 2})\} + \exp\{P(\mathbf{x}|\text{Model 3})\}\right)$$

$$\log P(\text{Model } i|\mathbf{x}) = \log P(\mathbf{x}|\text{Model } i) - \log\left(\sum_{j=1}^{3} \exp\{\log P(\mathbf{x}|\text{Model } j)\}\right) \tag{2.26}$$

For which the second argument can easily be calculated using the *logsumexp()* function from *scipy.special* library.

From the code used, we obtained the following results for the different models log likelihood, and for their probability (**not** in log), all summed in Table 2.1.

	Model 1	Model 2	Model 3
log likelihood	-4.436×10^3	-4.284×10^3	-3.851×10^3
MAP probability	9.143×10^{-255}	1.434×10^{-188}	$1.0 - 9.143 \times 10^{-255} - 1.434 \times 10^{-188}$

Table 2.1: Log Likelihood and MAP estimations of each model

Interpretation What we can deduce from these results is that it is highly likely the data follows a model with each component being Bernoulli distributed with a separate and unknown p_d, compared to Models 1 and 2. This makes sense since our data is diverse and hence make it impossible to follow Model 1.

Listing 2: Code - Q2

```python
import numpy as np
import scipy.special as sp
from scipy.special import betaln, logsumexp

def loglikelihood_model1(Y: np.ndarray, pd = 0.5):
    N, D = np.shape(Y)

    # p(x given model 1)
    log_like_model1 = N*D * np.log(pd)

    return log_like_model1


def loglikelihood_model2(Y: np.ndarray):
    N, D = np.shape(Y)
    sum_x = np.sum(Y).astype(int)

    log_like_model2 = betaln(1 + sum_x, 1 + (N*D - sum_x))

    return log_like_model2


def loglikelihood_model3(Y: np.ndarray):
    N, D = np.shape(Y)

    sum_xn = np.zeros((D, 1))
    log_betas = np.zeros((D, 1))

    for d in range(D):
        sum_xn[d] = np.sum(Y[:, d]).astype(int)
        log_betas[d] = betaln(1 + sum_xn[d], N + 1 - sum_xn[d])

    log_like_model3 = np.sum(log_betas)

    return log_like_model3

def map_prob_model(model_num: int, Y, pd=0.5, log=False):

    if model_num==1:
        log_like_numerator = loglikelihood_model1(Y, pd)
    elif model_num==2:
        log_like_numerator = loglikelihood_model2(Y)
    else:
        log_like_numerator = loglikelihood_model3(Y)

    log_like_all = np.array([loglikelihood_model1(Y, pd),
                             loglikelihood_model2(Y),
```

```
                                loglikelihood_model3(Y)])

    log_map_model = log_like_numerator - logsumexp(log_like_all)

    if not log:
        map_prob_model = np.exp(log_map_model)
    else:
        map_prob_model = log_map_model

    return map_prob_model

def main():
    map_model1 = map_prob_model(1, Y)
    print("The MAP probability of Model 1 is: ", map_model1, "\n")
    map_model2 = map_prob_model(2, Y)
    print("The MAP probability of Model 2 is: ", map_model2, "\n")
    map_model3 = map_prob_model(3, Y)
    print("The MAP probability of Model 3 is: ", map_model3, "\n")

    like1 = loglikelihood_model1(Y)
    print("The log likelihood of Model 1 is: ", like1, "\n")
    like2 = loglikelihood_model2(Y)
    print("The log likelihood of Model 2 is: ", like2, "\n")
    like3 = loglikelihood_model3(Y)
    print("The log likelihood of Model 3 is: ", like3, "\n")

main()
```

3 | Question 3 - EM for Binary Data

3.1 Question 3.(a)

We consider a mixture of K multivariate Bernoulli distributions. We use the parameters $\pi_1, ..., \pi_K$ to denote the mixing proportions, each comprised between 0 and 1 ($0 \leq \pi_k \leq 1$) and all summing to 1 ($\sum_{k=1}^{K} \pi_k = 1$).
We define the parameters vectors as $\mathbf{p}_k = (p_{k,1}, ..., p_{k,D})$, and the matrix P as:

$$\mathbf{P} = [\mathbf{p}_1, ..., \mathbf{p}_K]^T \tag{3.1}$$

Where each $p_{k,d}$ is defined as $0 \leq p_{k,d} \leq 1, \quad k \in [\![0; K]\!], \quad d \in [\![0; D]\!]$.
Each model is independent and identically distributed (*Assumption* 1), and pixels are independent of each other within each component distribution (*Assumption* 2).
Each of the N images (with index n), assumed to be independent (*Assumption* 3) are defined as:

$$x^{(n)} = (x_1^{(n)}, ..., x_D^{(n)}) \qquad x_d^{(n)} \in \{0, 1\}, \quad d \in [\![0; D]\!], \quad n \in [\![0; N]\!] \tag{3.2}$$

Once all these information have been made clear, we can make the following expression, due to *Assumption* 2:

$$p(x^{(n)}|k) = \prod_{d=1}^{D} p_{k,d}^{x_d^{(n)}} (1 - p_{k,d})^{1 - x_d^{(n)}} \tag{3.3}$$

Leading to the following expression for the likelihood of the mixture element π_k, which is the product between its proportion and its likelihood:

$$p(x^{(n)}|\pi_k) = \pi_k \times p(x^{(n)}|k) \tag{3.4}$$

$$p(x^{(n)}|\pi_k) = \pi_k \prod_{d=1}^{D} p_{k,d}^{x_d^{(n)}} (1 - p_{k,d})^{1 - x_d^{(n)}} \tag{3.5}$$

From that, the probability of x_i given by the set of Multivariate Bernoulli distribution used in the mixture is:

$$p(x^{(n)}|\boldsymbol{\pi}) = \sum_{k=1}^{K} p(x^{(n)}|\pi_k) \tag{3.6}$$

$$p(x^{(n)}|\boldsymbol{\pi}) = \sum_{k=1}^{K} \pi_k \prod_{d=1}^{D} p_{k,d}^{x_d^{(n)}} (1 - p_{k,d})^{1 - x_d^{(n)}} \tag{3.7}$$

Thus, for the whole set of images, the likelihood is given by:

$$p(\mathbf{x}|\boldsymbol{\pi}) = p(x^{(1)}, ..., x^{(N)}|\boldsymbol{\pi}) \tag{3.8}$$

And since *Assumption* 3:

$$p(\mathbf{x}|\boldsymbol{\pi}) = \prod_{n=1}^{N} p(x^{(n)}|\boldsymbol{\pi})$$

$$p(\mathbf{x}|\boldsymbol{\pi}) = \prod_{n=1}^{N} \left(\sum_{k=1}^{K} \pi_k \prod_{d=1}^{D} p_{k,d}^{x_d^{(n)}} (1-p_{k,d})^{1-x_d^{(n)}} \right) \tag{3.9}$$

Taking the log of this expression, we get the log likelihood expression:

$$\textbf{Log Likelihood} = \log(p(\mathbf{x}|\boldsymbol{\pi}))$$

$$\textbf{Log Likelihood} = \sum_{n=1}^{N} \log \left(\sum_{k=1}^{K} \pi_k \prod_{d=1}^{D} p_{k,d}^{x_d^{(n)}} (1-p_{k,d})^{1-x_d^{(n)}} \right) \tag{3.10}$$

For simplicity, we can introduce some matrix notations:

$$\mathbf{X} = [x^{(1)}, ..., x^{(N)}]^T \tag{3.11}$$

which will be a matrix of size $N \times D$.
We also notice the following:

$$p_{k,d}^{x_d^{(n)}} (1-p_{k,d})^{1-x_d^{(n)}} = \exp\left\{ \log p_{k,d}^{x_d^{(n)}} (1-p_{k,d})^{1-x_d^{(n)}} \right\}$$

$$= \exp\left\{ x_d^{(n)} \log p_{k,d} + (1-x_d^{(n)}) \log(1-p_{k,d}) \right\}$$

Thus, replacing in 3.7, we get:

$$p(x^{(n)}|\boldsymbol{\pi}) = \sum_{k=1}^{K} \pi_k \prod_{d=1}^{D} p_{k,d}^{x_d^{(n)}} (1-p_{k,d})^{1-x_d^{(n)}}$$

$$= \sum_{k=1}^{K} \pi_k \prod_{d=1}^{D} \exp\left\{ x_d^{(n)} \log p_{k,d} + (1-x_d^{(n)}) \log(1-p_{k,d}) \right\}$$

$$p(x^{(n)}|\boldsymbol{\pi}) = \sum_{k=1}^{K} \pi_k \exp\left\{ \sum_{d=1}^{D} \left(x_d^{(n)} \log p_{k,d} + (1-x_d^{(n)}) \log(1-p_{k,d}) \right) \right\} \tag{3.12}$$

And using now matrices $\mathbf{X}$ and $\mathbf{P}$ of dimensions $N \times K$ and $K \times D$ respectively:

$$p(x^{(n)}|\boldsymbol{\pi}) = \sum_{k=1}^{K} \pi_k \exp\left\{ (x^{(n)})^T \log(\mathbf{p}_k) + (1-x^{(n)})^T \log(1-\mathbf{p}_k) \right\} \tag{3.13}$$

Generalizing for the log likelihood [1] [2]:

$$\log p(\mathbf{x}|\boldsymbol{\pi}) = \sum_{n=1}^{N} \log \sum_{k=1}^{K} \pi_k \exp\left\{ (x^{(n)})^T \log(\mathbf{p}_k) + (1-x^{(n)})^T \log(1-\mathbf{p}_k) \right\} \tag{3.14}$$

[1] (since we take the log, it will be a sum of the logs instead of the product of the elements)
[2] x^n are vectors, although not written bold

3.2 Question 3.(b)

We would like to find an expression for the responsibility of the mixture component k, denoted as $r_{n,k}$, for data vector $\mathbf{x}^{(n)}$. Using Bayes rule:

$$r_{n,k} = P(s^{(n)} = k|\mathbf{x}^{(n)}, \boldsymbol{\pi}, \mathbf{P}) = \frac{P(\mathbf{x}^{(n)}|s^{(n)} = k, \boldsymbol{\pi}, \mathbf{P})P(s^{(n)} = k|\boldsymbol{\pi}, \mathbf{P})}{P(\mathbf{x}^{(n)}|\boldsymbol{\pi}, \mathbf{P})} \tag{3.15}$$

We notice that, since $s^{(n)}$ does not depend on $\mathbf{P}$:

$$P(s^{(n)} = k|\boldsymbol{\pi}, \mathbf{P}) = P(s^{(n)} = k|\boldsymbol{\pi}) = \pi_k$$

