FastLDA and SparseLDA

Luca Geminiani

January 2024

PhD Statistics and Computer Science, Bocconi
CS I Final Project

1 Introduction

This project aims to provide Julia codes for two CPU-enhancing algorithms for Latent Dirichlet Allocation: FastLDA and SparseLDA. My implementations are entirely based on the papers of Newman et al.(2008) for FastLDA, and Yao et al.(2009) for SparseLDA, both of which are cited in the README. Code is entirely of my making. I have only used two tarballs, written in C++ and C by the respective authors, as points of reference for the algorithms' structure and prior updating choices.

The purpose of this document is twofold. First, provide theoretical explanations for the functioning of the algorithms and my major implementation choices. Second, as this is a Computer Science project, discuss key problems of computational efficiency, my proposed solutions to them, and the limitations of my code.

In section 2 I explain the notation used in the code and this document for mathematical formulas.

In section 3 I briefly discuss my choices for data extraction and the utilized datasets.

In section 4 I provide some theoretical background for CPU-enhancing LDA methods.

In sections 5-6-7, the most important, I elaborate on the Gibbs sampling step of $FastLDA(2, 2, +\infty)$, $FastLDA(3, 3, 3)$, and $SparseLDA$, focusing on efficiency considerations.

In sections 8-9, I discuss Dirichlet Hyperparameters updating and the computation of Perplexity.

In section 10 I discuss results in terms of MCMC convergence of document-topic distributions, providing graphical representations(Trace Plots).

In section 11 I make final considerations and point out the major limitations of my code.

2 Notation and Fields

The notation I used in Julia is consistent with all the mathematical formulas present in this file. Here I describe the most important variables. Other key local variables, specific to each algorithm, are presented throughout the document.

```
T::Int64 #Number of topics (Choice of the user)
W::Int64 #Vocabulary size (number of unique words)
D::Int64 #Number of documents
Nt::Vector{Int64} #Number of words associated to each topic
Ntw::Matrix{Int64} #Number of topic assignments to each word
Ndt::Matrix{Int64} #Number of topic assignments to each document
Nd::Vector{Int64} #Total number of words in each document
Ntw_avg::Matrix{Float64} #Average of topic-word assignments (only after burn-in)
Ndt_avg::Matrix{Float64} #Average of topic-document assignments (only after burn-in)
z::Vector{Vector{Vector{Int64}}} #Topics assignments to each word, for each repetition, in each document
alpha::Vector{Float64} #Dirichlet prior parameters for the document-topic distribution
beta::Float64 #Dirichlet prior parameter for the word-topic distribution
I::Int64 #Current Iteration (used only after burn-in)
PX::Float64 #Perplexity
pdw::Vector{Vector{Float64}} #Probabilities of each word appearing in each document
Trace::Array{Int64, 3} #Entire Ndt distributions at each iteration, for Trace Plots

PTM(T, W) = new(T, W) #Mutable struct
```

The mutable struct PTM is used to store the variables above, as well as other algorithm-specific ones. Its role is crucial as it allows for their efficient updating and storage.

3 Data Extraction

The starting point of the project was to build a general framework to extract information from textual files of type ".csv". My choice is due to the popularity of .csv for storing textual data, as well as my general familiarity with such a format.

The dataset I used in all applications, $AbstractsMPCSST$ on GitHub, comes from kaggle.com, and contains a sample of 500 abstracts of scientific articles, selected by filtering on the topics of "Mathematics", "Computer Science", "Physics" and "Statistics". This induces an immediate choice for the number of topics T, which is set to 4 in $RunTests.jl$.
The choice of the number of topics is always left to the implementer, as I didn't include tools to estimate an optimal T(as in HierarchicalLDA).

Referencing the code in $RunTests.jl$, I explain how the main corpus of data is generated.
The first section, having removed all confounding characters, finds the abstracts row by row and stores them in the vector $cleaned\_texts$. It then collects all the unique words across all abstracts, generating the variable $vocabulary$. Finally, it creates two dictionaries of strings and integers, assigning to each word of the (unique) vocabulary an integer and vice-versa. It is possible to check which word was assigned to each integer using the variable $check\_word[number]$, and, vice-versa, $check\_id[word]$.

```
using CSV
using DataFrames

file_path = "AbstractsMPCSST.csv"
frame = CSV.File(file_path) |> DataFrame
abstracts = frame[:, "ABSTRACTS"]

chars_cut = r"[\\,\.!;:'\"()\[\]{}<>?/\|_+*=^%&$#~‘]"
cleaned_texts = [replace.(split(abstract), chars_cut => "") for abstract in abstracts]

unique_words = Set(vcat(cleaned_texts...))
vocabulary = collect(unique_words)

check_id = Dict(word => id for (id, word) in enumerate(vocabulary))
check_word = Dict(id => word for (id, word) in enumerate(vocabulary))
```

A second section creates the corpus used to perform Latent Dirichlet Allocation by all the algorithms. The variable "corpus" contains all the required information in the following format: for each document of text considered, and for each unique word appearing in such document, it assigns a vector of tuples of the form $(word\_id, counts)$.

Since the computation of Perplexity must be performed on a testing sample, while Gibbs Sampling on a training set, I arbitrarily split the main corpus into two parts, respectively of 15% and 85% of the original document's length.

```
corpus = []
for text in cleaned_texts
    words_id = map(w -> word_to_id[w], text)
    word_counts = Dict{Int, Int}()
    for word_id in words_id
        word_counts[word_id] = get(word_counts, word_id, 0) + 1
    end

    counts_tuple = [(word_id, counts) for (word_id, counts) in word_counts]
    push!(corpus, counts_tuple)
end

ratio = 0.85
corpus_train = []
corpus_test = []
for counts_tuple in corpus
    Nwd, = size(counts_tuple)
    length = Int(round(Nwd * ratio))
    push!(corpus_train, counts_tuple[1:length])
    push!(corpus_test, counts_tuple[length + 1:end])
end

T = 4
voc_size = length(vocabulary)
W = voc_size
```

The methods used to generate the corpus of data are easily extendable to other data formats, as long as the variable "abstracts" is correctly initialized from the data (just a vector containing all abstracts in string format).

4 Theoretical Introduction to SparseLDA and FastLDA

The goal of $FastLDA$ and $SparseLDA$ is to improve on implementations of LDA based on standard Collapsed Gibbs Sampling. Both algorithms exploit the sparseness of topic allocations over documents and words to reduce the computational costs of topic-assignment probabilities.
The methods employed allow, for each iteration over the corpus, to avoid computing all probabilities for all topics, leading to more efficient implementations, with almost no qualitative losses in results.

The underlying model for applying both algorithms is a probabilistic topic model, where each word in the considered sample of documents is represented as a mixture over T topic components, each of which is a Multinomial distribution over a vocabulary of unique words W. The probability of a word given topic t is given by $\phi_{w|t}$.
Each document d, in my applications each scientific abstract, is assigned a multinomial distribution over topics $\theta_{t|d}$.

