RICE UNIVERSITY

Large-Scale Online Aggregation Via Distributed Systems

by

Niketan Pansare

A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE

Doctor of Philosophy

APPROVED, THESIS COMMITTEE:

Christopher Jermaine, Chair
Associate Professor of Computer Science

Luay Nakhleh
Associate Professor of Computer Science

Erzsébet Merényi
Research Professor of Statistics, and
Electrical and Computer Engineering

Houston, Texas
July, 2014
ABSTRACT

Large-Scale Online Aggregation Via Distributed Systems

by

Niketan Pansare

To deal with huge amounts of data, many analysts use MapReduce, a software framework that parallelizes computations across a compute cluster. However, MapReduce is sometimes still not fast enough to perform the complicated analysis. In this thesis, I address this problem by developing a statistical estimation framework on top of MapReduce to provide for interactive data analysis. I present three projects that I have worked on under this topic.

My approach is based on Online Aggregation, a method that allows the user to compute an arbitrary aggregation function over the dataset and output probabilistic bounds in an online fashion. However, implementing Online Aggregation on MapReduce is non-trivial as classical sampling theory suffers from “inspection paradox”, which states that at any random time the block that takes longer to process is likely to be sampled. Since there is usually a correlation between processing time and output value from a data block, classical sampling theory will output biased estimates. Therefore, in the first project of my thesis, I propose a bayesian model that addresses the inspection paradox in MapReduce setting and outputs unbiased online estimates for the given aggregation function.

In the second project, I focus on applying OLA techniques to gradient descent, an optimization algorithm that finds the local minima of a function \( L(\Theta) \) by starting with an initial point \( \Theta_0 \) and then taking steps in direction of negative gradient of the function to
be optimized. Since the gradient descent algorithm is essentially an user-defined aggregate function, the OLA framework developed in the first part of my thesis can be used to speed up this algorithm in a MapReduce framework. The key technical question that must be answered is “When do we stop the OLA estimation for a given step (or epoch)?”. In this thesis, I propose and evaluate a new statistical model for addressing this question.

As feature selection is an important step in machine learning, my third project in this thesis focuses on building better topic models, a popular feature selection technique, for audio data by taking into account the inherent uncertainty of speech recognizer.
Dedicated to my parents
Namrata and Rambhau Pansare
for their unconditional love
and countless sacrifices.
Acknowledgments

With great humility, I would like to acknowledge the immense debt I owe to my thesis advisor and my mentor, Christopher Jermaine. He not only provided exceptional guidance in my research, but also helped me through turmoils of graduate life. I am extremely grateful for the amount of time and effort he invested in teaching me bayesian concepts, distributed systems and most importantly communication skills. This thesis would not have been possible without his encouragement, patience and understanding.

I would like to thank my uncle (Dilip Pansare) and my aunt (Suvarna Pansare) for giving me the necessary push to pursue higher education. Whole-hearted thanks to the most wonderful sister, Namita Pansare for willingly agreeing to proof-read endless pages of this thesis and my research papers. I am thankful to my beautiful wife Swati Ghule for her selfless love and understanding; especially for supporting me through the toughest of times. Also, I would like to thank my naughty kid brother Saurabh Pansare and my cute sister Anagha Pansare for their love and encouragement. I cannot thank enough my best friend and my brother, Rohan Kumar Singh, for being my harshest critic and my fiercest champion. I am also indebted to Y.K. uncle, for treating me like a son and especially for forcing me to continue further studies. I would like to extend special thanks to my in-laws, especially my parents-in-law, Shashikant and Mangal Ghule, for welcoming me with open arms and trusting me with your precious daughter/sister.

I would also like to convey my gratitude to Luay Nakhleh and Erzsébet Merényi for being in my committee and especially for providing constructive comments which helped improve the chapters 2 and 4. My deepest thanks goes to Peter Haas for providing valuable suggestions throughout my PhD.

I would like to thank Ravi Jampani, Kishore Malani, Parag Parekh, Alin Dobra, Keith
Cooper and Devika Subramanian for their invaluable advices on research and productivity. I would also acknowledge the support from Nitendra Rajput, Tyson Condie, Vinayak Borkar, Vivek Sarkar, Moshe Vardi, Alan Cox, Eugene Ng and extremely hardworking admin staff of Computer Science department. I would like to express my sincere gratitude to Sumedh Warudkar, Pooja Warudkar, Allwyn Carvallo, Nakul Jindal and Nikhil Limaye for their lasting friendship. I am especially grateful to my fellow labmates Zhuhua Cai, Anna Drummond, Risa Meyers, Arkabandhu Chowdhury, Letao Qi, Rahul Kumar and Shangyu Luo for their honest opinion and constructive criticism during my talks. I will forever cherish the friendship I made with all my colleagues at Rice University and University of Florida. Special thanks goes out to Deepak Majeti, Lung Li, Swati (Gambhir and Srivastav), Suchi Jain, Kuldeep Meel, Karthik Murthy and Shams Imam for their support during my PhD. I appreciate Faraz Makari, Punit Pania and Shaunak De taking time to proof-read my thesis.

I would like to thank my advisor, NSF and IBM for providing me with the necessary funding that made my PhD research possible.
Contents

Abstract ii
Acknowledgments v
List of Illustrations xi
List of Tables xiii

1 Introduction 1

1.1 Overview of Research Problems ................................. 2
  1.1.1 Online Aggregation for Large MapReduce Jobs ............... 2
  1.1.2 Distributed Gradient Descent Using Online Aggregation .... 5
  1.1.3 Topic Models Over Spoken Language .......................... 8

1.2 Specific Contributions ............................................ 11
  1.2.1 Online Aggregation for Large MapReduce Jobs ............... 11
  1.2.2 Distributed Gradient Descent Using Online Aggregation .... 11
  1.2.3 Topic Models Over Spoken Language .......................... 11

1.3 Outline of the thesis ............................................. 12

2 Background and Literature Overview 14

2.1 Online Aggregation ................................................. 14
2.2 MapReduce .......................................................... 15
2.3 Bayesian Machine Learning ........................................ 17
2.4 Gradient Descent .................................................. 22
2.5 Topic Models ....................................................... 26

3 Online Aggregation for Large MapReduce Jobs 28
3.1 Introduction ................................................. 28
  3.1.1 Online Aggregation for Large-Scale Computing .... 30
3.2 An Operational Model for OLA ......................... 31
  3.2.1 Why an Operational Model? ......................... 31
  3.2.2 Our Model for OLA Over MapReduce ................. 32
  3.2.3 Taking a Snapshot .................................. 33
  3.2.4 Why this model? Why these statistics? .............. 35
3.3 Implementing the Model ................................. 37
3.4 Estimation ................................................. 39
  3.4.1 Overview ............................................ 39
  3.4.2 Generative process .................................. 40
  3.4.3 Prior Distributions .................................. 43
  3.4.4 Posterior Distribution ............................... 44
  3.4.5 Putting It All Together .............................. 46
3.5 Experiments ................................................ 47
  3.5.1 Experiment One ...................................... 47
  3.5.2 Experiment Two ...................................... 52
3.6 OLA Programming Interface ............................ 54
  3.6.1 Map, Reduce and Combiner Contracts ................. 55
  3.6.2 Specifying the aggregate ............................ 56
  3.6.3 Configuration parameters ............................ 58
  3.6.4 EstimateReceiver interface .......................... 59
  3.6.5 Example ............................................. 60

4 Distributed Gradient Descent Using Online Aggregation 65
  4.1 Introduction ............................................. 65
  4.2 OLA-GD Overview ...................................... 69
  4.3 Background: OLA ....................................... 70
4.3.1 Online Aggregation ................................................. 70
4.3.2 Distributed OLA: Why is it difficult? ......................... 71
4.3.3 A Bayesian Model for OLA ................................. 72
4.3.4 OLA for GD: Challenges ........................................ 75
4.4 Modifying OLA for GD ........................................... 76
  4.4.1 Simplifying the Model ....................................... 76
  4.4.2 Estimating the Model Parameters .......................... 79
4.5 Ending an Epoch .................................................... 81
  4.5.1 Overview ...................................................... 81
  4.5.2 Estimating the Velocity .................................... 81
  4.5.3 Modeling the GD Walk .................................... 83
  4.5.4 Stopping Rules .............................................. 85
4.6 Hadoop Implementation ........................................... 89
  4.6.1 The Coordinator ............................................. 89
  4.6.2 The Mapper .................................................. 90
  4.6.3 The OLA-GD Server ...................................... 91
  4.6.4 The Model .................................................... 91
4.7 Gibbs Sampling Details .......................................... 92
  4.7.1 OLA Sampler ................................................. 92
  4.7.2 Sampler for GD Walk Model ............................... 96
4.8 Experimental Evaluation ......................................... 98
  4.8.1 Methods and Models Tested ............................... 98
  4.8.2 Data, Experimental Platform, Results .................... 100
  4.8.3 Discussion .................................................. 101
4.9 Conclusions ......................................................... 105

5 Topic Models Over Spoken Language 111
  5.1 Introduction ..................................................... 111
5.1.1 Topics Over Spoken Language ........................................ 112
5.1.2 Moving Beyond Text .................................................. 113
5.1.3 Taking Into Account Transcription Errors ...................... 113
5.2 The Spoken Topic Model .................................................... 114
  5.2.1 A Brief Introduction to LDA ......................................... 115
  5.2.2 The STM Generative Process ...................................... 115
5.3 To Speech And Back Again ............................................... 118
5.4 Inference ........................................................................ 121
  5.4.1 Markov Chain Monte Carlo .......................................... 121
  5.4.2 The Joint Density Function ......................................... 122
  5.4.3 Update Samplers ....................................................... 123
5.5 Experiments ................................................................... 125
  5.5.1 Experimental Setup .................................................... 126
  5.5.2 Discussion ............................................................... 133

6 Conclusions ...................................................................... 136

A Notation .......................................................................... 137

Bibliography ....................................................................... 139
Illustrations

3.1 Posterior query result distribution for number of Wikipedia page hits over the English language, at various query completion percentages, using both a randomized and arbitrary block ordering. The actual query result is a vertical black line. 48

3.2 Identical to above figure, except for the French language. 49

3.3 Posterior query result language for the French distribution, at various query completion percentages, taking into account and ignoring correlation between aggregate value and processing time. 50

4.1 Two statistical models used in determining the distance traveled toward $\Theta^*$ in epoch $i + 1$. The first is the OLA error model itself, which is used to obtain a distribution over $\epsilon_{i+1} = \Theta_{i+1} - \hat{\Theta}_{i+1}$. The second is a model for $\Delta_{i+1} = \Theta^* - \Theta_{i+1}$, the distance to the optimal solution. The figure shows three different samples from both of the models. The use of these samples to estimate the expected distance traveled is depicted in Figure 4.2. 106

4.2 Using samples from the two models depicted in Figure 4.1 to obtain a posterior distribution on the distance traveled toward $\Theta^*$ in epoch $i + 1$. In each case, $\Theta^*$ is computed as $\hat{\Theta}_{i+1} + \epsilon_{i+1} + \Delta_{i+1}$. From $\Theta^*$ we compute $d_i$, the distance to $\Theta^*$ at the end of the last epoch, and $d_{i+1}$, the distance at the end of the current epoch. The distance traveled toward $\Theta^*$ in epoch $i + 1$ is then $d_i - d_{i+1}$. In this example, the expected distance traveled is $(1.2 + 1.2 + 1.4)/3 = 1.27$. 107
4.3  Expected distance traveled to $\Theta^*$ as a function of the standard deviation of $
abla_i \epsilon_{i+1}$ in an example scenario. .................................................. 108
4.4  Expected distance traveled to $\Theta^*$ as a function of time in an example scenario. .................................................. 108
4.5  Loss versus time for the various learning task/GD framework combinations. 109
4.6  Effect of data skew with (and without) correlation in the OLA model. . . . 110

5.1  Average accuracy for synthetic data ................................................. 128
5.2  Average accuracy for TedTalks data .................................................. 130
5.3  Average accuracy for Yale data (2 classes) ....................................... 131
5.4  Average accuracy for Yale data (4 classes) ....................................... 132
Tables

3.1 Fraction of confidence bounds that are “incorrect”........................................ 54
3.2 Mapper, Reducer, Combiner class interfaces..................................................... 57
Chapter 1

Introduction

In this age of big data, every company is looking into ways to apply machine learning algorithms on their massive datasets. However, the problem is that this takes a significant amount of time due to the sheer volume of the data. To deal with huge amounts of data, many analysts use MapReduce [1], a software framework that parallelizes computations across a compute cluster. However, due to the sheer volume of data, MapReduce is sometimes still not fast enough to perform complicated analyses. One obvious way to address this problem is by making the MapReduce framework faster or by using faster hardware. Another way is to develop a statistical framework on top of MapReduce to provide for the interactive data analysis. In this thesis, I focus on latter approach to large-scale data analysis.

As part of my thesis, I will present three projects that I have worked on, under this topic:

1. Online aggregation for large MapReduce jobs

2. Distributed gradient descent using online aggregation

3. Topic models over spoken language

In each of the three projects, I have:

1. Developed a novel statistical model.
2. Implemented the changes to the underlying system necessary to support the above statistical model.

3. Validated the system implementation as well as the statistical model through experiments using synthetic and real data sets.

1.1 Overview of Research Problems

1.1.1 Online Aggregation for Large MapReduce Jobs

Introduction to Problem Domain

Online aggregation (OLA) allows a system to compute an arbitrary aggregation function over a data set and output probabilistic bounds on accuracy in online fashion. For example, consider the case where an aggregation function computes the average of employee salary over the Rice university database, and assume the answer is $40,000. After scanning over a small fraction of the data set, an OLA system might output the range [35000, 45000] with 95% confidence. This estimate means, that the final answer lies between 35000 and 45000 with probability 0.95. As the system processes more data, it can improve the estimate. For example, after a minute, it might output [38000, 41000], subsequently after two minutes [39500, 40050] and so on. This not only improves the interactivity of the system, but also provides the user with the option of stopping the aggregate query if the acceptable accuracy is reached early.

Though OLA has had significant scientific impact [2, 3, 4], its commercial impact has been limited. In my view, there are two main reasons for the limited impact. First, implementing OLA in relational database requires extensive changes to database kernel; in particular it requires random ordering of the blocks (or tuples). This would affect the implementation of several database optimization techniques (for example, indexing) which
many database vendors are not inclined to modify. Second, saving time (computer or human) at the expense of a little uncertainty has never been a pressing need for the database industry. Stopping a query early might save some CPU cycles and bandwidth for other users, but would not directly benefit the primary user who must then accept some uncertainty. Also, since the major cost in a classical database installation is the fixed cost associated with the maintenance of hardware and software, the cost of a given query does not decrease drastically with decreasing running time.

In recent years, however, many organizations have adopted MapReduce-based software frameworks (such as Hadoop) that run in the cloud rather than relational databases that run on top of self-managed systems. The, “Why stop early ?” argument is harder to make in the cloud since the cost of executing a query is proportional to the running time in cloud. Using OLA, the user can leverage the pay-as-you-go model of cloud services to their advantage. Also, the “We cannot change the kernel” argument is becoming less applicable when people are implementing all sorts of data oriented systems from scratch using MapReduce based software frameworks, for example, Hyracks [5], Spark [6], SimSQL [7], Hive [8], DryadLINQ [9], Shark [10], GraphLab [11], HAWQ [12], YARN [13], Blink [14], HaLoop [15], etc.

**Key Technical Challenge**

OLA in a relational database system [16] uses classical sampling theory to estimate the confidence bounds. The assumption is that the estimate is produced by analyzing an independently and identically distributed (iid) sample from a distribution, i.e. \( \{x_i \overset{iid}{\sim} f(.)\} \) where \( i \in \{1, \ldots, N\} \). OLA infers various characteristics of \( f(.) \) from the set \( \{x_1, \ldots, x_N\} \) and outputs confidence bounds based on them.

The key difference in a large-scale distributed computing environment, compared to a
classical relational database is the importance of block-based processing. Data in a distributed system are organized into atomic blocks. Since the block size is typically 64MB or larger, each block can contain millions of tuples, which results in non-negligible processing time for a block. Due to the possible need of moving the block to the compute location as well as heterogeneity of the cluster, there is a significantly large variability in the processing time across different blocks. This variation is important because one might expect that blocks that take longer to process have bigger aggregate values. For example, in case of a count query, the aggregate value of a block depends on the number of tuples, which is often directly proportional to the processing time. Thus, at a random time instance, the system is likely processing blocks that take longer to process. This statistical phenomenon is known as the inspection paradox of renewal reward theory [17].

