# ANALYSIS OF SCALABLE TOPIC MODELING IN LATENT DIRICHLET ALLOCATION

by

Richie Frost

A Senior Thesis Submitted to the Faculty of
The University of Utah
In Partial Fulfillment of the Requirements for the Degree

Bachelor of Computer Science

School of Computing
University of Utah
August 2017

Approved:

| | |
|---|---|
| _____ / _____ | _____ / _____ |
| Feifei Li<br>Supervisor | H. James de St. Germain<br>Director of Undergraduate Studies<br>School of Computing |

_____ / _____

Ross Whitaker
Director
School of Computing

**Abstract**

In natural language processing and machine learning, one of the more interesting problems to solve is determining the most relevant topics of documents (and words within those documents) with reasonable accuracy and at scale. Latent Dirichlet Allocation, also known as LDA, is one such algorithm that attempts to assign the most statistically likely topics to words and documents. Systems have been made to handle LDA for a smaller corpus of documents, but in an era when data quantity explodes at the scale of the web, more efficient methods must be used to model topics in a reasonable, practical way, even when the dataset is massive. This paper analyzes ways to scale LDA for different datasets, and examines a distributed architecture for asynchronous, non-locking LDA at scale.

# Contents

**1 Introduction** **4**

**2 Background** **6**

2.1 What is Topic Modeling? . . . . . 6

2.2 What is Latent Dirichlet Allocation? . . . . . 8

2.3 Using LDA to generate documents . . . . . 8

2.4 Why LDA? . . . . . 9

**3 LDA Inference** **13**

3.1 Markov Chain Monte Carlo Sampling . . . . . 16

3.1.1 Primer on Markov Chains . . . . . 16

3.1.2 Primer on Monte Carlo methods . . . . . 17

3.1.3 Markov Chain Monte Carlo methods - Putting it in context . . . . . 18

**4 Related Work** **19**

**5 Challenges** **22**

5.0.1 Runtime complexity . . . . . 22

5.0.2 Synchronizing globally shared $C_w$ (and $C_k$) . . . . . 22
5.0.3 Language of implementation . . . . . 24
5.0.4 Memory consumption . . . . . 24
5.0.5 Random memory access and cache misses . . . . . 25
5.0.6 Network load . . . . . 25
5.0.7 Token passing - correct worker likelihood . . . . . 26
5.0.8 Contribution 1: Nomadic Token Passing Map . . . . . 27

**6 Methods** **28**

6.1 Data Cleaning . . . . . 28
6.1.1 Lemmatization . . . . . 28
6.1.2 Part of Speech Tagging . . . . . 29
6.1.3 Stop words . . . . . 30
6.1.4 Tf-Idf . . . . . 30
6.1.5 Miscellaneous Cleaning . . . . . 31
6.2 Data Conversion . . . . . 31
6.2.1 Contribution 2: Data Structure for LDA Processing . 32
6.3 Markov Chain Monte Carlo Sampling . . . . . 32
6.3.1 Collapsed Gibbs Sampling (CGS) . . . . . 33
6.3.2 F+LDA . . . . . 34
6.3.3 Modified Fenwick (F+) Tree . . . . . 35
6.3.4 Sampling with F+LDA . . . . . 35
6.4 Distributed Architecture . . . . . 37
6.5 Load Balancing . . . . . 39

6.5.1 Contribution 3: Load balancing in a token passing architecture . . . . . 39

**7 Results** **41**

7.1 Data Cleaning . . . . . 41

7.1.1 The Process, a Lesson in Parallelization . . . . . 41

7.2 Sampler Comparison . . . . . 45

7.2.1 F+LDA vs CGS . . . . . 45

7.3 Distributed Architecture results . . . . . 49

7.3.1 Nomadic Token Passing . . . . . 49

7.3.2 Token Passing Delay . . . . . 50

7.3.3 Nomadic token passing - Excellent for memory consumption in distributed systems . . . . . 51

**8 Future Work** **52**

8.1 Porting to a faster language . . . . . 52

8.2 Online sampling . . . . . 52

8.3 Topic modeling as a service . . . . . 53

8.4 Flexibility . . . . . 53

**9 Conclusions** **54**

# Chapter 1

# Introduction

Data generated on the Internet and in business activity is a double-edged sword: the vast amount of data is only growing faster and faster, and so statistical methods of machine learning can theoretically converge to making more and more accurate predictions. However, data generated online, especially text data, is often the most prone to grammatical errors, misleading statements (such as sarcasm), emoticons, and a whole host of other issues. Therefore, even though the size of the data makes statistical machine learning seem more promising, there are still many challenges to be considered when attempting to process data generated online.

The method to be used for the topic modeling is Latent Dirichlet Allocation, or LDA. LDA uses a generative model for predicting the topics that correspond to a document, as well as the tokens in that document (individual words). Many statistical machine learning models attempt to use labeled instances of training data to teach models how to classify new information correctly. With labeled training data, these machine learning classifiers are

able to do supervised machine learning, and easily measure metrics like recall, precision, and overall accuracy. However, since labeled textual data is expensive and often small, with licensing issues creating even more obstacles, many web-scale textual machine learning is often done through unsupervised machine learning. LDA, as an unsupervised model, is an ideal algorithm for unsupervised topic modeling in web-scale datasets.

LDA is a type of hierarchical bayesian inference. Since the posterior distribution is not known, LDA has an inference step that approximates the posterior distribution. As there are many methods for the inference step of LDA, this paper aims to identify which methods perform best amongst different datasets. The key performance indicators for each method are 1) Log likelihood and 2) Elapsed time during the inference process. Log likelihood is a method used in unsupervised machine learning that approximates how closely a proposed distribution mirrors the ideal distribution, and the elapsed time metric is used for examining the runtime complexities of each method at different steps in the process. Every method to be examined in this paper uses a variant of Markov Chain Monte Carlo sampling methods, so I'll be examining the performance of several individual samplers as well as proposing some ways to improve them, with data to back up my results.

# Chapter 2

# Background

## 2.1 What is Topic Modeling?

In conversational language, topics are more familiarly known almost like semantic roles, where a topic is identified by a single word, such as "sports" or "cooking". However, topics in the sense of topic modeling are identified by a mixture words that are most likely to be found together in the wild, no matter which document they are found in. Intuitively, topic modeling is akin to the way grocery store aisles are organized. There are tens and hundreds of thousands of products in the grocery store, and as the manager of the store, you want to be able to group these products in such a way that they'll be easy to find, even if someone is coming from out of town and has never been to the store before. An aisle in a grocery store is typically identified by a number, and then related to that number are a few recognizable words that relate to products you'll find in that aisle - for example, aisle 2 may contain cookies, crackers, and chips. You likely won't find ice cream in

aisle 2, but aisle 6 may have a sign with words like ice cream, popsicles, whipped cream, and frozen berries. Thus, the manager of the store can group together similar products in batches, and most grocery stores will have similar groupings.

In topic modeling, documents are like grocery stores, words are like products, and topics are like aisles. We want to be able to distinguish which products should be grouped together, and each grocery store will have different numbers of aisles. Topic modeling often assumes a generative probabilistic model, which tries to model the kind of likelihood of grouping certain words together as well as the likelihood that each document will have such groupings. More formally, individual documents are represented as a probability distribution over topics, and topics are represented as a probability distribution over words. Since the actual probability distribution for documents, $\theta$ and the actual probability distribution for topics, $\phi$ are unknown at the beginning (when all we have is raw data), recent work has attempted to approximate the actual $\theta$ and $\phi$ distributions that would have generated the raw data. When these algorithms converge from a random proposed distribution to a close enough approximation of the actual distribution, we call this the posterior distribution. Topic modeling algorithms attempt to perform this convergence algorithm as efficiently and as accurately as possible, and Latent Dirichlet Allocation (LDA) is the most popular of them all.