The multinomial parameters $\theta_{t|d}$ and $\phi_{w|t}$ are drawn from Dirichlet Priors α_t and β respectively. The underlying Dirichlet distributions are:

$$P(\theta_{t|d}) = \frac{\Gamma(\sum_t \alpha_t)}{\prod_t \Gamma(\alpha_t)} \prod_t \theta_t^{\alpha_t - 1} \tag{1}$$

$$P(\phi_{w|t}) = \frac{\Gamma(W\beta)}{\prod_w \Gamma(\beta)} \prod_w \phi_w^{\beta - 1} \tag{2}$$

While z_d is distributed as $Multinomial(\theta_d)$ and w_d is distributed as $Multinomial(\phi_w)$.
We can obtain estimates for $P(w|t)$ and $P(t|d)$ through the posterior predictive distributions of the Dirichlet-Multinomial model, exploiting conjugacy.

$$\hat{\phi}_{w|t} = \frac{n_{w|t} + \beta}{n_{|t} + \beta W} \tag{3}$$

$$\hat{\theta}_{t|d} = \frac{\alpha_t + n_{t|d}}{\sum_{t'} \alpha_{t'} + n_{|d}} \tag{4}$$

From the above distributions, through various mathematical passages and the integration of multinomial factors out, we can obtain the following expression for the probability of topic assignment z, in document d, for a word w.

$$P(z = t|w) \propto (\alpha_t + n_{t|d}) \cdot \frac{n_{w|t} + \beta}{n_t + \beta W} \tag{5}$$

If we wanted to sample from the distribution in equation (5), we would have to compute its unnormalized weight, which can be labeled as $q(z = t)$ for each topic.

We would need to sample from $U(0, \sum_t q(t))$, and find the topic t for which $\sum_{t=1}^{t-1} q(t) < U < \sum_{t=1}^{t} q(t))$. Such an approach requires calculating $q(t)$ for each t to find the normalizing constant Z of $\sum_t q(t)$.
However, the probability mass is generally concentrated over very few topics, thus making some of these calculations redundant. This is the fundamental element of sparsity that both $FastLDA$ and $SparseLDA$ exploit to avoid computing all the $q(t)$ for every word.

5 Fast LDA (2,2,+∞)

FastLDA, rather than computing $\sum_{t=1}^{t} q(t)$ for each word, iteratively defines a sequence of refining approximations to it, by generating sequences of improving bounds on the normalizing constant $Z_1 \geq Z_2 \geq Z_3 \geq ... \geq Z_K = Z$. Bounds are achieved by a specification through norms of varying order, such as $(2, 2, +\infty)$ and $(3, 3, 3)$.

We can express the normalizing constant as $Z = \sum_t \vec{a}_t \vec{b}_t \vec{c}_t$, where:

$$\vec{a} = [N_{1d} + \alpha_1, \ldots, N_{Td} + \alpha_T] \tag{6}$$

$$\vec{b} = [N_{w1} + \beta, \ldots, N_{wT} + \beta] \tag{7}$$

$$\vec{c} = \left[\frac{1}{N_1 + W\beta}, \ldots, \frac{1}{N_T + W\beta}\right] \tag{8}$$

The construction of the initial upper bound Z_0 on Z relies on the Holder's Inequality (generalized version); $Z_0 = \|a\|_p \|b\|_q \|c\|_r$, where $1/p + 1/q + 1/r = 1$.

The formula for obtaining the bound at step l is:

$$Z_l = \sum_{i=1}^{l} (a_i b_i c_i) + \|a_{l+1:T}\|_p \|b_{l+1:T}\|_q \|c_{l+1:T}\|_r \geq Z \tag{9}$$

The sequence of bounds is decreasing and, when $l = T$, the constant Z is retrieved. The choice of (p, q, r) is a design choice, typically made to ensure computational efficiency.
For this project, I worked on $(2, 2, \infty)$, and $(3, 3, 3)$, the two applications discussed more in detail by Newman et al.

Whilst for $(r, q, p) < \infty$ norms can be updated in constant time at each step $l + 1$, this is impossible for $r = \infty$, for which we use $\|c_{l+1:T}\|_r = \min_t N_t$.
Updating $\|c_{l+1:T}\|_r$ under this specification involves checking the sorted order of the values of N_t and storing the new minimum. I found this to be a major problem in terms of efficiency, and have only partially been able to slow down the algorithm excessively, as will be discussed in later sections.

Similar computational costs must also be faced when sorting the indexes for topic assignments in each document. This allows to sample from the topic with the highest probability mass.

For *FastLDA* to be quicker than standard Collapsed Gibbs, the updates of norms and sorted indexes must occur efficiently; this is especially important as T becomes large.

5.1 Computation and Updating of Norms

The computation of the quadratic sums composing the norms only occurs in the first Gibbs iteration, as values must be initialized for all documents and all words. The document-level norm is only computed for the last word of each document, together with $F.indx\_dt[d, :]$, storing document-topic assignments in descending order.

```
#iter = 1
if ind_w == length(words) && i == Rw
    F.indx_dt[d, :] .= sortperm(F.Ndt[d, :], rev = true)
    for t = 1:T
        Ad[t] = (F.Ndt[d, t] + F.alpha[t])
    end
    F.a[d] = Ad' * Ad
end

for t in 1:T
    Bw[t] = (F.Ntw[t, w] + F.beta)
end
F.b[w] = Bw' * Bw
```

As discussed above, in the specification $(2, 2, \infty)$ only two of the three norms can be updated in constant time, whilst $\|c_{l+1:T}\|_r$ must be updated by reassessing which is the topic with the minimal number of word assignments. The quadratic sums of the norms, $F.a$ and $F.b$ in the code, are updated as follows:

```
function update_norms(F::PTM, d::Int64, w::Int64, t_new::Int64, t_old::Int64, d_last::Int64, w_last::Int64)
    T = F.T
    if d_last != t_old
        F.a[d] -= (F.alpha[d_last] + F.Ndt[d, d_last] - 1)^2
        F.a[d] -= (F.alpha[t_old] + F.Ndt[d, t_old] + 1)^2
        F.a[d] += (F.alpha[d_last] + F.Ndt[d, d_last])^2
        F.a[d] += (F.alpha[t_old] + F.Ndt[d, t_old])^2
        sort_update(T, F.Ndt[d, :], F.indx_dt[d, :], d_last, t_old)
    end
    if w_last != t_old
        F.b[w] -= (F.beta + F.Ntw[w_last, w] - 1)^2
        F.b[w] -= (F.beta + F.Ntw[t_old, w] + 1)^2
        F.b[w] += (F.beta + F.Ntw[w_last, w])^2
        F.b[w] += (F.beta + F.Ntw[t_old, w])^2
    end
end
```

Updates for a only occur when the last topic assignment in document d is different from the new assignment, and, for b, when the last topic assignment to word w is different from the new one. This enables to save up to 15% of total running time of the Gibbs Sampling step. (difference estimated through $@etime$ in Julia).