Thus, blocks that take longer to process are less likely to be taken into account when an estimate is generated. Since one might expect correlation between processing time and the aggregated value of a block, the estimates for the aggregate can be biased. This behaviour was observed experimentally in a previous paper [18]. Therefore, we need to deal with the inspection paradox in a principled fashion so as to obtain robust estimates.

Overview of the Approach

To address the inspection paradox, I propose a Bayesian model that utilizes a joint prior over the values \( x_i \) and time take to process/schedule the block \( i \). This prior is represented by \( f(x_i, t_i^{sch}, t_i^{proc}) \) where \( t_i^{sch} \) and \( t_i^{proc} \) represent the scheduling time and the processing time of the block \( i \) respectively. Thus, if a block has taken 5 seconds to be scheduled and has been processing for the last 999 seconds, then the posterior for the aggregate value \( x_i \) for that block is given by the distribution \( f(x_i|t_i^{sch} = 5, t_i^{proc} \geq 999) \). Since the model is taking timing information into account, the bias due to the inspection paradox is neutralized.
This model is implemented on Hyracks [5], an open-source project that supports map and reduce functions. The architecture of Hyracks is similar to Hadoop, the most popular implementation of MapReduce. Like Hadoop, Hyracks has a single master for submitting jobs and can interface with Hadoop Distributed File System (HDFS) [19]. To support OLA, the queue containing the list of blocks at master is shuffled at the start of the job. To schedule a new map task, master picks the block that is the head of this queue rather than using locality-based scheduling [20]. The output of the map task consists of two files: the meta file that contains the timing information and locality information, and the data file that contains the list of groups as well as values for each of those groups. These two files are available to all the reducers, which may invoke bayesian estimator when data from a new block is received. The bayesian estimator (which implements the above model) is implemented in C++ using GNU Scientific Library and Minuit2.

1.1.2 Distributed Gradient Descent Using Online Aggregation

Introduction to Problem Domain

Gradient descent [21, 22, 23] is an optimization algorithm that finds the local minima of a function $L(\Theta)$ by starting with an initial point $\Theta_0$ and then taking steps in the direction of negative gradient of the function to be optimized (i.e. $-\nabla L(\Theta)$). The computation of the gradient is referred to as epoch and gradient descent computes many epochs iteratively until completion. In the field of machine learning, the function $L(\Theta)$ is called as loss function and $\Theta$ is the model parameter. The goal here is to learn the optimal model parameter $\Theta^*$, so that one can use the associated target function of the machine learning model $g(x, \Theta^*)$ to perform predictions over test data. The above steps of the (batch) gradient descent
algorithm can be represented in the form of the following recursive equation:

$$\Theta_{k+1} = \Theta_k - \frac{1}{N} \sum_{n=1}^{N} \nabla L(y_n, g(x_n; \Theta_k))$$  \hspace{1cm} (1.1)

In the above equation, $\Theta_k$ denotes the parameter at $k^{th}$ epoch and $\nabla L(y_n, g(x_n; \Theta_k))$ represents the gradient computed using $n^{th}$ datapoint in $k^{th}$ epoch.

**Key Technical Challenge**

If the number of data points $N$ is very large it can take lot of time to compute the aggregate function $\frac{1}{N} \sum_{n=1}^{N} \nabla L(y_n, g(x_n; \Theta_k))$ for an epoch $k$. Since the gradient descent algorithm is essentially a user-defined aggregate function [24], the OLA framework developed in the first part of my thesis can be used to speed up this algorithm in a MapReduce framework. The key technical question that must be answered is “When do we stop the OLA estimation for a given epoch?”. This is a difficult question to answer, because the performance of an OLA-based gradient descent algorithm depends on both the properties of the loss surface, as well as the number of data points available during estimation, so a simple rule of thumb might give poor performance. For example, one can use a heuristic such as “stop if the standard deviation of the posterior of estimates of the aggregate function is less than a certain arbitrary threshold $\delta$”. However, there are couple of issues with this heuristic. First, the performance of the gradient descent would depend heavily on $\delta$, which is chosen arbitrarily. If $\delta$ is too small, then the algorithm would behave similarly to batch gradient descent, and if it is too large, then it would behave similar to stochastic gradient descent (SGD) [25, 26]. Second, the heuristic treats every epoch in the same way and hence ignores the underlying loss surface completely. This might be suboptimal especially if the loss surface is not well behaved. Therefore, a more principled approach is necessary to decide when to stop the OLA estimation for a given epoch.
Overview of the Approach

Gradient descent is implemented in MapReduce iteratively by starting a new job per epoch. Since there is an initialization and setup cost associated with beginning each MapReduce job, the rate* towards the optimal parameter is zero during the setup phase. The rate then increases quickly as the first data are processed. However, since the error in OLA estimates decreases linearly with the square root of the number of data points available during estimation, intuitively there should exist an optimal rate after which it is better to start with a new epoch than to continue with the next iteration of the current epoch. Therefore, it makes sense to stop when the rate of current iteration towards $\Theta^*$ falls below the average rate of current epoch towards $\Theta^*$, indicating that waiting longer for a more accurate OLA estimation is not efficient. It is better to start a new gradient descent epoch at that instant.

The key difficulty is that the optimal parameter $\Theta^*$ is not known. Precise estimation of $\Theta^*$ would require modeling the loss surface [27] which is a more difficult problem than the goal of gradient descent itself (i.e., finding the optimal parameter $\Theta^*$). So instead I choose to develop a simple model for the random walk of the gradient descent algorithm itself, and to use the model to produce the posterior of the optimal parameter $\Theta^*$. Even if this posterior is a crude approximation, it still provides us an idea of when to stop an epoch in a principled fashion. Given this random walk model, we use the posterior over the optimal parameter $\Theta^*$ to predict the average rate towards $\Theta^*$ of the epoch and also rate of current iteration towards $\Theta^*$ to decide whether to start a new gradient descent epoch.

*The rate that current epoch travels towards an optimal parameter $\Theta^*$ is defined as distance travelled towards $\Theta^*$ with respect to previous parameter divided by the time taken for that epoch.
1.1.3 Topic Models Over Spoken Language

Introduction to Problem Domain

The previous problems assumed that number of dimensions $D$ of the data points is manageable (in the order of hundreds to thousands), but the number of datapoints $N$ is extremely large. However, in text mining, $D$ is equal to the size of vocabulary (i.e. number of distinct words in the corpus), which is very large. To deal with this issue, a feature selection (or dimensionality reduction) methodology is often used. An extremely popular feature selection methodology is topic modeling [28]. A topic is defined as a probability distribution over the sets of words or phrases and each document in the corpus is drawn from the mixture of these topics. A topic model for a corpus specifies the set of topics, as well as the proportion in which they are present in any given document. For example, if a topic model were learned over a set of machine learning papers published in the last few years, one of the learned topics might put high probability on sets of words having to do with learning over graphs ("edge", "vertex", "SVD", "diameter"), one might favor the sets of words having to do with kernel methods ("SVM", "Gaussian processes", "hyperplane"), one might favor words associated with text processing ("LDA", "topic", "TF-IDF"), and so on. The topic model might further identify that 80% of the words in a specified document come from the graph topic, 10% from the kernel topic, and 10% from the text topic.

There are many good reasons for learning a topic model over a corpus. Often, topic modeling is used as a methodology for feature selection. A topic model is used to map each of the corpus’ documents to a feature vector in “topic space”, where the $i$th entry in the vector corresponds to the importance of the $i$th learned topic in the document. Then, the important learning tasks—document classification, clustering, outlier detection, recommendation, and so on—can be performed on the resulting, topic-based representations of
the documents. In this way, topic modeling serves as an alternative to other vector-based representations, such as mapping each dictionary word to a dimension and using the word’s TF-IDF value to position the document in the resulting vector space. In practice, performing learning tasks in topic space works well because the position of the document in topic space is semantically meaningful; documents that are close together presumably cover the same subject matter.

**Key Technical Challenge**

The recent interest in topic models has been driven by the explosion of electronic, text-based data that are available for analysis. From web pages to emails to microblogs, text-based data are everywhere. However, not all electronically-available natural language corpora are text-based. In this work, we consider the problem of learning topic models over spoken language.

Our work is motivated by our involvement with the *Spoken Web* (also called the *World Wide Telecomm Web*), which allows users in the rural India to post farming-related questions and responses to an audio forum using mobile phones [29, 30, 31]. More conventional, text-based social media are not convenient for Spoken Web users for many reasons, from lack of literacy to lack of SmartPhone or Internet access. We would like to learn topic models over users’ audio posts, so that we can subsequently perform learning tasks on the data.

Given the potential usefulness of topic modeling for spoken language, it is natural to ask: Are there significant technical challenges as we move beyond electronic text, or is it a simple matter of applying a speech-to-text software to the audio document, and then learning a standard topic model over the resulting document? I argue that the situation may not be simple at all. The difficulty is that even modern speech-to-text software is notori-
ously inaccurate, especially if recordings are made under non-optimal circumstances. Such inaccuracy can have an adverse effect on the quality of a topic model that is learned. The situation is particularly poor for non-English-language speech. In our motivating application to the Spoken Web, the spoken text we wish to analyze are in Gujarati, one of the dialect spoken in Indian.

Overview of the Approach

In this work, I propose a new topic model that leverages the statistical algorithms used in the most modern speech-to-text software. These algorithms typically consider all possible paths through a lattice [32], where each node in the lattice is a possible transcription of a spoken phrase. The most likely path is returned as the final transcription. It is easy to modify such a software to return not a single transcription but rather, for each phrase, a weighted set of possible transcriptions instead of just the most likely one. For example, imagine that a document from an audio corpus contains the sentence, “The cow jumped over the moon.” The speech-to-text software would break the file into a sequence of phrases. Each phrase is represented by a weighted set of possible values. For example, we might have:

\[
\Psi_1 = \{(0.4, \text{“bow wow”}), (0.2, \text{“the cow”}), (0.4, \text{“it’s now”})\} \\
\Psi_2 = \{(0.2, \text{“land rover”}), (0.2, \text{“lawnmower”}), (0.6, \text{“jumped over”})\} \\
\Psi_3 = \{(0.45, \text{“the moon”}), (0.45, \text{“too soon”}), (0.1, \text{“cocoon”})\}
\]

My idea is to exploit the additional information contained in such \(\Psi\) vectors. Specifically, we develop an alternative version of the popular LDA [33] topic model called the spoken topic model, or STM for short. This model uses a Bayesian interpretation of each \(\Psi\) vector that takes into account the vector’s explicit uncertainty description (the phrases and weights) in a principled fashion.
1.2 Specific Contributions

In this section, I list the specific contribution of this thesis.

1.2.1 Online Aggregation for Large MapReduce Jobs

For this problem, I:

1. Proposed a system model that is appropriate for OLA over MapReduce for large-scale problems.
2. Described in detail how the model is implemented in Hyracks [34], an open source project that supports map and reduce operations.
3. Discussed a Bayesian framework for producing estimates and confidence bounds within our model.
4. Offered experimental evidence that the OLA model produces an accurate and usable estimates very quickly.

1.2.2 Distributed Gradient Descent Using Online Aggregation

For this problem, I:

1. Extended the OLA model proposed in previous work [35] to support multidimensional aggregate function.
2. Proposed a novel bayesian model that predicts the optimal parameter of the gradient descent \( \Theta^* \) during OLA estimation. This parameter is then used to decide whether to stop an epoch of gradient descent.
3. Implemented necessary system changes on Hadoop [36], an open source MapReduce based project similar to Hyracks, to support the above mentioned models.

4. Offered experimental evidence that the OLA model in conjunction with the random walk model improves the performance of gradient descent on MapReduce.

1.2.3 Topic Models Over Spoken Language

For this problem, I:

1. Developed an alternative version of the popular LDA [33] topic model called the spoken topic model, or STM for short.

2. Derived a Markov Chain Monte Carlo algorithm that learns this spoken topic model from an audio document corpus.

3. Showed that the resulting model learns a more useful set of topics than those learned by simply applying standard learning methods to the final output of the speech-to-text software.

1.3 Outline of the thesis

In the remainder of the thesis I discuss the three projects that I worked on during my PhD studies in detail. Chapter 2 of the thesis covers the background and the related works. Chapter 3 contains my work in OLA over MapReduce. This chapter was originally published as “N. Pansare, V. R. Borkar, C. Jermaine, and T. Condie, “Online aggregation for large mapreduce jobs,” Proceedings of the VLDB Endowment, PVLDB vol. 4, no. 11, pp. 11351145, 2011.” [35]. Chapter 4 describes my work in distributed gradient descent using OLA. Chapter 5 includes my work on topic models for spoken language. Much of this
chapter was published as “N. Pansare, C. Jermaine, P. Haas, and N. Rajput, Topic models over spoken language, in *Proceedings of the 2012 IEEE 12th International Conference on Data Mining, ICDM 12, pp. 10621067.*” [37]. Chapter 6 concludes the thesis.
Chapter 2

Background and Literature Overview

In this chapter, I will review the related work in five key areas, namely Online Aggregation (section 2.1), MapReduce (section 2.2), Bayesian Machine Learning (section 2.3), Gradient Descent (section 2.4) and Topic Models (section 2.5).

2.1 Online Aggregation

Online Aggregation (OLA) has been studied for some time in the context of classic SQL databases [38, 39, 40, 41], and more recently for peer-to-peer systems [42]. But, in context of MapReduce, the only work that considers OLA (without ignoring MapReduce’s open programmability and fault tolerance) is the Hadoop Online Prototype (HOP) system [43]. HOP added a second execution plan to the Hadoop architecture that allowed pipelining in the map and reduce Hadoop operators [43]. Pipelining operators is a prerequisite to any OLA query plan, and a pipelining plan was shown to perform better in cases involving short jobs. HOP supported OLA queries by executing reduce tasks at data dependent intervals e.g., on 10%, 20%,...90% of the data. The query estimate assumed a uniform sample of the input data but made no modifications to enforce this in the Hadoop scheduler. This led to significant error in their estimates. To compensate, the authors modified the query to contain extra parameters that indicated how many samples of a particular aggregate group were present, and scaled the estimate accordingly.

OLA falls under the broad area of Approximate Query Processing, where the system
tries to give early (and possibly approximate) answers to the query. An alternative to OLA is precomputed synopsis, where the system uses summary statistics (computed prior to the execution of the query) to provide approximate answers [44, 45].

Other than using bayesian posterior as described in chapter 3, few of the widely used approaches to find confidence bounds are Central Limit Theorem [46], Chebyshev bounds, Hoeffding bounds [47] and Chernoff bounds [48]. All of these approaches are based on sampling [49], where the confidence bounds are computed using the statistics over a random block from the dataset. Central Limit Theorem assumes an underlying distribution for the error, whereas other bounds make no such assumption and hence are distribution-free bounds. Unlike Chebyshev bounds that requires estimate of the variance over the sample, Hoeffding and Chernoff bounds use maximum and minimum value over the dataset. Chernoff bounds work on binary data and hence are not as general as Hoeffding bounds. Since most database systems maintain statistics like maximum and minimum value over the data, Hoeffding bounds are very useful for performing OLA.

2.2 MapReduce

MapReduce is a programming model popularized by Google [1], for performing aggregate computations over large datasets. In 2005, Doug Cutting and Mike Cafarella created an open-source implementation of MapReduce framework called Hadoop [36]. The key features that made this framework so popular were its simple programming interface (map and reduce function), scalability and fault-tolerance. It is also important to note that unlike relational database, MapReduce does not require the data to be formatted in a fixed schema before processing. However, it is often criticized for its poor performance [50, 51] and the lack of support for the declarative language SQL. Several variants of MapReduce have been proposed to address this issue [52, 7, 8, 9, 10].
In the MapReduce framework, the programmer specifies a map function that processes input records and produces a list of intermediate key/value pairs, and a reduce function that is called once for each distinct map output key and the associated list of intermediate values. Optionally, the programmer can supply a combiner function, which is applied to the intermediate results between the map and reduce steps. The combiner interface is similar to reduce functions. Combiners are typically used to perform pre-aggregation, which can reduce the amount of network traffic when the map and reduce steps are executed in a distributed environment.