## 2.2 What is Latent Dirichlet Allocation?

Latent Dirichlet Allocation, or LDA, is a generative probabilistic model used to explain how computers might interpret how documents are written by humans. But since LDA can work with any discrete, finite set of data, it's not just limited to words. So, this axiom can be expanded to include any discrete set of elements, including documents in any language. The possibilities include image analysis, machine learning features, sounds, and more. Since the individual entries in the topic-document and topic-word matrices are probability values (when using MCMC sampling to approximate the posterior distribution), with the columns of such matrices representing topic distributions and the rows representing individual discrete collections (in our case, documents), all the user of LDA needs to do is provide some kind of mapping from indices of the rows/columns to line up with the patterns they're trying to find when mapping LDA's results back to a human-readable representation. However, for our purposes, we'll use the example of textual documents, particularly documents written in English.

## 2.3 Using LDA to generate documents

First, we need to understand how LDA generates documents. Let's say we have $D$ documents, $V$ words, and $K$ topics. Let $L_d$ be the number of words in document $d$, and $w$ be a word in $V$. Each document is generated word by word, with each word being generated from a topic-word distribution, and each topic is sampled from that document's document-topic distribution. Formally, the process is as follows:

1. Draw $K$ topics $\phi_k \sim Dirichlet(\beta)$

2. For each document $d \in D$:

   - Draw the topic distribution for $d$, $\theta_d \sim Dirichlet(\alpha)$
   - For each word $w$ in document $d$, from 1 to $L_d$:
     - Draw the topic $k$ to be used from the document's topic distribution, $\theta_d$
     - Draw the word $w$ to be used from the topic $k$ distribution $\phi_k$

## 2.4 Why LDA?

In natural language processing, a collection of documents is called a corpus. To be a sure, a computer typically sees individual documents differently than humans do. While a human sees context, structure, literary meaning and the subtleties of natural language, a computer (at least as of this writing) is designed to only process numbers in binary form. Humans need to then program computers to tell them what to do and create abstractions for different kinds of logic and functionality, but even then, computers have limitations in processing power, as well as the amount of information they're able to absorb. The intuition humans have when reading a document is something we take for granted; it's much more time and resource intensive for a computer to approach that same kind of intuition and off-the-cuff understanding of natural language. To put it simply, in order to process text at the scale of the web, computers need a way to distill the information found in documents into what's essential only, and in a format it can understand. For the pur-

poses of finding latent structure and relation amongst documents, LDA only needs to know about discrete instances of data elements and the context of co-occurring elements in the same document, as well as the context of similar data elements in varying contexts in other documents in the corpus. LDA doesn't need to know about what the elements themselves actually mean, just that the elements are indexed numerically in order to distinguish them. When data is converted this way, computers have a definite advantage over humans, at least in terms of the volume of document understanding.

For our purposes, and for many previous works using LDA for topic modeling on textual corpuses [1, 2, 3, 12, 4], this data format is known as a 'bag of words' representation. In an introductory probability and statistics course, one of the first analogies used to teach the idea of probability sampling is the parable of picking marbles out of a bag. Obviously, the marbles in the bag get jumbled around and there's not really any way of telling in which order the marbles will be sitting when one reaches into the bag. The probability question, in this nascent example, is typically something like *"What is the probability of picking a red marble, if we know there are 2 red marbles, 3 green, and 4 blue?"* Order doesn't matter, and the ratio of possible marbles chosen to the total number of marbles in the bag stays the same regardless of order. For LDA to be efficient, the elements of a data set are converted to a similar 'bag' of elements, where order is irrelevant and the 'marbles' are the numerical indices representing instances of unique words in the document. For textual documents like Wikipedia articles, for example, the 'bag' is analogous to an individual article (about Sea World, for example), and every time the word 'dolphin' shows up in the

article is analogous to the proverbial red marble in our previous example. In a corpus of documents, we're essentially looking to convert our original collection of documents into a collection of bags of words. Since frequencies are relative to individual documents as well as to the corpus as a whole, we can start to intuitively see how this bag of words representation translates into individual probability values that our computers can understand. To bridge the gap between the probabilistic values that a computer sees and the plain text representation that humans intuitively understand, the bag of words representation requires a dictionary, where each word is assigned the numerical ID used in LDA that can be used as a reference later and to convert future documents into a bag of words format. This dictionary is created at the same time as the bag of words conversion for each document, and is typically stored as a Hashmap data structure for quick lookup (O(1) time).

Whether using LDA or not, the goal of the topic modeling task is to produce a probabilistic model that can make it easier to group documents together, rather than produce single semantic-level words that can define a group of documents. This model is then used to infer the topic distribution, (or grouping, to put it colloquially) of new, previously unseen documents. Topic modeling makes it straightforward to make accurate recommendations in social networks, products on Amazon, and books to read based on books you've already read, among many other applications.

A topic, strictly defined, is a probabilistic mixture of words. Topic probabilities are assigned to documents as well as to individual words, and the power of LDA is its ability to infer similar words in many different contexts.

Each topic contains a probability mixture of every word in the vocabulary, with more likely words tending to cluster around individual topics with related words.

# Chapter 3

# LDA Inference

| Term | Definition |
|---|---|
| $\alpha$ | Hyperparameter to ensure $\theta$ sparsity |
| $\beta$ | Hyperparameter to ensure $\phi$ sparsity |
| $DW$ | Document-Word frequency matrix |
| $V$ | Size of vocabulary |
| $K$ | Number of topics |
| $D$ | Number of documents |
| $\phi$ | $K$x$V$ Topic-word probability matrix |
| $\phi_k$ | Word probability distribution for topic $k$ |
| $\theta$ | $D$x$K$ Document-topic probability matrix |
| $\theta_d$ | Topic probability distribution for document $d$ |
| $C_w$ | Word-topic count matrix |
| $C_d$ | Document-topic count matrix |
| $C_k$ | Topic total count array |
| $Z$ | Topic assignment matrix |
| $W$ | Word instances matrix |
| $C_{wk}$ | Topic count for word $w$ and topic $k$ |
| $C_{dk}$ | Topic count for document $d$ and topic $k$ |

Since we don't know the actual posterior distribution of $\phi$ and $\theta$ from the beginning, we need to infer an approximation of the posterior distribution using the data we are given. This paper examines one of two major inference methods, Markov Chain Monte Carlo sampling, and several algorithms for performing MCMC. In LDA, we are given hyper-parameters $\alpha$ and $\beta$, along with our data. $\alpha$ is used to ensure that the bulk of the topic distribution for a document centers around only a small subset of topics, and $\beta$ has the same effect on word distribution in a topic. This is known as sparsity. This sparsity is what sets LDA apart from Probabilistic Latent Semantic Analysis, or pLSA [13].

The bird's eye view of LDA inference goes like this (when using MCMC sampling):

1. For each word in the corpus, assign it a random topic.

2. For $N$ number of iterations:

    - For each word in the corpus, try to assign it a better topic than the previously assigned topic (using a sampling technique).

After some number of iterations, typically less than $N$ for sufficiently large $N$, the topic assignments become relatively stable, and don't change very much. This is because, although topic assignments are random at first, the power of co-occurrence of words in a document and in similar contexts across the corpus guides better and better topic assignments throughout the inference process. Typically, when measuring the progress of LDA inference, log likelihood is measured after each of $N$ iterations to see how relatively sta-

ble the topic assignments are, and therefore how close the current proposed distribution aligns with the most likely correct posterior distribution.

## 3.1 Markov Chain Monte Carlo Sampling

When you have an unknown probability distribution that you're trying to find, denoted as $p$, it's important to either a) have a simple, reasonable way to sample from $p$, or b) have a simple, reasonable way to sample from a proportional distribution to $p$ and compare it to $p$. Markov Chain Monte Carlo sampling is a method for simulating distribution sampling from $p$ when sampling from actual $p$ is intractable or impossible.

### 3.1.1 Primer on Markov Chains

A Markov Chain is a stochastic model in probability and artificial intelligence that is used to describe a sequence of possible events in a state space by starting with a strong dependence assumption and a random state, and progressively walking events in the state space by considering the following statement to be true:

$$p(s_t|s_{t-1}, s_{t-2}, ..., s_{t-n}) \equiv p(s_t|s_{t-1}) \tag{3.1}$$

That is to say, Markov Chains choose the next state in a transition by only considering the probability of the most recent state in the chain, rather than considering the entire history of state transition probabilities. This is useful for LDA inference sampling because it doesn't require the algorithm

to maintain a history of the topic assignments that change throughout the inference process, only the most recent topic assignment. This greatly simplifies the sampling process for LDA inference.