Instead, the norm C is originally computed and updated through:

```
#iter == 1
if d == D && ind_w == length(words) && i == Rw
    indx_t .= sortperm(F.Nt, rev = true)
    F.C = 1.0 / (F.Nt[indx_t[T]] + W * F.beta)
end

#iter > 1
if t_new != t_old
    sort_update(T, F.Nt, indx_t, t_new, t_old)
    F.C = 1.0 / (F.Nt[indx_t[T]] + W * F.beta)
end
```

For efficiency, $indx\_t$ is only computed once during the first iteration. It is then updated through the function $sort\_update$, which I discuss in the next section. The function is also used to update $F.indx\_dt[d, :]$.
The norm $F.C$ is updated accordingly, following the indicated minimal topic.
However, the value of $F.Nt$ changes at almost every iteration. Such a frequent substitution of values in $F.C$ makes its update more computationally expensive than that of the other two norms. This is a major limitation in the efficiency of the code, which I was not entirely able to fix. Still, the update of indexes and the if statement provide substantial performance improvements.

5.2 Updating Indexes

The update of norms requires resorting indexes in descending order, so to discover the minimum$(F.C)$ or maximum $(Mainloop)$ element within $F.Nt$ and $F.Ndt[d, :]$ respectively. To do so, instead of recurring again to $sortperm$, I built a function, $sort\_update$:

```
function sort_update(T::Int64, Nt::Vector{Int64}, ind::Vector{Int64}, t_new::Int64, t_old::Int64)
    prev = 0
    t_new = ind[t_new]

    while t_new > 1 && Nt[ind[t_new]] > Nt[ind[t_new - 1]]
        prev = ind[t_new]
        ind[t_new] = ind[t_new - 1]
        ind[t_new - 1] = prev
        t_new -= 1
    end

    t_old = ind[t_old]
```

```
    while t_old < T && Nt[ind[t_old]] <= Nt[ind[t_old + 1]]
        prev = ind[t_old]
        ind[t_old] = ind[t_old + 1]
        ind[t_old + 1] = prev
        t_old += 1
    end
end
```

Rather than going through the whole index, it exploits the fact that only two numbers are changed per each iteration (t_new,t_old involve either one or two elements of Nt or $Ndt[d,:]$). Starting from the indicated positions, it checks the descending orders of the variables one position at a time, and swaps elements in the considered index until the conditions for the increased and decreased indexes are respectively satisfied.

When T is equal to 4, there are no extensive computational advantages. However, as the number of chosen topics increases, the function allows to save a significant amount of computations.

5.3 Topic Assignment Determination

For the first iteration over the whole corpus, since norms haven't been defined yet, topic assignment occurs via standard Collapsed Gibbs, where probabilities are calculated for all topics through:

$$pbs[t] = \frac{(F.N_{dt}[d,t] + F.\alpha[t]) \cdot (F.N_{tw}[t,w] + F.\beta)}{F.N_t[t] + W \cdot F.\beta} \tag{10}$$

And the normalizing constant as: $Z = \sum_t pbs[t]$

Sampling from a uniform $U(0, Z)$, topic assignment is determined through:

```
Z = sum(pbs)
U = Z * rand()
currprob = pbs[1]
t_new = 1

while U > currprob
    t_new += 1
    currprob += pbs[t_new]
end
```

Starting from the second iteration over the corpus $FastLDA$ is applied, exploiting the constantly updated bounds on the normalization constant.

```
A = F.a[d]
B = F.b[w]
U = rand()
for t in 1:T
    q = F.indx_dt[d, t]
    if t != 1
        pbs[t] = pbs[t - 1]
    else
        pbs[t] = 0.0
    end
    pbs[t] += (F.Ndt[d, q] + F.alpha[q]) * (F.Ntw[q, w] + F.beta) / (F.Nt[q] + W * F.beta)

    A -= (F.alpha[q] + F.Ndt[d, q])^2
    B -= (F.beta + F.Ntw[q, w])^2
    A = max(0.0, A)
    B = max(0.0, B)
    Z_old = Z
    Z = pbs[t] + sqrt(A * B) * F.C

    if pbs[t] < U * Z
        continue
    elseif t == 1 || U * Z > pbs[t - 1]
        t_new = q
        break
    else
        U = (U * Z_old - pbs[t - 1]) * Z / (Z_old - Z)
```

```
        for j in 1:t
            if pbs[j] >= U
                t_new = F.indx_dt[d, j]
                break
            end
        end
    end
end
```

The loop begins by considering the number of topic assignments for each document, $F.Ndt$, which is often sparsely concentrated; only a few topics receive a high number of assignments. The index corresponding to the highest number of topic assignments in Ndt, for each document, is labeled q. The cumulative probability mass is initialized on the topic indicated by such index.

Only the part of the bounds relative to q is then subtracted to the norms. The old bound of the normalization constant is stored, and the new Z, based on the current sum of probabilities and the updated bounds is computed, at step t, as;

$$Zt = \sum_{i=1}^{t}(a_i b_i c_i) + \|a_{l+1:T}\|_2 \|b_{l+1:T}\|_2 \frac{1}{min_t N_t + W\beta} \tag{11}$$

Where $\sum_{i=1}^{t}(a_i b_i c_i) = \sum_{i=1}^{t} pbs[i]$ expresses the sum of probabilities up to topic t: not all have been computed.

At this point, if the current sum of probabilities is smaller than the value uniformly drawn from $U(0, Z)$, the loop repeats by doing computations on a new topic.
Else, if the uniform draw has surpassed the last sum of probabilities, the new topic is allocated as the current one. If it hasn't surpassed the last sum, but it is greater than the current, it means the correct topic was amongst the previous ones, and it is not the current one.
In this case, the threshold U is rescaled, and the topic is chosen amongst all the ones up to the current t.

It is important to stress that the majority of iterations stop at the first couple of topics; this is key for the velocity of the algorithm, and is induced by the sparse concentration of probability masses.

6 Fast LDA (3,3,3)

All the theoretical considerations discussed in the previous section and the methodologies to update indexes are also valid for $FastLDA(3,3,3)$. This section considers only the areas in which the two applications differ, namely the computation and update of norms, and the sampling step.

6.1 Computation and Updating of Norms

It is now possible to update all norms in constant time. The initialization of the (now cubic)sums and the index $H.indx\_dt[d,:]$, are all analogous to before, except for $F.c$ (lower case since it is now the summation component of the norm, and not the norm itself) which is computed as:

```
if g != 1
    for t in 1:T
        Ct[t] = 1.0 / (H.Nt[t] + W*H.beta)
    end
    H.c = sum(y -> y^3, Ct)
end
```