The MapReduce architecture consists of a query processing layer that is based on a data flow of map and reduce operations, and a Distributed File System (DFS) [53], which stores the input to the map function and the output of the reduce function. The intermediate data is typically stored on the local file system. The query processing layer consists of a single master node and many worker nodes [1]. The master is responsible for accepting jobs that specify the user-defined functions and automatically parallelizing those functions into units of work called tasks. Each task is assigned a portion of the relevant input and is individually scheduled on the worker nodes. Worker nodes are assigned a fixed number of slots for executing tasks (e.g., two maps and two reduces). A heartbeat protocol between each worker and the master is used to update the masters’ bookkeeping state of running tasks, and drive the scheduling of new tasks: if the master identifies a free worker slot, it will schedule further task on the worker.

A map task is assigned a portion of the input file called a split. By default, a split corresponds to a single DFS block (64MB by default), so typically the total number of file blocks determines the number of map tasks. The execution of a map task is divided into two phases. The map phase reads the assigned split from the DFS, parses it into records (key/value pairs), and applies the map function to each record. After the map function has
been applied to each input record, the *commit* phase executes the combiner function (if given) and registers the final output with the worker, which will then inform the master that the task has finished executing.

The execution of a reduce task is divided into three phases. In the *shuffle* phase, the reduce task receives its assigned key range from the output of each map task. This phase typically runs concurrently with the map tasks in a pipelined fashion. After receiving its partitions from all map tasks, the reduce task enters the *group* phase. This is commonly performed via sorting techniques. Finally, the *reduce* phase invokes the user-defined reduce function for each distinct key and associated list of values. The final output of the reduce function is written to the DFS and the master is informed on completion. In this design, the output of both map and reduce tasks is written to the disk before it can be consumed. This is particularly expensive for reduce tasks, because their output is written to DFS. Output materialization simplifies fault tolerance, because it reduces the amount of state that must be restored to consistency after a node failure. If any task (either map or reduce) fails, the master simply schedules a new task to perform the same work as the failed task. Since a task never exports any data other than its final answer, no further recovery steps are needed.

### 2.3 Bayesian Machine Learning

Given the data \( \{(x_1, y_1), \ldots, (x_n, y_n)\} \), the learning algorithm \( A \) picks a hypothesis \( g : x \rightarrow y \) that has *minimum risk*:

\[
g^* = \arg \min_g R(g) \\
= \arg \min_g \mathbb{E}[L(y, g(x))] 
\]
Note, the risk function \( R(g) \) is the expected loss of \( g \), and user specifies the loss function\(^*\) (also called as error measure) \( L : y \times y \rightarrow [0, +\infty) \) based on either mathematical convenience or application need. Here is the list of commonly used loss functions based on mathematical convenience:

<table>
<thead>
<tr>
<th>Used for</th>
<th>Name of the loss function</th>
<th>( L(g(x), y) = )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Regression</td>
<td>Squared loss</td>
<td>((g(x) - y)^2)</td>
</tr>
<tr>
<td></td>
<td>Absolute value loss</td>
<td>(</td>
</tr>
<tr>
<td></td>
<td>(\epsilon)-insensitive loss</td>
<td>(\max{</td>
</tr>
<tr>
<td>Classification</td>
<td>Squared loss</td>
<td>((g(x) - y)^2 = (1 - g(x)y)^2)</td>
</tr>
<tr>
<td></td>
<td>Hinge loss</td>
<td>(\max{1 - g(x)y, 0})</td>
</tr>
<tr>
<td></td>
<td>Logistic loss</td>
<td>((\ln 2)^{-1}\ln(1 + e^{-g(x)y}))</td>
</tr>
<tr>
<td></td>
<td>0/1 loss function</td>
<td>(\mathbb{I}(g(x) \neq y))</td>
</tr>
</tbody>
</table>

Similarly, you can define the loss function based on an application need. For example, as in the case of homeland security fingerprinting, false accepts are extremely expensive but false rejects are acceptable as the person can try the scanner again. So, in this case, the loss function can be described in the form of the following table:

<table>
<thead>
<tr>
<th>( g(x) )</th>
<th>( y )</th>
</tr>
</thead>
<tbody>
<tr>
<td>+1</td>
<td>+1</td>
</tr>
<tr>
<td>-1</td>
<td>99999</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( y )</th>
<th>+1</th>
<th>-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>+1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>-1</td>
<td>99999</td>
<td>0</td>
</tr>
</tbody>
</table>

The key idea of the above framework is that even if we dont know exactly what the function \( g^* \) looks like, we can still learn it by first defining the model we want to learn (i.e. restricting ourselves to a family of functions \( \mathcal{F} \)) and then minimizing the expected

---

\(^*\)An alternative to loss function is scoring function \( f : x \times y \rightarrow \mathbb{R} \) where \( g \) is the function that returns \( Y \) that gives the highest score.
empirical loss:

\[ g^* \approx \arg \min_{g \in \mathcal{F}} \mathbb{E}[L(g; \{ (x_1, y_1), \ldots, (x_N, y_N) \})] \]  

(2.1)

To simplify the mathematics, the data is assumed to be independent and identically distributed (i.e. \( iid \) assumption). Hence the above equation can be written in a much simpler form:

\[ g^* \approx \arg \min_{g \in \mathcal{F}} \frac{1}{N} \sum_{n=1}^{N} L(Y_n, g(X_n)) \]  

(2.2)

In this section, we will restrict ourselves to a parametric family described by parameter \( \Theta \), i.e. \( g = g(x_n, \Theta) \). So, the learning problem can now be described as follows:

\[ \Theta^* \approx \arg \min_{\Theta} \frac{1}{N} \sum_{n=1}^{N} L(y_n, g(x_n; \Theta)) \]  

(2.3)

One way to find the optimal parameter \( \Theta^* \) is by using an optimization technique called \textit{gradient descent}. The gradient descent finds the local minima of a function by starting with an initial point \( \Theta_0 \) and then taking steps in direction of negative gradient of the function to be optimized.
Algorithm 1: Gradient Descent

**Input:** Initial parameter $\Theta_0$, learning rate $\eta$, data $\{(x_1, y_1), \ldots, (x_N, y_N)\}$

**Output:** The optimal parameter $\Theta^*$

$k = 0$

while true do

    $\nabla L = 0$

    foreach training datapoint $n$ do

        $\nabla L = \nabla L + \nabla L(y_n, g(x_n; \Theta_k))$

    $\Theta_{k+1} = \Theta_k - \frac{\eta}{N} \nabla L$

    $k = k + 1$

    if $\Theta_k$ is converged then

        $\Theta^* = \Theta_k$

        Return $\Theta^*$

The above algorithm can be represented in form of following recursive equation and will be discussed in detail later:

$$\Theta_{k+1} = \Theta_k - \frac{\eta}{N} \sum_{n=1}^{N} \nabla L(y, g(x_n; \Theta_k))$$ \hspace{1cm} (2.4)

Once the optimal parameter $\Theta^*$ is determined, a new test example $x_{test}$ can be classified using the following formula:

$$y_{test} = g(x_{test}; \Theta^*)$$ \hspace{1cm} (2.5)

Another common approach to machine learning is by using probabilistic modeling. In probabilistic modeling, instead of focussing on the loss function, you start by positing a parametric model family with a set of parameters $\theta$. Based on the problem requirements, one of the two types of modeling approaches can be chosen: generative or discriminative [54, 55]. In generative model, the optimal parameter $\theta^*$ is chosen by maximizing (or
integrating over) the joint distribution:

\[ P(\theta, \{(x_1, y_1), \ldots, (x_N, y_N)\}) = P(\theta) \prod_{n=1}^{N} P(x_n, y_n | \theta) \]  (2.6)

Whereas, in a discriminative model, the optimal parameter \( \theta^* \) is chosen by maximizing (or integrating over) the conditional distribution:

\[ P(y, \theta | x) = P(\theta) \prod_{n=1}^{N} P(y_n | x_n, \theta) \]  (2.7)

Once the optimal parameter \( \theta^* \) is found using either of these two approaches, a new test example \( x_{test} \) can be classified by maximizing over the posterior:

\[ y_{test} = \arg \max_y P(y | x_{test}, \theta) \]  (2.8)

It is important to note that “maximizing or integrating over” the joint distribution is intractable in many cases. So, in most cases, the joint distribution is estimated by either using variational approximation [56] or by Markov Chain Monte Carlo (MCMC) methods [57, 58]. In variational approximation, the complicated posterior is approximated with a much simpler distribution by making more independence assumptions. Expectation propagation [59], belief propagation [60] and mean field approximation [61] are all examples of variational approximation methods. On the other hand, MCMC methods constructs a Markov chain and then uses the sampling based methods to learn the posterior. Two popular MCMC based methods are Metropolis-Hasting algorithm [62] and Gibbs Sampling [63]. Gibbs sampling learns the posterior by sequential sampling of all the conditional distributions of the posterior. The Gibbs sampler operates by continually maintaining candidate values for each of the variables to be inferred. The sampler repeatedly updates each of the variables for a sufficient number of iterations, until the chain has “mixed”. After this period, referred to as the “burn in,” one can take a snapshot of all of the current values to obtain a sample from the true posterior distribution.
2.4 Gradient Descent

Gradient descent is a first order optimization algorithm that finds a minima of a function and can be expressed in the form of a following recursive formula:

\[
\Theta_{k+1} = \Theta_k - \frac{1}{N} \sum_{n=1}^{N} \nabla L(Y, g(X_n; \Theta_k))
\]  

(2.9)

Since in most practical cases, only subset of the data is available to the gradient descent algorithm, the gradient is multiplied by a learning rate \(\eta\):

\[
\Theta_{k+1} = \Theta_k - \frac{\eta}{N} \sum_{n=1}^{N} \nabla L(Y, g(X_n; \Theta_k))
\]  

(2.10)

This algorithm is guaranteed to converge to the optimal value at the rate \(O\left(\frac{1}{k}\right)\) if the function \(g\) is convex and \(L\)-Lipschitz continuous with constant \(L\) using the step size \(\eta = \frac{1}{L}\). Nesterov [64] improved the convergence of this algorithm to \(O\left(\frac{1}{k^2}\right)\). This improved algorithm is referred to as the accelerated gradient method.

To improve the performance, the convergence rate \(O\left(\frac{1}{k^2}\right)\) (i.e. number of iterations to converge to optimal value) is sacrificed for computational speed by using the gradient of loss over a smaller subset of the data. The variant of gradient descent that uses a single random data point is called stochastic gradient descent (SGD) [21, 22, 23]. It is a special case of stochastic approximation algorithm proposed by Robbins and Monro [26]. For stochastic gradient descent, the learning rate \(\eta_k\) is decreased using the formula \(\eta_k = \frac{\eta_0}{1 + \gamma k}\). The effect of the hyper parameters \(\eta_0\) and \(\gamma\) was studied by Xu [65] and Bach [66]. Several works [67, 68, 69] proposed adapting the learning rate based on dimensions rather than using the above heuristic formula. If instead of a single data point, a random block is used to estimate the gradient, the algorithm is called as mini-batch gradient descent [70].

In case the gradient cannot be computed, finite difference [21] can be used as an approximation of gradient. In 2003, Spall [25] proposed an improvement over finite differ-
ence method called as simultaneous perturbation approximation (SPSA) which perturbs all
the coordinates along a random direction.

In 2011, Niu et al [24] showed that the recursive form of gradient descent can be ex-
pressed in form of an aggregate query in relational database. An empirical study by Niu
et al [71] claimed that this aggregate query can be performed without the need for global
coordination. Thus, the recursive equation 2.10 can be expanded as follows:

\[ \Theta_{k+1} = \Theta_k - \frac{\eta}{N} \sum_{n=1}^{N} \nabla L(Y, g(X_n; \Theta_k)) \]  \hspace{1cm} (2.11)

\[ \Theta_k \frac{\eta}{N} \sum_{j=1}^{k} \sum_{n=1}^{N} \nabla L(Y, g(X_n; \Theta_n)) \] \hspace{1cm} ... expanding the recursion (2.12)

\[ \Theta_0 - \frac{\eta}{n} \sum_{n=1}^{N} \sum_{j=1}^{k} \nabla L(Y, g(X_n; \Theta_j)) \] \hspace{1cm} ... switching the summation (2.13)

An important variant of gradient descent called incremental gradient descent [72] can
be derived by replacing the parameter \( \Theta_j \) inside the summation by its stochastic estimate
\( v_j \) in the equation 2.13.

\[ \Theta_{k+1} = \Theta_k - \frac{\eta_k}{N} \sum_{n=1}^{N} \nabla L(Y, g(X_n; v_n)) \]  \hspace{1cm} (2.14)

where \( v_0 = \Theta_k \)

and \( v_i = v_{i-1} - \eta_k \nabla L(Y, g(X_n; v_{n-1})) \)

Like incremental gradient descent, there are several variants of gradient descent [73]
that maintain a set of iterates \( v_i \). Since it is difficult to express these variants in the form
of an aggregate query that can be massively parallelized (i.e. work on MapReduce), we
will ignore them in our discussion. In their paper, Niu et al [24] focused only on stochastic
gradient descent and hence expressed gradient descent in form an aggregate query over
multiple iterations (i.e. \( \sum_{j=1}^{k} \)). Though this might work for a relational database that resides
on a single machine, it cannot be massively parallelized and hence will not perform well on MapReduce framework. A discussion of more traditional distributed gradient descent algorithms can be found in [74, Ch. 12]; these algorithms are primarily targeted for environments where evaluation of each term $x_i$ is very expensive—e.g., a lengthy stochastic simulation may be required—and communication is cheap. This is typically not the case in big-data environments such as Hadoop, Spark, and so on.

Hogwild! [75] is a framework for running parallel SGD on a single, multi-processor machine, or perhaps on a very closely linked cluster of machines where shared memory is a reasonable model. The Hogwild! developers argue that in such an environment, the right way to run GD is to run a single SGD calculation where each of the processors update a single version of the model, stored in RAM, without locks and without regard to race conditions. As long as the data are sparse so that the updates to the model are quite localized, the errors accumulated by races that do happen are shown both empirically and theoretically to be negligible. While Hogwild! should be considered an obvious choice for its intended environment, it does not extend naturally to a distributed setting.

Vowpal Wabbit (VW) [76], which is perhaps the best-known software package for large-scale GD. The VW system runs a distributed SGD, where models are computed independently at each location, and then averaged periodically. As we have argued in this paper, this is not a robust approach when the optimization problem is not convex. VW’s primary contribution to the state of the art in distributed GD is actually orthogonal to the subject of this paper; namely, VW implements a very efficient all-reduce\(^{\dagger}\) to average all of the models maintained across the cluster very quickly. In the VW Hadoop implementation, the various mappers build a spanning tree which is used outside of the context of Hadoop MapReduce.

\(^{\dagger}\)All-reduce has long been an important operation in the lore of parallel programming [77]. It is, for example, supported by MPI via the \texttt{MPI\_Allreduce} operation.
to average the models. We note that our own implementation of OLA-GD could make use of this all-reduce to achieve better speed, or, conversely, there is no reason that OLA-GD could not be plugged into a software such as VW to allow for use with non-convex models.

Additional methods for distributed optimization exist, but they tend to be either more involved to use, requiring a lot of problem-specific effort, or they are suitable primarily for convex models, or both. Consider the alternating direction method of multipliers (ADMM) [78]. ADMM can be viewed as a generalization of the averaging-based method we tested in the experimental section, in that heterogeneous optimization methods can be used by the different optimization processes and the method of combining the results is more sophisticated. ADMM is also related alternating minimization methods such as alternating least squares [79]. ADMM is quite popular, but in our experience it performs poorly on non-convex problems, and is not easy to use out of the box.