### 3.1.2 Primer on Monte Carlo methods

Monte Carlo methods approximate $E[X]$ for some random variable $X$ by taking the average of a number of samples from a proposal distribution. The equation is as follows:

$$\frac{1}{N}\sum_{i}^{N} q(X) \tag{3.2}$$

where $N$ is the number of monte carlo simulations, $q$ is the proposal probability distribution from which to sample, and $X$ is the random variable we're sampling with. The intuition is that as $N \to \infty$, $\frac{1}{N}\sum_{i}^{N} q(X) \approx E[X]$. More Monte Carlo simulations can lead to a closer approximation of the actual expected value of $X$, but we also need to keep in mind that in LDA inference, there is an inherent dependency between words in similar contexts, so we can't just use Monte Carlo to approximate all topic assignments, since the dependency relation makes takes away any independence assumptions across topic assignments in the corpus.

### 3.1.3 Markov Chain Monte Carlo methods - Putting it in context

In LDA inference, Markov Chains are used to determine the most likely correct topic assignments, and use Monte Carlo methods to approximate the best topic assignments over the inference process. If only Markov Chains were used, convergence would be less likely, so Monte Carlo methods are applied as well so that state transitions (i.e. new topic assignments) help the entire system converge to an approximately correct posterior distribution. Rather than just looking for locally optimal topic assignment choices at each iteration, we want our topic assignments at one iteration to provide meaningful context for topic assignments at the next iteration (and for subsequent words), hence the Markov Chain dependence assumption.

# Chapter 4

# Related Work

In an effort to streamline system performance, previous efforts have focused on the sampling technique used for LDA, even combining two different techniques as a hybrid sampling technique in order to take advantage of each technique's individual strengths in particular settings. LDA*, for example, uses a combination of two different sampling techniques to account for one technique's superior runtime efficiency on a dataset given a higher number of topics, and a separate sampling technique for better runtime efficiency on datasets with longer document length [1].

Most previous efforts have made an effort to pre-process the data, by removing stop-words - words that are overly frequent and not helpful in determining topics (words like 'the', 'and', 'if', etc.), and then sending the data to be processed and classified [2]. The idea here is that the stop-words skew the probability distribution unfairly amongst words that provide little meaning other than context, so removing them creates more clarity (sparsity) for topic-word distribution. Since LDA uses a bag of words model,

context is irrelevant, so removing stop-words before LDA sampling is ok here.

Systems have been built to handle large-scale LDA processing, using an asymmetric server/worker model with shared data between machines. These have focused on minimizing the cost of sending data between machines by compressing data, removing irrelevant tokens (i.e. overly frequent words), and representing data in clever data structures before sending bits over the wire. In addition, they have had to deal with the challenge of varying document sizes, as partitioning documents efficiently is largely dependent on the size of the individual documents being processed, not the number of documents in each partition. Load balancing must take place when distributing documents to workers. Since worker load is typically dominated by document length, a master dispatcher must keep track of the number of words that each worker will be processing, and distribute the next document to the worker with the least number of total words to process in its partition of the corpus.

Although many sampling algorithms for LDA run in O(1) time, work has been done to show that there are still improvements to be made when sampling. Chen, Li, Zhu and Chen developed WarpLDA, a cache-efficient algorithm for Latent Dirichlet Allocation [3]. They found that when other systems created caches to speed up their system, there was still a slight lag in processing. Their hypothesis was that the system slowed down due to cache misses, so their efforts exploited the speed of the L3 cache, achieving running times 5-15x faster than state of the art LDA systems, as of 2016 [3]. The interesting thing about these results is that WarpLDA outperformed two different world-class sampling methods - both a Metropolis-Hastings

based method and a sparsity aware method, and did so at scale. WarpLDA harnesses cache locality at the document level.

David Blei, the original author of the Latent Dirichlet Allocation paper [12], originally proposed variational inference as a means to approximate the posterior distribution. This is a method that foregoes sampling methods entirely, and focuses on bayesian inference and expectation maximization to get results. It scales easily with multiple workers due to the fact that sharing global state is more of a non-issue (individual partitions yield localized statistics that are combined with other workers). The E (expectation) step is the most expensive step in variational inference, and is typically passed off to workers before the workers return the results, where the M (maximization) step acts as a reducer for the global state. However, there are some drawbacks with variational inference that caused me to not go forward with this approach. The total memory consumption for the LDA system is proportional to O(W), where W is the number of workers. This means that the amount of memory required when using multiple workers increases with every worker, since every worker has to use its own copy of the same memory to perform the E step. Adding more workers increases speed for sure, and variational inference is typically faster than Markov Chain Monte Carlo sampling methods like Gibbs [9] and F+LDA [4], but the memory complexity makes scaling past a certain point prohibitive. Still, for some languages (such as Python) that lack the speed needed for efficiently distributing MCMC sampling over a cluster of machines, variational inference is a fine choice, as is evident in Python's most popular distributed LDA module, Gensim [10].

# Chapter 5

# Challenges

### 5.0.1 Runtime complexity

By itself, without any optimizations, LDA inference by MCMC sampling is very computationally expensive. In a corpus, each word instance must iterate through all possible topic assignment probabilities, and there are many words and documents in even a modestly-sized corpus. This constitutes only a single iteration of the corpus, and reasonable efforts for LDA inference often include hundreds or thousands of iterations [4, 3, 1]. For this reason, runtime complexity is a challenge.

### 5.0.2 Synchronizing globally shared $C_w$ (and $C_k$)

With a large corpus, LDA inference running on a single process quickly becomes intractable. For this reason, much research has been done in the way of distributed LDA inference [4, 3, 1, 5, 7, 10]. However, the problem

isn't solved with a simple map-reduce operation, because each worker thread needs to share some global state - namely, the $C_w$ matrix. It's possible to do LDA inference with independent $C_w$ matrices for each worker, but when combining workers, there's no guarantee that the word-topic rows will line up, even if the distributions are the same. For example, in worker 1, topic 1 could have the distribution heavily favored towards words 4, 5, and 6. In worker 2, topic 3 could have the distribution heavily favored towards words 4, 5 and 6. Clearly, topic 1 in worker 1 is the same distribution that worker 2 is trying to get in topic 3, but when combining the $C_w$ matrices from workers 1 and 2, the topics get jumbled up, and the distributions suffer from drift almost from the first iteration. So, it's clear that topic 1 in worker 1 needs to match topic 1 in worker 2, and so on. Although $C_d$ matrices are independent of each other and can be stored independently per worker [4], the probability distributions from $C_w$ are highly dependent and must therefore remain as one.

For this reason, one of the biggest challenges in distributed LDA is in synchronizing global state in such a way that performance isn't hindered and $C_w$ remains up to date at each worker. Many synchronization methods have been used to maintain $C_w$ with each worker. Yahoo! LDA uses a symmetric parameter server architecture, where the parameter server stores a single copy of $C_w$ and locks on updates from every worker [6]. LDA* uses an asymmetric parameter server with $C_w$ partitioned across several servers to create separate synchronization units [1], with sampling of less frequent words done on the parameter servers as well in order to reduce network communication cost. F+NOMAD uses a nomadic token passing scheme,

where each token consists of a 2-tuple of (token index, $C_w$ row). Each row corresponds to the token index except for token 0, which is designated as the $C_k$ token (topic assignment totals) [4]. All of these methods produce varying results of implementation difficulty and overall performance, synchronizing global state just as effectively as the next. I had the most success implementing the F+NOMAD architecture, and theoretically had the most potential for parallelization. However, this leads me to the next challenge.