All norms are then continuously updated via:

```
function update_norms(H::PTM, d::Int64, w::Int64, i::Int64, ind_w::Int64, t_new::Int64, t_old::Int64, d_last::Int
    T = H.T
    W = H.W
    if d == 1 && ind_w == 1 && i == 1
        H.c -= (1.0 / (W * H.beta + H.Nt[t_old] + 1))^3
        H.c += (1.0 / (W * H.beta + H.Nt[t_old]))^3
    elseif t_new != t_old
        H.c -= (1.0 / (W * H.beta + H.Nt[t_new] - 1))^3
        H.c -= (1.0 / (W * H.beta + H.Nt[t_old] + 1))^3
        H.c += (1.0 / (W * H.beta + H.Nt[t_new]))^3
        H.c += (1.0 / (W * H.beta + H.Nt[t_old]))^3
    end

    if d_last != t_old
        H.a[d] -= (H.alpha[d_last] + H.Ndt[d, d_last] - 1)^3
        H.a[d] -= (H.alpha[t_old] + H.Ndt[d, t_old] + 1)^3
        H.a[d] += (H.alpha[d_last] + H.Ndt[d, d_last])^3
        H.a[d] += (H.alpha[t_old] + H.Ndt[d, t_old])^3
        sort_update(T, H.Ndt[d, :], H.indx_dt[d, :], d_last, t_old)
    end

    if w_last != t_old
        H.b[w] -= (H.beta + H.Ntw[w_last, w] - 1)^3
        H.b[w] -= (H.beta + H.Ntw[t_old, w] + 1)^3
        H.b[w] += (H.beta + H.Ntw[w_last, w])^3
        H.b[w] += (H.beta + H.Ntw[t_old, w])^3
    end
end
```

The only relevant observation is that, as $H.c$ is recomputed once at each sampling iteration (except the first), when the first word in the first document is considered, it doesn't make sense to compare the current topic allocation t_old with the last t_new, and the update only concerns the current one. Following the very first word, all updates only occur when $t\_new != t\_old$.

6.2 Topic Assignment Determination

The first iteration exploits Collapsed Gibbs and initializes the norms, as above. Then, $FastLDA(3,3,3)$ is applied via:

```
A = H.a[d]
B = H.b[w]
C = H.c
U = rand()

for t in 1:T
```

```
    q = H.indx_dt[d, t]
    if t != 1
        pbs[t] = pbs[t - 1]
    else
        pbs[t] = 0.0
    end
    pbs[t] += (H.Ndt[d, q] + H.alpha[q]) * (H.Ntw[q, w] + H.beta) / (H.Nt[q] + W * H.beta)

    A -= (H.alpha[q] + H.Ndt[d, q])^3
    B -= (H.beta + H.Ntw[q, w])^3
    C -= (1.0 / (W*H.beta  + H.Nt[q]))^3
    A = max(0.0, A)
    B = max(0.0, B)
    C = max(0.0, C)
    Z_old = Z
    Z = pbs[t] + cbrt(A * B * C)

    if pbs[t] < U * Z
        continue
    elseif t == 1 || U * Z > pbs[t - 1]
        t_new = q
        break
    else
        U = (U * Z_old - pbs[t - 1]) * Z / (Z_old - Z)
        for j in 1:t
            if pbs[j] >= U
                t_new = H.indx_dt[d, j]
                break
            end
        end
    end
end
```

The reasoning is the same as before. The only difference is in the computation of the new bound for the normalization constant, which is:

$$Zt = \sum_{i=1}^{t}(a_i b_i c_i) + \|a_{l+1:T}\|_3 \|b_{l+1:T}\|_3 \|c_{l+1:T}\|_3 \tag{12}$$

With $\sum_{i=1}^{t}(a_i b_i c_i) = \sum_{i=1}^{t} pbs[i]$. The requirement for the order of the norms still holds, as their ratios add up to one.

Due to a more efficient updating of norms, this second specification, in my code, is faster than $FastLDA(2, 2, +\infty)$. This, in theory, should not happen, and is a major limitation of my project, as will be later discussed.

7 Sparse LDA

SparseLDA avoids computing $\sum_{t=1}^{t} q(t)$ for each word through a different approach. It does not rely on the calculation and update of bounds to the normalization constant but, following a simplification and three-stage division of the probability mass, it only computes the fraction of it that is required by each "random pointer".

First, the following decomposition can be obtained by rearranging the denominator of equation (5):

$$P(z = t|w) \propto \frac{\alpha_t \beta}{n_t + \beta W} + \frac{n_{t|d}\beta}{n_t + \beta W} + \frac{(\alpha_t + n_{t|d})n_{w|t}}{n_t + \beta W} \tag{13}$$

The full sampling mass is divided into three "buckets". The first, constant for all documents is called s (smoothing bucket), the second, constant for all words, is called r (document-topic bucket), and the third, whose updating is the most difficult, is called q (word-topic bucket).
The sparsity of topic assignments to words and documents is still exploited, as, for many words, certain "buckets" are more significant components of the overall probability mass than others.

The link between these buckets and the unnormalized weight is that $\sum_t q(t) = r + q + s$, where:

$$s = \sum_t \frac{\alpha_t \beta}{n_t + \beta W} \tag{14}$$

$$r = \sum_t \frac{n_{t|d}\beta}{n_t + \beta W} \tag{15}$$

$$q = \sum_t \frac{(\alpha_t + n_{t|d})n_{w|t}}{n_t + \beta W} \tag{16}$$

At each sampling step we draw from a $U(0, s + r + q)$, and check in which of the three buckets we have fallen into. Topics are assigned to each word based on three different criteria, for each of the buckets, which are discussed in detail in section 7.3 (Topic Assignment Determination).

In the next two sections, instead, I discuss how the three buckets and the variables $posNdt$ and $posNdw$, are computed and updated in constant time. These steps are fundamental to ensure $SparseLDA$ is faster than $FastLDA$, and of regular Collapsed Gibbs.

7.1 Computation and Updating of Buckets

The methods previously utilized to compute and update norms are not valid in $SparseLDA$, as all buckets must be recomputed, at different levels, for each MCMC iteration.

The smoothing bucket s, as well as the component f of bucket q, independent of word types, are computed only once at the beginning of each iteration.
Component f is then redefined for the topics with at least one positive assignment in the considered document. Along with f, bucket r is computed for all positive allocations in each document.

```
for g in 1:iter
    S.s = 0.0
    for t in 1:T
        S.s += S.alpha[t] * S.beta / (S.beta*W + S.Nt[t])
        S.f[t] = S.alpha[t] / (S.beta*W + S.Nt[t])
    end
    for d in 1:D
        S.r = 0.0
        for t in posNdt[d]
            S.r += S.beta * S.Ndt[d,t] / (S.beta*W + S.Nt[t])
            S.f[t] = (S.alpha[t] + S.Ndt[d,t]) / (S.beta*W + S.Nt[t])
        end
```

Instead, bucket q must be computed separately for each word (only for the corresponding positive topic assignments), due to the component $S.Ntw$.

```
q = 0.0
for t in posNtw[w]
    q += S.Ntw[t, w] * S.f[t]
end
```