And using the sum and product rules of probabilities, we get:

$$P(\mathbf{x}^{(n)}|\boldsymbol{\pi}, \mathbf{P}) = \sum_{i=1}^{K} P(\mathbf{x}^{(n)}, s^{(n)} = i|\boldsymbol{\pi}, \mathbf{P}) = \sum_{i=1}^{K} P(\mathbf{x}^{(n)}|s^{(n)} = i, \boldsymbol{\pi}, \mathbf{P}) \times P(s^{(n)} = i|\boldsymbol{\pi}, \mathbf{P})$$

Therefore, replacing in our expression, we get for the responsibility of mixture component k for data vector $\mathbf{x}^{(n)}$:

$$r_{n,k} = \frac{P(\mathbf{x}^{(n)}|s^{(n)} = k, \boldsymbol{\pi}, \mathbf{P}) \times P(s^{(n)} = k|\boldsymbol{\pi}, \mathbf{P})}{\sum_{i=1}^{K} P(\mathbf{x}^{(n)}|s^{(n)} = i, \boldsymbol{\pi}, \mathbf{P}) \times P(s^{(n)} = i|\boldsymbol{\pi}, \mathbf{P})} \tag{3.16}$$

$$r_{n,k} = \frac{\pi_k P(\mathbf{x}^{(n)}|s^{(n)} = k, \boldsymbol{\pi}, \mathbf{P})}{\sum_{i=1}^{K} \pi_i P(\mathbf{x}^{(n)}|s^{(n)} = i, \boldsymbol{\pi}, \mathbf{P})}$$

And since, using the E-step of the EM algorithm, we have:

$$P(\mathbf{x}^{(n)}|s^{(n)} = k, \boldsymbol{\pi}, \mathbf{P}) = \prod_{d=1}^{D} p_{k,d}^{x_d^{(n)}} (1 - p_{k,d})^{1-x_d^{(n)}}$$

Then:

$$r_{n,k} = \frac{\pi_k \prod_{d=1}^{D} p_{k,d}^{x_d^{(n)}} (1 - p_{k,d})^{1-x_d^{(n)}}}{\sum_{i=1}^{K} \pi_i \prod_{d=1}^{D} p_{i,d}^{x_d^{(n)}} (1 - p_{i,d})^{1-x_d^{(n)}}} \tag{3.17}$$

Taking the log and naming R_k the expression in the numerator, we get:

$$R_k = \pi_k \prod_{d=1}^{D} p_{k,d}^{x_d^{(n)}} (1 - p_{k,d})^{1-x_d^{(n)}}$$

$$\log R_k = \log \pi_k + \sum_{d=1}^{D} (x_d^{(n)} \log p_{k,d} + (1 - x_d^{(n)}) \log(1 - p_{k,d}))$$

And:

$$\log r_{n,k} = \log R_k - \log \left(\sum_{i=1}^{K} R_i \right) \tag{3.18}$$

3.3 Question 3.(c)

We would like to find the maximizing parameters for the expected log-joint with respect to parameters $\boldsymbol{\pi}$ and $\mathbf{P}$. It can be expressed as:

$$\langle \sum_{n=1}^{N} \log P(\mathbf{x}^{(n)}, s^{(n)}|\boldsymbol{\pi}, \mathbf{P})\rangle_{q(s^{(n)})} = \sum_{n=1}^{N} q(s^{(n)}) \log P(\mathbf{x}^{(n)}, s^{(n)}|\boldsymbol{\pi}, \mathbf{P}) \tag{3.19}$$

Using conditional probabilities:

$$\log P(\mathbf{x}^{(n)}, s^{(n)}|\boldsymbol{\pi}, \mathbf{P}) = \log P(\mathbf{x}^{(n)}|s^{(n)}, \boldsymbol{\pi}, \mathbf{P}) + \log P(s^{(n)}|\boldsymbol{\pi}, \mathbf{P})$$

for which we already found the expression before (in question (b)) being:

$$\log P(\mathbf{x}^{(n)}, s^{(n)}|\boldsymbol{\pi}, \mathbf{P}) = \log R_k$$

$$\log P(\mathbf{x}^{(n)}, s^{(n)}|\boldsymbol{\pi}, \mathbf{P}) = \log \boldsymbol{\pi} + \sum_{d=1}^{D} (x_d^{(n)} \log p_{k,d} + (1 - x_d^{(n)}) \log(1 - p_{k,d})) \tag{3.20}$$

Or better in matrix and vector form:

$$\log P(\mathbf{x}^{(n)}, s^{(n)}|\boldsymbol{\pi}, \mathbf{P}) = \log \boldsymbol{\pi} + \sum_{d=1}^{D} \log(\mathbf{P})^T \mathbf{x}^{(n)} + \log(1 - \mathbf{P})^T (1 - \mathbf{x}^{(n)})$$

From before, we know the corresponding E-step in our case is:

$$q(s^{(n)}) = \mathbf{r}_n = [r_{n,1}, ..., r_{n,K}]^T$$

Resulting in the expression of E:

$$E = \sum_{n=1}^{N} \mathbf{r}_n \left(\log \boldsymbol{\pi} + \sum_{d=1}^{D} \mathbf{x}^{(n)} * \log(\mathbf{P})^T + (1 - \mathbf{x}^{(n)})) \log(1 - \mathbf{P})^T \right) \tag{3.21}$$

Maximizing the E-step:

$$\text{argmax}_{\boldsymbol{\pi},\mathbf{P}} \langle \sum_{n=1}^{N} \log P(\mathbf{x}^{(n)}, s^{(n)}|\boldsymbol{\pi}, \mathbf{P})\rangle_{q(s^{(n)})} \tag{3.22}$$

Is equivalent to take the derivative of the expression of E and differentiate it with respect to $\boldsymbol{\pi} and \mathbf{P}$ and set the derivatives to 0.
Maximizing $\mathbf{P}$, by finding the optimal $\hat{p}_{k,d}$:

$$\frac{\partial E}{\partial p_{k,d}} = \frac{\partial}{\partial p_{k,d}} \sum_{n=1}^{N} \mathbf{r}_n \left(\log \boldsymbol{\pi} + \sum_{d=1}^{D} \mathbf{x}^{(n)} \log(\mathbf{P})^T + (1 - \mathbf{x}^{(n)})) \log(1 - \mathbf{P})^T \right)$$

$$= \sum_{n=1}^{N} r_{n,k} \frac{\partial}{\partial p_{k,d}} \left(\log(p_{k,d}) x_d^{(n)} + \log(1 - p_{k,d})(1 - x_d^{(n)}) \right)$$

$$\frac{\partial E}{\partial p_{k,d}} = \sum_{n=1}^{N} r_{n,k} \left(\frac{x_d^{(n)}}{p_{k,d}} + \frac{-(1 - x_d^{(n)})}{1 - p_{k,d}} \right) \tag{3.23}$$

And setting this equation to 0:

$$\frac{\partial E}{\partial p_{k,d}} = 0 \tag{3.24}$$

$$\sum_{n=1}^{N} r_{n,k} \left(\frac{x_d^{(n)}}{\hat{p}_{k,d}} + \frac{-(1 - x_d^{(n)})}{1 - \hat{p}_{k,d}} \right) = 0$$

$$\frac{(1 - \hat{p}_{k,d}) \sum_{n=1}^{N} r_{n,k} x_d^{(n)} - \hat{p}_{k,d} \sum_{n=1}^{N} r_{n,k} (1 - x_d^{(n)})}{\hat{p}_{k,d}(1 - \hat{p}_{k,d})} = 0$$

$$\sum_{n=1}^{N} r_{n,k} x_d^{(n)} - \hat{p}_{k,d} \sum_{n=1}^{N} r_{n,k} x_d^{(n)} + \hat{p}_{k,d} \sum_{n=1}^{N} r_{n,k} x_d^{(n)} - \hat{p}k, d \sum_{n=1}^{N} r_{n,k} = 0$$

$$\sum_{n=1}^{N} r_{n,k} x_d^{(n)} - \hat{p}k, d \sum_{n=1}^{N} r_{n,k} = 0$$

$$\hat{p}_{k,d} = \frac{\sum_{n=1}^{N} r_{n,k} x_d^{(n)}}{\sum_{n=1}^{N} r_{n,k}} \tag{3.25}$$

Doing the same for $\boldsymbol{\pi}$ to find the optimal $\hat{\pi}_k$:

$$\frac{\partial E}{\partial \pi_k} = 0$$

However, this equation result leads to the sum over n of $r_{n,k}$ being equal to 0, because:

$$\frac{\partial E}{\partial \pi_k} = \sum_{n=1}^{N} \frac{r_{n,k}}{\pi_k}$$

This doesn't solve our problem. We must redefine this problem as an optimization problem where we want to maximize E subject to the constraint $\sum_k \pi_k = 1$.
Using the Lagrangian multiplier λ to enforce normalization:

$$E_{Lagrangian} = E + \lambda \left(1 - \sum_{k=1}^{K} \pi_k \right) = 0 \tag{3.26}$$

And finding the conditions for stationarity:

$$\frac{\partial E_{Lagrangian}}{\partial \pi_k} = 0 \iff \sum_{n=1}^{N} \frac{r_{n,k}}{\hat{\pi}_k} - \lambda = 0$$

$$\sum_{n=1}^{N} r_{n,k} = \lambda \hat{\pi}_k$$

$$\hat{\pi}_k = \frac{\sum_{n=1}^{N} r_{n,k}}{\lambda} \tag{3.27}$$

And:

$$\frac{\partial E_{Lagrangian}}{\partial \lambda} = 0$$

Leads to:

$$\sum_{k=1}^{K} \pi_k = 1 \tag{3.28}$$

To find the parameter λ, we now replace in 3.28:

$$\sum_{k=1}^{K} \left(\frac{\sum_{n=1}^{N} r_{n,k}}{\lambda} \right) = 1$$

$$\lambda = \sum_{k=1}^{K} \sum_{n=1}^{N} r_{n,k} \tag{3.29}$$

Therefore, the final expression for the optimal $\hat{\pi}_k$ is:

$$\hat{\pi}_k = \frac{\sum_{n=1}^{N} r_{n,k}}{\sum_{k=1}^{K} \sum_{n=1}^{N} r_{n,k}} = \frac{\sum_{n=1}^{N} r_{n,k}}{\sum_{n=1}^{N} \sum_{k=1}^{K} r_{n,k}}$$

And since the sum over k of all the responsibilities $r_{n,k}$ is 1, then:

$$\hat{\pi}_k = \frac{\sum_{n=1}^{N} r_{n,k}}{\sum_{n=1}^{N} 1*}$$

$$\hat{\pi}_k = \frac{1}{N} \sum_{n=1}^{N} r_{n,k} \tag{3.30}$$

3.4 Question 3.(d)

After implementing the EM algorithm for a mixture of K multivariate Bernoullis, the obtained results are displayed in the figures below (Figures 3.1 to 3.5), where the code was executed for $K \in \{2, 3, 4, 7, 10\}$. The initial value of $\boldsymbol{\pi}$ has been chosen as:

$$\boldsymbol{\pi} = \left[\frac{1}{\sum_k k}, ..., \frac{K}{\sum_k k} \right]$$

and for $\mathbf{P}$ were for each $p_{k,d}$ the mean values of $x_d^{(n)}$ (the same values were given on dimension k).
We can see that for $K = 10$, the algorithm converges faster to a better value. This makes sense as there are 9 different single digits that can be drawn. The model would therefore find approximately a category for each, and one last category for the ones that were badly drawn.
A notable issue about it is the fact the optimization of the EM parameters only happened after around 5 to 10 iterations. We can suppose the model's performance depends on the initial values.