### 5.0.3 Language of implementation

There seems to be a tradeoff between time to implement and time to run - the programming languages that are quickest to implement are usually slowest to run, and the programming languages that are slowest to implement are usually the fastest to run. Debugging a distributed system can be tricky at best, so choosing a language that proves fairly easy to debug sometimes can take the place of choosing a faster language, which sacrifices system performance to some degree.

### 5.0.4 Memory consumption

When using a smaller corpus that fits entirely into memory, (in the order of megabytes), there isn't much need to worry about planning to scale with memory consumption. However, most companies will need to use a dataset with millions and even billions of documents, with hundreds of thousands of topics [3, 4, 5, 7, 1]. At this point, the designer of the system architecture needs to take several tradeoffs into account, such as load balancing amongst workers in a distributed system (running large corpuses in serial mode is

highly intractable), how large of a chunk of the corpus should each worker have, should the architecture be able to process in batches, should the architecture be able to handle batches? Should the system be able to handle streaming LDA inference? Most MCMC inference methods haven't done much to handle streaming LDA inference, but Y. Gao, et al presented an algorithm for streaming gibbs sampling that uses a decay factor to dampen the effect of each batch in online sampling [8].

### 5.0.5 Random memory access and cache misses

In any system with a large amount of data to be stored in memory, there are bound to be random memory accesses and cache misses. However, when the system scales, cache misses become more frequent with more random memory accesses, and this can slow down even an $O(1)$ sampler by a factor of 5-15x [3]. The challenge is to structure the sampler in such a way that memory is read more sequentially, which is no trivial task with a large $C_w$ matrix.

### 5.0.6 Network load

If the rest of the distributed system is highly performant, and each individual component is running at top speed, network load can be a bottleneck. Since each individual worker needs to synchronize with the global copy of $C_w$, there is often quite a large amount of data being sent over the wire in aggregate. Each individual network call can range anywhere from a subset of a single row of $C_w$ to all of $C_w$, and especially when updates are passed at each individual word update, such as in F+NOMAD [4], network load becomes

a high priority to optimize.

### 5.0.7 Token passing - correct worker likelihood

Specifically with the nomadic token passing scheme used in F+NOMAD and for my distributed architecture (see 6.4), I found that the likelihood of each worker receiving the token it needed became less and less when adding more workers when implementing the original F+NOMAD paper. Their architecture decides which worker to pass the token to next by uniformly sampling from $[1, W]$, where $W$ is the total number of workers. However, this led to 55% wasted network communication on a modestly sized, fairly dense corpus with only 4 workers, since 55% of the time tokens were being passed to a sampler that did not contain the corresponding word in its chunk of the corpus. The results got even worse, with a linear increase in the number of "misses" when adding more workers. The need each worker has for a token is defined by the unique words the worker processes in its chunk of the overall corpus. When more workers are added, each individual independent chunk of the corpus becomes more sparse relative to the corpus, which causes even more worker misses. I model the approximate probability of a token being passed correctly to a worker $w_{correct}$ with the following equation:

$$p(w_{correct}|t) = \frac{\frac{1}{W}\sum_{w=1}^{W} u_w}{T} \tag{5.1}$$

where $p(w_{correct}|t)$ means the worker $w$ selected can process the token $t$, $\frac{1}{W}\sum_{w=1}^{W} u_w$ is the average number of unique words per worker (with $u_w$ being the number of unique words at worker $w$), and $T$ is the total number

of tokens to be passed in the system. As the number of workers in the system increases, $\frac{1}{W}\sum_{w=1}^{W} u_w$ decreases, thereby decreasing the probability of a worker getting a token the corresponds to one of the unique words it's sampling. To solve this problem, I present my first contribution:

### 5.0.8 Contribution 1: Nomadic Token Passing Map

In my distributed system, I implemented a dispatcher that keeps track of global state by creating a map of sets, where each token is a key and the value is a set containing all of the workers that can process that token. This is used to route token passing only to workers that are able to process that token, thereby eliminating all worker misses and decreasing network load by 55%. Each worker keeps track of the number of iterations left to process that token, and when that number reaches 0 for worker $w$, $w$ is removed from the set at the dispatcher for token $t$. Once $w$ is removed from the queue for token $t$, $w'$ is unable to pass $t$ to $w$. Once the queue is empty for $t$, $t$ is passed back to the dispatcher, who marks that token's processing as complete, thereby halting all token passing for $t$ and decreasing the overall bandwidth needed for all remaining $t \in T$. Future work would be to decentralize this process, passing the queue of workers that may process this token as part of an extended version of the token structure, as a 3-tuple.

# Chapter 6

# Methods

## 6.1 Data Cleaning

To begin with, the data must be cleaned to produce any kind of reasonable results. As the saying goes, garbage in, garbage out. There are several steps to take to tailor data for effective LDA inference, and I'll outline the methods I used in the following sections.

### 6.1.1 Lemmatization

In natural language, words can take several different forms. They can be plural or singular, titlecase or lowercase, feminine or masculine, etc. Although these words have different derivations, their base form means the same thing [14]. For example, 'democratic', 'democrat' and 'democratization' all have the same base form 'democracy'. There are two ways of shortening words to get them to their base form - stemming and lemmatization. Stemming is more of a crude, quick way to chop off the ends of words to get them to look

more like their base, but isn't always correct. It's usually meant for massive data, where the amount of computation necessary prohibits any kind of morphological analysis or vocabulary use. Stemming is great for streaming algorithms when there is already a lot of data to counteract mis-transformed words, but even then, can produce semantic drift.

Lemmatization, on the other hand, is the right way to do things. Instead of using naive heuristics to chop the ends off of words, it uses morphological analysis to determine where the words need to be cut off, and uses context to determine if a word is being used as a noun or a verb, for example, and thus has a better foothold on making better decisions about how to find the base of the word. In LDA, there are already many words to consider in a corpus. But the most important thing we want to consider is what the words actually mean, and by lemmatizing words, we pare down the words to their essence. This enables us to not only considerably shorten the actual corpus for computation, but it also gives LDA inference clearer boundaries when assigning topics to words.

### 6.1.2 Part of Speech Tagging

For simplicity, and for our purposes, we are typically only interested in nouns and proper nouns when looking for topics. Verbs may sometimes come in handy, when the verb may be less common, but as a whole, nouns work better for LDA. This is because named entities, subjects, and direct/indirect objects are the most common identifiers when delineating topics. Preliminary Tf-Idf analysis shows that the most important words in the corpus are almost always nouns, so I chose to only use nouns in my datasets. For the

entire Wikipedia dataset, for example, choosing lemmatized nouns only for the corpus reduced the size of the dataset from 18 GB to 5.8 GB.

### 6.1.3 Stop words

Stop words are words that occur overly frequently in a corpus, and are therefore not useful. These include words like 'the', 'and', 'for', and 'they'. They don't provide any value for topic modeling, so they are automatically removed from the results in our corpus. These words may be part of some noun phrases in rare situations ("The Who" is a common example), but we are looking at more general patterns when topic modeling, and looking for more frequent words anyway, so removing stop words doesn't hurt the final result.

### 6.1.4 Tf-Idf

One technique that's very common when cleaning data is to use Tf-Idf values to target the most relevant words. Tf-Idf stands for term-frequency inverse document frequency, and is used to filter out which words are most representative of a document. The formula for computing a single word's Tf-Idf score is as follows.

Let $f_{t,d}$ = frequency of term $t$ in document $d$, $N = |D|$, and $D$ = the documents in the corpus.

$$tfidf(t, d, D) = tf(t, d) * idf(t, D) \tag{6.1}$$

$$tf(t,d) = f_{t,d} \tag{6.2}$$

$$idf(t,D) = log\frac{N}{|d \in D : t \in d|} \tag{6.3}$$

Inverse document frequency reflects how common a word is across the corpus, and term frequency is how common that word is in a single document $d$. The intuition is that words that are more important to a single document will have a higher Tf-Idf score, because they will be less frequent in the corpus but more frequent in that document. This heuristic works well for most applications, but we also want to make sure we only use lemmatized nouns as well with a minimum frequency so that we don't end up with characters like emoji and urls that don't really provide any meaning to the document but still might have high Tf-Idf scores.