An important alternative to gradient descent is the conjugate gradient method [80, 81] due to its iterative nature. Unlike gradient descent with constant learning rate, conjugate gradient method does not have performance degradation when the curvature of the loss function is very small or very large. The conjugate gradient method also performs better than approximated second-order optimization algorithm [82] like L-BFGS because computing conjugate information is computationally less expensive than estimating hessian especially for very large datasets [83]. Bunch et al. [84] implemented conjugate gradient method on MapReduce using three primitives: DAXPY, DDOT and MatVec where DAXPY computes $\alpha x + y$, DDOT computes a dot product and MatVec multiplies a matrix by a vector. Their implementation resulted in six MapReduce jobs per iteration of conjugate gradient and also significant communication overhead for even small matrices.
2.5 Topic Models

In text mining, topic modeling is a very useful feature selection technique which attempts to find “latent structure” (also referred to as topics or concepts) in a text document. These topics simplify a lot of machine learning tasks due to their intuitive representation as well as due to their low dimensionality. Earlier work in topic modeling were referred to as Latent Semantic Analysis (LSA) [85]. In 1999, Hofmann introduced a probabilistic version of latent semantic analysis called Probabilistic Semantic Indexing (PLSI) [86], where the document is modeled as a mixture of topics. In 2003, Blei introduced Latent Dirichlet Allocation (LDA) [87], which unlike PLSI, is a generative hierarchical bayesian model. The effectiveness of topics to represent the document for learning task have made LDA an extremely popular model for text mining. Since then several variants of LDA have been proposed. Dynamic topic models [88] learn how the topics evolve over time. Unlike LDA, Correlated topic models [89] assume correlation between topic proportions. Rather than treating s a document as bag-of-words, Structured Topic Model [90] models the documents as sequences of structures like paragraphs or sentences. My work on “Spoken Topic Models” involves extending Latent Dirchlet Allocation by taking into account the uncertainties that arise in speech-to-text transcriptions and output topics of better quality.

The existing work that is closest is work in the speech recognition area that focuses on segmenting streams of speech into coherent topics. This problem is long-studied [91, 92], though only a few papers make use of information regarding potential errors [93, 94]. Recent work has applied probabilistic LSA to modeling a spoken corpus [95], though no work that we are aware of combines an LDA-style generative model with an explicit representation of uncertainty.

The theory of reasoning under uncertainty has a long history (see Halpern [96] for a classic reference). Learning with label noise, where labels can be incorrect, has long
been studied in machine learning and there are many classic papers in the area [97, 98]. Learning in the case where several possible labels are explicitly given is a much more sparsely studied problem. See Cour et al. [99] for an example of such a methodology.
Chapter 3

Online Aggregation for Large MapReduce Jobs

3.1 Introduction

When running online aggregation (OLA) [38, 39, 40], at all times during query processing, a database system gives a user a statistically-valid estimate for the final answer to an aggregate query, along with confidence bounds of the form: “with probability $p$, the actual query answer is within the range low to high”. As the computation progresses, the bounds narrow, until (at query completion) the bounds are zero-width, indicating complete accuracy.

The main benefit of OLA is that if an acceptably accurate answer can be arrived at very quickly (perhaps in a tiny fraction of the time needed to run the entire query), the query can be aborted, saving significant computer and human time.

Although OLA arguably has had quite a bit of scientific impact (stimulating significant subsequent research), its commercial impact has been limited or even non-existent. In our view, there have been two main reasons for this lack of adoption:

1. First, implementing OLA within a database engine would likely require extensive changes to the database kernel. OLA requires some sort of statistically quantifiable randomness within the database engine. Most OLA algorithms require that the blocks (or tuples) in a relation be processed using a “random” ordering, where “random” has a very stringent mathematical definition. Since this would require significant changes to most kernels and would wreak havoc with techniques widely-implemented by database vendors (such as indexing), vendors and kernel developers
have justifiably viewed OLA with suspicion.

2. Second, the goal of saving human and computer time has never been as compelling as one might think. A user of an analytic database who writes a query that goes into a queue and finally makes it out into a big, production warehouse for evaluation has little motivation to kill the query early, even if the user is relatively happy with the results. Ending the query early might save some CPU cycles or disk bandwidth that can then be used by others, but the user who killed the query early may not benefit directly. Furthermore, the database hardware/software/maintenance costs in a self-managed system are not elastic, and do not decrease appreciably if many users decide to stop their queries early.

Significantly, I feel that these two impediments to widespread adoption of OLA may have become less important over time. The “We can’t change the kernel” argument is less important at a time when people are implementing all sorts of new databases or data-oriented systems from scratch, particularly for large-scale, shared nothing cluster environments. The “Why stop early?” argument is also harder to make nowadays, given the current move into the cloud. When someone other than the end-user’s organization is managing the compute infrastructure, as a query runs, dollars are quantifiable flowing from the end-user’s organization and into the cloud. Now that there may be a real and observable cost associated with every CPU cycle consumed and byte transferred, the end-user will likely have to justify those costs to the management. It stands to reason that being able to achieve 99% of the accuracy in 10% of the time will become much more attractive under such a cost model. Thus, I feel that OLA is an old idea whose time has come.
3.1.1 Online Aggregation for Large-Scale Computing

Given the potential for OLA to be newly relevant, and given the current interest on very large-scale, data-oriented computing, in this chapter I consider the problem of providing OLA in a shared-nothing environment. While I concentrate on implementing OLA on top of a MapReduce engine [100], many of our most basic research contributions are not specific to MapReduce, and should apply broadly.

Realizing OLA for large-scale, distributed computing is a challenging problem, and a simple extension to the classic work on OLA will not suffice. Classic work in OLA assumes that blocks or tuples are processed in a statistically random fashion, so that the set of data seen at any point in the computation is a random subset of the data in the system—if this is the case, then it is often easy to estimate the final answer using classic methods from survey (finite population) sampling theory [101]. The difference in a large-scale, distributed computing environment is the importance of elapsed time. In this type of environment, the basic unit of data that is processed is a block, which may contain millions of tuples and be a significant fraction of a gigabyte in size. When many machines are working in parallel, it is natural that there would be a lot of variation in the time taken to process each block. Some blocks could have a lot of data, and take longer to process. It is not unusual for machines to simply die, so they appear as if they have been processing a block forever. This variation in processing time is of tremendous importance if it is somehow correlated with the aggregate value of the block. Such correlation is to be expected: after all, blocks with a lot of data may have greater aggregate values, and take longer to process. In such a scenario, since those nodes that are processing large blocks with big aggregates are more likely to spend more time on those blocks, the set of blocks that have actually completed processing at any particular point are more likely to have small values, leading to biased estimates. This is an example of the well-known “inspection paradox” described by the renewal-reward theory.
[102]. Dealing with this in a principled fashion in a distributed environment is challenging, and requires innovation both in system design and in statistical analysis.

The remainder of this chapter describes a MapReduce OLA library, implemented on top of Hyracks, for providing estimates for the aggregates: \( \text{SUM} \), \( \text{COUNT} \), \( \text{AVG} \), \( \text{VARIANCE} \), or \( \text{STD} \_\text{DEV} \). To use this library, the user writes a Hadoop-style map-reduce job (that conforms to the programming interface discussed in the section 3.6) along with a class that is called-back by the MapReduce OLA library, every time an estimate is generated. Typically, this class can serve as an input to other map-reduce jobs or it can also pipe the estimates to a GUI front-end, thus providing a user interface similar to the classical online aggregation systems [38]. It is important to note that the programmer can halt the job when the estimation has reached a satisfactory level of confidence, à la CONTROL [41].

3.2 An Operational Model for OLA

3.2.1 Why an Operational Model?

The first step to getting OLA to work in a distributed, MapReduce environment is to define the abstract, operational model of the system. This model defines how data are processed in the system, and serves as a contract between the system implementors and the statistical analysis that underlies the OLA estimation process.

This operational model must meet two requirements:

1. First, the model must be amenable to statistical analysis. That is, at any point during the computation, it must be possible to take a snapshot of the system and to use that snapshot to predict the final output of the MapReduce program.

2. Second, the model must be amenable to implementation. It should impose little or no overhead on the system, so that there is little additional cost associated with
running a MapReduce OLA program, compared to a non-OLA MapReduce program. It should allow the actual implementation the freedom to deal with problems such as dead or slow machines and fluctuating resource availability, as well as allowing the implementation to take into account the physical placement of data in the system when assigning data to a CPU for processing. Furthermore, the model must be easy to implement, requiring little in the way of change or modification to the system it is built upon.

As discussed in the introduction to the chapter, I note that the “classic” OLA operational model (where data are simply processed in random order) is not directly applicable, because it ignores the “inspection paradox.”

### 3.2.2 Our Model for OLA Over MapReduce

As such, I must define a somewhat more complicated operational model, whose key ideas are as follows.

I assume that data have been organized into storage units that I refer to as *blocks*. A “block” is nothing more than an arbitrary subset of the data in the system; typically, we would expect a block to contain tens or hundreds of megabytes of data. I allow for the possibility that the data may have been organized into blocks in an adversarial fashion that the OLA software cannot control (that is, some blocks may be very large or very small, or may contain all of the data with the greatest aggregate values). However, all of the methods I describe in the chapter are also compatible with the case where the data have been placed into blocks in a fully random fashion. In that case, the aggregate values associated with each block would likely have low variance, and so our methods will converge much more quickly than if packed in an adversarial fashion. But our methods apply to both cases.

At the time that the OLA computation begins, all of the blocks are logically ordered in
a statistically random fashion, into a single queue of blocks. I assume the existence of a 
GetNext() operation that iterates through the blocks in the queue in order.

As in other MapReduce implementations, I assume a central scheduler whose job is 
creating mappers and reducers, supplying them with data, and scheduling all of them on 
the physical system hardware. When the scheduler decides that there are enough system 
resources to process the next block, it makes a call to GetNext() to obtain the identifier 
for a random block. The scheduler must schedule the block given to it by GetNext(); it 
cannot schedule the blocks out of order. After some arbitrary delay, this block is assigned to 
a mapper, at which time the block is “processed” by the mapper. This processing includes 
dead time while the block is read from disk and transferred over the network, and it includes 
all of the necessary processing of the actual bytes in the block. Once the scheduler has 
assigned a block to some mapper, it then calls GetNext() to obtain another unprocessed 
block to assign to another mapper. I assume that the scheduler only assigns one block 
at-a-time to each mapper, so that the processing times are independent across each mapper.

Although the scheduler is not allowed to schedule blocks out of order, nor can it have 
more than one block that has been obtained from GetNext() that has not been scheduled, 
it may wait as long as it wants to call GetNext(), and it may also wait as long as it wants 
to schedule a block once it has been obtained by GetNext(). This flexibility is important 
because it allows the scheduler to wait for a physical mapper to become available that is 
located close to some physical copy of the block.

3.2.3 Taking a Snapshot

When it is time for the statistical analysis software to estimate the final answer to the query, 
a snapshot must be taken of the system. This snapshot consists of all of the statistics that 
will be used by the software to compute its estimate. These statistics are collected on a per
block basis. For block $i$, they include:

1. The status of the block. This status is either **done** if the block has been fully processed, **processing** if the block is being processed by a physical mapper, or **unassigned** if the block has been obtained by `GetNext()` but is waiting to be assigned to a mapper.

2. If the block’s status is **done**, then for each group in the block, the snapshot contains $x_{i,j}$, which is the value obtained by aggregating all records in the block that fall in group $j$.*

3. The snapshot contains $t^{sch}_i$, which is the time taken to assign the block to a mapper. If the block’s status is **unassigned**, then a lower bound on $t^{sch}_i$ is given; this is the time elapsed waiting for assignment.

4. If the block’s status is not **unassigned**, then the snapshot contains the IP address (machine) of the mapper, as well as where the block was physically obtained from; this takes the value **local** (if the block was read from the same machine as the physical mapper), **rack** (if it was read from a different machine on the same rack), or **dist** (if it was read from a machine on a different rack).

5. If the block’s status is not **unassigned**, the snapshot also contains the time $t^{proc}_i$, which is the time required to process the block. If the block’s status is **processing**, then $t^{proc}_i$ is the time taken since the mapper was first given the block by the scheduler.

*For simplicity and clarity in the rest of the chapter, I will drop the $j$ and assume that all measured quantities, times, and statistics refer to a single group for which we have decided to perform estimation. The extensions to the multi-group case are straightforward—they require collecting all of the statistics on a per-group basis—and will only serve to complicate our notation.
3.2.4 Why this model? Why these statistics?

I end this section by considering some of the intuition behind the operational model.

From an implementor’s point of view, the model is compelling because it allows for a lot of freedom. The scheduler is able to process blocks on whatever machine it chooses. It can wait to schedule a block if no appropriate machine is available. It can ramp up the computation over time by adding more physical mappers, or ramp it down by simply not asking for blocks. The only real constraint is that when the scheduler makes a call to `GetNext()` to obtain a new block, it must assign the block it is given.

From a statistical point of view, the model has been designed with one singular goal in mind: at the time that a snapshot is taken, we wish to be able to (reasonably) view the $x_i$ values associated with each of the blocks that have been received by some call to `GetNext()` as a set of independent, identically distributed (iid) samples from a random variable with distribution function $f(x_i)$. As with any finite population, by randomly permuting the population and then traversing the items in order, we produce a set of (approximately) iid samples from a distribution where:

$$f(x_i) = \frac{\sum_j I(x_i = x_j)}{n}$$

In this expression, $n$ is the size of the population and the function $I$ returns the value 1 if the boolean argument is true, and 0 otherwise. The word “approximately” is necessary only because of the small correlation induced by the fact that the population is finite.

If this were the entire story, then we would essentially be done: we would obtain a set of iid samples from $f(\cdot)$, and hundreds of years of statistical theory would tell us exactly how to infer the various characteristics of $f(\cdot)$ and estimate the final query result.

Unfortunately, an added complication is that we have the so-called “inspection paradox” to deal with. While the aggregate values associated with blocks that have been ob-


\textit{tained} by \texttt{GetNext()} can be seen as iid samples from $f(\cdot)$, the aggregate values associated with the blocks that have been fully processed and have observable values cannot. That is, it may be the case that the time taken to schedule or to process a block is correlated with its contents. Thus, when we take a snapshot, some non-random set of the blocks returned by \texttt{GetNext()} may not yet have completed processing. For example, waiting for long time to schedule may, in fact, be the result of a block having a high aggregate value—the block may be in a busy part of the cluster and so it is difficult to schedule, but the reason that part of the cluster is busy could very well be that its blocks have more bytes, and hence higher aggregate values. Note that even in the case where blocks are uniformly sized, there may still be a correlation between processing time and aggregate value—imagine, for example, that the blocks on a slow machine tend to have a high aggregate value.

To take this into account, we allow for the scheduling and processing times to be correlated with the actual aggregate value, and we assume that the set of values associated with the blocks returned by \texttt{GetNext()} are samples from a three-dimensional distribution $f(x_i, t_{sch}^i, t_{proc}^i)$.

By using this three-dimensional distribution function, it will allow us to make predictions about the $x_i$ values that we have not seen, but for which we have information about $t_{sch}^i$ and $t_{proc}^i$, and hence we can deal with the inspection paradox in a principled fashion. For example, if we have a particular block that has been processed for 125 seconds, where it took 5 seconds to schedule, we can correctly view $x_i$ as a random sample from the distribution $f(x_i | t_{sch}^i = 5, t_{proc}^i \geq 125)$, thereby neutralizing the inspection paradox. This is precisely why all of the various timings are collected during the snapshot: they must be taken into account when estimates and confidence bounds are produced.

While this is a fairly thorough introduction to the intuition behind the model and the associated statistical considerations, the actual estimation process will be described in detail
3.3 Implementing the Model

In this section I describe our implementation of the OLA model in Hyracks [34]. Hyracks is a new open source project that supports map and reduce operations, along with higher level relational operations such as filter (selection), projection, and join. The Hyracks architecture is similar to Hadoop—it has a single master node for submitting jobs (queries) and housing the task scheduler, which executes tasks on worker nodes running in the cluster. Hyracks tasks support read and write operations in HDFS, which we leverage to store the input to the map tasks and the output of the reduce tasks. Like Hadoop, when a client submits a MapReduce job, Hyracks assigns a single map task to a given block in the input data, and creates a configurable number of reduce tasks that are assigned specific groups using some partitioning function.

I modified the Hyracks implementation in two ways. First, I created a single queue containing the blocks in the input data. The order of the blocks in the queue is uniformly shuffled using the `java.util.Collections.shuffle` routine from the Java Standard Library. When Hyracks schedules a map task, it assigns the current block at the head of the queue. The map task’s execution time includes the time to obtain its assigned block from HDFS, the execution of the map function on each input record, and the execution of the combiner on the complete map function output. In this work I ignore performance issues involving locality; although I do account for block locality in our model. In future work, I plan on investigating locality scheduling techniques that use multiple (shuffled) queues—scheduling from the queue with the optimal locality placement—and Delay Scheduling [103].