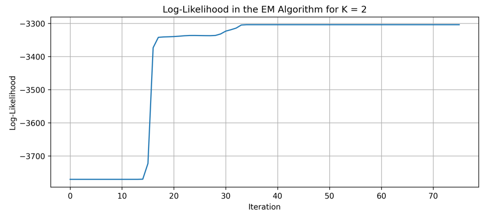


Figure 3.1: K=2

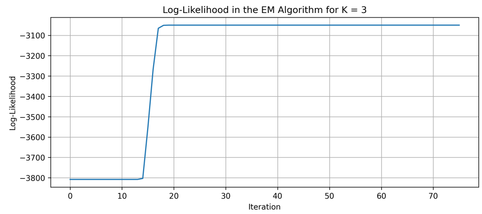


Figure 3.2: K=3

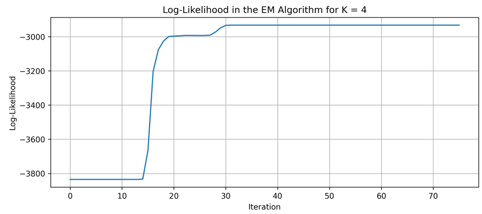


Figure 3.3: K=4

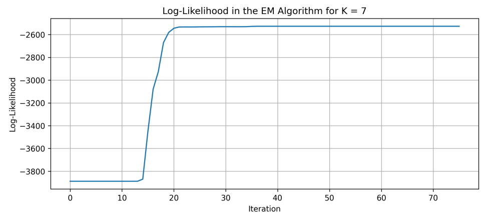


Figure 3.4: K=7

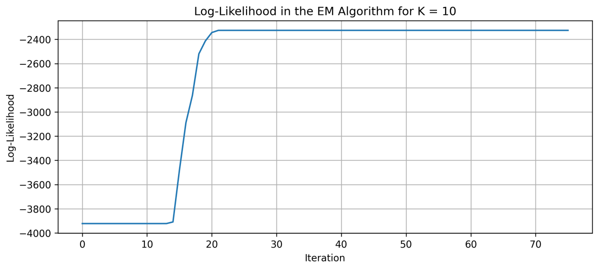


Figure 3.5: K=10

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

# Defining the algorithm function, which takes K (number of
# mixture components), the matrix X (containing dataset) and
# the maximum iterations to run

def EM_algorithm(K: int,
                 X: np.ndarray,
                 n_iter: int=300,
                 epsilon: float=1e-12,
                 min_break = 75
                 ):
    """
    Main function fo executing the EM algorithm

    Inputs:
        - K: int, the number of mixture components
        - X: int, the matrix containing the dataset
        - n_iter: int, the maximum number of iterations
        - epsilon: float, the precision after which we can stop
            the algorithm from running.

    Outputs:
        - pi_k: array, size (K,1) containing all mixing proportions
        - p_kd: array, size (K,D) containing all p_kds (equivalent to
                matrix P in problem wording)
        - log_likelihood: list, containing all the log likelihood
                updates of the model.
    """

    # Finding dimensions of initial dataset
    N, D = np.shape(X)

    # Defining initial pi_k as an array
    # with increasing values
    pi_k = np.arange(1,K+1)
    sum_pi = np.sum(pi_k)
    pi_k = (1/sum_pi)*pi_k

    # Initializing P's values as mean of
    # element d in all images
    p_kd = np.zeros((K, D))
    for d in range(D):
        p_kd[:, d] = (1/N)*X[:, d].sum()

    # Initializing the final
```

```
    log_likelihood = []

    # Checking for any 0 or 1 in the values of pi
    # and P, and if so, replacing them by very small
    # numbers. It avoids numerical instability from
    # Dividing by a 0 or log(1)=0
    for k in range(K):
        if pi_k[k] < 1e-10:
            pi_k[k] = 1e-10
        elif pi_k[k] > (1- 1e-10):
            pi_k[k] = 1- 1e-10

        for d in range(D):
            if p_kd[k,d] < 1e-10:
                p_kd[k,d] = 1e-10
            elif p_kd[k,d] > 1-1e-10:
                p_kd[k,d] = 1-1e-10

    # Loop of the EM algorithm, which will stop when the
    # max number of iterations will be reached OR when the
    # updates won't have a difference superior to epsilon
    for i in range(n_iter):

        # Calculating the responsibilities (E-step)
        r_nk = E_step(X, pi_k, p_kd, K)

        # Calculating the new pi and P values (M-Step)
        pi_k, p_kd = M_step(X, r_nk)

        # Checking for invalid values to avoid numerical
        # instability
        for k in range(K):
            if pi_k[k] < 1e-10:
                pi_k[k] = 1e-10
            elif pi_k[k] > (1- 1e-10):
                pi_k[k] = 1- 1e-10
            for d in range(D):
                if p_kd[k,d] < 1e-10:
                    p_kd[k,d] = 1e-10
                elif p_kd[k,d] > 1-1e-10:
                    p_kd[k,d] = 1-1e-10

        # Calculating the new log likelihood, given the calculated
        # optimal parameters and storing it in the list
        log_like = log_likelihood_EM(X, pi_k, p_kd, r_nk)
        log_likelihood.append(log_like)

        # Breaking the loop if the updates don't bring any change
        if i > min_break and abs(log_likelihood[-1] - log_likelihood[-2]) < epsilon:
```

```
            break

    return pi_k, p_kd, log_likelihood

def E_step(X: np.ndarray, pi_k, p_kd, K):
    """
    Calculating the E-step of the algorithm, using the equations found earlier

    Inputs:
    - X: np.ndarray, shape (N,D), contains initial data
    - pi_k: list, shape(K,), contains current pi values for each mixture
    - p_kd: array, shape(K, D), contains current probabilities
    - K: int, number of models in the mixture

    Outputs:
    - r_nk: np.ndarray, shape(N,K), contains responsibilities of each model
    """
    # Defining the values of N, being number of rows in dataset
    N, _ = np.shape(X)

    # Initializing the responsibilities matrix
    log_r_nk = np.zeros((N, K))

    # Reuniting the different rows of pi_k
    # as a single row
    log_pi_k = np.log(pi_k).ravel()

    # Calculating the logs of each responsibility
    log_r_nk = log_pi_k + (X @ np.log(p_kd.T)) + ((1 - X) @ np.log(1 - p_kd.T))

    # Normalizing the logs of responsibilities
    log_r_nk = log_r_nk - logsumexp(log_r_nk, axis=1, keepdims=True)

    # Returning to the non-log domain, by calculating the exponential
    # of the responsibilities
    r_nk = np.exp(log_r_nk)

    return r_nk

def M_step(X: np.ndarray, r_nk):
    """
    Calculates the M-step of the EM algorithm, using the equations defined in report

    Inputs:
    - X: np.ndarray, shape (N,D), contains initial data
    - r_nk: array, shape (N, K), responsibilities calculated of each model on
                                each data point
    Outputs:
    - pi_k: list, shape (K,), contains current pi values for each mixture
```

```
    - P: array, shape (K, D), contains current probabilities
    """

    N, D = np.shape(X)
    K = np.shape(r_nk)[1]

    # Verifying the validity of the values
    # of the responsibilities
    for n in range(N):
        for k in range(K):
            if r_nk[n,k] < 1e-10:
                r_nk[n, k] = 1e-10
            elif r_nk[n,k] > (1-1e-10):
                r_nk[n,k] = 1-1e-10

    # Calculating the updates of the parameters
    pi_k = (1/N)*np.sum(r_nk, axis=0)
    log_p_kd = np.log(r_nk.T @ X) - np.log(r_nk.sum(axis=0)[:, np.newaxis])
    p_kd = np.exp(log_p_kd)

    # Verifying the validity of the values
    # of the newly calculated parameters (in pi and P)
    for k in range(K):
            if pi_k[k] < 1e-10:
                pi_k[k] = 1e-10
            elif pi_k[k] > (1- 1e-10):
                pi_k[k] = 1- 1e-10
            for d in range(D):
                if p_kd[k,d] < 1e-10:
                    p_kd[k,d] = 1e-10
                elif p_kd[k,d] > 1-1e-10:
                    p_kd[k,d] = 1-1e-10

    return pi_k, p_kd


def log_likelihood_EM(X, pi_k, p_kd, r_nk):
    """
    Calculating the log likelihood of the data being given the parameters
    of the mixture model

    Inputs:
    - X: np.ndarray, shape (N,D), contains initial data
    - pi_k: list, shape (K,), contains current pi values for each mixture
    - p_kd: array, shape (K, D), contains current probabilities
    - r_nk: array, shape (N, K), responsibilities calculated of each model on
                                each data point
    Outputs:
    - log_likelihood: float, log likelihood probability calculated
```

```
    """

    # Calculating N, D, K and initializing the value of the log likelihood
    N, D = np.shape(X)
    K = np.shape(pi_k)[0]
    log_likelihood = 0

    # Looping through the r_nk dimensions, since there will be as many elements to add
    # as there is responsibilities
    for n in range(N):
        for k in range(K):

            # Calculating the log of element k in pi matrix
            log_pi = np.log(pi_k[k])

            # Calculating P(x \vert s=k, pi, P), the main element in the calculation
            log_px_given_k = np.sum(X[n] * np.log(p_kd[k])
            + (1 - X[n]) * np.log(1 - p_kd[k]))

            # Weighting the log of probabilities by the responsibilities, and
            # adding them to the log likelihood.
            log_rnk_contribution = r_nk[n,k] * (log_pi + log_px_given_k)
            log_likelihood += log_rnk_contribution

    return log_likelihood


def plot_EM(log_likelihoods, pi_k, p_kd, K, save_like=False, save_prob=False):
    """
    Plots the EM results obtained
    """

    # Plotting the obtained log likelihoods as a function of the iterations
    plt.figure(figsize=(10,4))
    plt.plot(log_likelihoods)
    plt.xlabel("Iteration")
    plt.ylabel("Log-Likelihood")
    plt.title(f"Log-Likelihood in the EM Algorithm for K = {K}")
    plt.grid()
    if save_like:
        plt.savefig(f'Q3/Loglikelihood_{K}.png',
                    format="png",
                    dpi=300,
                    bbox_inches="tight")
    plt.show()