### 6.1.5 Miscellaneous Cleaning

I also made sure to remove all non-ASCII characters, since they didn't provide any contextual clues in plain English text. URLs were also removed, as were numbers. To make the data more consistent and predictable, I also made sure to put every word in lowercase.

## 6.2 Data Conversion

Even with all the effort of paring down documents to their essence, documents still need to be converted into a format that's suitable for LDA

inference. Each word is converted to a number, and a $D$x$V$ matrix is created to store the counts of words that occur in each document, where $D$ is the number of documents and $V$ is the number of unique words. This matrix will be called $DW$

### 6.2.1 Contribution 2: Data Structure for LDA Processing

In order to make the corpus more compact, and reduce the number of random memory accesses, I used a set of 3 arrays to store the word, document, and topic assignment indices where the value at that index is nonzero, based on the optimizations in [11]. With this setup, there's no need to run several for loops, and greatly reduces the computation space - especially for highly sparse datasets. Also, every iteration in the loop is guaranteed to process only nonzero entries in each document (in other words, unique words from the vocabulary that have at least one instance in the document). This is because, when the arrays are created, they are done so by finding the row and column indices of nonzero entries in the $DW$ matrix. This not only reduces the runtime complexity, but also the memory complexity, and ensures more sequential memory accesses. These optimizations ensure a significant speed boost in each MCMC sampling iteration.

## 6.3 Markov Chain Monte Carlo Sampling

Only Markov Chain Monte Carlo samplers were used for analyzing LDA inference in this paper. In particular, Collapsed Gibbs Sampling (CGS) and F+LDA samplers were used. A summary of how each sampler works is now

presented.

### 6.3.1 Collapsed Gibbs Sampling (CGS)

Collapsed Gibbs Sampling is a very common sampling method for LDA inference with MCMC methods, due to its simplicity and high availability [4]. The general algorithm is as follows:

1. Given word $w$ in document $d$, decrease $C_{wk}$, $C_{dk}$ and $C_k$ by one.

2. Resample the topic assignment for word $w$ in document $d$ with the equation

$$P(Z_{dw}|W_{dw}, \alpha, \beta) \propto \frac{(C_{dk} + \alpha)(C_{wk} + \beta)}{C_k + \beta * V} \tag{6.4}$$

3. Increase $C_{wk}$, $C_{dk}$ and $C_k$ by one.

The first step in the resampling process is to build a cumulative probability distribution

$$\forall k \in K : p(k) = \sum_{i}^{k-1} \frac{(C_{dk} + \alpha)(C_{wk} + \beta)}{C_k + \beta * V} \tag{6.5}$$

where $p(k)$ is the value at each entry, stored in a $K$ sized array denoted as **cumsum**. Then, take a uniform sample $u \sim uniform([0, \textbf{cumsum}(argmax(k))])$, where $argmax(k)$ is the last entry in the cumulative probability distribution array, denoting the unnormalized cumulative probability $\forall k \in K$ for equation 6.4.

With $u$, perform a binary search on **cumsum** to find the index at which $u$ would be inserted to maintain sorted order in the array, i.e. the index

before the first entry $i$ where $\textbf{cumsum}[i] >= u$. The index returned from this binary search is the new topic assignment for the word at position $w$ in document $d$.

### 6.3.2 F+LDA

F+LDA sampling is an optimized version of Collapsed Gibbs Sampling that takes advantage of the fact that the approximate posterior distribution can be factored from equation 6.4 into two parts:

$$p = \frac{(C_{dk} + \alpha) * (C_{wk} + \beta)}{C_k + \beta * V} \tag{6.6}$$

$$= \beta(\frac{C_{dk} + \alpha}{C_k + \beta * V}) + C_{wk}(\frac{C_{dk} + \alpha}{C_k + \beta * V}) \tag{6.7}$$

Notice that this factorization conveniently puts the probability distribution from which to sample into two parts that can be computed independently, then added later for sampling:

$$\beta(\frac{C_{dk} + \alpha}{C_k + \beta * V}) \tag{6.8}$$

$$C_{wk}(\frac{C_{dk} + \alpha}{C_k + \beta * V}) \tag{6.9}$$

F+LDA exploits this fact by optimizing for equation 6.8, since the computation only requires $C_{dk}$, and is completely independent of computations involving $C_{wk}$. $C_{dk}$ is always dense, but its portion of the probability distribution can still be optimized by knowing that, when sampling document by document, the data structure used to store the document-topic portion of

the probability distribution only needs to be updated twice for each word - once when decrementing the current topic counts, and once when incrementing the current topic counts. F+LDA uses a modified Fenwick Tree to get sampling and maintenance of the document-topic portion of the probability distribution down to $O(logK)$. [4]

### 6.3.3 Modified Fenwick (F+) Tree

The Modified Fenwick Tree, or F+ Tree [4], is a tree that is stored as an array of length $2K$, where $K$ is the number of topics. Each entry $k \in K$ is the probability $\beta(\frac{C_{dk}+\alpha}{C_k+\beta*V})$, for the document $d$ and with a vocabulary size $V$. Each of the last $K$ entries in the array are the leaves of the tree and the actual probabilities for $k \in K$, and the first $K$ entries in the array comprise the ancestors of the leaves, where each parent node's value is the sum of all of its descendants. The root of the tree contains the total cumulative probability for $\forall k \in K$.

### 6.3.4 Sampling with F+LDA

When using F+LDA, there is a 2-level sampling paradigm. As mentioned in the previous section, the document-topic portion of the sampling process is stored in an F+ Tree. Since the probability for each $C_{wk}$ needs to be computed for every topic $k \in K$, an array structure is built and rebuilt with every word instance sampled in the corpus. The array is of size $K$, where each entry in the array is the unnormalized cumulative probability

$p(k) = \sum_{i=1}^{k-1} p(i)$. In other words, each entry in the array is only the cumulative probability of $k \in K$ up until the $k^{th}$ entry in the array. For simplicity, let $q$ be the document-topic portion of the cumulative distribution $\forall k \in K$ at word $DW_{dw}$, and $r$ be the word-topic portion of said distribution. When this array is populated, a uniform sample is then drawn

$$u \sim uniform(max(q) + max(r)) \tag{6.10}$$

where $max(q)$ is the last entry in $q$ and $max(r)$ is the last entry in $r$. Then, if $u < max(q)$, sample from $q$ using the modified Fenwick Tree. Otherwise, sample from $r$, using the $K$-dim array and a binary search. Then with the new topic $z_{new}$ as the result of that sample, assign $z_{new}$ for word $w$ at $DW_{dw}$ and increment $C_{wz_{new}}$, $C_{dz_{new}}$ and $C_{z_{new}}$.

As a summary, here's the general algorithm for F+LDA sampling (for one word $w \in DW_{dw}$:

1. Decrement $C_{wk}$, $C_{dk}$ and $C_k$
2. Build $q$, $\forall k \in K$ : Equation 6.8
3. Build $r$, $\forall k \in K$ : Equation 6.9
4. Sample $u \sim uniform(max(q) + max(r))$
    - If $u \leq max(q)$, sample $\sim q$
    - Otherwise, sample $\sim r$
    - $z_{new}$ is the sample returned

5. Assign $z_{new}$ to $Z_{dw}$

6. Increment $C_{wz_{new}}$, $C_{dz_{new}}$, $C_{z_{new}}$

## 6.4 Distributed Architecture

The architecture I chose to use was based off of the F+NOMAD [4] architecture. This decision was driven by the need to address some of the challenges in section 5, such as memory complexity per worker and synchronizing $C_w$. The F+NOMAD approach was taken originally from a matrix completion algorithm [18], and generalized for synchronizing multiple variables $C_w$ and $C_z$. Token passing also has its roots in network architecture [17].

