Assignment 4

Callum Lau, 19102521

Approximate Inference

December 12, 2019

1 Deriving Gibbs Sampling for LDA

1.1 Part (a)

From the graphical model, we can infer that the joint distribution takes the form (note that we leave out the conditioning on α and β for notational simplicity):

$$p(\boldsymbol{z}, \boldsymbol{x}, \boldsymbol{\phi}, \boldsymbol{\theta}) = \left(\prod_{k=1}^{K} p(\boldsymbol{\phi}_k|\beta)\right)\left(\prod_{d=1}^{D} p(\boldsymbol{\theta}_d|\alpha)\prod_{i=1}^{N_d} p(z_{id}|\boldsymbol{\theta}_d)p(x_{id}|z_{id}, \boldsymbol{\phi}_{1:K})\right) \tag{1}$$

Since $\boldsymbol{\phi}_k \sim Dir(\beta)$:

$$\prod_{k=1}^{K} p(\boldsymbol{\phi}_k|\beta) = \prod_{k=1}^{K} \frac{\Gamma(\sum_{w=1}^{W}\beta)}{\prod_{w=1}^{W}\Gamma(\beta)}\prod_{w=1}^{W}(\phi_{kw})^{\beta-1} = \left(\frac{\Gamma(W\beta)}{\Gamma(\beta)^W}\right)^K \prod_{k=1}^{K}\prod_{w=1}^{W}(\phi_{kw})^{\beta-1} \tag{2}$$

Since $\boldsymbol{\theta}_d \sim Dir(\alpha)$:

$$\prod_{d=1}^{D} p(\boldsymbol{\theta}_k|\alpha) = \prod_{d=1}^{D} \frac{\Gamma(\sum_{k=1}^{K}\alpha)}{\prod_{k=1}^{K}\Gamma(\alpha)}\prod_{w=1}^{W}(\theta_{dk})^{\alpha-1} = \left(\frac{\Gamma(K\alpha)}{\Gamma(\alpha)^K}\right)^D \prod_{d=1}^{D}\prod_{k=1}^{K}(\theta_{dk})^{\alpha-1} \tag{3}$$

Since $z_{id}|\boldsymbol{\theta}_d \sim Discrete(\boldsymbol{\theta}_d)$:

$$\begin{aligned}\prod_{d=1}^{D}\prod_{i=1}^{N_d} p(z_{id}|\boldsymbol{\theta}_d) &= \prod_{d=1}^{D}\prod_{i=1}^{N_d}\prod_{k=1}^{K}(\theta_{dk})^{\delta(z_{id}=k)} = \prod_{d=1}^{D}\prod_{k=1}^{K}(\theta_{dk})^{\sum_{i=1}^{N_d}\delta(z_{id}=k)} \\ &= \prod_{d=1}^{D}\prod_{k=1}^{K}(\theta_{dk})^{A_{dk}}\end{aligned} \tag{4}$$

Since $x_{id}|z_{id}, \boldsymbol{\phi}_{z_{id}} \sim Discrete(\boldsymbol{\phi}_{z_{id}})$:

$$\prod_{d=1}^{D}\prod_{i=1}^{N_d} p(x_{id}|z_{id}, \boldsymbol{\phi}_{1:K}) = \prod_{d=1}^{D}\prod_{i=1}^{N_d}\prod_{k=1}^{K} p(x_{id}|\phi_k) = \prod_{d=1}^{D}\prod_{i=1}^{N_d}\prod_{k=1}^{K}\prod_{w=1}^{W}(\phi_{kw})^{\delta(x_{id}=w)\delta(z_{id}=k)}$$

$$= \prod_{k=1}^{K}\prod_{w=1}^{W}(\phi_{kw})^{\sum_{d=1}^{D}\sum_{i=1}^{N_d}\delta(x_{id}=w)\delta(z_{id}=k)} = \prod_{k=1}^{K}\prod_{w=1}^{W}(\phi_{kw})^{B_{kw}} \tag{5}$$

Then combining equations (2) - (5) we can write (1) as:

$$p(\boldsymbol{z},\boldsymbol{x},\boldsymbol{\phi},\boldsymbol{\theta}) = \underbrace{\left(\frac{\Gamma(W\beta)}{\Gamma(\beta)^W}\right)^K \prod_{k=1}^{K}\prod_{w=1}^{W}(\phi_{kw})^{B_{kw}+\beta-1}}_{(A)} \underbrace{\left(\frac{\Gamma(K\alpha)}{\Gamma(\alpha)^K}\right)^D \prod_{d=1}^{D}\prod_{k=1}^{K}(\theta_{dk})^{A_{dk}+\alpha-1}}_{(B)} \tag{6}$$

1.2 Part (b)

We now reintroduce the N_d and M_k terms, by rewriting (A) and (B) as posterior Dirichlets on $\boldsymbol{\phi}$ and $\boldsymbol{\theta}$. In the following we simply multiply by a constant 1 (leaving (A) and (B) unchanged). However, we write this constant in terms of the normalisation term for the Dirichlet distribution:

$$\begin{aligned}
(A) &= \left(\frac{\Gamma(W\beta)}{\Gamma(\beta)^W}\right)^K \prod_{k=1}^{K}\prod_{w=1}^{W}(\phi_{kw})^{B_{kw}+\beta-1} \\
&= \left(\frac{\Gamma(W\beta)}{\Gamma(\beta)^W}\right)^K \prod_{k=1}^{K}\frac{\prod_{w=1}^{W}\Gamma(B_{kw}+\beta)}{\Gamma(\sum_{w=1}^{W}B_{kw}+W\beta)} \prod_{k=1}^{K}\left(\frac{\Gamma(\sum_{w=1}^{W}B_{kw}+W\beta)}{\prod_{w=1}^{W}\Gamma(B_{kw}+\beta)}\prod_{w=1}^{W}(\phi_{kw})^{B_{kw}+\beta-1}\right) \\
&= \left(\frac{\Gamma(W\beta)}{\Gamma(\beta)^W}\right)^K \prod_{k=1}^{K}\frac{\prod_{w=1}^{W}\Gamma(B_{kw}+\beta)}{\Gamma(M_k+W\beta)} \prod_{k=1}^{K}p(\phi_k|\boldsymbol{z},\boldsymbol{x})
\end{aligned} \tag{7}$$

Where the final line follows from the independence of each ϕ_k and the conjugacy of the Dirichlet distribution for the multinomial.

$$\begin{aligned}
(B) &= \left(\frac{\Gamma(K\alpha)}{\Gamma(\alpha)^K}\right)^D \prod_{d=1}^{D}\prod_{k=1}^{K}(\theta_{dk})^{A_{dk}+\alpha-1} \\
&= \left(\frac{\Gamma(K\alpha)}{\Gamma(\beta)^K}\right)^D \prod_{d=1}^{D}\frac{\prod_{k=1}^{K}\Gamma(A_{dk}+\alpha)}{\Gamma(\sum_{k=1}^{K}A_{dk}+K\alpha)} \prod_{d=1}^{D}\left(\frac{\Gamma(\sum_{k=1}^{K}A_{dk}+K\alpha)}{\prod_{k=1}^{K}\Gamma(A_{dk}+\alpha)}\prod_{k=1}^{K}(\theta_{dk})^{A_{dk}+\alpha-1}\right) \\
&= \left(\frac{\Gamma(K\alpha)}{\Gamma(\beta)^K}\right)^D \prod_{d=1}^{D}\frac{\prod_{k=1}^{K}\Gamma(A_{dk}+\alpha)}{\Gamma(N_d+K\alpha)} \prod_{d=1}^{D}p(\boldsymbol{\theta}_d|\boldsymbol{z})
\end{aligned} \tag{8}$$

Where the final line follows from the independence of each θ_d and the conjugacy of the Dirichlet distribution for the multinomial.
Since we have equations (7) and (8), we can quickly write down the sampling updates for $\boldsymbol{\theta}$ and $\boldsymbol{\phi}$:

$$(7) \implies p(\boldsymbol{\phi}_k|\boldsymbol{z},\boldsymbol{x}) = \frac{\Gamma(\sum_{w=1}^{W}B_{kw}+W\beta)}{\prod_{w=1}^{W}\Gamma(B_{kw}+\beta)}\prod_{w=1}^{W}(\phi_{kw})^{B_{kw}+\beta-1} = Dir(B_{k\cdot}+\beta)$$

$$(8) \implies p(\boldsymbol{\theta}_d|\boldsymbol{z}) = \frac{\Gamma(\sum_{k=1}^K A_{dk} + K\alpha)}{\prod_{k=1}^K \Gamma(A_{dk} + \alpha)} \prod_{k=1}^K (\theta_{dk})^{A_{dk}+\alpha-1} = Dir(A_{d\cdot} + \alpha)$$