Our second modification involves running the estimator in the reduce task during the subsequently.
shuffle phase. In the shuffle phase, the reduce task is continuously receiving the output of completed map tasks. The output of a map task includes a data file containing the groups assigned to the reduce task and a meta-data file containing timing and locality information. If the map output contains no groups for a given reduce task then an empty data file is given along with a complete meta-data file. The meta-data file contains the block identifier, the time it took to schedule the block and the block locality relative to the map task execution: machine-local, rack-local, or distant. Also included is the map task IP address, start time and end time. Finally, we include the time when the estimator is called on the reduce task. The reduce task executes the estimator when it receives a new map output. The location of all data and meta-data files received thus far is given to the estimator when it is called. After the estimator completes, its output can be written to HDFS or forwarded to a downstream operator using the user-defined call-back class described in the introduction.

One issue that caused us some headache during the debugging of our system is that a reasonably synchronized global time must be maintained for the system. Since block processing times are typically on the order of minutes, this synchronization need only be accurate to within a few seconds. But a significant drift can indeed cause problems. The reason is that when a block arrives at the reducer, the total processing time is computed by subtracting the time that the block is received from the time that estimation began. Likewise, at estimation time, the reduce task performing the estimation must subtract the start time for the block from the current time to obtain a lower bound on the total processing time for the block. Due to the way I implemented this originally, these lower bounds were not consistent with the total processing time recorded for each block—the bounds tended to be much too large. Since a correlation between processing time and aggregate value had been observed, the result was that at estimation time the system “guessed” that the aggregate value for these unfinished blocks was very large, and significant over-estimates
3.4 Estimation

In this section, I consider how estimates and confidence bounds for those estimates can be obtained. As intimated previously, this is a challenging problem, as we must take into account processing times as well as observed aggregate values in order to circumvent the inspection paradox.

3.4.1 Overview

We will apply a Bayesian approach for estimation [104]; for brevity, this section will assume that the reader has some very basic familiarity with Bayesian statistics. The Bayesian approach has several obvious benefits for this particular problem. Most significant is the fact that the inspection paradox “goes away” under the Bayesian approach if one takes into account the time spent waiting for each block to be processed as observed data.

In standard Bayesian fashion, I will first describe a stochastic, parametric process that we imagine was used to produce the “observed” as well as the “hidden” data. The “observed data” will collectively be referred using variable $X$. This set includes all of the known aggregate values and processing times. Our generative process will also produce a set of unobserved variables collectively referred to as $\Theta$. $\Theta$ includes any data that is unobserved (for example, the processing time for a block that has not yet finished)—this data is collectively referred to as $Y$—as well as any unknown parameters required by the generative process (for example, the mean aggregate value per block). In Bayesian fashion, I will then attempt to infer the distribution $P(\Theta|X)$, which is referred to as a posterior distribution for $\Theta$. Then, given $X$ as well as $P(\Theta|X)$, it is possible to obtain a posterior distribution over the actual query result, which can be used to obtain confidence bounds.
that are reported to the user.

Note that the discussion in this section is directly applicable only to `SUM` and `COUNT` queries, which are both evaluated by simply summing \( x_i \) values (in the `SUM` case, \( x_i \) will contain the total aggregate value for the block, and in the `COUNT` case, \( x_i \) will contain the tuple count for the block). Extensions to other aggregates such as `AVG`, `VARIANCE` and `STD_DEV` are straightforward; in general they require that we maintain zeroth, first and second moments for each block\(^\dagger\).

3.4.2 Generative process

To obtain the data that we must analyze to produce estimates and confidence bounds, we imagine that the following steps are repeated, once for each of the \( n \) blocks in the system:

1. \( Z_i \sim \text{Normal}(\mu, \Sigma) \)

2. \((X_i, Y_i) \leftarrow \text{PostProcess}(Z_i)\)

“\( \sim \)” should be read as “is sampled from”. After this process has been repeated \( n \) times (once for each block)—our goal is then to infer the posterior distribution for \( \Theta \) using \( X \).

This process requires some additional explanation. We begin by describing the vector \( Z_i \). If there are \( m \) machines being used to execute a query, we imagine that associated with the \( i \)th block is a vector \( Z_i \) with \( 3m + 2 \) entries, which contains both observed and hidden data. \( Z_i \) takes the form:

\[
Z_i = (x_i, t_{sch}^{loc}, t_{i,1}^{rack}, t_{i,1}^{dist}, t_{loc}^{rack}, t_{i,2}^{rack}, t_{dist}^{rack}, \ldots, t_{i,m}^{loc}, t_{i,m}^{rack}, t_{i,m}^{dist})
\]

The vector has the following components:

1. \( x_i \) is the value that is obtained when the block is aggregated.

\(^\dagger\)The zeroth, first and second moments are count, sum and sum of squares respectively.
2. $t_{i}^{sch}$ is the time required to schedule the block, once it has first been selected for scheduling.

3. $t_{i,j}^{loc}$ is the time taken to actually process the block by a mapper on machine $j$, given that the block is to be read locally from machine $j$.

4. $t_{i,j}^{rack}$ is the time taken to process the block by a mapper on machine $j$, given that the block is to be read from a machine on the same rack as machine $j$.

5. $t_{i,j}^{dist}$ is the time taken to process the block by a mapper on machine $j$, given that the block must be read from a machine on a different rack.

Note that $\mathbf{Z}_i$ has $(3m + 2)$ dimensions, rather than the three dimensions one might expect after reading Section 3.3 of the chapter. The reason is that we do not have a single processing time distribution; rather, we have $3m$ such distributions, with three distributions for each machine, depending on where the actual data comes from. This provides for a very fine-grained model, where processing times can differ from machine to machine.

Also note that we assume that $\mathbf{Z}_i$ is normally distributed, with mean vector $\mu$ and covariance matrix $\Sigma$. At first glance, assuming normality may seem questionable, but in practice this is not a particularly significant assumption because we are aggregating over many $\mathbf{Z}_i$ values—one for each block. Assuming normality here is similar to appealing to the Central Limit Theorem [105] when applying more traditional, non-Bayesian methods.

Finally, note that in step (2) of the generative process, $\mathbf{Z}_i$ is “post-processed” to actually produce the observable data $\mathbf{X}_i$ that is associated with the $i$th block. This removes the data from $\mathbf{Z}_i$ that could not/should not be observed, and puts this unobservable data into $\mathbf{Y}_i$. For example, given that each block is processed only once, no one is ever going to observe both $t_{i,1}^{loc}$ and $t_{i,5}^{loc}$ for a given block—we might imagine that both values exist, but they will never
be observed together. Hence, $X_i$ will never contain both of these values, and one or the other must end up in $Y_i$.

In fact, there are four different ways in which the “post-processing” will be performed, depending upon the state of block $i$ at the time that the estimation is performed:

*i in case 1: (No information) $X_i = \langle \rangle$*

In this case, the block has not been chosen by the scheduler and so no information is available. $X_i$ is empty, and $Y_i = Z_i$.

*i in case 2: (Scheduling) $X_i = \langle [t^{sch}_i] \rangle$*

In this case, the block is at the head of the scheduler’s queue and is waiting for a map task to be assigned to it. Thus, we have a lower bound on the scheduling time, denoted by $[t^{sch}_i]$. This is simply the amount of time the block has been waiting to be scheduled. Again in this case, $Y_i = Z_i$.

*i in case 3: (Scheduled and processing) $X_i = \langle t^{sch}_i, [t^{L_i}_{i,W_i}] \rangle$*

In this case, a map task has been assigned to the block and processing has begun. Thus, we have access to an exact value for $t^{sch}_i$. We also have $W_i$, which is the identity of the machine on which the block is being processed, and $L_i$, which is the locality information for the block (loc, rack, or dist). Finally, we know $[t^{L_i}_{i,W_i}]$, which is a lower bound on the processing time for the block—one can view $t^{L_i}_{i,W_i}$ as being equivalent to $t^{proc}_{i}$ from Section 3. In case 3, $Y_i$ contains everything in $Z_i$ except for $t^{sch}_i$.

*i in case 4: (Scheduled and processed) $X_i = \langle x_i, t^{sch}_i, t^{L_i}_{i,W_i} \rangle$*

In this case, the map task has finished processing the data and the aggregate value has finally arrived at the reducer. Hence, in addition to $W_i$ and $L_i$, we know exact
values for the scheduling time, the processing time, and the aggregate value for the block. Here, \( Y_i \) contains everything in \( Z_i \) except for the three values in \( X_i \).

### 3.4.3 Prior Distributions

To make our model fully Bayesian, we must supply priors on \( \mu \) and \( \Sigma \). In our implementation, each \( \mu_k \sim \text{InvGamma}(1, 1) \) (where \( k \) refers to the \( k \)th dimension in \( Z_i \)). The inverse Gamma distribution is a standard, uninformative prior for values that must be non-negative—it makes sense to have non-negative means for all of the time values in the \( Z_i \) vector. It will also usually make sense to have a non-negative mean for \( x_i \); if not, then another suitable, uninformative prior can be used.

Handling the covariance matrix \( \Sigma \) is a bit trickier. The standard prior distribution for a covariance matrix is the inverse Wishart distribution, because it is “conjugate” for the normal. This means that under certain conditions, upon observing the output from a normal distribution with an inverse Wishart prior on the covariance, the posterior on the covariance is still inverse Wishart. Conjugacy is convenient because it can make inference much easier. Unfortunately, these “certain conditions” are not met in our application because we do not always have actual observations from the normal—we may only know, for example, that the processing time has a lower bound (if we are in “case three” from the previous subsection). Thus, we choose to use an application-specific prior that is easily factorizable; that is, where we can easily write the marginal distribution for each entry in the covariance matrix. This makes deriving a Gibbs sampler for inference much easier (see the next subsection).

Specifically, we let \( \sigma_k \sim \text{InvGamma}(1, 1) \), where \( \Sigma_{k,k} = \sigma_k^2 \). Then, we assume that the following process is used to generate the rest of \( \Sigma \):
Algorithm 2: Generative process for $\Sigma$

while true do
    foreach $k_1 \in \{1 \ldots (3m + 2)\}$ do
        foreach $k_2 \in \{(k_1 + 1) \ldots (3m + 2)\}$ do
            $\rho_{k_1,k_2} \sim$ GenBeta$(-1, 1, 1, 1)$;
            $\Sigma_{k_1,k_2} = \Sigma_{k_2,k_1} = \rho_{k_1,k_2} \times \sigma_{k_1} \times \sigma_{k_2}$;
        endforeach
    endforeach
    if $\Sigma$ is positive definite then
        break;
end

Here, GenBeta$(-1, 1, 1, 1)$ refers to a generalized Beta$(1, 1)$ distribution, stretched to cover the range from $-1$ to $1$ (rather than the usual $0$ to $1$). What this process does is to essentially sample a correlation $\rho$ for each of the pairs of variables in $Z_i$, and to then check whether a valid covariance matrix has been obtained (one that is positive definite). If it has not, then the whole process is repeated again. The PDF for $\Sigma$ can then be written as:

$$P(\Sigma) \propto \begin{cases} 
0 & \text{if } \Sigma \text{ is not positive-definite} \\
\prod_k \text{InvGamma}(\sigma_k|1,1) \\
\times \prod_{k_1,k_2} \text{GenBeta}(\rho_{k_1,k_2}|-1,1,1,1) & \text{otherwise}
\end{cases}$$

3.4.4 Posterior Distribution

In this subsection, I tackle the problem of obtaining a formula for the desired posterior distribution, $P(\Theta|X)$. Recall that $X = \bigcup_i \{X_i\}$, and the unobservable data set $\Theta$ contains $Y = \bigcup_i \{Y_i\}$, as well as the normal parameters $\mu$ and $\Sigma$.

From elementary probability, we know that:

$$P(\Theta|X) = \frac{P(X|\Theta)P(\Theta)}{P(X)}$$
This means that there are three quantities that we must derive expressions for: \( P(X|\Theta) \), \( P(\Theta) \), and \( P(X) \).

We deal with \( P(X|\Theta) \) first. From the generative process, we know that \( P(X|\Theta) = \prod_i P(X_i|\Theta) \). We can easily write an expression for each \( P(X_i|\Theta) \), which will depend upon the case that holds for block \( i \):

\textbf{i in case 1: (No information)} \( P(X_i|\Theta) = P(\emptyset|\Theta) = 1 \) since \( X_i \) is empty.

\textbf{i in case 2: (Scheduling)} Here, we have only a lower bound on the scheduling time. Thus,

\[ P(X_i|\Theta) = P(\langle t^{sch}_i \rangle|\Theta) = 1 \text{ if } t^{sch}_i \geq |t^{sch}_i|, \]

and 0 otherwise since this is impossible.

\textbf{i in case 3: (Scheduled and processing)} In this case,

\[ P(X_i|\Theta) = P(\langle t^{sch}_i, |t^{L_i}_{i,W_i}| \rangle|\Theta) = \text{Normal}(t^{sch}_i|\mu, \Sigma, Y_i) \text{ if } t^{L_i}_{i,W_i} \geq |t^{L_i}_{i,W_i}|, \]

and 0 otherwise since this is again impossible.

\textbf{i in case 4: (Scheduled and processed)} Here, we evaluate a normal distribution:

\[ P(X_i|\Theta) = P(\langle x_i, t^{sch}_i, t^{L_i}_{i,W_i} \rangle|\Theta) = \text{Normal}(x_i, t^{sch}_i, t^{L_i}_{i,W_i}|\mu, \Sigma, Y_i). \]

Now, we move onto deriving an expression for \( P(\Theta) \). From the last few subsections, we have:

\[
P(\Theta) = P(Y|\mu, \Sigma)P(\mu)P(\Sigma) \]

\[= P(\mu)P(\Sigma) \prod_i P(Y_i|\mu, \Sigma) \]

\[= P(\Sigma) \prod_j \text{InvGamma}(\mu_j|1, 1) \prod_i \text{Normal}(Y_i|\mu, \Sigma) \]

where an explicit formula for \( P(\Sigma) \) was given previously.
This gives us expressions for $P(X|\Theta)$ and $P(\Theta)$. In standard Bayesian fashion, we ignore $P(X)$, which would be very difficult to compute since it would involve integrating over $\Theta$. But since $P(X)$ does not depend upon $\Theta$, it is merely a normalizing constant that is necessary for the total mass of $P(\Theta|X)$ to be one, and is not needed to compare the relative merits of candidate $\Theta$ values.

### 3.4.5 Putting It All Together

Since our goal is to produce estimates and confidence bounds for the actual query result, we are not interested in the posterior distribution $P(\Theta|X)$ for its own sake. Rather, we will use $P(\Theta|X)$ to produce estimates and confidence bounds for the answer.

To describe how this is done, note that given a possible value for $\Theta$—combined with the visible data $X$—we have access to each and every $x_i$ value in the database. Thus, given a particular $\Theta$ as well as $X$ it is very easy to compute the query answer as:

$$Q(\Theta, X) = \sum_i x_i$$

Then by integrating $P(\Theta|X)$ over all possible $\Theta$, we obtain various statistics describing the eventual query result. For example, the following gives us the expected value of the query result:

$$\int_{\Theta} P(\Theta|X)Q(\Theta, X)d\Theta$$

And we can obtain the lower end $l$ for a 95% confidence bound on the query result by computing $\Lambda$ and $l$ so that:

$$\int_{\Theta \in \Lambda} P(\Theta|X)d\Theta = 0.025$$

where

$$\max_{\Theta \in \Lambda}\{Q(\Theta, X)\} \leq l$$ and $$\min_{\Theta \in \Lambda}\{Q(\Theta, X)\} \geq l$$

The upper end could be computed in a similar fashion.
Unfortunately, performing this sort of computation exactly is difficult. The difficulty is often circumvented using so-called “Markov Chain Monte Carlo” (MCMC) methods [106] that sample directly from a distribution such as $P(X|\Theta)$. In our case, we apply a particular MCMC method called a Gibbs sampler to the problem [107]. The samples obtained from a Gibbs sampler are easily used to compute expected value and confidence bounds. For example, we can run the sampler to produce several hundred candidate $\Theta$ values, and then average the associated query results—this is equivalent to the expected value computation described above. Cutting off the top 2.5% and the bottom 2.5% of the set of query results, and then taking the highest and lowest remaining results, gives 95% confidence bounds on the query answer.