    # Plotting the mixture components probabilities as heat maps
    K, D = p_kd.shape
```

```
    image_size = int(np.sqrt(D)/2)
    figs, axs = plt.subplots(1, K, figsize=(4*K, 4))
    figs.suptitle(f"Mixture components probabilities for K = {K})", fontsize=14)
    for i in range(K):
        ax = axs[i] if K > 1 else axs
        ax.imshow(p_kd[i].reshape((image_size, image_size)), cmap='viridis')
        ax.set_title(f"pi: {pi_k[i]:.2f}", fontsize=10)
        ax.axis('off')
    if save_prob:
        plt.savefig(f'Q3/prob_{K}.png', format="png", dpi=300, bbox_inches="tight")
    plt.show()


def main():
    Ks: list = [2, 3, 4, 7, 10]

    for K in Ks:
        print("=== EM Algorithm for K =", K, " ===")
        pi_k, p_kd, log_likelihoods = EM_algorithm(K, Y, n_iter=75, epsilon=1e-6)
        print("For K=", K, ", \n pi_k = ", pi_k, "\n p_kd = ", p_kd) print("\n and log_likelihoods =
        plot_EM(log_likelihoods, pi_k, p_kd, K)


main()
```

3.5 Question 3.(e)

In this section, we used the code from the previous questions and slightly modifying it (see the code following the figures below) to make it generate random initial parameters. The different initial random probabilities were normalized after being generated to still be valid probabilities.

In Figure **??** are displayed the initial random $p_{k,d}$ used in the EM algorithm. We can see there aren't any relevant pattern. Then, by running the algorithm on such initial values, we obtain the final parameters learned by the EM algorithm and displayed in Figure 3.11. The resulting parameters can be good for low dimensions (*e.g.* 3.6), but for larger dimensions, rapidly, it has learned parameters that do not look very good (with extreme probabilities, *i.e.* either 0 or 1). Comparing them to that of our fairly smart proposal (the data-driven one, shown in Figure 3.17) and by comparing both, it seems each category the data-driven look like a real handwritten digit.
We therefore conclude we got different solutions depending on the starting point. This algorithm therefore has some flaws, and the main flaw is its over-dependence on the data. By choosing the wrong data, the model won't be able to classify well the different categories of handwritten digits. From looking at the pictures, we also see it fails at finding clusters, since the handwritten digits are all similar in the different categories. Thus, overall, it would perform pretty badly in a real life scenario.

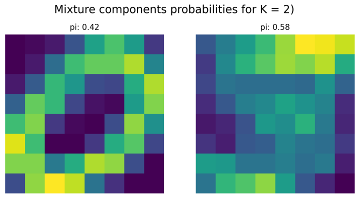


Figure 3.6: K=2

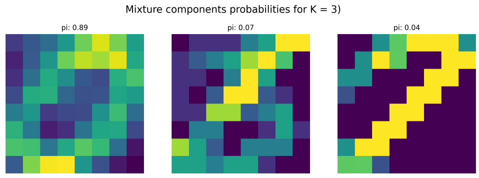


Figure 3.7: K=3

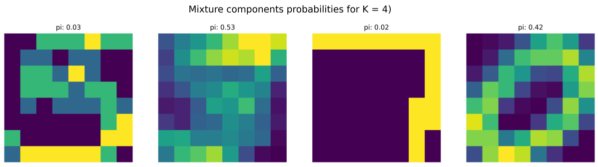


Figure 3.8: K=4

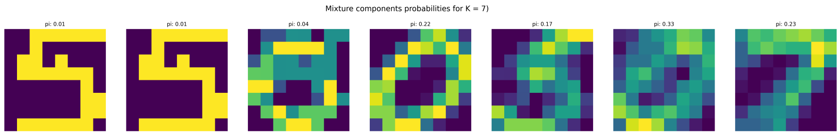


Figure 3.9: K=7

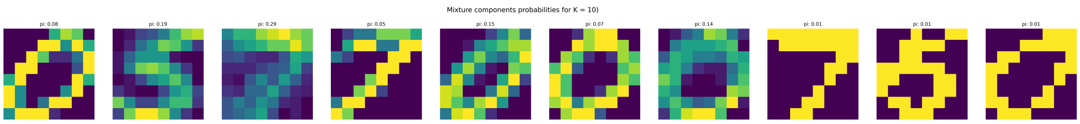


Figure 3.10: K=10

Figure 3.11: Learned Parameters from Random Initial Parameters

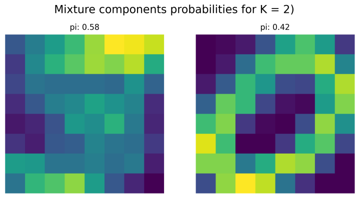


Figure 3.12: K=2

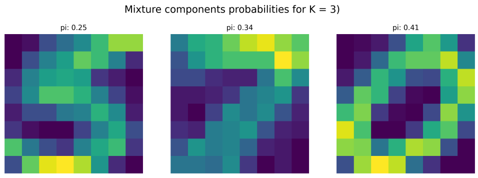


Figure 3.13: K=3

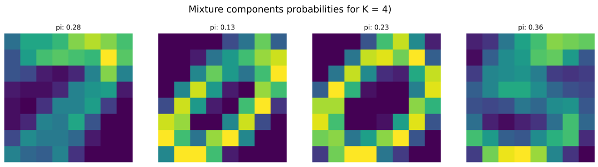


Figure 3.14: K=4

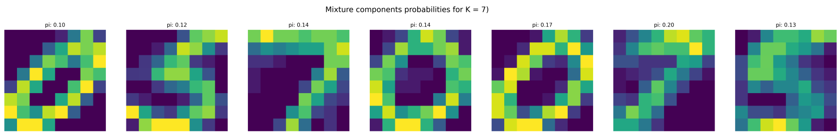


Figure 3.15: K=7

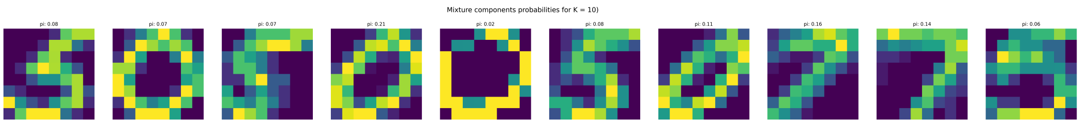


Figure 3.16: K=10

Figure 3.17: Parameters Learned by the EM Algorithm for Data-Driven Initialization

4 | Question 5 - Decrypting Messages with MCMC

We are given an encrypted passage of English text. The mapping of this encryption is one-to-one, in a sense that all encrypted symbols are assigned to a unique other different symbol. The English text, composed of s_i symbols as $s_1 s_2 ... s_n$, is modelled as a first-order Markov Chain:

$$p(s_1 s_2 ... s_n) = p(s_1) \prod_{i=2}^{n} p(s_i | s_{i-1}) \tag{4.1}$$

4.1 Question 5.(a)

We define the transition probabilities in the studied text (*i.e. "War and Peace"*, by Leo Tolstoy), as $p(s_i = \alpha | s_{i1} = \beta) = \psi(\alpha, \beta)$ and the stationary distributions of the symbols as $\lim_{i \to \infty} p(s_i = \gamma) = \phi(\gamma)$. We assume the first letter of the encrypted text is sampled from the stationary distributions.

The formulae for the ML estimates of $p(s_i = \alpha | s_{i-1} = \beta) = \psi(\alpha, \beta)$ are given by:

$$\psi_{ML}(\alpha, \beta) = \frac{\text{counts of pair } (\alpha, \beta) \text{ as } \beta\alpha \text{ in the text}}{\text{counts of symbol } \beta \text{ in the text}} \tag{4.2}$$

The stationary probabilities were calculated in the code by calculating, using the *np.linalg.eig()* function on the transpose of the transition probability matrix, finding the eigenvector corresponding to the eigenvalue with the value the closest to 1 and normalizing the values in that eigenvector by dividing them by their entire sum.
In Table 4.1, a few of the transition probability matrix are displayed. Since the table is too large, the csv file containing the whole matrix is joint to the report. In addition, Figure 4.1 shows the heat map plot corresponding to the matrix, which is clearer.
In Table 4.3 are displayed all the entries of the stationary distribution. In addition, the bar plot shown in Figure 4.2 shows the stationary distribution probabilities for each of the corresponding symbols.

	a	b	c	d	e	f	g	h
a	3.401e-05	1.681e-02	3.410e-02	5.500e-02	8.405e-04	8.104e-03	1.709e-02	1.526e-03
b	9.254e-02	6.608e-03	8.657e-05	4.617e-04	3.288e-01	0.000e+00	0.000e+00	2.886e-05
c	1.091e-01	0.000e+00	1.702e-02	1.460e-04	2.132e-01	0.000e+00	0.000e+00	1.883e-01
d	2.083e-02	1.014e-04	8.454e-05	1.102e-02	1.163e-01	7.186e-04	3.669e-03	4.227e-04
e	4.231e-02	8.088e-04	1.725e-02	9.085e-02	2.564e-02	1.005e-02	6.835e-03	2.265e-03
f	7.022e-02	4.918e-04	0.000e+00	0.000e+00	8.418e-02	5.163e-02	0.000e+00	5.465e-05
g	6.606e-02	0.000e+00	0.000e+00	8.183e-04	1.146e-01	0.000e+00	8.670e-03	1.161e-01
h	1.657e-01	3.703e-04	3.703e-04	3.584e-04	4.496e-01	4.062e-04	5.973e-06	3.584e-05

Table 4.1: A few transition probabilities displayed

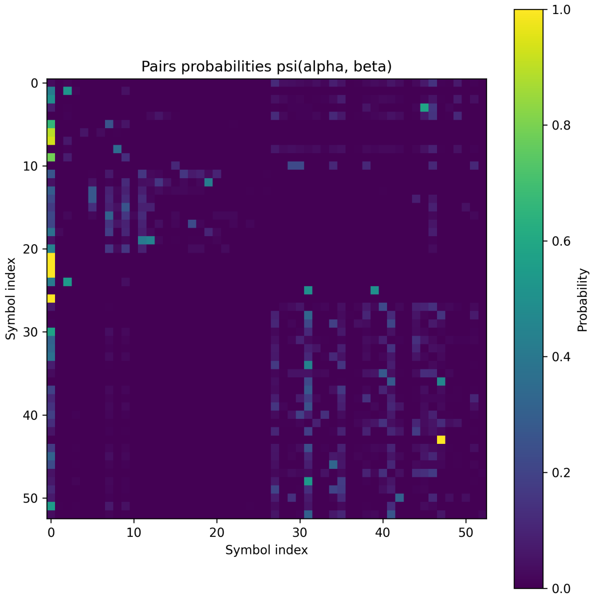


Figure 4.1: Heat Map of the Transition Probabilities for each probability $\psi(\alpha, \beta)$

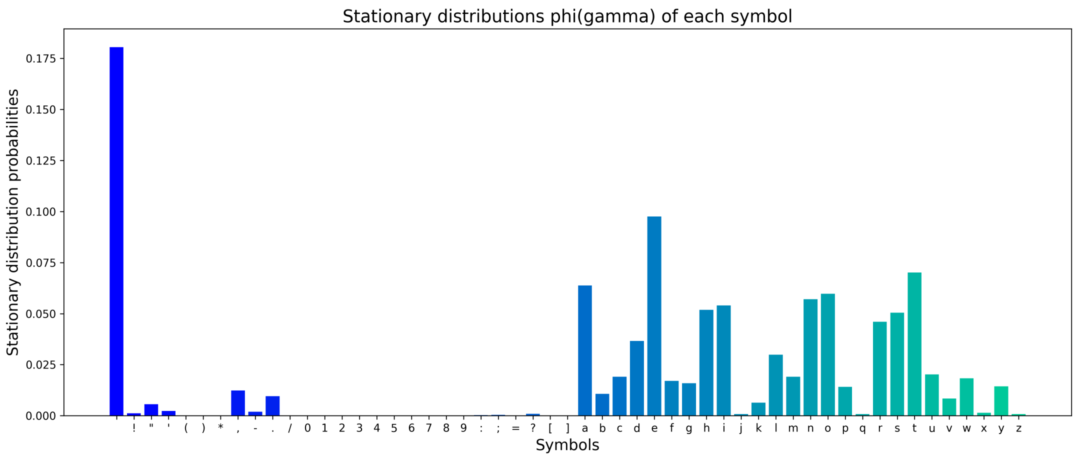


Figure 4.2: Stationary distributions figure, showing the $\phi(\gamma)$ probability for each corresponding symbol.

Symbol	Stationary Probability
	1.805e-01
!	1.216e-03
""	5.565e-03
’	2.331e-03
(	2.065e-04
)	2.065e-04
*	8.917e-05
-	1.880e-03
.	9.561e-03
/	2.787e-06
0	5.264e-05
1	1.208e-04
2	4.459e-05
3	1.827e-05
4	7.122e-06
5	1.579e-05
6	1.641e-05
7	1.177e-05
8	5.945e-05
9	9.908e-06
:	3.118e-04
;	3.542e-04
=	6.193e-07
?	9.707e-04
[	6.193e-07
]	6.193e-07
a	6.373e-02
b	1.073e-02
c	1.908e-02
d	3.663e-02
e	9.762e-02
f	1.700e-02
g	1.589e-02
h	5.184e-02
i	5.395e-02
j	7.973e-04
k	6.328e-03
l	2.989e-02
m	1.909e-02
n	5.703e-02
o	5.973e-02
p	1.410e-02
q	7.218e-04
r	4.596e-02
s	5.044e-02
t	7.012e-02
u	2.026e-02
v	8.386e-03
w	1.834e-02
x	1.357e-03
y	1.433e-02
z	7.391e-04

Table 4.2: Stationary Probabilities

4.2 Question 5.(b)

We represent the mapping of s onto its corresponding encoding as $\sigma(s)$. We define each state of the MCMC sampler as a permutation between two mapping variables, meaning we simply exchange randomly the encoding of two distinct variables to their corresponding encoding symbols.
Assuming a uniform prior distribution over the permutations, we ask ourselves if the latent variables $\sigma(s)$ for different symbols s independent. The answer is no: the latent variables are not independent because assigning a latent variable to a symbol s reduces by one choice the possible assignment of the next latent variable to the next symbol s.
Let $e_1...e_n$ be an encrypted English text. We recall the equation (2.1), $p(s_1 s_2 ... s_n) = p(s_1)\prod_{i=2}^{n} p(s_i|s_{i-1})$. Then relating them to the encoding function, the joint probability of $e_1...e_n$ given σ is:

$$e_i = \sigma(s_i) \iff s_i = \sigma^{-1}(e_i)$$

And thus:

$$p(e_1, ..., e_n, |\sigma) = \phi(\sigma^{-1}(e_1)) \prod_{i=2}^{n} p(\sigma^{-1}(e_i)|\sigma^{-1}(e_{i-1})) \tag{4.3}$$

4.3 Question 5.(c)

We use the Metropolis-Hastings algorithm with the proposal given by choosing two symbols s and s^2 at random and swapping the corresponding encrypted symbols.
The proposal probability $S(\sigma \to \sigma')$ depends on the permutations σ and σ' since we swap the encoding of two symbols in the mapping. Therefore, the proposal probability is the probability of choosing those two symbols simultaneously, given by:

$$S(\sigma \to \sigma') = \frac{1}{\binom{n}{2}} \tag{4.4}$$

where n is the number of symbols. In our case, we get:

$$S(\sigma \to \sigma') = \frac{1}{\binom{53}{2}} = \frac{2! \times 51!}{53!} = \frac{2}{53 \times 52}$$

$$S(\sigma \to \sigma') = \frac{1}{1378} \tag{4.5}$$

The MH acceptance probability, which depends on the current mapping (while the proposal probability did not) for a given proposal is given by:

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

We notice $S(\sigma' \to \sigma) = S(\sigma \to \sigma')$. Thus, we are left with:

$$A(\sigma \to \sigma'|e_1, ..., e_n) = \min\left(1, \frac{p(\sigma'|e_1, ..., e_n)}{p(\sigma|e_1, ..., e_n)}\right) \tag{4.6}$$

Recalling Bayes' Theorem:

$$p(\sigma|e_1, ..., e_n) = \frac{p(e_1, ..., e_n|\sigma)p(e_1, ..., e_n)}{p(\sigma)}$$

And similarly for σ':

$$p(\sigma'|e_1,...,e_n) = \frac{p(e_1,...,e_n|\sigma')p(e_1,...,e_n)}{p(\sigma')} \iff p(e_1,...,e_n) = \frac{p(\sigma')p(\sigma'|e_1,...,e_n)}{p(e_1,...,e_n|\sigma')}$$

Combining the two expressions (by replacing $p(e_1,...,e_n)$):

$$p(\sigma|e_1,...,e_n) = \frac{p(e_1,...,e_n|\sigma)}{p(\sigma)} \times \frac{p(\sigma')p(\sigma'|e_1,...,e_n)}{p(e_1,...,e_n|\sigma')} \tag{4.7}$$

And reinjecting them into the expression of A (equation (2.6)):