The two buckets s and r are kept updated, after being reinitialized at each iteration (s), and each document (r), through the following function:

```
function update_buckets(S::PTM, d::Int64, last_d::Int64, ind_w::Int64, i::Int64, t_old::Int64, t_new::Int64)
    W = S.W
    if d == 1 && ind_w == 1 && i == 1
        S.s -= S.alpha[t_old] * S.beta / (S.beta * W + S.Nt[t_old] + 1)
        S.s += S.alpha[t_old] * S.beta / (S.beta * W + S.Nt[t_old])
    elseif t_new != t_old
        S.s -= S.alpha[t_new] * S.beta / (S.beta * W + S.Nt[t_new] - 1)
        S.s -= S.alpha[t_old] * S.beta / (S.beta * W + S.Nt[t_old] + 1)
        S.s += S.alpha[t_new] * S.beta / (S.beta * W + S.Nt[t_new])
        S.s += S.alpha[t_old] * S.beta / (S.beta * W + S.Nt[t_old])
    end

    if last_d != d
        S.r -= S.beta * (S.Ndt[d, t_old] + 1) / (S.beta * W + S.Nt[t_old] + 1)
        S.r += S.beta * S.Ndt[d, t_old] / (S.beta * W + S.Nt[t_old])
    elseif t_new != t_old
        S.r -= S.beta * (S.Ndt[d, t_new] - 1) / (S.beta * W + S.Nt[t_new] - 1)
        S.r -= S.beta * (S.Ndt[d, t_old] + 1) / (S.beta * W + S.Nt[t_old] + 1)
        S.r += S.beta * S.Ndt[d, t_new] / (S.beta * W + S.Nt[t_new])
        S.r += S.beta * S.Ndt[d, t_old] / (S.beta * W + S.Nt[t_old])
    end
end
```

Similar updating is impossible for q, whose computation is remarkably less efficient. Its word-independent component, f, can however be updated in continuous time. I discuss this and the update of relevant indexes in the next segment.

7.2 Updating Indexes and Component "f"

The indexes $posNdt$ and $posNtw$ are crucial in the computation of buckets, as well as in the Gibbs Sampling step.
The two variables, which store all topics receiving at least one assignment in each document and word respectively, are computed only once, before any of the MCMC iterations, by exploting the function $findall$.

```
posNdt = [findall(e -> e != 0, S.Ndt[d, :]) for d in 1:D]
posNtw = [findall(e -> e != 0, S.Ntw[:, w]) for w in 1:W]
```

Their update then occurs through the following function, including also adjustments for component f, since they must occur in the same positions in the code. "step" is a Boolean indicator indicating whether we are at the stage where a topic has been removed from all counts(once t_old is assigned) or whether it has been added (assignment of t_new).

```
function update_pos_f(S::PTM, posNdt::Vector{Vector{Int64}}, posNtw::Vector{Vector{Int64}},
    d::Int64, w::Int64, t_old::Int64, t_new::Int64, step::Int64)
    W = S.W
    if step == 0
        S.Ndt[d, t_old] == 0 && filter!(e -> e != t_old, posNdt[d])
        S.Ntw[t_old, w] == 0 && filter!(e -> e != t_old, posNtw[w])

        S.f[t_old] = (S.alpha[t_old] + S.Ndt[d, t_old]) / (S.beta * W + S.Nt[t_old])
    else
        S.Ndt[d, t_new] == 1 && push!(posNdt[d], t_new)
        S.Ntw[t_new, w] == 1 && push!(posNtw[w], t_new)

        S.f[t_new] = (S.alpha[t_new] + S.Ndt[d, t_new]) / (S.beta * W + S.Nt[t_new])
    end
end
```

I tried to furtherly improve the update of $posNdt$ and $posNtw$ by excluding checks on the topics to which each Ndt is converging to, but I was ultimately unable to implement efficient calculations, and my approach actually worsened the performance of the code, so I avoided it entirely.

7.3 Topic Assignment Determination

Following the computation of buckets and indexes, the key assignment of topics in $SparseLDA$ occurs through three separate criteria for each bucket the random draw falls into.

```
Z = (s + r + q)
U = rand() * Z
t_new = 0

if U < s  #smoothing only bucket
    U = U + r + q
    for t in 1:T
        U += S.alpha[t]*S.beta / (S.beta*W + S.Nt[t])
        t_new = t
        if U >= Z
            break
        end
    end
elseif U < s + r #document topic bucket
    l = 1
    U = U + q
    while U <= Z
        t_new = posNdt[d][l]
        U += S.beta*S.Ndt[d, t_new] / (S.beta*W + S.Nt[t_new])
        l += 1
    end
else  # U > (s + r)  "topic-word" bucket
    l = 1
    while U <= Z
        t_new = posNtw[w][l]
        U += S.Ntw[t_new, w] * S.f[t_new]
        l += 1
    end
end
```

First, if $U < s$ we skim through all topics, adding up $\frac{\alpha_t \beta}{n_t + \beta W}$ until we reach a value greater than U and assign the corresponding topic.

Second, if $s < U < (s + r)$, we only iterate over topics for which the current document has a positive assignment $posNdt$. We reason similarly to before, but adding up $\frac{\beta n_{t|d}}{n_t + \beta W}$ until we reach a value greater than U.

Lastly, if $U > (s+r)$, we have reached the topic-word bucket. We only iterate over topics that have a positive assignment $posNtw$ for the considered word w, and, as above, we add up $\frac{(\alpha_t + n_{t|d}) n_{w|t}}{n_t + \beta W}$ until the uniform's value is reached.

The sampling step of $SparseLDA$ is typically quicker than that of $FastLDA$. Instead, the computation and update of buckets can be more cumbersome than that of norms, since these are only computed once at the first MCMC step, and are then only updated. Nonetheless, as outlined by Yao et al., the former mismatch seems to prevail over the latter, making $SparseLDA$ generally faster.

8 Dirichlet Parameters: Definition and Updating

As specified by Yao et al. , I allow α_t parameters to vary for each of the T topics, and I only admit one value of β in all the algorithms.
This implies that, in the Dirichlet distributions of topics over words, all topics have an identical weight. Instead, in the Dirichlet for topics and documents, each topic is allowed to have a different weight. Nonetheless, in the code, all Dirichlet parameters are initialized as random numbers between 0 and 1, implying a strong concentration of probabilities around the extremes.
After having performed Collapsed Gibbs over the entire corpus, the prior parameters are updated, before a new simulation starts, through the following formulas:

$$\alpha_t = \alpha_t(old) \cdot \frac{\sum_{d=1}^{D} \psi(N_{dt} + \alpha_t) - D\psi(\alpha_t)}{\sum_{d=1}^{D} \psi(\sum_{t=1}^{T} N_{dt} + \sum_{t=1}^{T} \alpha_t) - D\psi(\sum_{t=1}^{T} \alpha_t)} \tag{17}$$

$$\beta = \beta(old) \cdot \frac{\sum_t \sum_w \psi(N_{tw} + \beta) - TW\psi(\beta)}{\sum_{t=1}^{T} \psi(N_t + W\beta) - TW\psi(W\beta)} \tag{18}$$

Where ψ is the digamma function.

These expressions originate from the conjugacy of the Dirichlet-Multinomial model and the maximization of the logarithm of the posterior distributions of the parameters.
The code to perform such operations is in the function *prior_update*, which is identical in all scripts.

The specification I used, proposed by Yao et al., was not implemented by Newman et al., who only allowed for one value of α.
I adapted $FastLDA$ by allowing more than one value as I think this adds a further layer of modeling flexibility. The user can specify even very heterogeneous prior beliefs about each of the underlying topic parameters, with very limited additional costs, simply by manually inserting the desired values in the function *init_vars*.