### 3.5 Experiments

In this section, I describe a set of experiments on the software that I have developed. Our experiments are designed to answer the following questions: Can the confidence bounds that our system reports be trusted? How important is it to take into account the correlation between processing time and data value in both synthetic and real data? How important is choosing blocks in a statistically randomized order? In a realistic setting, is the system able to produce accurate results quickly?

#### 3.5.1 Experiment One

**Basic Setup.**

In the first experiment, I run our version of Hyracks with OLA over six months of data from the Wikipedia page traffic data set (available at http://aws.amazon.com/datasets/4182), with the simple goal of counting the number of Wikipedia page hits on a per-language basis over
Figure 3.1: Posterior query result distribution for number of Wikipedia page hits over the English language, at various query completion percentages, using both a randomized and arbitrary block ordering. The actual query result is a vertical black line.
Figure 3.2: Identical to above figure, except for the French language.
Figure 3.3: Posterior query result language for the French distribution, at various query completion percentages, taking into account and ignoring correlation between aggregate value and processing time.
those six months. Six months of Wikipedia data take up about 220GB (compressed), and are stored in 3,960 blocks. I run our software on an eleven node cluster, with one master node and ten slaves. Each machine has four disks, four cores running at 2.3 GHz, and 12GB of RAM. I use 80 mappers and ten reducers (with one reducer available for each of the ten languages that are to be counted). The entire MapReduce process takes approximately 46 minutes to run to completion.

To demonstrate the relative importance of the different components of our software, I run three different versions of our OLA software. The first uses every method described in the chapter. In the second version, randomization of blocks is not performed by the scheduler, so blocks are scheduled in arbitrary (but non-random) order. In the third version, randomization is used, but the (possible) correlation between processing/scheduling time and the aggregate value is not taken into account by the system, leaving the software vulnerable to the “inspection paradox” described earlier.

**Results.**

A subset of the observed results are given in Figures 3.1, 3.2, and 3.3. Figure 3.1 shows the posterior distribution of possible query results computed by our system, for the English language, at various times during the MapReduce task (I show results after 10% of the task is complete, after 20% is complete, after 30%, and so on). Two posterior distributions are plotted: one computed running the first version (all features from the chapter), and a second computed running the second version (no randomization). Each plot also shows the true query result for the English language. Figure 3.2 is similar to Figure 3.1, but it shows the results for the French language. Figure 3.3 also shows results for the French language, but it shows the computed posterior distribution for the query result after a much smaller portion of the task has completed: 1%, 2%, 3%, and so on. This plot also shows both
version one of our software, and version three (randomization, but no correlation).

Discussion.

It is clear that without randomization, severe bias is possible, and so confidence bounds obtained without a random scheduling order are useless. With randomization, the confidence bounds from Figures 3.1 and 3.2 seem remarkably accurate. It is significant that the bounds obtained are quite narrow, very quickly. Take the English language. After 10% of the blocks have been processed, the bounds go from approximately $4.1 \times 10^4$ to $4.4 \times 10^{10}$, which represents a possible error of only $\pm 3\%$. For many applications, it may be acceptable to simply kill the computation with this level of accuracy. In general, convergence could be made to happen even more quickly by increasing the number of blocks used to store the same data set, though this could have a negative effect on the overall processing time of the MapReduce job.

Even for this particular data set (where the correlation between processing time and aggregate value is quite weak) there is still a clear benefit to taking into account the correlation, particularly when only a very small fraction of the data has been processed. Consider the plots corresponding to finishing 3% and 4% of the MapReduce job. Without correlation, the posterior distribution almost totally misses the actual query result. Taking into account the correlation, the distribution is neatly bisected by the correct result.

3.5.2 Experiment Two

Basic Setup.

This experiment is somewhat similar to the first, except that our goal is to try to determine, in a systematic fashion, how accurate the computed posterior distribution is when used to compute 95% confidence bounds. Since testing accuracy requires many, many repetitions
of the MapReduce task, instead of actually running the task in a real cluster, I use a simulator. In our simulation, a random aggregate value is associated with each block, and random processing times are associated with each block as well; the correlation between aggregate value and processing time is set to be 0.7. Under the same setup as above (80 mappers, 3,960 blocks), I repeat the MapReduce and the estimation process 100 times. Every time that the estimation is re-run, I consider several different task-completion percentages (1% done, 2% done, 3% done, and so on). At each task-completion percentage, I use the computed posterior distribution to obtain 95% confidence bounds by “chopping off” the top 2.5% and bottom 2.5% of the posterior distribution. I then determine whether or not the actual query result is within the 95% confidence bounds, and compute the fraction of the time that the query result is not within the 95% confidence bounds over each of the 100 repetitions. If our estimation process worked perfectly, then the 95% confidence bounds would cover the actual answer 95% of the time, with a 5% error rate. This whole process is repeated twice: once using the full estimation process, and a second time ignoring the possibility of correlation between processing time and aggregate value.

Results.

The results are given in Table 3.1, and are mostly self-explanatory. For each of the listed task-completion percentages, the fraction of “incorrect” 95% confidence intervals is given, for both of the software versions.

Discussion.

As can clearly be seen, the confidence bounds computed when taking into account correlation are accurate, coming very close to the expected 5% throughout query execution. On the other hand, the results obtained without taking into account the correlation are very
poor, particularly when only a small fraction of the MapReduce task has been completed. This mirrors the results shown in Figure 3.2, but under much more extreme circumstances.

### 3.6 OLA Programming Interface

In this section I describe the interface that Hyracks exports for running online aggregation (OLA) queries. I begin with the programming interface, which is very similar to conventional Hadoop\(^\dagger\), except for three key differences:

1. The user’s Mapper, Reducer and Combiner classes have to obey the contract given in section 3.6.1.

2. The user can optionally write a class that implements the `EstimateReceiver` interface (see section 3.6.4). This will be called whenever a new estimate is generated. I provide a default implementation of this interface that simply writes the estimates to HDFS.

3. The user also needs to specify few additional properties in the job configuration file (see section 3.6.2 and 3.6.3).

\(^\dagger\)I will use the terms *Hadoop* and *Hyracks* interchangeably since Hyracks supports the Hadoop API.

---

<table>
<thead>
<tr>
<th>Percentage of MapReduce task complete</th>
<th>2%</th>
<th>3%</th>
<th>4%</th>
<th>5%</th>
<th>10%</th>
<th>20%</th>
<th>30%</th>
</tr>
</thead>
<tbody>
<tr>
<td>w corr</td>
<td>.03</td>
<td>.06</td>
<td>.05</td>
<td>.05</td>
<td>.02</td>
<td>.05</td>
<td></td>
</tr>
<tr>
<td>w/o corr</td>
<td>.70</td>
<td>.62</td>
<td>.61</td>
<td>.37</td>
<td>.22</td>
<td>.13</td>
<td></td>
</tr>
</tbody>
</table>
3.6.1 Map, Reduce and Combiner Contracts

In this subsection, I describe the input and output of the mapper, the combiner, and the reducer in our OLA implementation. Our system supplies default combiner and reducer implementations; the mapper is always user-supplied.

Currently, our OLA system designed to handle computations that are equivalent to the following SQL.§

\[
\begin{align*}
\text{SELECT } & \text{AGG} (g(y)) \\
\text{FROM } & \text{MY_TABLE AS } y \\
\text{GROUP BY } & h(y)
\end{align*}
\]

To implement such a query, the user would first need to supply a mapper. The mapper would simply read in the data, and then output a stream of \( (h(y), g(y)) \) pairs.

Now consider the combiner. For the time being, I restrict ourselves to the case where \text{AGG} is \text{SUM}. Section 3.2.3 of the chapter describes a number of statistics that must be collected by the system to perform OLA over such a query. The various timing values are hidden from the Hyracks programmer who uses our system, and are collected automatically. Thus, assuming for a moment that the system is restricted to handling \text{SUM} queries, the only one of these parameters that a Hyracks programmer would actually be concerned with producing is \( x_{i,j} \). Recall that \( x_{i,j} \) is the aggregate value for group \( j \) over block \( i \), which would be produced (along with the group key \( j \)) by the combiner. Thus, the output of the

---

§Note that while this query does not contain an explicit \text{WHERE} clause, a boolean predicate can implicitly be present in \( g(y) \) by making this function return zero if the record in question does not match a selection condition. I also plan to generalize beyond such simple queries to support multiple aggregation operations, as well as joins.
combiner whose input key is $j$ and input values are $\langle g(y_1), g(y_2), \ldots \rangle$, is:

$$x_{i,j} = \sum_k g(y_k)$$

along with the key for group $j$.

Likewise, the reducer accepts a set of $x_{i,j}$ values (all having the same group key $j$), aggregates them and then outputs a final aggregate value along with the group key.

In our current implementation, the group key type must be `Text`, and $x_{i,j}$ must be of type `Double`. These types are a bit restrictive, and I plan to generalize this in the future.

While the prior discussion is adequate to handle SUM queries, (and, if $x_{i,j}$ is binary, it suffices for COUNT queries), in order to support AVG, VARIANCE and STD_DEV we need some additional information. Note that $x_{i,j}$ is the so-called “first moment” of the distribution of the values in block $i$. To handle the additional aggregates, we generalize $x_{i,j}$ to $x_{i,j}^m$ so that:

$$x_{i,j}^m = \sum_k g^m(y_k),$$

where $x_{i,j}^m$ denotes the $m$th moment of the values in the block. To support the full set of five aggregate functions described above, we need the zero-th, first, and second moments of each block. Thus, the user-supplied combiner does not only output $x_{i,j}$ (which is equivalent to $x_{i,j}^1$), it must actually output the triple $\langle x_{i,j}^0, x_{i,j}^1, x_{i,j}^2 \rangle$ in a comma separated vector (CSV) record format. This triple is then used by the reducer to produce the final aggregate.

The inputs and outputs for the mapper, combiner, and reducer are summarized in the table 3.2.

### 3.6.2 Specifying the aggregate

Internally, our OLA library provides implementations of the combiner and reduce functions for the five aggregates described previously. I provide a helper class `OAJobConfigurer`
Table 3.2: Mapper, Reducer, Combiner class interfaces.

<table>
<thead>
<tr>
<th></th>
<th>Mapper</th>
<th>Combiner</th>
<th>Reducer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input key</td>
<td>N/A</td>
<td>$j$</td>
<td>$j$</td>
</tr>
<tr>
<td>Input value</td>
<td>N/A</td>
<td>$y_k$</td>
<td>$x_{i,j}^0, x_{i,j}^1, x_{i,j}^2$</td>
</tr>
<tr>
<td>Output key</td>
<td>$j$</td>
<td>$j$</td>
<td>$j$</td>
</tr>
<tr>
<td>Output value</td>
<td>$y_k$</td>
<td>$x_{i,j}^0, x_{i,j}^1, x_{i,j}^2$</td>
<td>Final value</td>
</tr>
</tbody>
</table>

that allows the user to indicate the aggregate choice, and which asks the Hyracks engine to use the default implementation of combiner and reduce functions for that aggregate. Use of this class is demonstrated in the following code snippet:

```java
Job job = new Job();
job.setMapperClass(mapperClass);
OAJobConfigurer.configure(job, "SUM");
...
master.submitJob(job);
```

The first line creates a new job configuration object, which is submitted to the master scheduler by the last line. The configuration parameters are set by the client prior to the final submission of the job. Here, we select the (user-defined) mapper function implementation and also the (default) SUM aggregate implementation from the OLA library. It should be noted that the clients can still implement their own reduce (and combiner) function as long as it conforms to contract given in the section 3.6.1. This is useful, for example, when one needs to implement a HAVING clause for a group-by aggregate.
3.6.3 Configuration parameters

Next, I describe the configuration parameters exported by the system. These parameters must be set by the client prior to the final job submit call to the master scheduler. Our goal is to give confidence bounds of the form “with probability $p$, the actual query answer is within the range from $low$ to $high$”, where the bounds are updated each $ms$ milliseconds during the execution of the query.

The user can specify this using the following properties in the job configuration file:

1. `mapreduce_online.estimation_interval = ms`
   
   This property specifies the number of milliseconds after which the Bayesian estimator should be called by the Hyracks engine. If the value is large, the estimator is rarely called and it improves the performance of map-reduce job. This is because the Bayesian estimator is CPU intensive code and will compete with the map-reduce workers for CPU cycles. On the other hand, if the value is low, the estimator is effectively called after the processing of every block, hence giving more estimates to the user.

2. `mapreduce_online.confidence_interval = range`
   
   Another way to improve the performance is to ask the Hyracks engine not to call the estimator code once a desired accuracy is reached. That is, estimation is stopped once the width of the interval returned by the Bayesian estimator is less than or equal to the specified range. To disable this feature (i.e. to obtain estimates until the end of the map-reduce job irrespective of accuracy), the user has to set this value to 0.

\*The probability $p$ is fixed during the execution of the query, and the estimator outputs the confidence interval $[low, high]$ based on the value of $p$.\*
A feature slated for future work, which could potentially reduce the cost of estimation further, is to allow the user to switch the scheduling policy from random to a more conventional locality-based scheduling, once all the groups have reached the desired accuracy.

3. `mapreduce_online.confidence_level = conf`
   The system defaults to `conf = 95%`.

4. `mapreduce_online.reduce_operator = AGG`
   Any one of the five aggregates defined previously can be used for `AGG`.

5. `mapreduce_online.estimate_receiver.class = RC`
   `RC` gives the name of the class that will be called back by the Hyracks engine to communicate the estimates with the user. This class has to implement the `EstimateReceiver` interface, described in the section 3.6.4.

### 3.6.4 EstimateReceiver interface

Whatever mapper, combiner, and reducer are specified by the user, the output of the various combiners is sent both to the user- or system-supplied reducer, and to our Bayesian estimation code which uses the combiner-supplied moments to estimate the final answer to the specified aggregate query. Somehow, the system needs to do something with these estimates. It is the `EstimateReceiver` class that is tasked with accepting an estimate from the OLA engine and processing it as necessary. The `EstimateReceiver` class, for example, might write the estimate to a file, or it might display the result graphically in a web page so that the user can see the progression of the computation.

The interface for this class is:

```java
public interface EstimateReceiver {
```
Whenever the estimator finishes processing, the Hyracks engine calls `Open` function and then calls `NextGroup` for each group. Finally, it calls `Close` to notify that there are no more groups found in the current estimation cycle. If the job is completed (i.e. there are no more blocks remaining to be processed) or if all the groups have reached desired accuracy, the Hyracks engine will call `NoMoreBlocks` method.

### 3.6.5 Example

To make this interface concrete, I describe the steps involved to implement the following simple SQL query:

```sql
SELECT SUM(page_hits)
FROM WIKIPEDIA_LOG
GROUP BY LANGUAGE
```

The above SQL query uses the Wikipedia page traffic data set, that stores each log entry in CSV format. The two fields in the log that we are interested in, are the language of the Wikipedia page pointed to by that entry (`LANGUAGE`) and the number of times that page has been viewed (`page_hits`). The above query returns the total number of page hits for each language in the Wikipedia data set. This query can be implemented by a single
MapReduce job.