$$\begin{aligned} A(\sigma \to \sigma'|e_1,...,e_n) &= \min\left(1, \frac{p(\sigma'|e_1,...,e_n)}{\frac{p(e_1,...,e_n|\sigma)}{p(\sigma)} \times \frac{p(\sigma')p(\sigma'|e_1,...,e_n)}{p(e_1,...,e_n|\sigma')}}\right) \\ &= \min\left(1, \frac{1}{\frac{p(e_1,...,e_n|\sigma)}{p(\sigma)} \times \frac{p(\sigma')}{p(e_1,...,e_n|\sigma')}}\right) \end{aligned}$$

Leading to:

$$A(\sigma \to \sigma'|e_1,...,e_n) = \min\left(1, \frac{p(e_1,...,e_n|\sigma')p(\sigma)}{p(\sigma')p(e_1,...,e_n|\sigma)}\right)$$

And since we assumed a uniform prior distribution over the permutations, then $p(\sigma) = p(\sigma')$ and therefore, the expression of the MH acceptance probability simplifies to:

$$A(\sigma \to \sigma'|e_1,...,e_n) = \min\left(1, \frac{p(e_1,...,e_n|\sigma')}{p(e_1,...,e_n|\sigma)}\right) \tag{4.8}$$

4.4 Question 5.(d)

This section's goal is to implement the MH sampler and running it on the encrypted text. In Table 4.3 are displayed the results every 100 iterations, using a random initial proposal, for the proposal encoding shown in 4.4.

Iteration	**First 60 characters decrypted**
200	"enla;l;uingstlonblautsl""icnsto’csl;sotdla;lmo ystlgo""slasldu"
300	"enly;l;uongstlinblyutsl""ocnsti’csl;sitdly;lmi astlgi""slysldu"
400	"enly;l;uoncstlinblyutsl’ognsti""gsl;sitdly;lki astlci’slysldu"
500	lnew.e.ouncsteinbewotse’ugnsti;gse.sitdew.eki asteci’sewsedo
800	lnew.e.ounksteinyewotse’ugnstibgse.sitmew.epi asteki’sewsemo
900	ln w. .ounkes inh wose ’ugnesibge .eism w. pitaes ki’e we mo
1000	ln wh hounkes in. wose cugnesibge heism wh pitaes kice we mo
1100	ln wa aounkes inf wose cugnesibge aeism wa pithes kice we mo
1200	ln wa aounkes inf wose cudnesibde aeism wa pithes kice we mo
1300	on wa alunker inf wlre cudneribde aeirm wa pither kice we ml
1400	on wa alunker inf wlre cudneribde aeirm wa pither kice we ml
1500	on wa alunver inf wlre mudneribde aeirc wa pither vime we cl
1600	on wa alunver inf wlre mudnerisde aeirc wa pither vime we cl
1700	on wa alunver inf wlre mudnerisde aeirc wa yither vime we cl
1800	on wl launver inf ware mudnerisde leirc wl yither vime we ca
1900	on wc caunver inf ware mudnerisde ceirl wc yither vime we la
2000	on wf faunver inc ware mudnerisde feirl wf yither vime we la
2100	on wl launver ind ware mucnerisce leirf wl yither vime we fa
2200	on wl launver ind ware mucnerisce leirf wl yither vime we fa
2300	on ws saunver ind ware mucnerilce seirf ws yither vime we fa
2400	on ws saunver ind ware mulnericle seirf ws yither vime we fa
2500	on ws saunver ind ware mulnericle seirf ws yither vime we fa
2600	on ws saunver ind ware mulnerible seirf ws yither vime we fa
2700	on ws saunver ind ware mulnerible seirf ws yither vime we fa
2800	on ws saunver ind ware mulnerible seirf ws yither vime we fa
2900	in ws sounger and wore mulnerable searf ws yather game we fo
3000	in ws sounger and wore mulnerable searf ws yather game we fo
3100	in ws sounger and wore mulnerable searf ws yather game we fo
3200	in ws sounger and wore mulnerable searf ws yather game we fo
3300	in ws sounger and wore mulnerable searf ws yather game we fo
3400	in ws sounger and wore mulnerable searf ws yather game we fo
3500	in ws sounger and wore mulnerable searf ws yather game we fo
3600	in ws sounger and wore mulnerable searf ws yather game we fo
3700	in ws sounger and wore mulnerable searf ws yather game we fo
3800	in ws sounger and wore mulnerable searf ws yather game we fo
3900	in wf founger and wore mulnerable fears wf yather game we so
4000	in wf founger and wore mulnerable fears wf yather game we so
4100	in wf founger and wore mulnerable fears wf yather game we so
4200	in wf founger and wore mulnerable fears wf yather game we so
4300	in wf founger and wore mulnerable fears wf yather game we so
4400	in wf founger and wore mulnerable fears wf yather game we so
4500	in wf founger and wore mulnerable fears wf yather game we so
4600	in wf founger and wore mulnerable fears wf yather game we so
4700	in wf founger and wore mulnerable fears wf yather game we so
4800	in wf founger and wore mulnerable fears wf yather game we so
4900	in wf founger and wore mulnerable fears wf yather game we so
5000	in wy younger and wore mulnerable years wy father game we so
5100	in my younger and more wulnerable years my father gawe me so
5200	in wy younger and wore mulnerable years wy father game we so
5300	in wy younger and wore vulnerable years wy father gave we so
5400	in wy younger and wore vulnerable years wy father gave we so
5500	in wy younger and wore vulnerable years wy father gave we so
5600	in wy younger and wore mulnerable years wy father game we so
5700	in wy younger and wore mulnerable years wy father game we so
5800	in wy younger and wore mulnerable years wy father game we so
5900	in wy younger and wore mulnerable years wy father game we so
6000	in wy younger and wore mulnerable years wy father game we so
6100	in my younger and more wulnerable years my father gawe me so
6200	in my younger and more wulnerable years my father gawe me so
6300	in my younger and more wulnerable years my father gawe me so
6400	in my younger and more kulnerable years my father gake me so
6500	in my younger and more kulnerable years my father gake me so
6600	in my younger and more kulnerable years my father gake me so
6700	in my younger and more kulnerable years my father gake me so
6800	in my younger and more kulnerable years my father gake me so
6900	in my younger and more kulnerable years my father gake me so
7000	in my younger and more kulnerable years my father gake me so
7100	in my younger and more kulnerable years my father gake me so
7200	in my younger and more kulnerable years my father gake me so
7300	in my younger and more kulnerable years my father gake me so
7400	in my younger and more kulnerable years my father gake me so
7500	in my younger and more kulnerable years my father gake me so
7600	in my younger and more kulnerable years my father gake me so
7700	in my younger and more kulnerable years my father gake me so
7800	in my younger and more kulnerable years my father gake me so
7900	in my younger and more kulnerable years my father gake me so
8000	in my younger and more kulnerable years my father gake me so
8100	in my younger and more kulnerable years my father gake me so
8200	in my younger and more kulnerable years my father gake me so
8300	in my younger and more kulnerable years my father gake me so
8400	in my younger and more kulnerable years my father gake me so
8500	in my younger and more kulnerable years my father gake me so
8600	in my younger and more kulnerable years my father gake me so
8700	in my younger and more kulnerable years my father gake me so
8800	in my younger and more vulnerable years my father gave me so
8900	in my younger and more vulnerable years my father gave me so
9000	in my younger and more vulnerable years my father gave me so
9100	in my younger and more vulnerable years my father gave me so
9200	in my younger and more vulnerable years my father gave me so
9300	in my younger and more vulnerable years my father gave me so
9400	in my younger and more vulnerable years my father gave me so
9500	in my younger and more vulnerable years my father gave me so
9600	in my younger and more vulnerable years my father gave me so
9700	in my younger and more vulnerable years my father gave me so
9800	in my younger and more vulnerable years my father gave me so
9900	in my younger and more vulnerable years my father gave me so
10000	in my younger and more vulnerable years my father gave me so

Table 4.3: First 60 characters of Decrypted Text

Encoded symbol	**Original symbol**
9	=
i	
r	-
""""	;
!	:
f	!
;	?
c	/
u	,
?	""""
*	(
s	)
z	[
p	]
6	*
0	1
q	2
o	3
a	4
w	5
h	6
=	7
3	8
2	9
g	a
]	b
y	c
k	d
l	e
j	f
)	g
x	h
b	i
7	j
v	k
1	l
/	m
.	n
	o
e	p
5	q
-	r
t	s
[	t
4	u
:	v
(	w
,	x
n	y
m	z
8	

Table 4.4: Random initial mapping used for Iteration 0

Listing 3: Code - Q5.(d)

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import matplotlib.cm as cm
import matplotlib.colors as mcolors
from collections import Counter
from unidecode import unidecode
from operator import itemgetter
import random
import math
import csv

def text_cleaning(
symbol_file='symbols.txt',
message_file='message.txt',
training_file='war_and_peace_tolstoi.txt'
):
    # Loading the text, and estimating the transition probabilities

    with open('war_and_peace_tolstoi.txt', 'r', encoding='utf-8') as file:
        warpeace = file.read().lower()

    with open('symbols.txt', 'r', encoding='utf-8') as symbol_file:
        symbols_list = symbol_file.read().splitlines()

    with open('message.txt', 'r', encoding='utf-8') as message_file:
        message = message_file.read()

    # Cleaning the text from all unwanted accents, but it makes new symbols appear
    # so we also remove them.
    warpeace_clean = unidecode(warpeace)
    warpeace_clean = warpeace_clean.replace('\n', ' ').replace('#', '')
    warpeace_clean = warpeace_clean.replace('%', '').replace('$', '')

    return symbols_list, message, warpeace_clean

def transition_counts(warpeace_clean):
    # Keeping track of the number of counts of:
    # 1) Each symbol
    symbols_found = Counter()
    # 2) Each pair of symbols
    pair_found = Counter()

    for i in range(len(warpeace_clean)-1):
        s_i_minus_1 = warpeace_clean[i]
        s_i = warpeace_clean[i+1]
        pair_found[(s_i_minus_1, s_i)] += 1
```

```python
            symbols_found[s_i_minus_1] += 1

    return symbols_found, pair_found

def psi_phi_calculations(symbols_found, pair_found, precision=1e-9):

    transition_probabilities = {}

    for (s_i_minus_1, s_i), frequency_pair in pair_found.items():
        transition_probabilities[s_i_minus_1, s_i] = frequency_pair/(symbols_found[s_i_minus_1])

    unsorted_symbol_states = set([i[0] for i in transition_probabilities.keys()] \
    + [i[1] for i in transition_probabilities.keys()])
    symbol_states = sorted(unsorted_symbol_states)
    n_symbols = len(symbol_states)