Where $A_{d\cdot}, B_{k\cdot}$ indicate summations over the hidden variable. We derive the Gibbs update for $z_{id} = k$ in a slightly different way. We note that, via the graphical model:

$$\begin{aligned} p(z_{id} = k|z_{\neg(i,d)}, \theta_d, \phi_{1:K}) &= p(z_{id}|\theta_d, x_{id}, \phi_{1:K}) \\ &\propto p(x_{id}|z_{id}, \phi_{1:K}) \times p(z_{id}|\theta_d) \\ &= \phi_{kw} \cdot \theta_{dk} \end{aligned}$$

We can then normalise, to produce the true (normalised) update:

$$p(z_{id} = k|z_{\neg(i,d)}, \theta_d, \phi_{1:K}) = \frac{\phi_{kw} \cdot \theta_{dk}}{\sum_{k'=1}^K \phi_{k'w} \cdot \theta_{dk'}}$$

1.3 Part (c)

Integrating out is now simple since the joint is now factored into independent $\boldsymbol{\theta}$ and $\boldsymbol{\phi}$ Dirichlet posteriors:

$$\begin{aligned} p(\boldsymbol{z}, \boldsymbol{x}) &= \int_{\boldsymbol{\phi}} \int_{\boldsymbol{\theta}} p(\boldsymbol{z}, \boldsymbol{x}, \boldsymbol{\phi}, \boldsymbol{\theta}) d\boldsymbol{\theta} d\boldsymbol{\phi} = \int_{\boldsymbol{\phi}} (A) d\boldsymbol{\phi} \int_{\boldsymbol{\theta}} (B) d\boldsymbol{\theta} \\ &= \left(\frac{\Gamma(W\beta)}{\Gamma(\beta)^W}\right)^K \prod_{k=1}^K \frac{\prod_{w=1}^W \Gamma(B_{kw} + \beta)}{\Gamma(M_k + W\beta)} \int_{\boldsymbol{\phi}} \prod_{k=1}^K p(\phi_k|\boldsymbol{z}, \boldsymbol{x}) d\boldsymbol{\phi} \\ &\times \left(\frac{\Gamma(K\alpha)}{\Gamma(\beta)^K}\right)^D \prod_{d=1}^D \frac{\prod_{k=1}^K \Gamma(A_{dk} + \alpha)}{\Gamma(N_d + K\alpha)} \int_{\boldsymbol{\theta}} \prod_{d=1}^D p(\boldsymbol{\theta}_d|\boldsymbol{z}) d\boldsymbol{\theta} \end{aligned}$$

Now since all the $\int_{\boldsymbol{\phi}_k} p(\phi_k|\boldsymbol{x}, \boldsymbol{z}) d\boldsymbol{\phi}_k$ and $\int_{\boldsymbol{\theta}_d} p(\boldsymbol{\theta}_d|\boldsymbol{x}, \boldsymbol{z}) d\boldsymbol{\theta}_d$ terms are all independent we can integrate over their supports independently. The integral over the support of a distribution must be one and therefore the all integral terms just integrate to one. Therefore we get that:

$$p(\boldsymbol{z}, \boldsymbol{x}) = \left(\frac{\Gamma(W\beta)}{\Gamma(\beta)^W}\right)^K \prod_{k=1}^K \frac{\prod_{w=1}^W \Gamma(B_{kw} + \beta)}{\Gamma(M_k + W\beta)} \left(\frac{\Gamma(K\alpha)}{\Gamma(\beta)^K}\right)^D \prod_{d=1}^D \frac{\prod_{k=1}^K \Gamma(A_{dk} + \alpha)}{\Gamma(N_d + K\alpha)} \tag{9}$$

1.4 Part (d)

Dropping the normalisation constants we can write (9) as:

$$p(\boldsymbol{z}, \boldsymbol{x}) \propto \prod_{k=1}^K \frac{\prod_{w=1}^W \Gamma(B_{kw} + \beta)}{\Gamma(M_k + W\beta)} \prod_{d=1}^D \frac{\prod_{k=1}^K \Gamma(A_{dk} + \alpha)}{\Gamma(N_d + K\alpha)}$$

And rewriting the equation in terms of Beta functions:

$$= \prod_{k=1}^{K} B(B_{k\cdot} + \beta) \cdot \prod_{d=1}^{D} B(A_{d\cdot} + \alpha) \tag{10}$$

Let the superscript $^{-(i',d')}$ stand for omitting the i'th word in the d'th document out of any calculations. Using this notation we calculate the Collapsed Gibbs update for $z_{i'd'} = k'$ where $x_{i'd'} = w'$:

$$p(z_{i'd'} = k' | \boldsymbol{z}^{-(i',d')}, \boldsymbol{x}^{-(i',d')}, x_{i'd'} = w') = \frac{p(\boldsymbol{z}, \boldsymbol{x})}{p(\boldsymbol{z}^{-(i',d')}, \boldsymbol{x})}$$

Using (10) we can express this in terms of Beta functions:

$$= \underbrace{\prod_{k=1}^{K} \frac{B(B_k + \beta)}{B(B_k^{-(i',d')} + \beta)}}_{(C)} \cdot \underbrace{\prod_{d=1}^{D} \frac{B(A_d + \alpha)}{B(A_d^{-(i',d')} + \alpha)}}_{(D)}$$

We now evaluate terms (C) and (D) separately. Note that for (C) all the numerator and denominator terms are identical except when $k = k'$. Therefore we get:

$$\begin{aligned}
(C) &= \frac{B(B_{k'} + \beta)}{B(B_{k'}^{-(i',d')} + \beta)} = \frac{\prod_{w=1}^{W} \Gamma(B_{k'w} + \beta)}{\prod_{w=1}^{W} \Gamma(B_{k'w}^{-(i',d')} + \beta)} \times \frac{\Gamma(\sum_{w=1}^{W} B_{k'w}^{-(i',d')} + W\beta)}{\Gamma(\sum_{w=1}^{W} B_{k'w} + W\beta)} \\
&= \frac{\Gamma(B_{k'w'} + \beta)}{\Gamma(B_{k'w'}^{-(i',d')} + \beta)} \times \frac{\prod_{w \neq w'} \Gamma(B_{k'w} + \beta)}{\prod_{w \neq w'} \Gamma(B_{k'w}^{-(i',d')} + \beta)} \times \frac{\Gamma(\sum_{w=1}^{W} B_{k'w}^{-(i',d')} + W\beta)}{\Gamma(\sum_{w=1}^{W} B_{k'w} + W\beta)}
\end{aligned}$$

Note that for $w \neq w'$ then $B_{k'w}^{-(i',d')} = B_{k'w}$, and so all the products over $w \neq w'$ cancel. Therefore this equation simplifies to:

$$\begin{aligned}
&= \frac{\Gamma(B_{k'w'} + \beta)}{\Gamma(B_{k'w'}^{-(i',d')} + \beta)} \times \frac{\Gamma(\sum_{w=1}^{W} B_{k'w}^{-(i',d')} + W\beta)}{\Gamma(\sum_{w=1}^{W} B_{k'w} + W\beta)} \\
&= \frac{\Gamma(B_{k'w'}^{-(i',d')} + \beta + 1)}{\Gamma(B_{k'w'}^{-(i',d')} + \beta)} \times \frac{\Gamma(\sum_{w=1}^{W} B_{k'w}^{-(i',d')} + W\beta)}{\Gamma(\sum_{w=1}^{W} B_{k'w}^{-(i',d')} + W\beta + 1)}
\end{aligned}$$