To implement this query, the user-supplied map function parses the data, and then outputs the language (the key) and the number of page hits. The reduce function accepts a single language and list of page hit values. It sums the page hit values and writes a result (key-value) record to the output. Finally, the WikipediaMain class stitches the job together and submits it to the master scheduler for execution.

```java
public class WikipediaMapper extends Mapper<LongWritable, Text, Text, DoubleWritable> {
  // Line number is the key and
  // the content of the line is the value
  public void map(LongWritable key, Text value, Context ctx)
      throws IOException, InterruptedException {
    // LANG and PAGE_HITS are integer constants
    // that depend on the format of the log entry
    Array[String] fields = value.split(",");
    ctx.write(fields[LANG],
              Double.parseDouble(fields[PAGE_HITS]));
  }
}

public class WikipediaCombiner extends Reducer<Text, DoubleWritable, Text, Text> {
  public void reduce(Text key,
                      Iterable<DoubleWritable> values, Context ctx)
      throws IOException, InterruptedException {
  }
}
double sum = 0;
for (DoubleWritable val : values) {
    sum += val.get();
}
ctx.write(key, new Text("" + sum + ","));
}

public class WikipediaReducer extends Reduce<Text, Text, Text, DoubleWritable> {
    public void reduce(Text key, 
                       Iterable<DoubleWritable> values, Context ctx) 
                       throws IOException, InterruptedException {
        double sum = 0;
        for (Text val : values) {
            Array<String> moments = val.toString().split(",");
            sum += Double.parseDouble(moments[1]);
        }
        ctx.write(key, new DoubleWritable(sum));
    }
}

public class WikipediaMain {
    public static void main(String[] args) 
        throws Exception {

...  
Job job = new Job();  
job.setMapperClass(WikipediaMapper);  
job.setCombinerClass(WikipediaCombiner);  
job.setReducerClass(WikipediaReducer);  
...  
master.submitJob(job);  
}
}

To obtain online estimates for the same program, the user needs to modify the WikipediaMain class to specify the aggregate function (as discussed in section 3.6.2) and set the job configuration properties (as discussed in section 3.6.3). Note, that using this default configuration, the user need not write the reducer and the combiner class.