9 Computation of Perplexity

In this section I provide mathematical expressions for the formulas underlying the computation of perplexity, which is only computed after the burn-in has been discarded and only on a testing subset of the corpus, to avoid over-fitting. The probabilities $P(w|t)$ and $P(d|t)$ are estimated as in equations (3) and (4) above.

Instead, we can compute the probability $P(w|d)$ as:

$$P(w|d) = \frac{I-1}{I} \cdot p(w|d) + \sum_t \frac{\phi_{tw} \cdot \theta_{dk}}{I} \tag{19}$$

Where $p(w|d)$ is the previous probability of w in document d, and I is the iteration number (sampling number).

When $I == 1$, that is if the burn-in has been discarded and it's the first iteration in which we begin computing Perplexity, $p(w|d)$ is initialized as a random number. The log-likelihood of the newly sampled corpus is then computed as:

$$log(L) = \sum_d \sum_w R_{d,w} \cdot \log(P(w|d)) \tag{20}$$

Where $R_{d,w}$ is the number of repetitions of word w in document d. Finally, perplexity is computed as:

$$Perplexity = \exp\left(\frac{-log(L)}{N}\right) \tag{21}$$

Where N is the total number of words in the corpus (different from W).
All the operations mentioned above occur through the function $PPLEX$, identical in all scripts.

10 MCMC Results and Diagnostics

To compare results across the three algorithms and check MCMC convergence, I ran 400 Monte Carlo simulations with a $burnin = 200$ and $sample = 200$.
First, I show general results of document-topic allocations for $FastLDA(2, 2, +\infty)$, $FastLDA(3, 3, 3)$ and $SparseLDA$. I only show a sample of results, but identical patterns of similarity can be observed for all other documents.

```
julia> F.Ndt_avg
500 4  Matrix{Float64}:
0.17    14.72    61.39    42.72
0.07     0.315   96.51     0.105
0.16     0.47   152.505   83.865
0.22    47.87     0.4      0.51
0.195    3.55   145.205    2.05
73.755    4.52    53.21     2.515
0.475  122.15     9.265   51.11
0.895    0.63     9.33   142.145
8.67    50.295   72.235    1.8
1.73     0.515    0.525   96.23
1.37     0.58    89.56     0.49
4.21     1.14   202.035    0.615
0.885    4.25   120.925    9.94
13.15     1.655   51.14     9.055
...
0.325    1.505  216.115    0.055
1.095    1.365  219.475    0.065
146.315    0.28     1.22     2.185
6.515   40.65     0.245   21.59
0.145   19.79     0.255    1.81
15.845    0.47    88.85     2.835
63.94     2.865   11.71     0.485
0.34    52.95     0.095    0.615
1.59    74.265    0.435   56.71
0.24    72.02     0.15    12.59
0.295    0.36   164.935    0.41
205.045    0.055    8.31     7.59
0.135   28.41     0.17     1.285
7.305   20.18   127.86     0.655
```

```
julia> H.Ndt_avg
500 4  Matrix{Float64}:
2.315   12.455   83.395   20.835
0.675    0.095   94.275    1.955
13.345    9.525  213.855    0.275
0.985   35.98    10.86     1.175
0.885    0.76   143.1      6.255
75.41     8.3     21.685   28.605
1.325  180.15     0.35     1.175
0.62    16.195    7.63   128.555
4.09     6.055   73.395   49.46
4.19     7.1      3.135   84.575
0.52     1.455   69.5     20.525
3.11     2.355  174.3     28.235
12.37     7.56   109.21     6.86
3.55     8.105   61.765    1.58
...
0.295    0.585  146.8     70.32
31.4     46.475  141.58     2.545
145.93     0.39     1.735    1.945
5.825   39.74    21.175    2.26
2.43    18.47     0.93     0.17
24.165    1.92    75.815    6.1
62.1      1.455    7.985    7.46
9.345   42.88     1.225    0.55
7.91    74.425   50.135    0.53
20.75    58.755    4.945    0.55
1.765    1.04   162.45     0.745
198.9      1.57     3.875   16.655
0.81    27.265    1.245    0.68
25.3      0.645   76.75    53.305
```

```
julia> S.Ndt_avg
500 4  Matrix{Float64}:
24.835    4.9     85.335   3.93
1.155    1.85    92.845   1.15
67.51    14.48   149.51    5.5
1.145   43.29     1.94    2.625
2.385    5.295  140.825   2.495
99.285    7.195   13.635  13.885
3.305  118.795   57.675   3.225
36.165   15.575    3.77   97.49
5.7     59.04    64.595   3.665
10.26     6.645    9.48   72.615
2.24     4.335   81.51    3.915
12.15     8.245  179.555   8.05
44.255   18.53    67.42    5.795
2.74    26.68    28.945  16.635
...
2.98    10.165  202.575   2.28
1.45    26.14   179.8    14.61
127.845    2.17     6.51   13.475
5.065   35.08    18.03   10.825
1.175   14.77     3.84    2.215
15.755    3.51    72.2    16.535
50.27     6.615   11.085  11.03
2.195   46.225    2.35    3.23
3.545   73.4     50.615   5.44
1.22    67.585    9.68    6.515
4.485    5.675  153.88    1.96
166.79     3.74    17.835  32.635
1.805   22.725    2.005   3.465
1.06    18.84   106.41   29.69
```

It is important to stress that, to achieve these tables, I manually fixed the random seed to uniform the labeling of topics. This is not automatically done in the code and is the reason why, whilst numbers might appear similar overall across documents, the topics to which they are assigned will differ, since they are assigned at random when utilizing the function $init\_vars$.
If one wants to reproduce similar results, the seed must be fixed, so to assign the same "qualitative labels" (topic 1, 2, 3, 4) to all algorithms in the same order.

Results seem overall consistent, as all algorithms unanimously indicate the same topic as the one with the largest importance in 99% of all documents. In some instances, however, they differ quite significantly concerning the weights of less important topics. This could be a major limitation of my code, as, even by increasing the size of Monte Carlo simulations, differences don't disappear.
Whilst all algorithms produce consistent and sensible results for topics with largest weights, findings do not always match concerning topics of "less importance", even with an increased number of iterations.

To confirm the validity of MCMC results, it is also important to verify the convergence of document-topic Markov chains (assumed ergodic and π-invariant by construction, due to Gibbs Sampling).
I thus provide Trace Plots for six documents chosen at random in the corpus; 294, 106, 402, 277, 112, 484. Similar converging trends hold for any other document. Other metrics to check convergence could be used, but I found this to be the most appealing.