    symbol_indexes = {symbol: i for i, symbol in enumerate(symbol_states)}

    # Transition probabilities matrix
    psi = np.zeros((n_symbols, n_symbols))
    for (s_i_minus_1, s_i), p_current_given_previous in transition_probabilities.items():
        i = symbol_indexes[s_i_minus_1]
        j = symbol_indexes[s_i]
        psi[i, j] = p_current_given_previous

    # Adapting for ergodicity, after answering question 5.(e)
    for i in range(np.shape(psi)[0]):
        for j in range(np.shape(psi)[0]):
            if psi[i,j] == 0:
                psi[i,j] += precision


    eigenvalues, eigenvectors = np.linalg.eig(psi.T)
    stationary_vector = eigenvectors[:, np.isclose(eigenvalues, 1)]
    stationary_distribution = (stationary_vector.real / np.sum(stationary_vector))

    stationary_distribution_dict = \
    {symbol: stationary_distribution[symbol_indexes[symbol]].real for symbol in symbol_states}

    stationary_probabilities = [i for i in stationary_distribution_dict.values()]

    return psi, stationary_distribution_dict, symbol_indexes, stationary_probabilities


def save_csv_data_transition(psi, filename="transition_probabilities.csv"):
    psi_array = np.array(psi)
    clean_data_transition = [[f"{value:.2e}" for value in row] for row in psi_array]

    with open(filename, mode="w", newline="") as file:
```

```
        writer = csv.writer(file)
        writer.writerows(clean_data_transition)

def save_csv_data_stationary(
stationary_distribution_dict,
filename="stationary_probabilities.csv"
):
    with open(filename, mode="w", newline="") as file:
        writer = csv.writer(file)
        writer.writerow(["Symbol", "Stationary Probability"])  # Header row
        for symbol, probability in stationary_distribution_dict.items():
            clean_data=f"{probability:.3e}"
            writer.writerow([symbol, clean_data])

# Summary tables - Q1.a)
def plot_heatmap_transition(psi, save=False):
    plt.figure(figsize=(7, 7))
    plt.imshow(psi, cmap="viridis", interpolation='nearest')
    plt.colorbar(label='Probability')
    plt.title('Pairs probabilities psi(alpha, beta)')
    plt.xlabel("Symbol index")
    plt.ylabel("Symbol index")
    plt.tight_layout()
    if save:
        plt.savefig('heatmap_transition.png', format="png", dpi=300, bbox_inches="tight")
    plt.show()


def plot_bar_stationary(symbols_list, stationary_probabilities, cmap_colors='winter', save=False):
    stationary_probabilities = [float(prob) for prob in stationary_probabilities]
    cmap = plt.get_cmap(cmap_colors, int(np.ceil(len(symbols_list))))
    colors = [cmap(i) for i in np.linspace(0, 0.8, len(symbols_list))]

    plt.figure(figsize=(14, 6))
    plt.bar(symbols_list, stationary_probabilities, color=colors)
    plt.title("Stationary distributions phi(gamma) of each symbol", fontsize=16)
    plt.xlabel("Symbols", fontsize=14)
    plt.ylabel("Stationary distribution probabilities", fontsize=14)
    plt.tight_layout()
    if save:
        plt.savefig('bar_stationary.png', format="png", dpi=300, bbox_inches="tight")
    plt.show()

# Q5.d)
def random_proposal(target_list):
    indexes = np.arange(0, len(target_list)).tolist()
    new_indexes = random.sample(indexes, len(indexes))
    proposal_list = [target_list[i] for i in new_indexes]
    proposal_mapping = {proposal_list[i] : target_list[i] for i in range(len(target_list))}
```

```python
    return proposal_mapping

def proposal_mechanism(mapping):
    mapping_proposal = mapping.copy()
    keys_list = list(mapping.keys())

    indexes = np.arange(0, len(mapping)).tolist()
    swap_index = random.sample(indexes, 2)

    tmp = mapping_proposal[keys_list[swap_index[0]]]
    mapping_proposal[keys_list[swap_index[0]]] = mapping_proposal[keys_list[swap_index[1]]]
    mapping_proposal[keys_list[swap_index[1]]] = tmp

    return mapping_proposal


def jointprob_e_given_sig(
        message : str,
        symbol_indexes : dict,
        mapping : dict,
        stationary_distribution_dict : list,
        psi: np.ndarray
        ):

    log_jointprob_e = 0

    reversed_symbol_indexes = {index: symbol for symbol, index in symbol_indexes.items()}

    first_key = message[0]
    inverse_mapping_first_key = mapping[first_key]
    first_key_symbol = reversed_symbol_indexes[symbol_indexes[inverse_mapping_first_key]]

    stationary_first_key = stationary_distribution_dict[first_key_symbol]

    log_jointprob_e += math.log(stationary_first_key)

    prev = message[0]
    for i in range(1, len(message)):
        prev = message[i-1]
        current = message[i]
        reverse_map_i_minus_1 = mapping[prev]
        reverse_map_i = mapping[current]
        index1, index2 = symbol_indexes[reverse_map_i_minus_1], symbol_indexes[reverse_map_i]

        transition_prob_pair = psi[index1, index2]

        if transition_prob_pair > 0:
            log_jointprob_e += math.log(transition_prob_pair)
        else:
```

```
                log_jointprob_e += math.log(1e-10)

    return log_jointprob_e

def mcmc_step(
        current_mapping,
        stationary_distribution_dict,
        psi,
        message,
        symbol_indexes,
        ):

    accepted = False
    new_mapping = proposal_mechanism(current_mapping)

    log_joint_sig = jointprob_e_given_sig(message,
                                          symbol_indexes,
                                          current_mapping,
                                          stationary_distribution_dict,
                                          psi
                                          )
    log_joint_sig_prime = jointprob_e_given_sig(message,
                                          symbol_indexes,
                                          new_mapping,
                                          stationary_distribution_dict,
                                          psi
                                          )

    if log_joint_sig_prime > log_joint_sig:     # The ratio was not possible to compute without
                                                # introducing a runtime error.
        A = 1
    else:
        A = math.exp(log_joint_sig_prime - log_joint_sig)

    U_i = random.uniform(0, 1)

    if U_i <= A:
        returned_mapping = new_mapping
        accepted = True
    else:
        returned_mapping = current_mapping
        accepted = False

    return returned_mapping, accepted


def decrypting_first60(mapping, message, end=60):

    decrypted_message = ""
```

```
    for char in message[:end]:
        decrypted_char = mapping[char]
        decrypted_message += decrypted_char

    return decrypted_message


def run_mcmc(
        n_iterations,
        initial_mapping,
        stationary_distribution_dict,
        psi,
        message,
        symbol_indexes
        ):

    current_mapping = initial_mapping
    mappings_list = [current_mapping]
    n_accepted = 0

    for i in range(n_iterations):
        current_mapping, accepted = mcmc_step(current_mapping,
                                              stationary_distribution_dict,
                                              psi,
                                              message,
                                              symbol_indexes
                                              )
        mappings_list.append(current_mapping)
        if i % 100 == 0:
            print("Iteration ", i, " : ", decrypting_first60(current_mapping, message))
        n_accepted += accepted

    return mappings_list
def decrypting_message_final(message,
                             symbols_list,
                             stationary_distribution_dict,
                             stationary_probabilities,
                             psi,
                             symbol_indexes,
                             n_iterations=2000,
                             random_map=False
                             ):

    proposal_encoding = random_proposal(symbols_list)

    mapping_chain = run_mcmc(n_iterations,
    proposal_encoding,
    stationary_distribution_dict,
    psi,
```

```
    message,
    symbol_indexes)

    return mapping_chain

def main():
    # Opening all the files and cleaning the training text
    symbols_list, message, warpeace_clean = text_cleaning()

    # Counting the frequencies of each symbol and each pair
    symbols_found, pair_found = transition_counts(warpeace_clean)
    # Calculating the transition probability matrix, the stationary distribution's
    # probabilities (stored in a dictionary), the symbol's indexes (in the original symbol file)
    # and the corresponding stationary probabilities.
    psi, stationary_distribution_dict, symbol_indexes, stationary_probabilities
    = psi_phi_calculations(
    symbols_found,
    pair_found
    )

    # Explanatory plots to understand better the data
    plot_heatmap_transition(psi)
    plot_bar_stationary(symbols_list, stationary_probabilities)

    # Saving the plots if wanted
    save_plots = False
    if save_plots:
        save_csv_data_transition(psi, symbols_list)
        save_csv_data_stationary(stationary_distribution_dict)

    # Choosing the initial mapping as random
    proposal_encoding = random_proposal(symbols_list)
    # Decrypting the whole message with 30000 iterations
    A = decrypting_message_final(message,
    symbols_list,
    stationary_distribution_dict,
    stationary_probabilities,
    psi,
    symbol_indexes,
    n_iterations=30000,
    random_map=False)

    # Decrypting the entire message and comparing it to the original.
    print("Original encoding : ", decrypting_first60(mapping=proposal_encoding,
    message=message,
    end=(len(message)-1)))
    print("Decoded text : ", decrypting_first60(mapping=A[-1],
    message=message,
    end=(len(message)-1)))
```

```

# Running the main() function to observe all results
main()
```