Where we have used the fact that not counting (i', d') means that $B_{k'w'}^{-(i',d')} + 1 = B_{k'w'}$. Now using the identity $\Gamma(x + 1) = x\Gamma(x)$, we get:

$$\begin{aligned}
&= \left(\frac{B_{k'w'}^{-(i',d')} + \beta}{\sum_{w=1}^{W} B_{k'w}^{-(i',d')} + W\beta} \right) \cancel{\frac{\Gamma(B_{k'w'}^{-(i',d')} + \beta)}{\Gamma(B_{k'w'}^{-(i',d')} + \beta)}} \times \cancel{\frac{\Gamma(\sum_{w=1}^{W} B_{k'w}^{-(i',d')} + W\beta)}{\Gamma(\sum_{w=1}^{W} B_{k'w}^{-(i',d')} + W\beta)}} \\
&= \frac{B_{k'w'}^{-(i',d')} + \beta}{\sum_{w=1}^{W} B_{k'w}^{-(i',d')} + W\beta}
\end{aligned} \tag{11}$$

We proceed in a similar way for term (D):

$$\begin{aligned}(D) &= \frac{B(A_{d'} + \alpha)}{B(A_{d'}^{-(i',d')} + \alpha)} = \frac{\prod_{k=1}^{K} \Gamma(A_{d'k} + \alpha)}{\prod_{k=1}^{K} \Gamma(A_{d'k}^{-(i',d')} + \alpha)} \times \frac{\Gamma(\sum_{k=1}^{K} A_{d'k}^{-(i',d')} + K\alpha)}{\Gamma(\sum_{k=1}^{K} A_{d'k} + K\alpha)} \\ &= \frac{\Gamma(A_{d'k'} + \alpha)}{\Gamma(A_{d'k'}^{-(i',d')} + \alpha)} \times \frac{\Gamma(\sum_{k=1}^{K} A_{d'k}^{-(i',d')} + K\alpha)}{\Gamma(\sum_{k=1}^{K} A_{d'k}^{-(i',d')} + K\alpha + 1)} \\ &= \frac{\Gamma(A_{d'k'}^{-(i',d')} + \alpha + 1)}{\Gamma(A_{d'k'}^{-(i',d')} + \alpha)} \times \frac{1}{\sum_{k=1}^{K} A_{d'k}^{-(i',d')} + K\alpha} \\ &= \frac{A_{d'k'}^{-(i',d')} + \alpha}{\sum_{k=1}^{K} A_{d'k}^{-(i',d')} + K\alpha} \end{aligned} \tag{12}$$

Combining (11) and (12) we get the final expression:

$$p(z_{i'd'} = k' | \boldsymbol{z}^{-(i',d')}, \boldsymbol{x}^{-(i',d')}, x_{i'd'} = w') = \frac{A_{d'k'}^{-(i',d')} + \alpha}{\sum_{k=1}^{K} A_{d'k}^{-(i',d')} + K\alpha} \cdot \frac{B_{k'w'}^{-(i',d')} + \beta}{\sum_{w=1}^{W} B_{k'w}^{-(i',d')} + W\beta} \tag{13}$$

1.5 Part (e)

- α is the hyperprior controlling the Dirichlet distribution over topics for each document. Since the α values are symmetric, we can say that increasing α makes each document more likely to contain a mixture of topics. On the other hand decreasing α would make the distribution of θ more sparse, and therefore we would expect to see less topics per document.

- β controls the Dirichlet distribution over words for each topic. For larger values of β we would expect to see topics containing a larger mixture of words, and for smaller β, fewer words corresponding to each topic.

Therefore the choice of α would depend on how many topics we expect to identify per document, and the choice of β would depend on how specific we would want the words for each topic to be. Generally, we want to encourage sparsity in the topic distributions and word distribution and therefore a sensible choice would be to set α and β to be less than 1. We could generate samples of α and β by performing Gibbs sampling on these parameters as well. Starting from the collapsed joint we would use:

$$p(\alpha = \alpha' | \boldsymbol{x}, \boldsymbol{z}, \boldsymbol{\beta}) \propto p(\boldsymbol{x}, \boldsymbol{z} | \alpha = \alpha', \beta) p(\alpha = \alpha') p(\beta)$$

Now, using appropriate priors for α and β which encourage small values, and the distribution $p(\boldsymbol{x}, \boldsymbol{z} | \alpha = \alpha', \beta)$ which we have already found in Part (c), we can generate samples for α (after normalisation). The same procedure would apply to β.

2 Decrypting Messages with MCMC

2.1 Part (a)

We wish to estimate

$$p(s_1 s_2 ... s_n) = p(s_1) \prod_{i=2}^{n} p(s_i | s_{i-1}) \tag{14}$$

and we denote $\Psi(\alpha, \beta) \coloneqq p(s_i = \alpha | s_{i-1} = \beta)$. For the ML estimates, denote the counts

$$C(\alpha, \beta) \coloneqq \#\text{times string } \beta\alpha \text{ observed}$$
$$P(\alpha) \coloneqq \#\text{times string } \alpha \text{ observed}$$

Then the ML estimates are-

$$\Psi(\alpha, \beta) = \frac{C(\alpha, \beta)}{\sum_{i=1}^{n} C(\alpha, s_i)} \tag{15}$$

$$\phi(\alpha) = \frac{P(\alpha)}{\sum_{i=1}^{n} P(s_i)} \tag{16}$$

For the text *War and Peace*, we kept a count of the transition and occurrence statistics and estimated the corresponding probabilities, using the ML formulae (15) - (16). The transition and stationary ML probability estimates are displayed in the diagrams below:

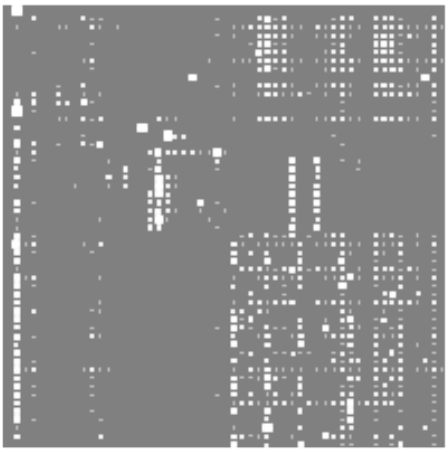


Figure 1: Hinton Diagram of Transition Probabilities

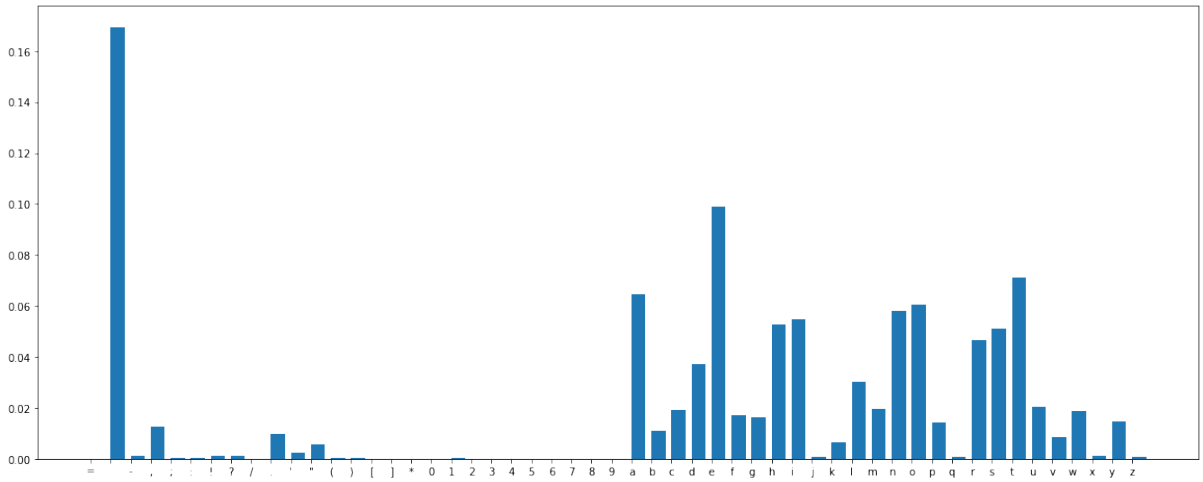


Figure 2: Bar Chart of Stationary Probabilities

2.2 Part (b)

For a ciphertext $\boldsymbol{e}$ and plaintext $\boldsymbol{s}$ we know $\exists\sigma$ such that $\sigma(\boldsymbol{s}) = \boldsymbol{e}$. Let $p(s_i|s_j) = M(s_i, s_j)$. The latent variables for different symbols are not independent because the underlying symbols themselves are not independent, and the encryption function σ is one-to-one. Note that for some encyption text $e_1...e_n$ we are interested in calculating $p(e_1...e_n|\sigma)$, which can be written down equivalently as $p(\sigma^{-1}(\boldsymbol{e}))$. The joint probability can therefore be written down by inverting the encryption:

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

$$\implies p(e_1 e_2 ... e_n|\sigma) = p(\sigma^{-1}(e_1))\prod_{i=2}^{n} M(\sigma^{-1}(e_i), \sigma^{-1}(e_{i-1})) \tag{17}$$

where we assume for notational simplicity that $\sigma(s_i) = e_i$.