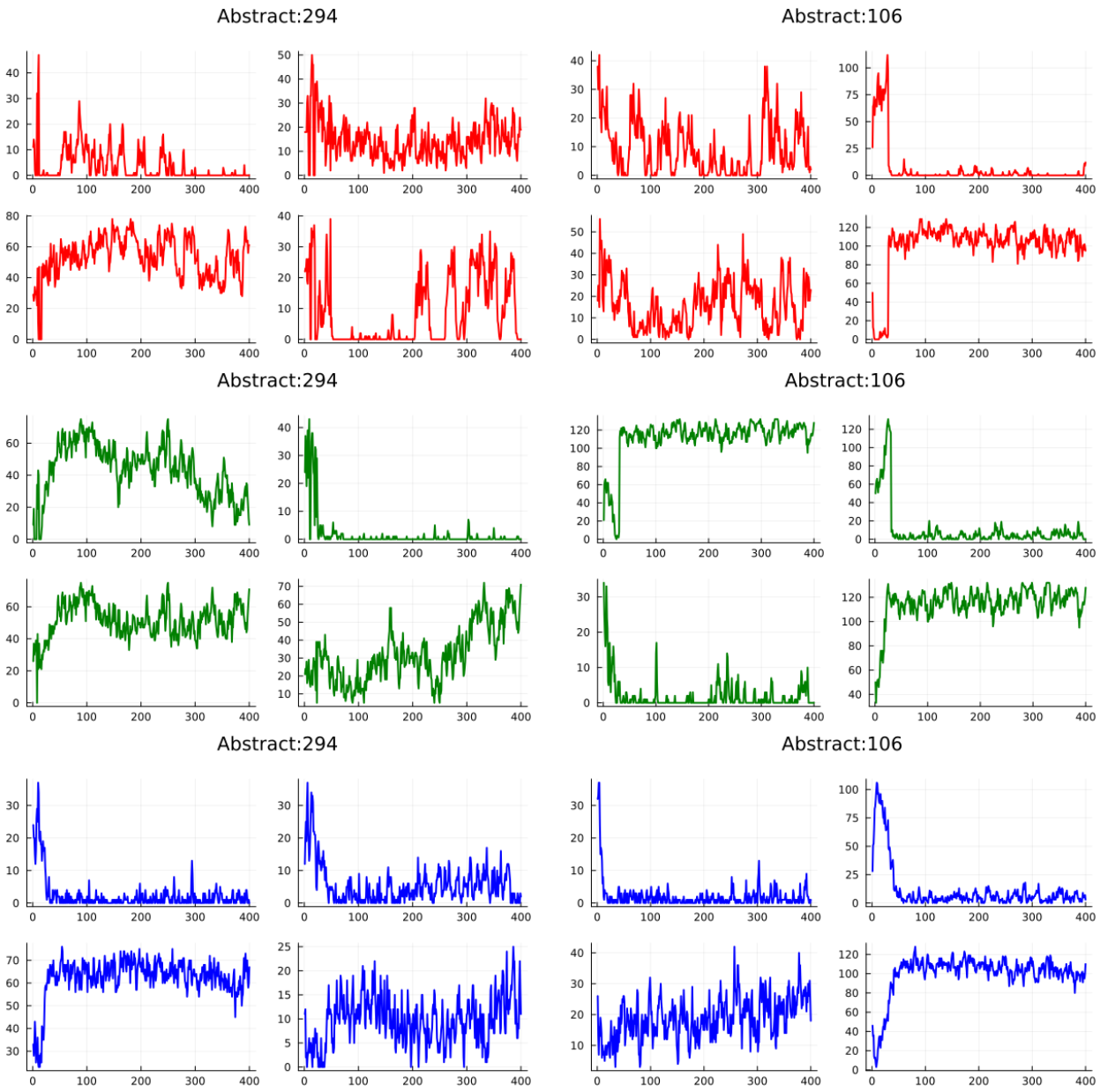

Figure 1: MCMC Trace Plots: Documents 294, 106. Y-axis: Frequencies of topic assignments, Topics = 4, Red = $FastLDA(2, 2, +\infty)$ Green = $FastLDA(3, 3, 3)$, Blue = $SparseLDA$

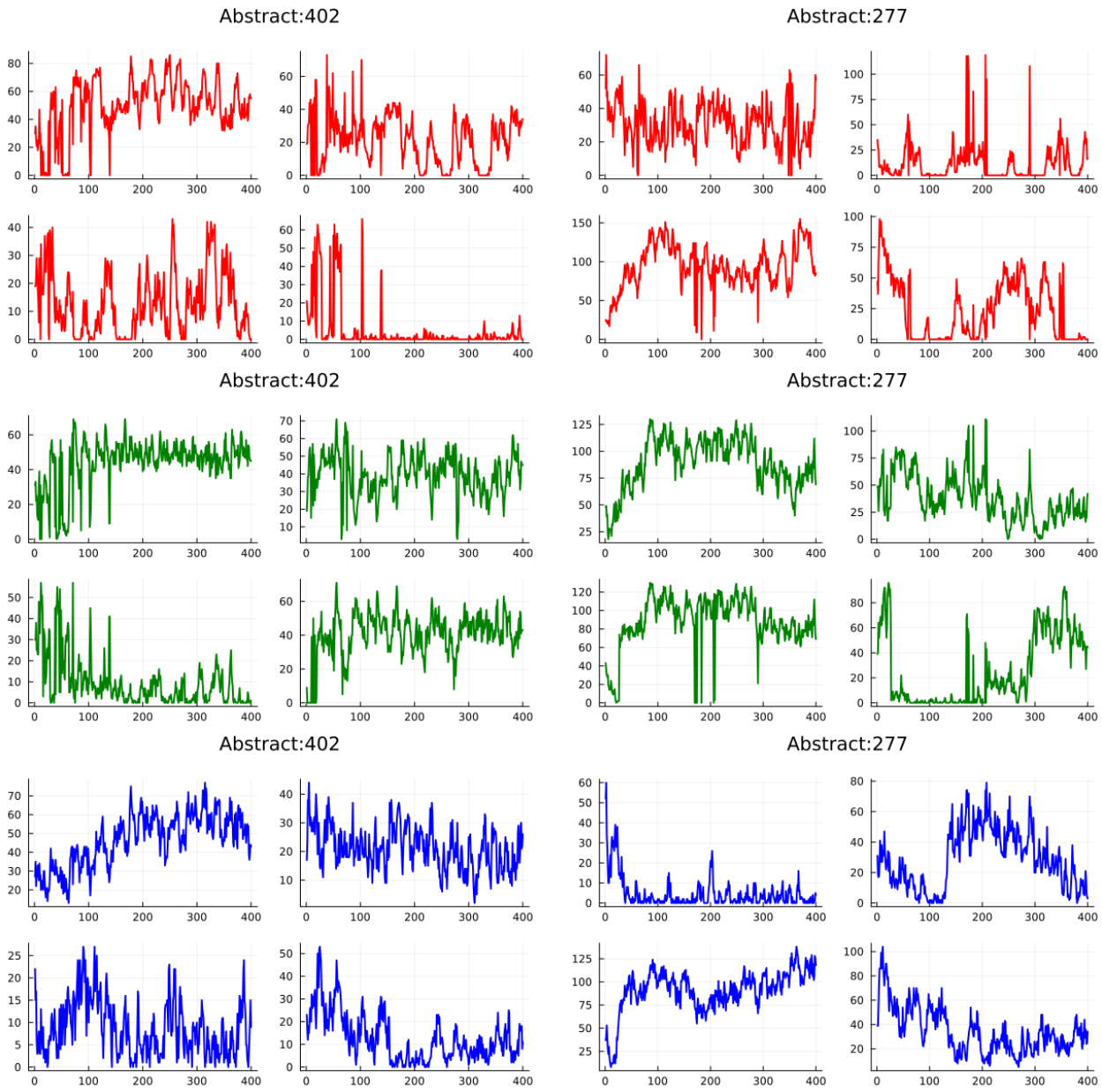

Figure 2: MCMC Trace Plots: Documents 402, 277. Y-axis: Frequencies of topic assignments, Topics = 4, Red = $FastLDA(2, 2, +\infty)$ Green = $FastLDA(3, 3, 3)$, Blue = $SparseLDA$