4.5 Question 5.(e)

By definition, if a Markov Chain is ergotic, it can reach a unique steady-state, independent of the starting point. To be ergotic, a Markov Chain must be:
1) aperiodic, *i.e.* $P(X_n = s|X_{n-1} = s) = p_{s\to s} > 0$, implying it must also have a non-zero diagonal. In other words, it must be able to stay in every state with a non-null probability.
2) irreducible, *i.e.* if there is a path (with non-zero probability) from each state to every other state in the transition graph.
In our case, if there are some zero entries for $\phi(\alpha, \beta)$, then our distribution is not irreducible nor aperiodic, meaning $p_{s\to s}$ is not > 0. This makes sense since, in English, and especially in books such as War and Peace by Leo Tolstoy, there is a very low chance (if not a probability 0) to see a sequence of identical symbols (e.g. 'aaaaaaaa', 'bbbb', ...) for every symbol (53 in total). Moreover, these sequences of identical symbols must be seen much more than once for their probabilities to be non-negligible compared to other transitions and thus not to be neglected by the computer's precision, e.g. in the book's summary, the chapter titles XII, XIII, etc. introduce a transition probability between some letters, but it might not be sufficient enough since it happens a few times only in the whole book. Regarding the irreducibility of the chain, the same reason applies: states cannot be reached from all possible other states ('xyz' never happens a lot in a well-written book).

A possible solution to restore the ergodicity of the chain is to make the transition probabilities non-zero. For instance, we could set a threshold (e.g. ranging from 1e-8 to 1e-10), and correct all transition probabilities (including the diagonal elements) below that threshold to make them equal to the threshold. It must be done before calculating the stationary distributions. This will force back the ergodicity of the chain by permitting a restoration of both aperiodicity and irreducibility of the chain. However, this threshold must be low enough and have no incidence on the other symbols probability of occurring, because the only goal of the threshold is to restore ergodicity and not replace symbols we are sure of.

4.6 Question 5.(f)

As stated for the previous question, this method is not perfect for decoding, and there are flaws in the approach we will study.
The approach relies on analyzing how english words are formed to obtain a sequence of symbols making sense. Symbol probabilities alone won't be sufficient, we might obtain a sequence of spaces with a few of most-used letters in the english language (a, e, etc.), and there might even be no letters at all since the space probability is very high compared to them. The sequences of symbols won't make sense and the program won't fulfill the initial task, a sufficient reason to refute this idea.
Using a second-order Markov chain can introduce some computation issues, because the program will have to work in three dimensions for transition probabilities (one for each symbol s_i, s_{i-1} and s_{i-2}). This makes the computation much more complex because the transition probability matrix will have $N \times N \times N$ in addition to a third dimension. Since eigenvalues are calculated for 2D matrices, we might also have to adapt the code to fit 3D eigenvalues/eigenvectors methods. An additional information to refute this idea of second-order Markov chain is the fact the training data must be much higher for the model to be as accurate as a first-order Markov Chain.
If the encryption scheme allows two symbols to be mapped to the same encrypted value, the approach will simply first attribute spaces, and then decrypt the most used word in the training set that has the same number of letters. We will obtain a sentence which won't make sense, and thus it will not decrypt the message as intended. Or we could even simply have a sequence of random spaces and letters, since

only depending on stationary distributions.
If we use the approach on Chinese language for instance, it his highly probable the program won't be able to decrypt the message. Firstly, this is because Chinese signs are unique and don't work as english and other western languages where letters and symbols represent how we pronounce the word. Chinese language associates images (each having a full meaning alone) to each symbol, which makes their use much less frequent in sentences than letters in english and will be much more complicated for the computer to grasp. For example, if the sentence to decrypt is "The dog eats outside.", the computer won't make the difference between "cat" and "dog", between "eating" and "playing", etc. It will likely decrypt the wrong sentence. Additionally, their transition probabilities and stationary probabilities might end up to be very low values, which will be harder for the program to decrypt. Using a two-symbol swap as we did in the approach is also too low, and the time for the program to decrypt the language will be too higher. Hence, to resume what will happen knowing all this information: the program will offer a two symbol swap; the difference in probability will be so low, that they will most likely be rejected during the acceptance step of the Metropolis-Hasting algorithm. We will end-up in an infinite loop.
A remark we can make when displaying the whole decrypted message is the fact some letters still can't be decrypted right by the program since they are not very used (e.g. j, q, z, k and probably x), even after a high number of iterations. What we can see in common for these letters is their very low stationary probabilities. Their probabilities are somehow lower or equal to punctuation probabilities. We could also apply the threshold method to the stationary vector, but I doubt it will solve the issue, because they might have an impact on highly-probable symbols by replacing them where they were expected.

5 | Question 7 - Optimization

5.1 Question 7.(a)

We want to find the local extrema of the function $f(x, y) = x + 2y$ subject to the constraints $y^2 + xy = 1$. This is a constrained optimization problem, with an equality constraint.

We name $g(x, y) = y^2 + xy - 1$, where finding the values of x and y such that $g(x, y) = 0$ is a reformulation of the constraint of our problem. Introducing a Lagrange multiplier, denoted by λ, the problem is equivalent to solving the following system of equations:

$$\begin{cases} \nabla f(x, y) + \lambda \nabla g(x, y) = 0 & (E_1) \\ \\ g(x, y) = 0 & (E_2) \end{cases} \tag{5.1}$$

∇f and ∇g are in three dimensions, one for each of the following variables: x, y, λ.
Differentiating with respect to each variables the equations, we get:

$$\frac{\partial E_1}{\partial x} = 0 \iff \frac{\partial}{\partial x}\big(x + 2y + \lambda(y^2 + xy - 1)\big) = 0 \iff \lambda y + 1 = 0 \tag{5.2}$$

Calculated a similar manner:

$$\frac{\partial E_1}{\partial y} = 0 \iff 2 + 2\lambda y + \lambda x = 0 \tag{5.3}$$

$$\frac{\partial E_1}{\partial \lambda} = 0 \iff y^2 + xy - 1 = 0 \tag{5.4}$$

Thus:

$$\begin{cases} \lambda y + 1 = 0 \\ 2 + 2\lambda y + \lambda x = 0 \\ y^2 + xy - 1 = 0 \end{cases} \tag{5.5}$$

Equivalent to:

$$\begin{cases} \lambda = -\frac{1}{y} \\ 2 + 2\left(-\frac{1}{y}\right) y + (-\frac{1}{y})x = 0 \\ y^2 + xy - 1 = 0 \end{cases} \quad \begin{cases} \lambda = -\frac{1}{y} \\ -\frac{x}{y} = 0 \\ y^2 + xy - 1 = 0 \end{cases} \quad \begin{cases} \lambda = -\frac{1}{y} \\ x = 0 \\ y^2 - 1 = 0 \end{cases}$$

We obtain as results $y = \pm 1,\ x = 0,$ and $\lambda = \mp 1$. Thus, we obtained the solutions to the system and to the optimization problem, *i.e.* the locations of the local extrema:

$$(x, y) \in \mathcal{S} = \{(0, 1), (0, -1)\}$$

5.2 Question 7.(b)

We assume we dispose of a method to evaluate the exponential function $\exp(x) = e^x$. We would like to evaluate the function $\ln a$, for a given $a \in \mathbb{R}_+$ using Newton's method.