In the F+NOMAD token passing architecture, tokens are 2-tuples that consist of an (index, $C_w$ row) pair. The token index is $w + 1$ for some indexed $w \in V$, and the $C_w$ row corresponds to word $w$. In order to synchronize $C_k$ as well, a special token with index 0 is passed, with the row containing the latest version of $C_k$. In lay terms, a token is essentially the permission a worker needs to process sampling for word $w$. Possession of a token indicates permission for a worker to sample the word corresponding to that token. This paradigm is known as "worker computes" [4]. With no locking necessary, and with the token passing done asynchronously amongst workers, there is a very high potential for parallelization.

Each worker has a job queue, where incoming tokens are pushed onto the queue and then popped for sampling. When the row contained in the token has been sampled, updates are stored in the token, and the token is then passed to the next worker who can receive it. The decision of which

worker receives the token next comes from Contribution 1, a map of sets, where each entry in the dictionary points to a set of workers that can process that token. When a worker receives the token with index 0, it computes its updates to $C_k$ and then passes it along to the next worker. This is also done with the dictionary of sets, and every worker is contained in the set until it has processed the token $N$ number of times, where $N$ is the number of iterations to process the entire corpus.

Because each document in the corpus is independent of all other documents in the corpus, each worker can have as little as 1 document in its chunk and still produce an accurate topic model for its chunk. This is largely because of the token passing architecture, knowing that every time the worker needs to compute topic assignment updates for the documents in its chunk, it only needs to receive a valid token. This is highly scalable because the memory consumption is low for each worker in terms of $C_w$, and as more workers are added, memory consumption per worker only decreases in terms of $C_d$. The token mapping paradigm outlined in contribution 1 also ensures that no network bandwidth is wasted when sharing $C_w$ amongst the workers when more workers are added, since each token pass is guaranteed to reach a worker that can process that token. Each of these workers can sample in parallel with the tokens that they possess, and as tokens are passed asynchronously on separate threads, the potential for parallelization is unrivaled.

This architecture solves problems from section 5 by decreasing memory consumption for each worker by distributing a single copy of $C_w$ and $C_k$ amongst all workers with $|w| + 1$ shards as tokens, removing the need for

locking to decrease runtime complexity and removing potential for deadlocks, making for easy system implementation, synchronizing $C_w$ and $C_k$, and reducing the number of random memory accesses by only sampling from one row of $C_w$ at a time. The only potential remaining challenge pertains to network load, which will be covered in section 7.

## 6.5 Load Balancing

### 6.5.1 Contribution 3: Load balancing in a token passing architecture

In addition to the highly scalable, asynchronous and non-locking architecture described in section 6.4, each worker needs to have a balanced workload in order to optimize overall system performance. Since infrequent words were already removed in section 6.1, we don't need to worry about the same kind of skew-aware partitioning that was done in LDA* [1]. Furthermore, since the parameter $C_w$ is decentralized, there is no need for any parameter servers to sample infrequent words [1]. This greatly simplifies the complexity of the system in terms of developer responsibility.

Let $L$ denote the length of a document, where $L_d$ is the number of words in document $d$. Research has shown that LDA inference performance is dominated by $max(L_d) \forall d \in D$. For this reason, documents are distributed amongst workers based on $L_d$, trying to get the average number of words per worker as equal as possible. The dispatcher in charge of distributing documents maintains an array of word totals per worker. Documents are processed one-by-one at system initialization, with the dispatcher assigning

subsequent documents to the worker with the lowest total number of words to be processed across each of its documents.

# Chapter 7

# Results

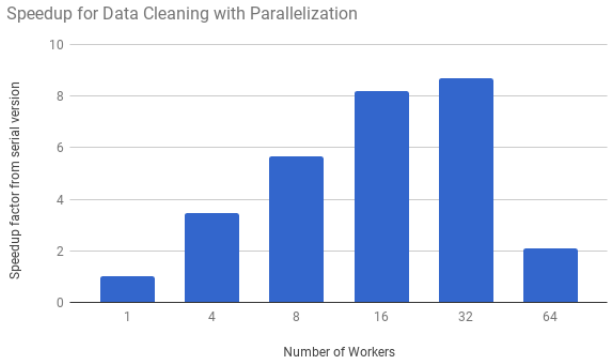
## 7.1 Data Cleaning

### 7.1.1 The Process, a Lesson in Parallelization

Starting out, the goal was to use tweets as the main data source for training LDA. However, it soon became apparent that the dataset, even with significant cleaning, was both overly sparse and overly messy. Words that weren't even words were showing up (emoji, for example, was notoriously hard to clean up), and the time it took for the data cleaning program to run was painfully slow for the amount of data retrieved. Tweets were cleaned by removing all urls, email addresses, handles, and emoji, and then using spaCy [15] to ensure that only lemmatized nouns and proper nouns were left.

However, this left each individual tweet only 3.5 words long on average, which isn't nearly long enough to gather any meaningful representation of

Figure 7.1: Speedup of the data cleaning process on the Wikipedia Dataset, using multiple workers




a vocabulary, and makes the matrices in LDA overly sparse. My initial solution was to cluster the documents based on geolocation, and that created document sizes that were significant enough for LDA. However, after running some trial runs with LDA, the models being generated produced less than desirable results in terms of their accuracy in processing new documents.

At this point, I decided that my LDA model needed to be trained using a more mature dataset. Wikipedia has publicly available data dumps of all of its articles, and various open-source software projects to extract the text from the data. So, I decided to use this dataset instead, seeing that words were generally spelled correctly (the open source community is constantly checking each other's grammar), and that the subjects were very broad - making it a great candidate for forming an all-purpose vocabulary that could be used to train LDA more intelligently. Not only that, articles in Wikipedia are conveniently typically about only one topic, with maybe a few mentions of subtopics within an article. This makes it so LDA can learn

discrete topics without having to worry about so much semantic drift.

From here, I started using Python's Gensim module [10] to extract the wikipedia data into sparse TfIdf (term-frequency inverse-document frequency) vectors, but soon found that there were a few problems with the dataset:

1. Words were not lemmatized, so the vocabulary was overstuffed with variations on the same word. The data extraction program also had a cap on the vocabulary size, so some words were left out in favor of these word variations.

2. The words returned were often still messy. For example, one of the main topics generated in LDA had one of its most salient words contain byte content, completely unusable by humans. I later found out that these byte-content tokens are metadata in Wikipedia, and therefore even less useful on training a topic model.

3. Most of the words in the dataset were non-nouns. The most common words were actually often pronouns, which are very irrelevant for topic modeling. After running LDA on this pronoun-heavy vocabulary, some of the main topics' most salient words were "you" and "he", for example. Garbage in, garbage out.

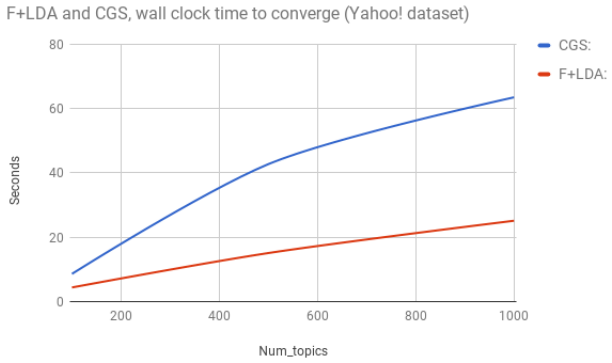
In order to clean the data more thoroughly, I decided I needed finer grained control over how the data was being processed, even if it took more time. I knew that just extracting nouns and proper nouns from documents would take days at least, let alone removing email addresses, urls, etc. I

used Python's textacy [16] module to clean the data in this way, combined with spaCy to extract nouns and proper nouns. My data cleaning program could only clean about 175 documents per minute, so after running the data cleaning program for a whole night, I only had just over 122,000 documents cleaned.

With a large dataset, unless you have the resources to put a lot of computational power towards the problem, you need to parallelize your data cleaning process in order to make it efficient. I realized my previous attempts weren't going to be fast enough, so I decided to parallelize the process with 32 workers on an 8 core server. Previous projections showed that the serial version of my data cleaner would've taken roughly 2 months to complete with the Wikipedia articles dataset, but parallelizing the process got that time to only 28 hours. Figure 1 shows the speedup results when parallelizing the data cleaning process over several workers. Since there were 8 cores used and each core had 4 processors, 32 showed to be the magic number for the number of workers to parallelize. When using 64, performance significantly decreased. This shows that the speed of the cleaning process depends heavily on the number of processors available for computation, since this is a very CPU-intensive task. Clearly 28 hours is nowhere near real-time data cleaning, but with more workers in a distributed system, this could quickly converge into a more real-time process, since individual documents are independent of each other and can be split up into a map-reduce operation in a cluster with ease.