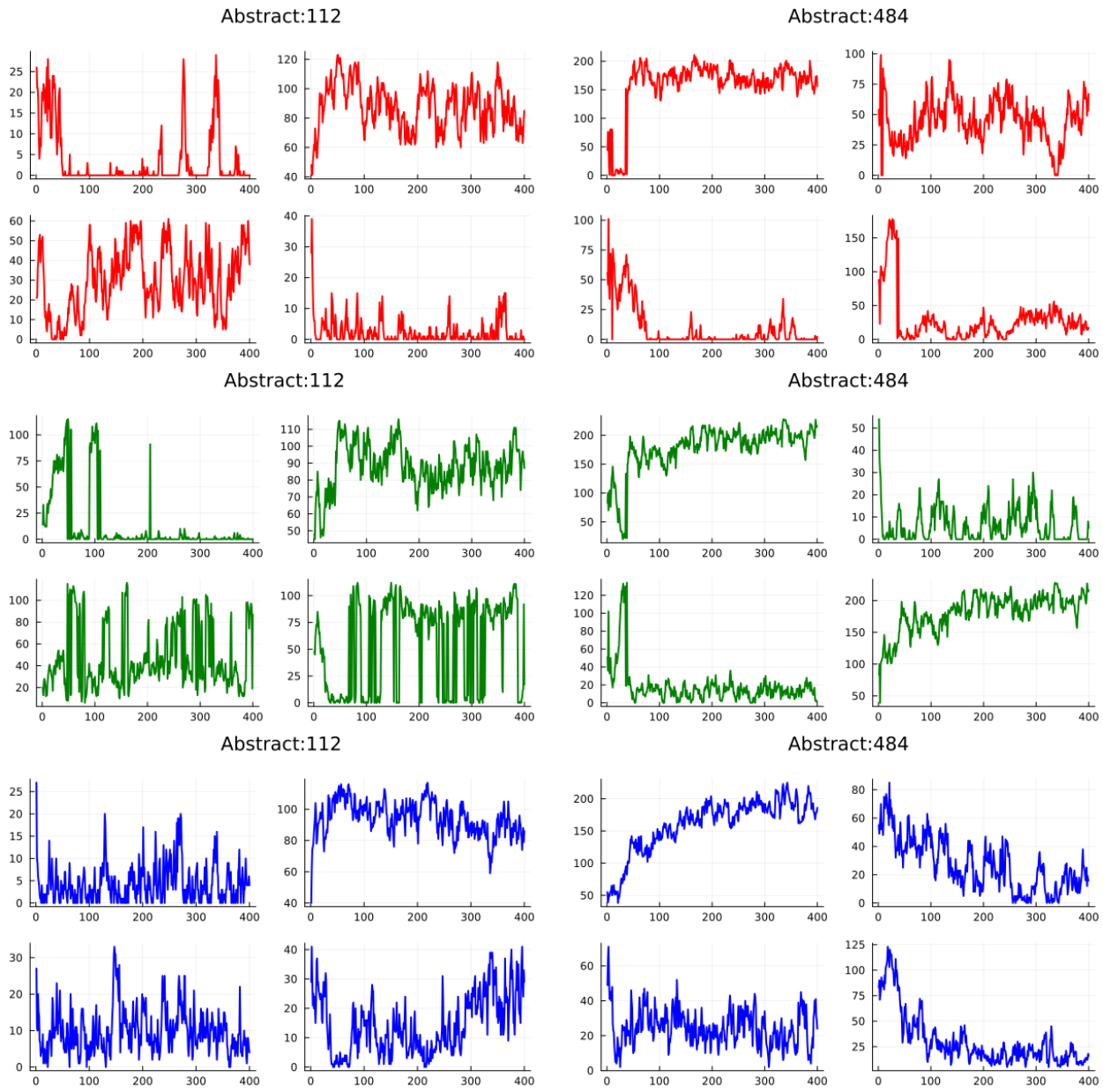

Figure 3: MCMC Trace Plots: Documents 112, 484. Y-axis: Frequencies of topic assignments, Topics = 4, Red = $FastLDA(2, 2, +\infty)$ Green = $FastLDA(3, 3, 3)$, Blue = $SparseLDA$

The following general observations about Trace Plots can be made.
First, convergence is evident for all topic assignments in all considered documents. The only exception is topic 4 in document 112 for $FastLDA(3, 3, 3)$, which is still in the transient phase after 400 iterations.
Second, the transient phase generally terminates before the 200th MCMC iteration in all algorithms. Such a pattern can be generally observed for all chains, with few exceptions due to the stochastic nature of the process.
Finally, whilst in the provided plots chains of $FastLDA$ seem to converge at a slightly faster rate compared to those of $SparseLDA$, this pattern doesn't hold on average, as the length of transient phases of Markov chains is, overall, similar across all algorithms.

11 Limitations of the Project and Final Considerations

My code has several important limitations, particularly concerning the efficiency of $FastLDA$.
Whilst, since the beginning of the project, I have made substantial improvements to the velocity of the algorithms through the corrections discussed in sections (5.1,2) and (6.1,2), I am aware that $FastLDA(3,3,3)$ and, particularly, $FastLDA(2,2,+\infty)$ are still too slow, especially when compared to $SparseLDA$.

My implementation of $SparseLDA$ is, in fact, up to 12 times faster compared to that of $FastLDA$, regardless of the chosen number of topics and size of the dataset.
Whilst a difference should appear, as also highlighted by Yao et al., in my case the mismatch is excessive, and calls, most likely, for some issues in the $FastLDA$ algorithms.

Concerning $FastLDA(2,2,+\infty)$, focus should be put on enhancing the update of the norm-component $F.C$.
Having tested many approaches, I couldn't come up with valid alternatives to what is already done in the code, and the final results in terms of efficiency are not entirely satisfactory, especially when the number of chosen topics T is augmented.
$FastLDA(3,3,3)$ might be similarly improved since $F.c$ is computed once for each Gibbs iteration, whilst it should be possible to compute it only once. However, in this case, the efficiency loss is almost negligible.

A second issue in the project is that, as mentioned above, topic assignments tend to not be matched across less important topics. Nonetheless, I am globally satisfied with the quality of results since the algorithms, on average, provide the same answer for the topic with the largest weight in each document.

Finally, my algorithms rely on an already quite polished structure for textual data, which in my applications are already well sorted into separate abstracts with a relatively uniform textual structure. An interesting addition to my project could be that of providing Julia code to draw and reorder data from more complex sources, such as unorganized web pages, to extend the applicability of the algorithms.