5.2.1 Question 7.(b).(i)

Firstly, we can recall the exponential function's inverse mapping is the logarithmic function, *i.e.* $\exp(\ln x) = x$, and we can additionally write $\exp^{-1} x = \ln x$.
Thus, we can use the following function $f(x, a)$ to which Newton's Method can be applied to x such that $x = \ln a$:

$$f(x, a) = e^x - a \tag{5.6}$$

And by solving the equation $f(x, a) = 0$, we obtain the logarithm of a, *i.e.* $\ln a$.

5.2.2 Question 7.(b).(ii)

The update equation in Newton's method to search for the root of f, $f(x, a) = 0$, is found using the following formula:

$$x_{n+1} = x_n - \frac{f(x_n, a)}{f'(x_n, a)} \tag{5.7}$$

Calculating the first derivative of f with respect to x_n:

$$f'(x_n, a) = \frac{\partial \left(e^{x_n} - a\right)}{\partial x_n}$$

$$f'(x_n, a) = e^{x_n} \tag{5.8}$$

Therefore, we obtain:

$$x_{n+1} = x_n - \frac{e^{x_n} - a}{e^{x_n}} \quad \Longleftrightarrow \quad x_{n+1} = x_n - 1 + \frac{1}{e^{x_n}} \tag{5.9}$$

6 | Question 8 - [BONUS] Eigenvalues as solutions of an optimization problem

We define A as a symmetric $n \times n$ - matrix, and:

$$q_A(x) = x^T Ax \qquad \text{and} \qquad R_A(x) = \frac{q_A(x)}{||x||^2} = \frac{x^T Ax}{x^T x} \quad \text{for } x \in \mathbb{R}^n. \tag{6.1}$$

We remind the purpose of this problem is to verify the fact: *If A is a symmetric $n \times n$-matrix, the optimization problem $x^* = \arg\max_{x \in \mathbb{R}^n} R_A(x)$ has a solution, $R_A(x^*)$ is the largest eigenvalue of A, and x^* is a corresponding eigenvector.*

6.1 Question 8.(a)

The **Extreme Value theorem** states the following:

If a function f is continuous on a closed interval $[a;b]$, then f attains both a maximum and a minimum in that interval.

$$i.e. \quad \exists x_{max} \in [a;b] \quad / \quad \forall x \in [a;b] \, f_{max} \geq f(x) \tag{6.2}$$

$$and$$

$$\exists x_{min} \in [a;b] \quad / \quad \forall x \in [a;b] \, f_{min} \leq f(x) \tag{6.3}$$

The conditions for the validity of such theorem are requirements int he function f that f should be both **continuous** and **defined on a closed interval** (the function must be "*compact*").

We would like to show that $\sup_{x \in \mathbb{R}^n} R_A(x)$ using the extreme value theorem. Firstly, since $\mathbb{R}^n$ is not compact, we need to find an interval of definition to be compact while having an equivalent supremum. Let's prove the unit sphere $S = \{x \in \mathbb{R}^n | \|x\| = 1\}$, which is a compact set, has an equivalent supremum.

Then, defining $s \in \mathbb{R}^n$, we notice $\forall s \in \mathbb{R}^n, \exists x \in \mathbb{S} \; / \; x = \frac{s}{\|s\|}$. This expression is true, because by dividing by the norm of each vector in the space $\mathbb{R}^n$, we obtain a unit vector comprised in the interval S. By replacing in $R_A(x)$'s expression, we get:

$$\begin{aligned}
\sup_{\{x \in S\}} R_A(x) &= \sup_{\{x \in \mathbb{R}^n | \|x\| = 1\}} \frac{x^T Ax}{x^T x} \\
&= \sup_{\{\frac{s}{\|s\|} \in \mathbb{R}^n | \left\| \frac{s}{\|s\|} \right\| = 1\}} \frac{\left(\frac{s}{\|s\|}\right)^T A \frac{s}{\|s\|}}{\left(\frac{s}{\|s\|}\right)^T \frac{s}{\|s\|}} \\
&= \sup_{\{\frac{s}{\|s\|} \in \mathbb{R}^n | \left\| \frac{s}{\|s\|} \right\| = 1\}} \frac{\frac{s^T}{\|s\|} A \frac{s}{\|s\|}}{\frac{s^T}{\|s\|} \frac{s}{\|s\|}}
\end{aligned}$$

And since $\|s\|$ is a scalar:

$$= \sup_{\{\frac{s}{\|s\|}\in\mathbb{R}^n | \left\|\frac{s}{\|s\|}\right\|=1\}} \frac{\frac{s^T}{\|s\|^2} A s^T}{\frac{s^T s}{\|s\|^2}}$$

$$= \sup_{\{\frac{s}{\|s\|}\in\mathbb{R}^n | \left\|\frac{s}{\|s\|}\right\|=1\}} \frac{s^T A s}{s^T s}$$

And switching back to the domain of definition of s, since the :

$$= \sup_{\{s\in\mathbb{R}^n\}} \frac{\frac{s^T}{\|s\|^2} A s^T}{\frac{s^T s}{\|s\|^2}}$$

$$= \sup_{\{s\in\mathbb{R}^n\}} R_A(s)$$

Leading to the final expression of:

$$\sup_{x\in S} R_A(x) = \sup_{s\in\mathbb{R}^n} R_A(s) \tag{6.4}$$

Which proves the supremum of the set $S = \{x \in \mathbb{R}^n | \|x\| = 1\}$ is equivalent to that of $\mathbb{R}^n$.

Since the set S is continuous (because its expression, the unit sphere, is valid for all element of $\mathbb{R}^n$), and is bounded (its images limits are finite numbers, *i.e.* all equal to 1), then the **Extreme Value theorem applies** to our case and therefore, the supremum of $R_A(x), x \in \mathbb{R}^n$ is attained when the supremum of $R_A(x_s), x_s \in S$ is attained.

6.2 Question 8.(b)

We define $\lambda_1 \geq ... \geq \lambda_n$ as the eigenvalues of A in descending order, and $\xi_1, ..., \xi_n$ their corresponding eigenvectors that form an ONB.
Using our previous results, we start from the unit sphere:

$$\sup_{x\in\mathbb{R}^n | \|x\|=1} \frac{x^T A x}{\|x\|} = \sup_{x\in\mathbb{R}^n | \|x\|=1} x^T A x \tag{6.5}$$

For all $x \in \mathbb{R}^n$, we can rewrite using A's eigenvectors (since they form an ONB) as:

$$x = \sum_{i=1}^{n} (\xi_i^T x)\xi_i \tag{6.6}$$

Thus, replacing in our problem, we get:

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

If we recall the following property of eigenvectors of the matrix A:

$$A\xi_i = \lambda_i \xi_i$$

Then since A is symmetric, the property is valid for the transpose of eigenvectors:

$$A\xi_i^T = \lambda_i \xi_i^T$$

Therefore:

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

Using the property of transpose:

$$(AB)^T = B^T A^T$$

Then:

$$\begin{aligned} x^T A x = \left(\sum_{i=1}^{n} \xi_i^T (\xi_i^T x)^T\right) \sum_{i=1}^{n} \lambda_i \xi_i^T x \xi_i &= \left(\sum_{i=1}^{n} \xi_i^T x^T (\xi_i^T)^T\right) \sum_{i=1}^{n} \lambda_i \xi_i^T x \xi_i \\ &= \left(\sum_{i=1}^{n} \xi_i^T x^T \xi_i\right) \sum_{i=1}^{n} \lambda_i \xi_i^T x \xi_i \end{aligned}$$

$$x^T A x = \left(\sum_{i=1}^{n} \xi_i^T x^T \xi_i\right) \left(\lambda_i \sum_{i=1}^{n} \xi_i^T x \xi_i\right) \tag{6.7}$$

Since λ_i is a scalar.
And, since $\xi_i, i \in \{1, ..., n\}$ form an orthonormal basis (ONB), then:

$$\xi_i^T \xi_j = \begin{cases} 1, & i = j \\ 0, & i \neq j \end{cases} \tag{6.8}$$

Thus:

$$x^T A x = \sum_{i=1}^{n} \lambda_i \xi_i^T x^T x \xi_i$$

$$x^T A x = \sum_{i=1}^{n} \lambda_i x_i^T \xi_i$$

And once again using the property of the ONB vectors 6.8, then:

$$x^T A x = \sum_{i=1}^{n} \lambda_i \tag{6.9}$$

Since we defined the eigenvalues of A previously as $\lambda_1 \geq ... \geq \lambda_n$ in descending order, then:

$$R_A(x \in S) = \sum_{i=1}^{n} \lambda_i \leq \lambda_1 \tag{6.10}$$

Which was what we intended to prove.

6.3 Question 8.(c)

We consider $x \in \mathbb{R}^n \backslash \text{span}\{\xi_1, ...\xi_k\}$, which means all x in $\mathbb{R}^n$ that is not a part of $\text{span}\{\xi_1, ...\xi_k\}$. The $\text{span}\{\xi_1, ...\xi_k\}$ contains all the linear combinations of linearly independent eigenvectors corresponding to λ_1, $k \leq n$.
Therefore, x can be rewritten as we did before for Question 8.(b), but excluding all the eigenvectors corresponding to λ_1:

$$x = \sum_{i=k+1}^{n} (\xi_i^T x)\xi_i \tag{6.11}$$

Using our previous result for Question 8.(b), we can state (remembering the eigenvalues are ordered in descending order and thus λ_{k+1} is the maximum):

$$R_A(x) = x^T Ax = \sum_{i=k+1}^{n} \lambda_i \leq \lambda_{k+1}$$

And since $\lambda_{k+1} < \lambda_1$, then:

$$R_A(x) < \lambda_1 x \in \mathbb{R}^n \backslash \text{span}\{\xi_1, ...\xi_k\} \tag{6.12}$$

Which is what we needed to prove.