2.3 Part (c)

Since the proposal is given by choosing two symbols uniformly at random then the proposal probability does not depend on the permutation and all proposal probabilities are $\frac{1}{\binom{n}{2}}$ since for any proposal we simply choose two symbols to swap. Therefore we have that $S(\sigma' \to \sigma) = S(\sigma \to \sigma')$. The MH acceptance probability for a given proposal is defined to be:

$$A(\sigma', \sigma) \coloneqq min\left(1, \frac{S(\sigma' \to \sigma)p(\sigma')}{S(\sigma \to \sigma')p(\sigma)}\right) \tag{18}$$

And given $S(\sigma' \to \sigma) = S(\sigma \to \sigma')$, we can cancel these terms:

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

$$= min\left(1, \frac{p(\boldsymbol{e}|\sigma')}{p(\boldsymbol{e}|\sigma)}\right)$$

We have already computed the final expressions in equation (17), and so can write down the MH acceptance probability as:

$$A(\sigma', \sigma) = min\left(1, \frac{p(\boldsymbol{e}|\sigma')}{p(\boldsymbol{e}|\sigma)}\right) \tag{19}$$

which makes intuitive sense since $\frac{p(\boldsymbol{e}|\sigma')}{p(\boldsymbol{e}|\sigma)}$ is just the ratio of the likelihood of the encryption given the two permutation functions.

2.4 Part (d)

The first step in the MH sampler was pre-processing the data. The transition and stationary probabilities were calculated via ML estimates, and the encypted text was loaded:

```
import numpy as np

symbols = []
with open('symbols.txt') as f:
    for line in f:
        for ch in line:
            if(ch != '\n'):
                symbols.append(ch)

transition_count = np.zeros(shape=(len(symbols), len(symbols))).astype(int
                                                    )
stationary_count = np.zeros(len(symbols)).astype(int)
prev_ch = '"'
with open('war_peace_text.txt') as f:
    for line in f:
        line = line.rstrip('\n').lower()
        for ch in line:
            if(ch in symbols):
                r = symbols.index(ch)
                stationary_count[r] += 1
                if(prev_ch in symbols):
                    c = symbols.index(prev_ch)
                    transition_count[r, c] += 1
            prev_ch = ch

M = transition_count/transition_count.sum(axis=1, keepdims=True) + 1.0e-20
P = stationary_count/np.sum(stationary_count)

symbol_length = len(symbols)
cipher_text = None
with open('message.txt') as f:
    cipher_text = [line.strip() for line in f]
cipher_text = cipher_text[0]
```

We then wrote a class MCMC to decrypt ciphertext using the MH algorithm:

```
import copy

class MCMC(object):

    def __init__(self, M, P, symbol_length, cipher_text, symbols):
        """
        --M is the transition probability matrix
        --P is the invariant stationary matrix
        --symbol_length is the number of symbols in the alphabet
        --cipher_text is the encypted text we wish to decrypt
        --f is the current inverse sigma (permutation) function
        --cipher_ints is the equivalent representation of the symbols
          in terms of integers {0,1,..,54}
        """
        self.M = M
        self.P = P
        self.symbol_length = symbol_length
        self.cipher_text = cipher_text
        self.f = np.random.choice(symbol_length, symbol_length, replace=
                                                    False).astype(int)
        self.cipher_ints = np.zeros(len(cipher_text)).astype(int)
        self.symbol_list = symbols

    def m_h_step(self):
        """
        Performs the Metropolis-Hastings update step:
        1. A proposal function test_f is generated
        2. The scores p(e|f) for the proposal (f_test) and current
           (self.f) permutations are computed.
        3. The MH acceptance probability is computed.
        4. The current permutation function is copied through or
           replaced by test_f for the next step via rejection sampling.
        """
        index_1 = random.randint(0, self.symbol_length-1)
        index_2 = random.randint(0, self.symbol_length-1)
        test_f = self.sample_f(index_1, index_2)

        score_1, pred_1 = self.score(self.f)
        score_2, pred_2 = self.score(test_f)
        A = np.minimum(1, np.exp(score_2 - score_1))

        if(np.random.uniform() <= A):

            self.f[index_1] = np.take(test_f, index_1)
            self.f[index_2] = np.take(test_f, index_2)

        pass

    def sample_f(self, index_1, index_2):
        """
        Function to randomly sample a proposal permutation
        by swapping the two symbols (represented by integers)
        at index_1 and index_2 in the current permutation function.
```

```
        --test_f is the proposal
        """

        test_f = np.zeros(self.symbol_length).astype(int)

        for i in range(self.symbol_length):
            if(i == index_1):
                test_f[i] = np.take(self.f, index_2)
            elif(i == index_2):
                test_f[i] = np.take(self.f, index_1)
            else:
                test_f[i] = np.take(self.f, i)

        return test_f

    def initialise_f(self):
        """
        Function to intitialse f 'cleverly'. After randomly
        intialising f, certain values of f(e) = s are instantiated.
        For example, in our encyption, we want 'p' in the encyption
        to map to '<space>' in the decrpytion.
        """

        ### p symbol 42 stand for <space> 1
        self.swap_char_map('p', ' ')

        pass

    def swap_char_map(self, ch, ch_new):
        """
        Forces the permutation function f to map the encypted
        symbol 'ch' to the decrpyted symbol 'ch_new'.
        """
        val_char_to_swap = self.symbol_list.index(ch)
        val_char_to_place = self.symbol_list.index(ch_new)

        value_in_old_f = np.take(self.f, val_char_to_swap)
        index_to_swap = np.where(self.f == val_char_to_place)

        self.f[val_char_to_swap] = val_char_to_place
        self.f[index_to_swap] = value_in_old_f

        pass

    def generate_string_representation(self):
        """
        Generates a string representation of the decryption
        of the string.
        """
        str_rep = ""
        for i in self.cipher_ints:
            index_of_symbol = self.f[i]
            string_of_symbol = self.symbol_list[index_of_symbol]
            str_rep += string_of_symbol
```

```
        return str_rep

    def score(self, f):
        """
        Score function to compute the value of p(e|f) using
        the transition matrix M, stationary probabilities P
        and the formulae derived in the question.
        """
        j = 0
        prediction = np.zeros(len(cipher_text)).astype(int)
        for i in self.cipher_ints:
            prediction[j] = np.take(f, i)
            j += 1

        total_score = np.log(self.P[prediction[0]])
        for i in range(1, np.size(prediction)):
            total_score += np.log(self.M[prediction[i]][prediction[i-1]])

        return total_score, prediction

    def create_cipher_numbering(self):
        """
        Creates an integer representation of the encypted
        text by using substituting each symbol with the map:
        {'=', ' ', ...,'z'} -> {0, 1, ..., 54}
        """
        i = 0
        for ch in self.cipher_text:
            self.cipher_ints[i] = self.symbol_list.index(ch)
            i += 1

        pass
```

The MH algorithm was then performed for ten runs, and the decrypted text stored every 100th iteration:

```
final = [[]]*10
for t in range(10):
    metroMCMC = MCMC(M, P, symbol_length, cipher_text, symbols)
    metroMCMC.create_cipher_numbering()
    metroMCMC.initialise_f()
    for i in range(4000):
        c = metroMCMC.m_h_step()
        if(i%100==0):
            x = metroMCMC.generate_string_representation()[0:62]
            final[t].append(x)
            print(x)
```

The code essentially works by having the encyption $\rightarrow$ decryption permutation function f as a one-to-one mapping between integer representations of the symbols, and means we can avoid working with the characters themselves.

2.4.1 Description of Code

1. For each run an MCMC object is created. This class contains all the methods required to perform the Metropolis-Hastings Algorithm on the encypted text. The attribute $self.f$ is the permutation function from encypted to decrypted symbols.