In the end, with the Wikipedia dataset, my data cleaner processed 4,368,441 documents, yielding 869,461,874 lemmatized nouns, with 7,560,052

Figure 7.2: F+LDA vs CGS, Log Likelihood Convergence Time




unique words. This method of data cleaning reduced each document to about 25% of its original size. The resources for performing LDA at this scale were not available at the time of this thesis, so I chose to work on subsets of the data for my experiments. I chose to use only the 100,000 most frequent words in the dataset, then stripped down the dataset to only include those words that were in the top 100,000. Even though this made the dataset more reasonable for LDA, stripping out low-frequency/high cost words, the dataset was still too large for the resources available. For this reason, I chose to do experiments with the data in chunks rather than use the entire dataset.

## 7.2 Sampler Comparison

### 7.2.1 F+LDA vs CGS

One of the biggest revelations in this process was to figure out how much better F+LDA was than CGS at sampling when scaling document size and

Figure 7.3: F+LDA vs CGS, Log Likelihood Comparison on Sparse Data

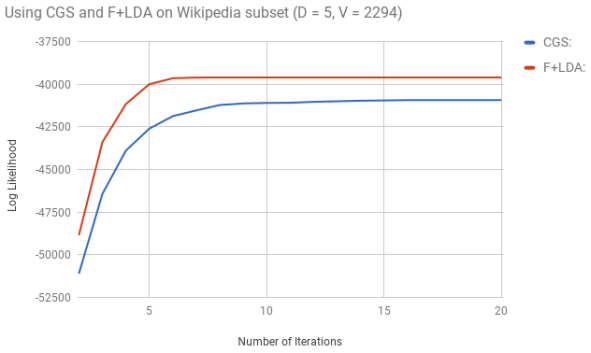
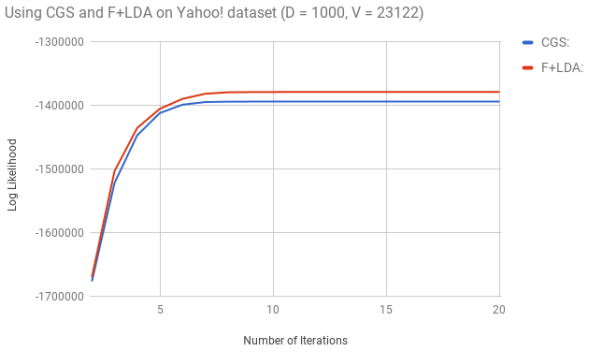



Figure 7.4: F+LDA vs CGS, Log Likelihood Comparison on Longer Corpus




number of topics - in terms of log likelihood convergence, stationary log likelihood, and clock time to converge. Figures 7.2, 7.3, and 7.4 show the results. F+LDA is faster than CGS with longer documents and larger $K$ because of the modified Fenwick Tree used to speed up sampling, as well as maintaining the distribution for each word to sample. Because the document portion of the probability distribution is stored in a tree, sample generation takes place in $O(log(K))$ time, where $K$ is the number of topics. CGS also uses a binary search variant to sample from the proposal distribution,

Figure 7.5: F+LDA vs CGS Inference Method, Wikipedia Articles (average 110 words per document)

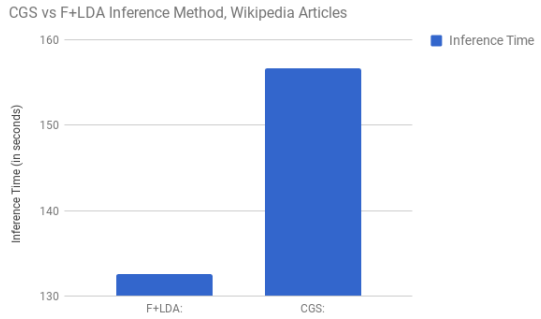
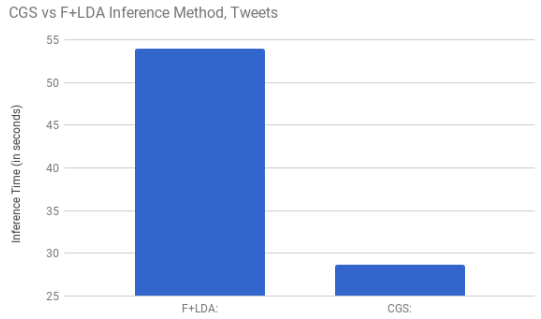


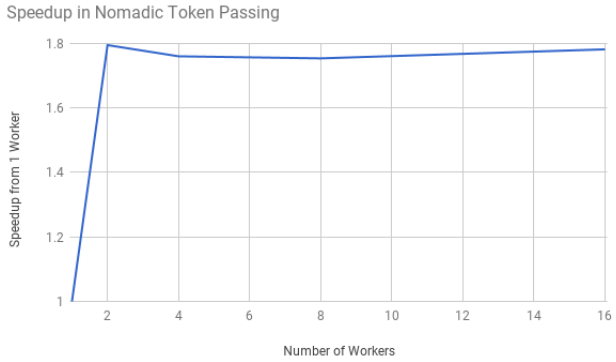

Figure 7.6: F+LDA vs CGS Inference Method, Tweets (average 10 words per document)




so it also samples in $O(log(K))$ time, but F+LDA only needs $O(log(K))$ for updates to the proposal distribution as well, whereas CGS needs $O(K)$ to rebuild the proposal distribution for each word in the corpus. This is because CGS has to reload an entire array with probability values for the next token, whereas the Fenwick Tree only needs to update every time the topic assignment changes for a word and when starting sampling on a new document. CGS needs to update the distribution array with every word iteration. Although the uniform sample doesn't always fall in the document-

portion of the proposal distribution (see the section on 2-level sampling for F+LDA), F+LDA samples more often from this distribution with shorter documents and larger $K$, and thus can save a lot of time for the overall LDA inference process. With longer documents, the sampling done for $z_{new}$ is sampled from the F+ Tree up to 18x more frequently than with shorter documents. As Figure 7.2 shows, F+LDA far outpaces CGS with larger $K$ as well. Figure 7.3 shows that F+LDA converges to a better log likelihood with a highly sparse dataset. With the dataset used for figure 7.3, the corpus is intentionally highly sparse by comparison. I took only 5 documents from the entire Wikipedia corpus, with each document being fairly different from the others. This ensured sparsity, and intuitively would mean that topic distribution would be much more sparse, since the individual articles are so different from each other. Figure 7.4 shows that CGS and F+LDA have fairly similar log likelihood values when the corpus is more dense, although F+LDA is still slightly better. This illustrates that F+LDA excels with more sparse data, whereas the results are much more similar with more dense data. Figure 7.5 shows the inference speed comparison with a specific dataset - 10,000 Wikipedia articles. Since the articles are longer, F+LDA is about 12% faster than CGS. In Figure 7.6, it's easy to tell that CGS performs much better on very short documents - in this case, a corpus of 10,000 tweets. This is because F+LDA still has some overhead when building the tree and rebuilding it with every document, so the overall runtime is much slower with F+LDA.