2. In order to avoid working with the symbol characters, the method create_cipher_numbering was used to convert each symbol to an integer, where we have ('=', ' ', ..., 'z') $\rightarrow$ (0, 1, ..., 54). The permutation function $self.f$ can then be represented as a one-to-one mapping between integers in the range [0,54]. Therefore $self.f$ was represented as a list containing some permutation of the numbers in that range, where the index of the list corresponds to the integer value of the encrypted symbol, and the integer at that index corresponds to the mapped value. An example f is given below:

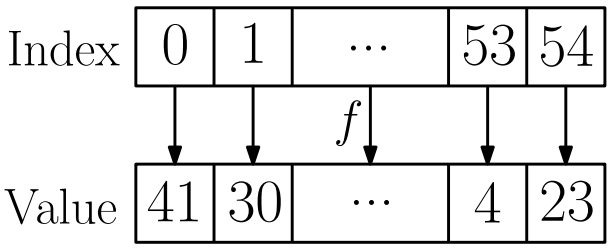


Figure 3: Example permutation function

3. The list $self.f$ was initialised at random for each run, and 'clever' instantiation choices were enforced via the function initialise_f. One such choice was to (intially) force the character $'p'$ in the cipher text to be mapped to $' < space >'$. This was because the $'p'$ character in the cipher was by far the most frequent, and from Figure 2 we see that $' < space >'$ is the most common symbol in the English corpus. Therefore it was a sensible choice to initialise the chain with this value.

4. At each iteration, the algorithm then calls m_h_step which is the function for performing the MH updates. Firstly a proposal f_test is generated, via the function sample_f. This works by swapping the values at two indices in $self.f$ at random. Secondly the scores $p(e|f)$, and $p(e|f\_test)$ were calculated via the function score. The mappings f_test and $self.f$ were applied to the integer representation of the cipher text - $cipher\_ints$, to produce an integer representation of the decrpyted string $s_1 s_2 .. s_n$. The scores $p(e|f)$ and $p(e|f\_test)$ could then be computed using the ML estimated transition matrix M, the stationary probabilities P, and the formula in equation (17).

5. Finally, the proposal permutation f_test could then be accepted with probability according to equation (20). This process was re-iterated 4000 times. (Some stopping criteria relating to testing the 'convergence' of $self.f$ - i.e. if $self.f$ remained unchanged for a certain number of iterations - could also have been implemented).

This process was performed for 10 runs, and the decrypted integers restored into character representation via the function generate_string_representation every 100 iterations per run.

2.4.2 Results

The outputs of one such run that produced an accurate decryption is given below:

Decrypted String
5* qy y](*8e' ;*k q]'e 9(s*e';7se ye;'d qy =;t[e' 8;9e qe d]qe
vn *g g]unse' fnk *]'e luine'f4le gef'd *g aft.e' sfle *e d]*e
ma g; ;suayer fak gsre lulaerf4le ;efrd g; nft.er yfle ge dsge
yu g; ;sfumer auk gsre lfluera'le ;eard g; natver male ge dsge
ou w; ;shumer auk wsre bhluera-le ;eard w; natcer mabe we dswe
ou w. .lhumer auk wire bhluera-le .ears w. natcer mabe we slwe
ou w. .lhumer auk wire bhluera-le .ears w. natcer mabe we slwe
ou w. .lkumer aug wire dkluera-le .ears w. natcer made we slwe
ou w. .lkumer aug wire dklueravle .ears w. natcer made we slwe
ou w. .lkumer aun wire dklueragle .ears w. vather made we slwe
ou w. .lkumer aun wire dklueragle .ears w. vather made we slwe
ou w. .lfuder aun wire kflueramle .ears w. vather dake we slwe
ou w. .lfuder aun wire kflueramle .ears w. vather dake we slwe
ou w, ,lfuder aun wire kflueramle ,ears w, vather dake we slwe
ou w, ,lfuder aun wire kflueramle ,ears w, vather dake we slwe
ou w, ,lfuger aun wire kflueramle ,ears w, vather gake we slwe
ou w, ,lfuger aun wire kflueramle ,ears w, vather gake we slwe
ou w, ,lfuger aun wire vfluerable ,ears w, kather gave we slwe
ou w, ,lfuger aun wire vflueradle ,ears w, kather gave we slwe
ou w, ,lfuger aun wire vflueradle ,ears w, kather gave we slwe
ou w, ,lfuger aud wire vflueranle ,ears w, kather gave we slwe
on w, ,lfnger and wire vflneraule ,ears w, kather gave we slwe
on w, ,lfnger and wire vflnerayle ,ears w, kather gave we slwe
on w, ,lfnger and wire vflneramle ,ears w, kather gave we slwe
on w, ,lfnger and wire vflneramle ,ears w, kather gave we slwe
on w, ,lfnger and wire vflneramle ,ears w, kather gave we slwe
on w, ,lmnger and wire vminerafle ,ears w, kather gave we slwe
on w, ,lmnger and wire vminerafle ,ears w, kather gave we slwe
on w, ,lmnger and wire vminerafle ,ears w, kather gave we slwe
on w, ,lknger and wire vklnerable ,ears w, mather gave we slwe
on w, ,lknger and wire vklnerable ,ears w, mather gave we slwe
on w, ,lunger and wire vulnerable ,ears w, mather gave we slwe
on w, ,lunger and wire vulnerable ,ears w, mather gave we slwe
on w, ,lunger and wire vulnerable ,ears w, mather gave we slwe
on w, ,lunger and wire vulnerable ,ears w, mather gave we slwe
on w, ,lunger and wire vulnerable ,ears w, mather gave we slwe
on w, ,lunger and wire vulnerable ,ears w, mather gave we slwe
in w, ,ounger and wore vulnerable ,ears w, mather gave we sowe
in w, ,ounger and wore vulnerable ,ears w, mather gave we sowe
in m, ,ounger and more vulnerable ,ears m, wather gave me some
in m, ,ounger and more vulnerable ,ears m, wather gave me some
in my younger and more vulnerable years my wather gave me some
in my younger and more vulnerable years my pather gave me some
in my younger and more vulnerable years my father gave me some

2.5 Part (e)

If we have $\Psi(\alpha, \beta) = 0$ then for some proposals σ' which decrypt the string such that $\sigma'^{-1}(e_i) = \alpha, \sigma'^{-1}(e_{i-1}) = \beta$, then $M(\sigma'^{-1}(e_i), \sigma'^{-1}(e_{i-1})) = \Psi(\alpha, \beta) = 0$, and hence from formula (17) we will have $p(e|\sigma') = 0$. As a result the MH acceptance probability will also be zero for this proposal and hence this state σ' will be impossible to reach and therefore the ergodicity is broken. We could restore ergodicity by simply adding a single pseudo-count to those transitions which we do not observe when scraping 'War and Peace' for the ML

estimates. Upon normalisation to produce the transition probabilities, these single counts will produce tiny transition probabilities and therefore will not affect the accuracy of the decrypter.

2.6 Part (f)

1. Symbol probabilities alone would not be sufficient using an MCMC sampler because we would have to assume the state at any one point is independent of the previous state, and therefore could not exploit the conditional dependence structure of the chain. We could still evaluate the conditionals as just the product of the independent frequency probabilities, i.e. $\Psi(\alpha, \beta) = \phi(\alpha) \cdot \phi(\beta)$ (where ϕ is the stationary probability), but then using this algorithm would just assign the most common encrypted symbol to the decrypted symbol with the highest stationary/frequency probability and so on. This assignment would maximise the log likelihood however it would clearly yield an incorrect decryption. Therefore this approach would not work.

2. A second order Markov chain would introduce the following problems. Firstly, many of the transition probabilities would be zero, since most sequences of three letters would be highly unlikely to see in the English language. This would potentially break the ergodicity of the chain. Rectifying this via the method used before would introduce enough inaccuracy in the real transition probabilities so as to make the MH algorithm useless. Additionally most of the probabilities will become much lower, making the algorithm much more likely to reach many local 'maxima'. As a result, decrypting algorithm may need an extreme amount of runs to be successful.

3. The algorithm will not work if two symbols can be mapped to the same encypted value. This is because in formula (17) we rely on a unique inverse $\sigma^{-1}(e_i)$ of each encrypted algorithm in order to calculate the scores for the MH rejection probability. We may try to run the algorithm anyway and produce 'some' decryption. However in the absolute *best* case scenario, the decryption may be correct apart from the symbols which are mapped to the same value.

4. This approach would not be feasible for the Chinese language. Even for 56 symbols, the algorithm can fail to find the correct decryption. Expanding to 10,000 symbols would make the possible number of substitutions $x \to x'$ much higher, and so although theoretically possible, the algorithm in practice would be exceedingly unlikely to find the correct transitions.

3 Implementing Gibbs Sampling for LDA

3.1

The python code for the sampling updates for the non-collapsed LDA algorithm is given below:

```
    def update_params(self):
        """
```

```
        Samples theta and phi, then computes the distribution of
        z_id and samples counts A_dk, B_kw from it
        """
        # todo: sample theta and phi
        for k in range(self.n_topics):
            self.phi[k] = self.rand_gen.dirichlet(self.beta + self.B_kw[k]
                                                  )

        for d in range(self.n_docs):
            self.theta[d] = self.rand_gen.dirichlet(self.alpha + self.A_dk
                                                    [d])

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

These updates simply follow equations (7) and (8)

```
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)

        for d in range(self.n_docs):
            for w in range(self.n_words):
                 sample = self.rand_gen.choice(self.topics_space,
                 size=self.docs_words[d,w],
                 p = self.topic_doc_words_distr[:,d,w])
                 for k in sample:
                     self.A_dk[d, k] += 1
                     self.B_kw[k, w] += 1
```

In order to update the counts, we loop through every (doc, word) pair. For each pair we generate self.doc_words[d,w] samples, over K (self.topics_space) topics using the current estimate for the distribution (self.topic_words_distr[:,d,w]). For each topic in this sample, we update the corresponding A_dk and B_kw count.

```
def update_loglike(self, iteration):
        """
        Updates loglike of the data, omitting the constant additive term
        with Gamma functions of hyperparameters
        """
        loglike = 0

        for k in range(self.n_topics):
            for w in range(self.n_words):
                loglike += (self.B_kw[k, w] + self.beta - 1)*np.log(self.
                                                    phi[k, w])
            for d in range(self.n_docs):
                loglike += (self.A_dk[d, k] + self.alpha - 1)*np.log(self.
                                                    theta[d, k])
        self.loglike[iteration] = loglike
```

```
        test_likelihood = 0
        for d in range(self.n_docs):
            for w in range(self.n_words):
                test_likelihood += self.docs_words_test[d,w]* \
                np.log(np.sum(np.dot(self.phi[:,w], self.theta[d,:])))
        self.loglike_test[iteration] = test_likelihood
```

Plots for the log likelihood for the training test data are given below:

Figure 4: Log Likelihood for Training Data

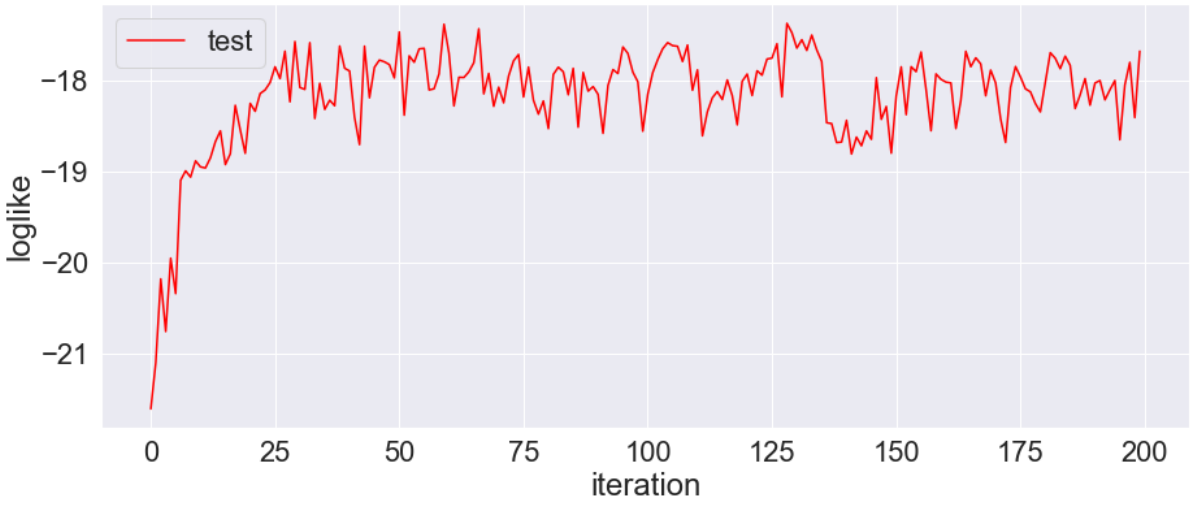


Figure 5: Log Likelihood for Test Data

Similarly the sampling updates for the collapsed Gibbs for LDA and the log likelihood updates are given below:

```
def update_params(self):
    """
    Computes the distribution of z_id.
    Sampling of A_dk, B_kw is done automatically as
    each new z_id updates these counters
    """
    for d in range(self.n_docs):
        for w in range(self.n_words):
            topic = self.doc_word_samples[(d,w)]
            i = 0
            for k in topic:
                p_z = np.zeros(len(self.topics_space))
                for k_prime in range(len(self.topics_space)):

                    if(k == k_prime):
                        self.A_dk[d, k_prime] -= 1
                        self.B_kw[k_prime, w] -= 1

                    dom_1 = np.sum(self.A_dk[d,:]) + self.alpha*self.
                                                        n_topics
                    dom_2 = np.sum(self.B_kw[k_prime,:]) + self.beta*self.
                                                        n_words
                    num_1 = (self.A_dk[d][k_prime] + self.alpha)
                    num_2 = (self.B_kw[k_prime][w] + self.beta)
                    p_z[k_prime] = num_1*num_2/(dom_1*dom_2)

                    if(k == k_prime):
                        self.A_dk[d, k_prime] += 1
                        self.B_kw[k_prime, w] += 1


                p_z /= np.sum(p_z)
                sample = self.rand_gen.choice(self.topics_space, size=1,
                                              p = p_z)
                self.A_dk[d, sample] += 1
                self.B_kw[sample, w] += 1
                self.A_dk[d, k] -= 1
                self.B_kw[k, w] -= 1
                self.doc_word_samples[d, w][i] = sample
                i += 1
    pass
```

1. Some explanation of how this code works is required. Essentially the algorithm loops through every document d' and every word w'. For each word we receive a list of the topic assignments of this word in a particular document, i.e. topic assignments k' for each token (i', d') corresponding to the same word.

2. For k in the topic space:

   (a) If the topic assignment of the word $k' == k$, we update the counts A_dk and B_kw by subtracting 1 at the index where $d = d', k = k', w = w'$.

(b) For each (i', d'), we compute the probabilities $p(z_{i'd'} = k|\boldsymbol{z}^{-(i',d')}, \boldsymbol{x}^{-(i',d')}, x_{i'd'} = w')$ using the formulae derived in Question 1.

(c) If we subtracted one from the count previously, we restore the counts by adding one back.

3. After normalising, we now have a distribution for $z_{(i',d')}$ (called p_z in the code), and can therefore sample a new topic for this token. We therefore update the count for $A\_[d'][sample]$ and $B\_kw[sample, w']$ by adding one, and subtract one from the count of the original topic assignment for that token ($A\_dk[d', k']$, $B\_kw[k', w']$).

4. Finally, we update the topic assignment for that token in doc_word_samples.

The log likelihood updates for train and test sets are given below:

```
def update_loglike(self, iteration):
    """
    Updates loglike of the data, omitting the constant additive term
    with Gamma functions of hyperparameters
    """
    # todo: implement log-like

    loglike = 0
    for k in range(self.n_topics):
        for w in range(self.n_words):
            loglike += gammaln(self.B_kw[k][w] + self.beta)
        loglike -= gammaln(np.sum(self.B_kw[k]) + self.beta*self.n_words)
    for d in range(self.n_words):
        for k in range(self.n_topics):
            loglike += gammaln(self.A_dk[d][k] + self.alpha)
        loglike -= gammaln(np.sum(self.A_dk[d]) + self.alpha*self.n_topics
                                                )
    self.loglike[iteration] = loglike


    theta = self.alpha + self.A_dk
    ss = np.sum(theta, axis = 1)[:,None]
    theta = theta/ss

    phi = self.beta + self.B_kw
    ss = np.sum(phi, axis = 1)[:,None]
    phi = phi/ss


    test_likelihood = 0
    for d in range(self.n_docs):
        for w in range(self.n_words):
            test_likelihood += self.docs_words_test[d,w]* \
            np.log(np.sum(np.dot(phi[:,w], theta[d,:])))
    self.loglike_test[iteration] = test_likelihood
    pass
```

The log likelihood for the collapsed sampling for the training data is given by formula (9) in Question 1. We computed the log likelihood for the training data by computing ML

estimates for *theta* and *phi*, and then applying the same method for the standard updates. The resulting plots are given below:

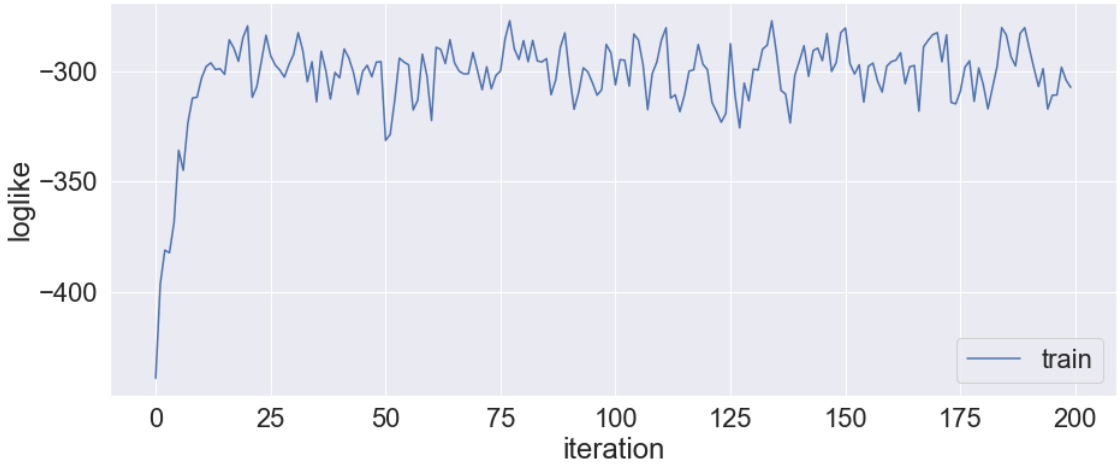


Figure 6: Log Likelihood for Training Data (Collapsed)

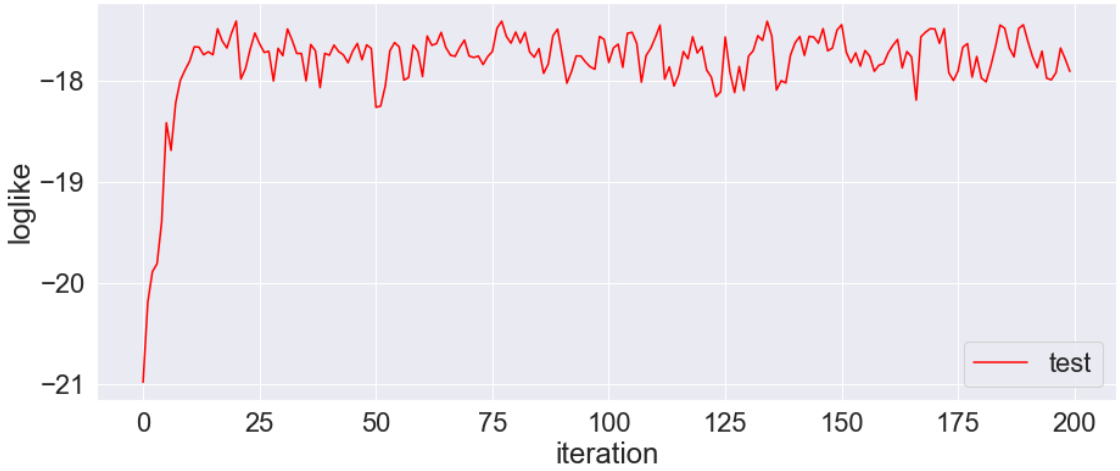


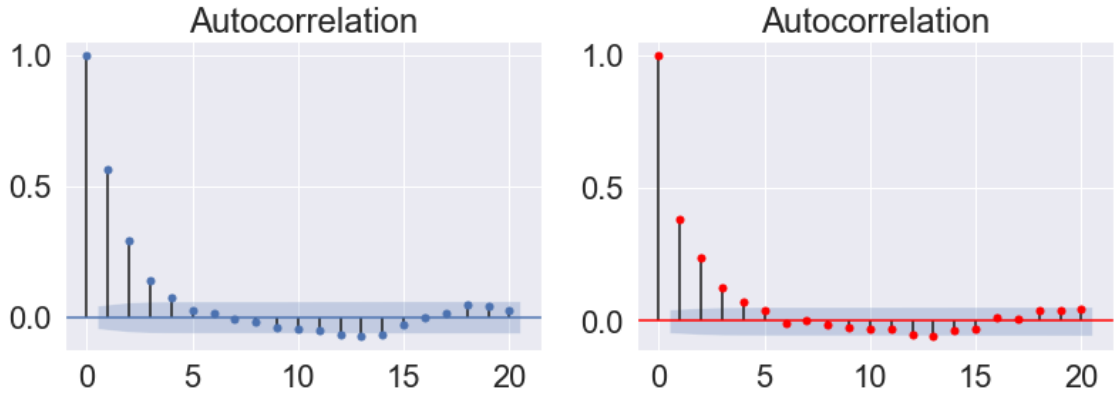
Figure 7: Log Likelihood for Test Data (Collapsed)

3.2 Part (b)

Based on the plots of the log predicted and log joint for the non-collapsed sampler, we believe that 25 burn-in iterations are sufficient. After 25 iterations, we see that the log-likelihood in all cases stops increasing and instead follows a stochastic oscillation for the remaining iterations. Since the log joint and predicted act as a proxy for assessing the likelihood of the samples, after 25 iterations we can say that each sample becomes somewhat independent of the previous one. Therefore we can conclude that sufficient mixing has taken place, and the chain has reached an invariant distribution. However, the burn in rate of the collapsed

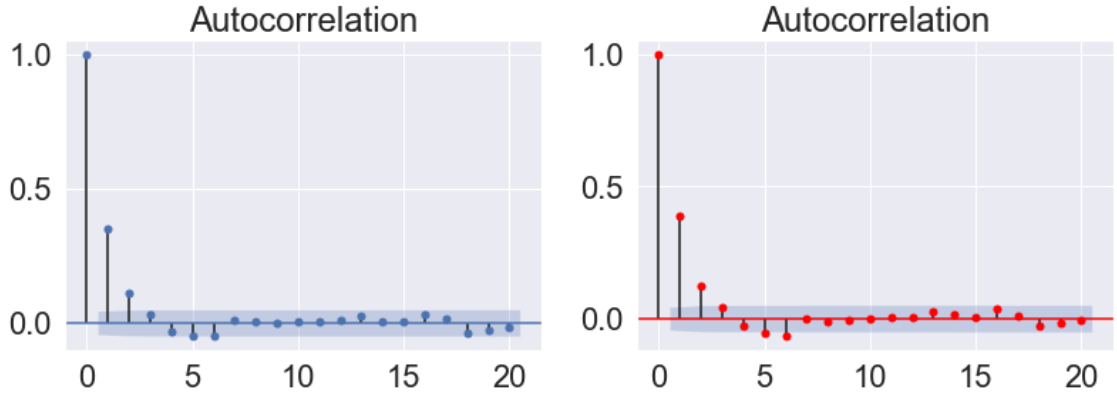
sampler seems to be faster, and we estimate around 18 is the minimum number of iterations required for burn in based on similar analysis.

The auto-correlation plots for the Log joint and Log predicted values under the non-collapsed Gibbs sampler are given below. 30 burn-in iterations were disregarded before computation (just to be safe), and a total of 2000 iterations for the sampler were performed.



(a) Log joint (b) Log Predicted

The auto-correlation plots for the Log joint and Log predicted values under the collapsed Gibbs sampler are given below. 30 burn-in iterations were disregarded before computation, and a total of 2000 iterations for the sampler were performed.



(a) Log joint (b) Log Predicted

In both plots, we used the library `statsmodels.graphics.tsaplots import plot_acf`, which also provides a 95% confidence interval for the auto-correlation for different lags (shaded blue area). For the non-collapsed Gibbs, we see that the first Lag for both the joint and the predicted with an autocorrelation within the 95% confidence interval (of being zero) is $L = 5$. For the collapsed Gibbs, we see that the first lag within the 95% confidence interval is $L = 3$.