Figure 7.7: Speedup from 1 worker when using Nomadic Token Passing architecture

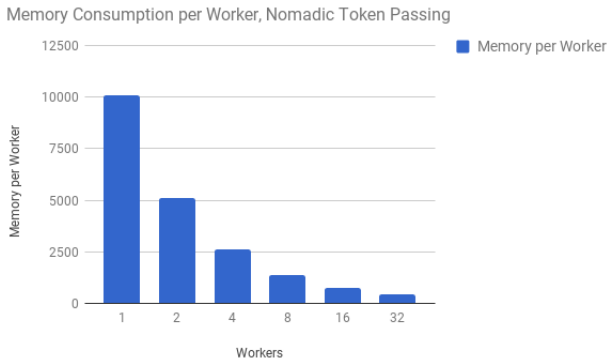



## 7.3 Distributed Architecture results

### 7.3.1 Nomadic Token Passing

Although exceptionally parallel in theory, there are many challenges with parallelizing with the nomadic token passing architecture. Python as a language is incredibly fast to implement, but much slower when running. Although certain critical parts can be written in C via a Cython wrapper, there are some parts that need to stay in pure Python (such as the dispatcher and workers), and those tend to slow down the entire system process. Although the samplers used for the distributed system were written in Cython and were thus very fast, they made it so the load on the system was something that couldn't be optimized past a certain point, and the entire system suffered, in terms of runtime complexity. Figure 7.7 shows that there is significant linear speedup when moving from 1 worker to 2 workers, but after that, the speedup levels off at around 2x, no matter how many workers are introduced. There are several reasons this happened, and they are now

Figure 7.8: Memory consumption trend per worker, Nomadic Token Passing architecture




discussed.

### 7.3.2 Token Passing Delay

Because the nomadic token passing architecture has so much data in flux between workers and the dispatcher, and because the samplers are so much faster than the rest of the pure-Python architecture, the runtime complexity of the entire system is dominated by what happens with the tokens while they are being transferred from one worker to another, and especially when maintaining state of the overall system. Both with using a central dispatcher to maintain global state as well as decentralizing the global state amongst workers, the data is the same - the underlying language to be used is the underlying bottleneck of the system. One solution would be to rewrite the samplers in pure python to show more of a speedup in the overall system when more workers are added, but that would make the system even slower, and only serve to show that the architecture works, which has already been

proven in [4]. The goal of using token passing was to parallelize the entire system asynchronously and without locking, but I failed to realize how much the choice of language would affect the overall running time of the system. Python is slower because there are no static types and there is a global interpreter lock (GIL) that make Python great for building prototypes and for making single-threaded applications, but when the Python interpreter needs to infer type checking in a distributed architecture where milliseconds matter, that adds significant lag to the overall system when run at scale.

### 7.3.3 Nomadic token passing - Excellent for memory consumption in distributed systems

One advantage that I consistently found with the nomadic token passing architecture was that the memory consumption was very minimal, and the amount of memory needed at each worker is $O(K + d_w)$, where $K$ is the number of topics and $d_w$ is the number of documents assigned to each worker $d$. The good news is, the memory gets even better for each worker as more workers are added, since each worker only takes a chunk of the entire corpus $|d_w| = \frac{D}{W}$, where $D$ is the total number of documents and $W$ is the total number of workers. Figure 7.8 shows that memory consumption decreases linearly with the number of workers added.

# Chapter 8

# Future Work

## 8.1 Porting to a faster language

For the future, this project needs to be ported from Python/Cython to C++, or at least Java, in order to circumvent the slowness of Python. This would improve the network lag experienced in this implementation, and it'd be easier to see how much faster the system would perform with the addition of more workers.

## 8.2 Online sampling

A major improvement to this system would be to enable online sampling, or to transition to variational bayes for easier online inference. This would be very helpful in situations where data is streaming into a system, and the model needs to be built based on continuous data rather than ad hoc requests.

## 8.3 Topic modeling as a service

To make this system industrial strength, this should be made into topic modeling as a service. Right now, the workers and dispatcher cease once the algorithm completes, since this is primarily a system to observe tradeoffs in different datasets when considering LDA inference. An improved version of this would perform well under any kind of dataset like in [1], and be able to produce meaningful $\phi$ and $\theta$ by writing them to disk or providing them as downloadable bytes, perhaps as a flat file, or even storing them in a database. Ideally, this would be some kind of RESTful API, or for more speed and security, RPC connections could be used.

## 8.4 Flexibility

One of the other drawbacks of the F+NOMAD architecture being implemented in Python, using Pyro python remote objects, is that outside systems can't directly connect to the LDA inference system unless they are also written in Python. Changing the communication framework to something RPC based, like gRPC, makes the system flexible enough to accept communication from any other program that can use the same RPC framework. This is ideal when connecting with HDFS, for example, and as a component in Spark, since gRPC can talk to programs written in Java as well.

# Chapter 9

# Conclusions

In this paper, I examined the inference process for LDA using MCMC sampling, and explored some methods used to optimize for sparse datasets vs denser datasets. F+LDA is better for sampling more sparse datasets, longer documents and when $K$ is large, due to its $O(logN)$ modified Fenwick tree sampler, as well as being able to update the proposal distribution in $O(logN)$ time. Collapsed Gibbs Sampling is better for more dense datasets, since the frequency in sampling from $C_d$ and $C_w$ centric distributions is roughly the same.

I also reviewed the F+NOMAD distributed architecture for LDA inference, using token passing combined with F+LDA sampling. There are performance considerations to keep in mind when choosing a language in which to implement the architecture, and the choice of language largely determines the speed of the overall system.

Future work aims to optimize this parallelized architecture by porting it to a faster, compiled language like C++, and making the system flexible

enough to act as a service, rather than being geared towards individual ad hoc tasks.

# Bibliography

[1] L. Yu, C. Zhang, Y. Shao, and B. Cui. LDA*: A Robust and Large-Scale Topic Modeling System.

[2] N. Ebrahimi and F. Li. Research Interest Detection From Grad Admission Database: Text Classification at Scale.

[3] J. Chen, K. Li, J. Zhu, and W. Chen. WarpLDA: a Cache Efficient O(1) Algorithm for Latent Dirichlet Allocation.

[4] H.-F. Yu, C.-J. Hsieh, H. Yun, S. Vishwanathan, and I.S.Dhillon. A scalable asynchronous distributed algorithm for topic modeling. In WWW, pages 1340-1350. ACM, 2015.

[5] J. Yuan, F. Gao, Q. Ho, W. Dai, J. Wei, X. Zheng, E.P. Xing, T.-Y. Liu, and W.-Y. Ma. Lightlda: Big topic models on modest computer clusters. In WWW, pages 1351-1361. ACM, 2015.

[6] A. J. Smola and S. Narayanamurthy. An architecture for parallel topic models. In Proceedings of the VLDB, 2010.

[7] A. Q. Li, A. Ahmed, S. Ravi, and A. J. Smola. Reducing the sampling complexity of topic models. In SIGKDD, pages 891-900. ACM, 2014.

[8] Y. Gao, J. Chen, and J. Zhu. Streaming Gibbs Sampling for LDA Model

[9] Gibbs Sampling, Wikipedia `https://en.wikipedia.org/wiki/Gibbs_sampling`

[10] Radim Řehůřek and Petr Sojka Software Framework for Topic Modelling with Large Corpora. In Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks, pages 45-50. ELRA, 2010

[11] Topic modeling with Latent Dirichlet allocation using Gibbs sampling `https://github.com/lda-project/lda`

[12] D. Blei, A. Ng, and M. Jordan. Latent Dirichlet Allocation. Journal of Machine Learning Research, 3:993-1022, January 2003.

[13] T. Hofmann Probabilistic Latent Semantic Analysis. In Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence. Pages 289-296, August 01, 1999.

[14] Stemming and Lemmatization Stanford University `https://nlp.stanford.edu/IR-book/html/htmledition/stemming-and-lemmatization-1.html`

[15] spaCy: Industrial-Strength Natural Language Processing `https://spacy.io/`

[16] textacy: Higher-level NLP built on spaCy `https://github.com/chartbeat-labs/textacy`

[17] Littlejohn Shinder, Debra (2001). Computer Networking Essentials. Cisco Press. p. 123. ISBN 978-1587130380. Retrieved 31 July 2017.

[18] H. Yun, H.-F. Yu, C.-J. Hsieh, S. V. N. Vishwanathan, and I. S. Dhillon. Nomad: Non-locking, stochastic multi-machine algorithm for asynchronous and decentralized matrix completion. CoRR, abs/1312.0193, 2013.