In order to obtain a respresentative set of samples from the posterior we need enough i.i.d samples so that the posterior can be adequately approximated. Therefore the number of

samples we need will be the result of discarding the burn in samples, the lag, and the number of i.i.d samples necessary to approximate the posterior. Hence a suitable formula would be:

$$B + L \times K \tag{20}$$

where B is the number of burn-in samples, L is the lag, and K is the number of i.i.d samples required to accurately estimate the posterior. In the case where the posterior can be approximated by a Gaussian, for instance, a minimum number for K is ~ 30. Hence for the non-collapsed Gibbs the minimum number of samples required is:

$$25 + 5 \times 30 = 175$$

And also for the collapsed sampler, we need at least:

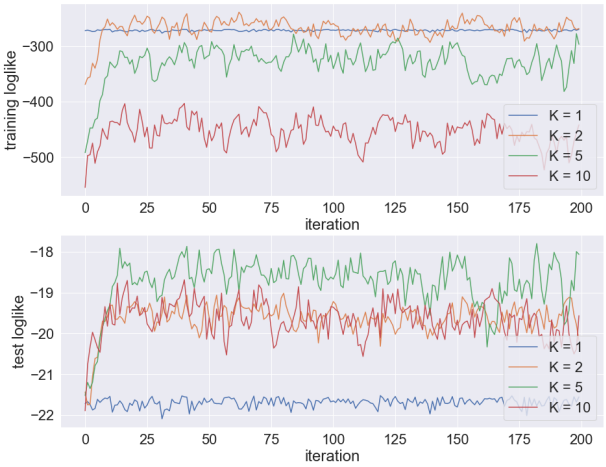
$$18 + 3 \times 30 = 108$$

3.3 Part (c)

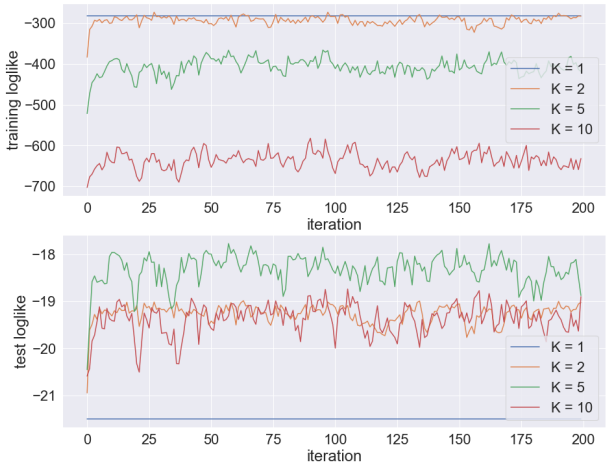
Based on the analysis in the previous question, we see that the minimum number of samples needed for the collapsed sampler (108) is lower than for the non-collapsed sampler (175), and therefore we conclude that the collapsed sampler converges quicker. However, regardless of the N value we posit, we see that the burn-in and lag of the collapsed sampler is lower, and so will require fewer samples and will converge quicker. The collapsed sampler converges faster because we are reducing the number of parameters we have to sample by integrating them out. Upon drawing 'bad' samples of θ and ϕ, it will take a longer number of iterations to break out of this 'bad' loop which produces poor z_{id} assignments (and thus a lower log likelihood), whereas within the collapsed version we weight the settings of θ and ϕ by integrating them out. This can also be seen in the Log likelihood plots for the test data, where the log likelihoods for the non-collapsed sampler varies much more widely than for the collapsed version.

3.4 Part (d)

In the following section we vary one of the parameters, while keeping the others the same. We produce the plots for the log likelihood of the training and test data for both the standard and collapsed versions:

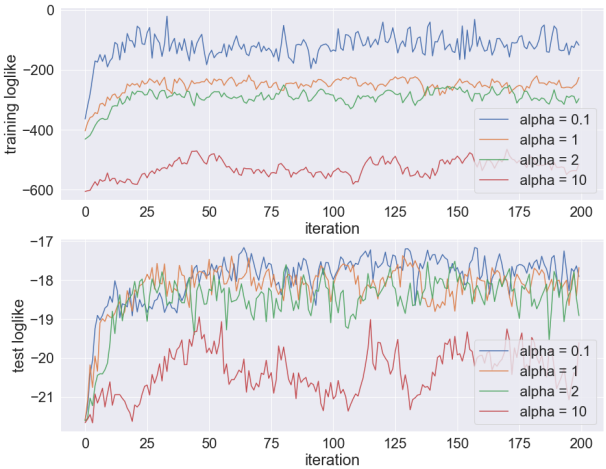


(a) Varying K for standard

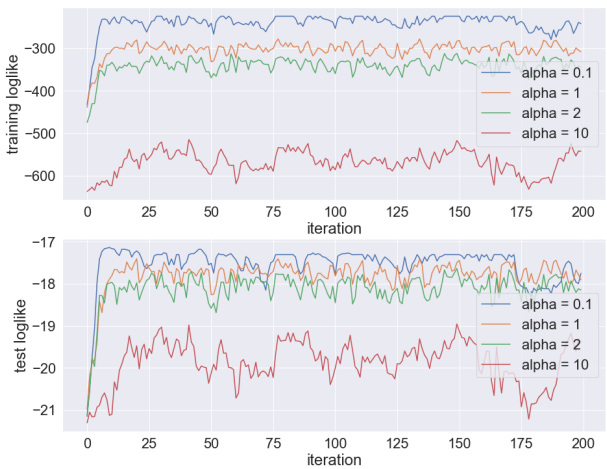


(b) Varying K for collapsed

For K=1 we see that the log likelihood for the training data remains constant. The algorithm is forced to pick one topic for all documents and therefore there cannot be any variation in the predictive power. Generally we see that K = 5 produces the greatest predictive power, since there is some leeway in the possible number of topics to assign. For K = 10, it seems that the log-likelihood decreases again, perhaps because too many topics are assigned to each document.

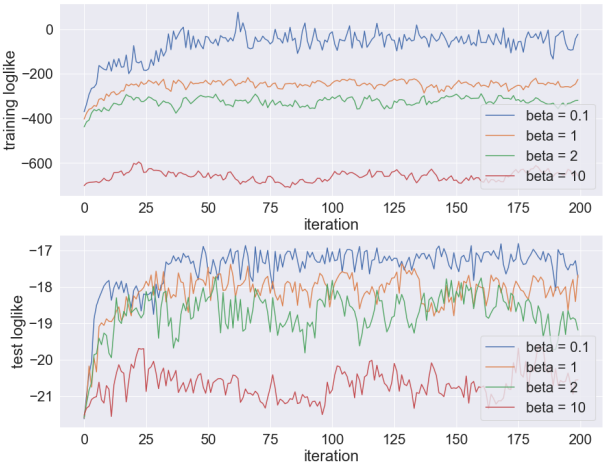


(a) Varying alpha for standard

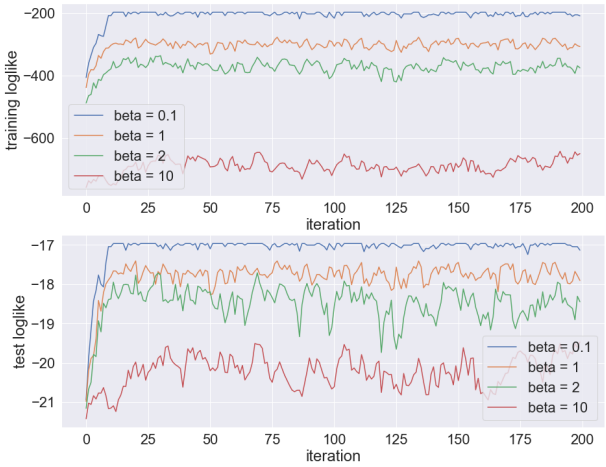


(b) Varying alpha for collapsed

The value of alpha dictates the sparsity of topic assignments within each document. For smaller alpha (0.1) the posterior performance is best because having a smaller selection of topics within each document is more realistic - generally we would expect to see each document talking about only a few topics.



(a) Varying beta for standard



(b) Varying beta for collapsed

Similar analysis is true for the value of beta. The smallest value of Beta tested - 0.1 - seems to produce the greatest predictive power of the model. Beta dictates the sparsity of words assigned to topics, and therefore for smaller values of beta (0.1) fewer words are associated to each topic. Again this is an assignment that corresponds to a realistic scenario, where we expect to see topic characterised by only a certain few key words.