```java
public class WikipediaMain {
    public static void main(String[] args) throws Exception {
        ...
        // Set job configuration properties
        Configuration conf = new Configuration();
        conf.setProperty("mapreduce_online.reduce_operator", "SUM");
        conf.setProperty("mapreduce_online.estimate_receiver.class", WikipediaEstimateReceiver);
```
// Specify aggregate
Job job = new Job(conf);
job.setMapperClass(WikipediaMapper);
OAJobConfigurer.configure(job, "SUM");
...

The user also needs to specify an extra class
WikipediaEstimateReceiver that handles the estimates returned by the OLA library.

public class WikipediaEstimateReceiver
    implements EstimateReceiver {
    ...
    public void NextGroup(Text key, Double estimate, Double low, Double high) {
        // Do something with estimate
        // Example: display it to a GUI
    }
}
Chapter 4

Distributed Gradient Descent Using Online Aggregation

4.1 Introduction

Optimization lies at the heart of data mining, machine learning, statistical analysis, and decision support, and gradient descent (GD) [82, 26] is perhaps the most widely-used large-scale optimization methodology. Given a data set $T$ with elements $t_j$ (e.g., a database table comprised of tuples), the goal is to learn a $d$-dimensional vector ($d \geq 1$) of model parameters $\Theta = (\Theta^{(1)}, \Theta^{(2)}, \ldots, \Theta^{(d)})$ that minimize an objective function (also called a loss function) of the form $\sum_j L(t_j|\Theta)$. To this end, GD performs the following simple update* repeatedly until convergence:

$$\Theta_{i+1} \leftarrow \Theta_i - \sum_j \nabla L(t_j|\Theta_i).$$

Each update (i.e., each downhill step) marks the end of a processing epoch. $\Theta_i$ denotes the set of model parameters learned after the $i$th epoch.

Anecdotal evidence suggests that in industry, “plain vanilla” GD is the go-to choice for the task of learning standard, predictive models over very large data sets, such as logistic regression models and support vector machines. The GD has many advantages, but perhaps the most important—as observed in [108]—is that, at its core, GD requires the iterative execution of a series of computations, each of which looks essentially like a simple SQL $\text{SUM}\,$.

---

*In the usual GD representation, the term $\delta_j = \sum_j \nabla L(t_j|\Theta_i)$ is multiplied by a “learning rate” (also called a “step size” or “gain parameter”) that converges to 0 over time for stochastic GD algorithms or is fixed for batch GD algorithms. For simplicity, we absorb this factor into the value of $\delta_j$. 

aggregate query. The only substantial difference is that the items being \textit{SUM}ed (that is, the \( \nabla L(t_j|\Theta_i) \) terms) are typically vectors rather than scalars. But as long as the vectors being \textit{SUM}ed fit into RAM, GD is as simple to implement and as embarrassingly parallelizable as a classical, single-table SQL \textit{SUM} query.

Furthermore, GD is a widely-applicable framework. Other optimization strategies might converge to a solution more quickly for specific problems, for example, strategies that make use of second-order partial derivatives (the “Hessian matrix” [82]) or that exploit special problem structure as found, for example, in matrix completion problems [109]. But, in general, second-order methods are infeasible past a few thousand dimensions, and it can take a lot of time and effort to identify the special structure in a particular optimization problem, if it exists at all.

**Gradient descent in a distributed environment.** GD in its simplest form is known as \textit{batch GD}, because the set of model parameters \( \Theta \) is updated only after the entire data set has been processed. In a single-processor environment, alternatives that update \( \Theta \) after processing only a small, random subset of the data (called \textit{mini-batch GD}) or after processing a single random data point (called \textit{stochastic GD}) are typically preferred, because they converge faster than batch GD [26]. Intuitively, the reason is that one or a few points can yield an unbiased estimate for \( \sum_j \nabla L(t_j|\Theta_i) \); additional data can reduce the variance of this estimate, but then the updates to \( \Theta \) that move it closer to the optimal value \( \Theta^* \) are less frequent. In a simple, one-processor environment, one can argue mathematically that a strategy having less accurate gradient estimates but more frequent epochs is superior to a strategy with fewer epochs and more accurate gradients [23].

The picture, however, is less clear in massively distributed environments where systems such as Hadoop [100], Spark [6], and so on are used to run the computation on dozens or hundreds of loosely-coupled machines. Compared to a one-processor environment, the key
difference is that ending an epoch is not costless. Each individual processor can compute a summation in isolation over the data it has been assigned, but when the epoch ends, these local summations must be summed up via a distributed aggregation, and then the resulting updated $\Theta$ must be broadcast back to each of the processors in a distributed system. After this broadcast, the next epoch can begin.

The time required to end an epoch will vary based upon the platform being used, but may be anywhere from a couple of seconds to the better part of a minute for naive algorithm implementations. This fundamentally changes the situation. In a single processor environment, the point-at-a-time stochastic GD algorithm is typically preferred, but in a massively distributed environment, it would obviously be silly to use a hundred machines to each process one out of $10^{14}$ data points, then wait a few seconds for the next epoch before processing the next data point.

**Managing the cost of ending an epoch.** There are three general approaches to addressing the high cost of ending an epoch in a distributed system. The first approach is the simplest: do a classical batch GD, and process the entire data set in parallel before ending the epoch; see, e.g., [110]. This has the clear advantage that it maximizes the ratio of time spent processing data to time spent waiting between epochs. The problem, however, is the same as with batch GD in a single-processor environment: one wastes a tremendous amount of time computing a precise value for each $\sum_j \nabla L(t_j|\Theta_i)$, when one would travel more quickly toward the optimal solution by executing more epochs using less accurate gradient estimates in each.

A second approach, called *distributed mini-batch* [111], attempts to remedy this situation by randomly organizing the data into blocks of some pre-defined size (where the number of blocks in the system is considerably larger than the number of processors). In an epoch, each processor is randomly assigned one block to process; after each processor
has finished its block, the epoch ends. The problem with this approach is that it requires that the data be randomized, which can be very expensive, and the user must somehow choose a good block size a priori, at which time it is fixed. The algorithm cannot adapt to the problem.

The third approach, called distributed stochastic GD [112], is remarkably simple. Here the idea is to have each of the processors run a stochastic GD independently and in parallel on a subset of the data. Once all of the data have been processed, the resulting parameter sets are averaged. This turns out to be an excellent approach, preserving the key benefits of single-processor stochastic GD, provided that the learning problem is convex. Otherwise, distributed stochastic GD is a very poor choice, since each individual stochastic GD process converges to a locally optimal solution, the average of which is meaningless. Crucially, many important models (particularly those used in recommender systems [113]) are not convex.

**Distributed GD via Online Aggregation.** Because each GD epoch requires essentially the computation of an aggregate query, it is natural to ask whether distributed online aggregation (OLA) as in [35] can be used to improve the efficiency of distributed GD. The idea in OLA is to process the data in random order, using the subset of the data processed so far to estimate the final answer to an aggregate query [16]; processing can terminate as soon as the estimate is sufficiently accurate. Thus one could envision using distributed OLA to estimate the quantity $\sum_j \nabla L(t_j|\Theta_i)$ in an online fashion, stopping when the estimate is good enough that it makes sense to update the parameters and start the next epoch. It is this idea that we explore in this chapter.

Our specific contributions are as follows:

1. We consider how to modify the Bayesian statistical model used in ordinary distributed OLA for use in distributed GD.
2. We especially focus on the difficult problem of using an OLA-style estimate for the gradient (as well as an estimate of the accompanying error) to adaptively determine when to stop one epoch and start another. To do this, we utilize a high-level Bayesian model of the walk being performed by the GD algorithm. The system can decide, in a principled fashion, when the current set of parameter values is close to the optimal set. At this point, more accuracy is required and epochs become longer.

3. We implement OLA-GD as well as the three other options described above, and compare them experimentally across a variety of learning tasks. We show that OLA-GD is the most widely applicable distributed GD framework. For simple, convex learning problems, it is not much slower than distributed stochastic GD; for difficult, non-convex problems, where distributed stochastic GD is not usable, OLA-GD is the best choice.

4.2 OLA-GD Overview

At the highest level, OLA-GD exploits the equivalence between GD and database aggregation [108], executing each GD epoch as a separate, distributed OLA task. The OLA-GD software monitors the accuracy of the OLA computation, using the accuracy to decide when the estimate provided by the OLA task is good enough to end the epoch, and move on to the next one. In this way, the OLA-GD software adaptively decides when to end an epoch, naturally taking into account the cost associated with ending one epoch and starting another in a distributed environment (see Section 4.5 of the chapter).

To make this decision, OLA-GD fundamentally relies on two different, Bayesian statistical models:

1. The OLA model (Sections 4.3 and 4.4 of the chapter) allows us to choose a reasonable
parameter vector $\hat{\Theta}_i$ after processing a (possibly small) subset of the data, where $\hat{\Theta}_i$ is an estimate for $\Theta_i$. Here, $\Theta_i$ is the parameter vector that would result if we were to run the $i$th epoch to completion. This model provides for a principled way to counteract the bias that might occur when running OLA in a distributed environment (see Section 4.3.2). It also provides for a posterior distribution over $\epsilon_i = \hat{\Theta}_i - \Theta_i$, which is the error of the OLA estimate (Section 4.5.1).

2. The random walk model (Section 4.5.3) is a stochastic model for the sequence of parameter vectors $\Theta_{i+1}, \Theta_{i+2}, \ldots$ that would result if we were to run a batch GD starting with the set of model parameters $\Theta_i$. The utility of this model is that we can run it forward from a given set of model parameters, giving us a distribution over the optimal parameter set $\Theta^*$.

Taken together, these two models can be used to compute a distribution over the distance traveled toward $\Theta^*$, were we to stop the current epoch now and use $\hat{\Theta}_i$ to approximate $\Theta_i$ (Section 4.5.2). By dividing the expected distance traveled by the time taken, it tells us the average velocity with which we have traveled toward $\Theta^*$. As we will argue in Section 4.5.4, a very simple rule-of-thumb for stopping the current epoch—end it when the current velocity falls below the average velocity over the epoch—tends to maximize the average velocity with which we travel toward $\Theta^*$, and hence it minimizes the expected time for the OLA-GD algorithm to run to completion.

4.3 Background: OLA

4.3.1 Online Aggregation

*Online aggregation* (OLA) [16] refers to a family of “anytime algorithms” [114] for use in database aggregation query processing. In OLA, data are processed in random order so
that, at any time, the set of data processed thus far constitutes a random sample (without replacement) of all of the data in the database. Thus statistical techniques can be used in conjunction with the subset of the data that have been processed to produce an online estimate of the final answer to an aggregate query, and to give error bars describing the accuracy of the estimate. Further, since computing the estimate (and error bars) often requires only a small set of aggregate values that are easily maintained with only a few CPU cycles per database tuple, OLA incurs little additional cost above and beyond the cost of computing the aggregate query itself. The original motivation for OLA was to allow users to watch a query progress and then kill the computation as soon as they are happy with the result, saving human and computer time.

4.3.2 Distributed OLA: Why is it difficult?

If we restrict ourselves to single-table selection queries (possibly with \texttt{GROUP BY} clauses) in a non-distributed environment, the statistical methods and analysis needed to make OLA work are surprisingly simple. Part of this simplicity is due to the fact that, in a centralized environment, one can reasonably assume that the time required to process the atomic data units that are being sampled (whether they be individual tuples or blocks) is constant. Thus, at any time instant, the set of data units processed can be taken as a random sample of the entire database. In a distributed environment, this assumption is of questionable validity.

Consider the problem of implementing OLA in a MapReduce-like computation [100], run over data stored in the Hadoop distributed file system (HDFS) [19]. In HDFS, a large data set will comprise hundreds or thousands of files, distributed around a compute cluster. These files can vary widely in size and may also be distributed to machines in a non-random way, especially if the compute cluster grows over time through the addition of additional machines. To run OLA, one might imagine assigning waves of mappers randomly to the
files in the directory. When a mapper finishes processing a file, it sends statistics that it has collected on the file to a central coordinator for processing, who uses all of the statistics collected thus far over all of the files to produce an estimate for the final query result. Clearly the assumption of uniform processing time is unreasonable in this setting. Some mappers will be run on slower machines, some mappers will operate over larger files. Some of the data will be fetched locally, some from the same rack, and some fetched from machines on a different rack altogether.

Consequently, when OLA estimate is computed at a given time point, the set of files that have been processed may be quite non-random, especially early on during the OLA computation, even when files are assigned randomly to mappers. The reason is that at a given instant, the set of files currently being processed is more likely to include those that take longer to process; hence the set of files that were used to compute the current OLA estimate is more likely to contain those that take a shorter time to process. If the processing time for a block is correlated with the aggregate computed over a file, estimation bias will result. This phenomenon is sometimes referred to as the inspection paradox [17]. It is not hard to imagine a case where such bias exists: if the goal is to compute a \( \text{SUM} \) aggregate, then files with more data take longer to process and also have larger \( \text{SUMs} \). Since larger blocks are underrepresented in the set of blocks that have been processed early on during an OLA job, the running OLA estimate for a \( \text{SUM} \) query over data with highly varying file sizes underestimates the final result in the early stages of the computation.

### 4.3.3 A Bayesian Model for OLA

Pansare et al. [35] proposed handling this bias using Bayesian methods. The Bayesian approach to OLA is quite natural. Suppose the goal is to provide online estimates to the result of an aggregation query of the form
SELECT SUM expr 
FROM myTable

Denote by \( x_j \) the value of \( expr \) for the \( j \)th tuple, so that the query result can be written as \( Q = \sum_{j=1}^{n} x_j \). Before processing begins, we have not yet inspected any of the \( x_j \), and so are uncertain of their value. The Bayesian approach associates an initial probability distribution \( x_j \sim p(x) \) with each \( x_j \) that quantifies our preliminary uncertainty about \( x_j \) values. When we observe \( x_j \), our uncertainty distribution becomes a degenerate probability distribution concentrated at the observed value; that is, there is no more uncertainty. During query processing, we continually update our probability distribution for \( x_j \) as we obtain more and more information, for example, when we observe a data value \( x_j \) that is correlated with \( x_j \).

It is easy to see that our abstraction \( Q \) encompasses a much broader class of queries than shown above. Indeed, even the foregoing query can encode a \texttt{WHERE} clause by letting \( expr = 0 \) for any tuple that is not selected by the predicate. \texttt{GROUP}\textsc{ing} can be encoded by essentially having a separate query per group, where \( expr = 0 \) for any tuple is that not a member of the group. \texttt{myTable} itself might be the result of a \texttt{GROUP BY} query over some other table. Thus, without loss of generality, an \( x_j \) can represent a function of a single tuple or a function over a horizontal partition of a table, a physical HDFS block, or an entire HDFS file; the models and math are the same. Moreover, our results can be extended without too much difficulty to queries \( Q \) that use other aggregation operators besides \texttt{SUM}.

Given this setup, Pansare et al. propose the use of a Markov chain Monte Carlo (MCMC) algorithm, specifically, Gibbs sampling, to sample possible values of the complete data set \( Z = \{x_1, x_2, \ldots, x_n\} \), including those blocks that have not been processed yet. The query \( Q \) can then be computed over those sampled data sets, which gives us an empirical probabil-
ity distribution of possible values for $Q$. By taking, e.g., the middle 95% of this empirical
distribution, confidence bounds can be produced.

In standard Bayesian parlance, the initial distribution $p(x)$ is called the *prior distribu-
tion* on the $x_j$ values, which in turn gives us a prior on $Q$. After some portion of $Z$ has
been processed (we call this subset $V$), we update $p(x)$ to obtain $p(x|V)$. This is our *post-
terior distribution* on the database blocks, and in turn, gives us a posterior distribution on
$Q$, which is then presented to the user.

The Bayesian approach makes particular sense if we want to counteract bias due to
variance in processing time. Pansare et al. propose modeling not only the distribution of
each $x_j$, but rather the joint distribution over the $x_j$ values and the times required to process
these values. Processing time can be taken into account to counteract bias. If, for example,
data objects that take longer to process tend to have larger $x_j$ values, then a block that has
not completed processing but has been with a mapper for a very long time will likely have
a larger $x_j$ value, as captured by the joint distribution.

In more detail, Pansare et al. view the complete data set $Z$ as comprising not $x_j$ values,
but rather $Z_j$ vectors of the form

$$Z_j = \langle x_j, t_{scht}^{j,1}, t_{loc}^{j,1}, t_{rack}^{j,1}, t_{dist}^{j,1},$$
$$t_{loc}^{j,2}, t_{rack}^{j,2}, t_{dist}^{j,2}, \ldots,$$
$$t_{loc}^{j,m}, t_{rack}^{j,m}, t_{dist}^{j,m} \rangle.$$ 

The vector has the following components:

1. $x_j$ is the value that is obtained when the block is aggregated.

2. $t_{scht}^{j}$ is the time required to schedule the block, once it has first been selected for
   scheduling.
3. $t_{j,k}^{\text{loc}}$ is the time taken to actually process the block by a mapper on machine $k$, given that the block is to be read locally from machine $k$.

4. $t_{j,k}^{\text{rack}}$ is the time taken to process the block by a mapper on machine $k$, given that the block is to be read from a machine on the same rack as machine $k$.

5. $t_{j,k}^{\text{dist}}$ is the time taken to process the block by a mapper on machine $k$, given that the block must be read from a machine on a different rack.

A Normal($Z_j|\mu, \Sigma$) prior over $Z_j$ is used; note that $\mu$ and $\Sigma$ are also treated as random in the Bayesian framework. At each instant during the OLA, the state of the computation splits each $Z_j$ into an ordered pair ($V_j$, $U_j$), where $V_j$ is the observable part, and $U_j$ is the unobservable part. For example, if $Z_j$ corresponds to a data block and the block as not yet been processed, then $V_j$ is empty and $U_j = Z_j$. If, on the other hand, the block is currently being processed, a map task has been assigned to the block and processing has begun. Thus, $V_j$ contains an exact value for $t_{j}^{\text{sch}}$. The vector $V_j$ also contains $[t_{j}^{k}]$, which is a lower bound on the processing time for the block on a particular machine $j$ for some $k$ in $\{\text{loc, rack, dist}\}$. Given this setup, an MCMC algorithm (or another Bayesian inference framework) can be used to estimate the distribution $p(Z|V)$, which in turn can be used to produce confidence bounds on $Q$.

### 4.3.4 OLA for GD: Challenges

There are two key technical challenges when attempting to adapt distributed OLA to the problem of distributed GD.

1. In the aforementioned work on distributed OLA, it was assumed that the $x_j$ values being aggregated are one-dimensional. In GD, where $x_j = \nabla L(t_j|\Theta_i)$, the $d$-dimensional parameter vector $\Theta_i$ often has hundreds, thousands, or even millions of
entries; hence, the OLA model and computation must be modified to handle multi-dimensional data.

2. The output of an OLA algorithm is an estimate plus confidence bounds. It is traditionally assumed that a user is watching the confidence bounds, and stops the OLA computation when the bounds are tight enough. When OLA is used for GD, these bounds must be used to automatically decide when the estimate is good enough to end the current epoch. The OLA-GD software must take into account the cost associated with ending the current epoch and starting a new one, and use this information so as to end an epoch in a way that attempts to minimize the time until convergence of the overall computation. This is a challenging problem.

We address these issues in the next two sections of the chapter.

4.4 Modifying OLA for GD

4.4.1 Simplifying the Model

In standard Bayesian OLA, $x_j$ is modeled as a scalar-valued random variable, e.g., the value of a scalar expression $expr$ evaluated on the $i$th tuple in a table. In OLA-GD, $x_j = \nabla F(t_j|\Theta)$ is a vector-valued random variable, whose dimensionality is equivalent to $d$, the number of parameters in $\Theta$. This is problematic: if we try to directly use the model from the previous section (where each $Z_j$ has $3m + 2$ values when there are $m$ machines in the system) and we have a $\text{Normal}(Z_j|\mu, \Sigma)$ prior over $Z_j$, then the covariance matrix $\Sigma$ can potentially have up to $\binom{3m+1+d}{2}$ unique entries to estimate. If $m$ is 100 and $d$ is 1000, this is around 850,000 entries. For $m = 100$, $d = 10,000$ this is 53 million entries. For tractability, this number must be reduced.
Fortunately, we do not need to model correlations between processing times on different machines, since, in practice, we never observe the same block being processed on different machines. This reduces the number of correlations significantly, to \((3m + 1)d + \binom{d}{2}\), or around 800,000 in our \(m = 100, d = 1,000\) example.

We can reduce the number of correlations further by collapsing the processing time and the scheduling time to a single value. Moreover, we can consider ignoring the differences in processing time on different machines, as well as the difference in processing time observed depending upon whether the block being processed is local to the machine, local to the rack, or on a remote machine. With this simplification, each \(Z_j\) has the form \((x_j, t_j)\), and there are only \((d+1)\) correlations in \(\Sigma\). This collapsing may be justified by arguing that in the case when many blocks are processed, the differences between machines and block locations “average out”. Indeed, in our experience on Amazon EC2, the machines in a rented cluster have few (if any) differences, and the network is very fast, meaning that the biases from block location diminish importance.

Even with the above simplification, \(\Sigma\) may still contain too many nonzero correlations when \(d\) is large. We therefore simplify the problem even more by assuming zero correlation between each pair of dimensions in \(\nabla L(t_j|\Theta_i)\). This assumption of independence between various update dimensions is not going to be true in every case. The “walk” performed by a GD algorithm may tend to update the various parameters in a correlated fashion. However, this will not affect the model’s ability to counteract the inspection paradox because model will still be able to adequately take into account the statistical relationship between processing time and aggregate value, which is usually the main source of error. The obvious benefit of this approach is that the number of correlations in \(\Sigma\) is then
only \( d \), and \( \Sigma \) takes the form:

\[
\begin{pmatrix}
\sigma_1^2 & 0 & 0 & 0 & \ldots & \rho_1 \sigma_1 \sigma_t \\
0 & \sigma_2^2 & 0 & 0 & \ldots & \rho_2 \sigma_2 \sigma_t \\
0 & 0 & \sigma_3^2 & 0 & \ldots & \rho_3 \sigma_3 \sigma_t \\
0 & 0 & 0 & \sigma_4^2 & \ldots & \rho_4 \sigma_4 \sigma_t \\
\vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\
\rho_1 \sigma_1 \sigma_t & \rho_2 \sigma_2 \sigma_t & \rho_3 \sigma_3 \sigma_t & \rho_4 \sigma_4 \sigma_t & \ldots & \sigma_t^2
\end{pmatrix}
\]

Here, \( \sigma_j^2 \) for \( j \leq d \) denotes the variance in the \( i \)th component of the gradient, and \( \sigma_t^2 \) denotes the variance of the block processing time. \( \rho_j \) denotes the correlation between the \( j \)th component of the gradient and the block processing time.

Under this simplified regime, at a particular time instant when we are attempting to estimate the gradient \( Q = \sum_j^n x_j \), there are three cases for an \( x_j \) that is computed over a block of data:

1. The block has been fully processed; here, \( V_j = Z_j = \langle x_j, t_j \rangle \) since we know both the aggregated value of the block, as well as the processing time.

2. The block is currently being processed; here \( V_j \) contains \( [t_j] \), since we know the lower bound on the processing time—this is taken as the time we’ve waited so far for the block to process.

3. Finally, in the case that the block has yet to be processed at all, \( V_j \) is empty and \( U_j = Z_j \).

To complete the model, we must assign prior probability distributions to \( \mu \) and \( \Sigma \). Following standard practice, we assume that \( \mu_t \sim \text{Normal}(\mu_{0,t}, \sigma_{0,t}^2) \) and \( \sigma_t^2 \sim \text{InvGamma}(1,1) \). Moreover, we assume for \( 1 \leq k \leq d \) that \( \mu_k \sim \text{Normal}(\mu_{0,k}, \sigma_{0,k}^2) \), \( \sigma_k^2 \sim \text{InvGamma}(1,1) \),
and \((\rho_k + 1)/2 \sim \text{Beta}(1, 1)\). Here \(\mu_{0,t}, \sigma_{0,t}^2\), and \(\{\mu_{0,k}, \sigma_{0,k}^2\}_{1 \leq k \leq d}\) are fixed “hyperparameters” of the model, as are the parameters of the Inverse Gamma and Beta distributions, all of which are equal to 1.

The problem is then to estimate the posterior distribution \(P(Z|V)\). Practically speaking, this involves estimating each \(Z_j\), as well as the parameters \(\mu\) and \(\Sigma\) for the Normal(\(\mu, \Sigma\)) prior on each \(Z_j\). The exact mechanism for doing this is considered in the next subsection.

### 4.4.2 Estimating the Model Parameters

We use Gibbs sampling for estimating \(P(Z|V)\). Briefly, a Gibbs sampler is a Markov chain Monte Carlo (MCMC) algorithm for drawing samples from some target joint probability density function (PDF) whose value can be evaluated up to a constant term. The procedure is as follows. Assume that \(Y = (y_1, y_2, \ldots, y_m)\) is a vector of random variables. To generate \(k\) samples from \(P(Y)\), the Gibbs sampler proceeds as follows:

1. Select an initial value \(y^{(0)} = (y_1^{(0)}, y_2^{(0)}, \ldots, y_m^{(0)})\).

2. For each \(i = \{1 \ldots k\}\), choose some subset \(Y'\) of \(Y\). Then update each \(\{y_j^{(i)} \text{ s.t. } y_j \in Y'\}\) by sampling from

   \[P(y_j^{(i)} \text{ s.t. } y_j \in Y'|y_j^{(i)} \text{ s.t. } y_j \in Y \setminus Y').\]

The most common Gibbs sampler repeatedly cycles through the \(m\) vector entries, updating exactly one entry at each step. After an initial “burn-in” period, the probability density of the sampled values is approximately equal to the target density. In the case of OLA for GD, the set of variables \(Y\) to estimate via a Gibbs sampler includes the hidden data \(U\) as well as the entries of the covariance matrix \(\Sigma\) and the mean vector \(\mu\).

Since we are running a distributed GD algorithm, we must run our Gibbs sampler for \(Y\) in a distributed environment. In our distributed implementation, one particular node is
designated as the coordinator, and is responsible for managing the overall GD computation (see Section 4.6 for more details on our Hadoop-based implementation). A number of workers are tasked with scanning the data blocks in random order, computing \( x_j \) over each, and periodically communicating back to the coordinator the \( \langle x_j, t_j \rangle \) values computed from the blocks that have been fully processed, as well as the \(|t_j|\) values for the blocks that have only been partially processed. After receiving this data, the coordinator uses the completed \( x_i \)'s and corresponding \( t_i \)'s to incrementally update the various statistics needed by the Gibbs sampler—see Section 4.7.1 for a mathematical description of the sampler—and then discards the \( \langle x_j, t_j \rangle \) pairs. Finally, the coordinator runs multiple iterations of the sampler, generating sampled \( x_j \) values for each block that has not yet completed processing, as well as sampled values for \( \Sigma \) and \( \mu \).

Of course, we are ultimately interested in the posterior distribution for the gradient \( Q \). After a suitable burn-in period, the current set of values \( Z \) can be used to used to generate a sampled realization \( \hat{Q} \) for \( Q \) as follows. First, let \( \hat{Q}_{\text{done}} \) denote the sum \( \sum_j x_j \) for the blocks that have completed processing (the vector \( \hat{Q}_{\text{done}} \) is maintained as an aggregate value by the coordinator). Second, let \( \hat{Q}_{\text{proc}} \) denote the sum \( \sum_j x_j \) for the blocks that are currently being processed; the constituent \( x_j \) values are obtained by examining the current state of the \( x_j \) values in the set \( Y \) that are being maintained by the Gibbs sampler. Finally, let \( \hat{Q}_{\text{unproc}} \) denote the sum \( \sum_j x_j \) for the blocks where no information is known because they are still unprocessed and untouched; assume that there are \( m \) such blocks. This value is not maintained by the Gibbs sampler but we know that, given \( \mu \) and \( \Sigma \), each \( x_i \) has a Normal\((\mu, \Sigma)\) distribution\(^\dagger\), so it follows that \( P(Q_{\text{unproc}}|\mu, \Sigma) \) will have a Normal\((m\mu, m\Sigma)\) distribution. Thus, it is easy to generate a sampled value for \( \hat{Q} \) as \( \hat{Q} = \)

\(^\dagger\)More precisely, each \( x_j \) will have a Normal\((\mu, \Sigma)\) distribution if we discard the last entry in \( \mu \) and the last row and column in \( \Sigma \), as these deal with the quantity \( t \).
\( \hat{Q}_{\text{done}} + \hat{Q}_{\text{proc}} + \hat{Q}_{\text{unproc}} \). By running the sampler for many iterations and generating many values for \( \hat{Q} \), I obtain an approximation for the posterior distribution of \( Q \).

4.5 Ending an Epoch

4.5.1 Overview

In Section 4.4 I showed how to construct running Bayesian estimates of the gradient during an epoch of distributed GD; more precisely, the Bayesian OLA technique provides running estimates of the distribution of the gradient. In this section, I address the key technical challenge of using this information to decide when to terminate the current epoch. As discussed previously, possibilities range from stopping early, after only a tiny fraction of the data have been processed, running each epoch to completion, or stopping at some point in between.

In what follows, we describe a principled stopping framework predicated on (approximately) maximizing the expected velocity at which the GD random walk travels toward \( \Theta^* \); that is, we seek to maximize the change in distance to \( \Theta^* \) per unit of time. The fundamental challenge is to estimate the expected distance to \( \Theta^* \) at various time points during the gradient computation. To this end, we show in Section 4.5.2 how the distance and velocity can be computed from a high-level statistical model of the overall GD random walk. This model is developed in detail in Section 4.5.3. Given the resulting method for estimating velocity, we provide some stopping rules in Section 4.5.4.

4.5.2 Estimating the Velocity

In Bayesian fashion, the OLA model of the prior section can be used to obtain a posterior distribution over the true gradient \( Q_i = \sum_j \nabla F(t_j|\Theta_i) \). Since \( \Theta_{i+1} = \Theta_i - Q_i \) for each \( i \),
it follows that, after observing $\Theta_i$, we can directly obtain the posterior distribution of the parameter vector $\Theta_{i+1}$ that would result if we were to run the epoch to completion. We might choose to end the epoch sooner, however. In any case, ending the epoch requires choosing a particular point estimate $\hat{\Theta}_{i+1}$ for $\Theta_{i+1};$ a reasonable choice is the mean of the posterior distribution of $\Theta_{i+1}$. This step incurs a random error of $\epsilon_{i+1} = \Theta_{i+1} - \hat{\Theta}_{i+1}$. The distribution of $\epsilon_{i+1}$ can easily be determined, since $P(\Theta_{i+1}|\cdot)$ is known and $\hat{\Theta}_{i+1}$ is an observed value (hence treated as a constant); see Figure 4.1(a). I will subsequently denote by OLAError$(i + 1)$ the posterior distribution of $\epsilon_{i+1}$.

Thus we can obtain the posterior distribution of the distance $||\epsilon_{i+1}||$ from $\hat{\Theta}_{i+1}$ to $\Theta_{i+1}$. An additional statistical model is needed, however, to further obtain the distribution of the distance $||\Delta_{i+1}||$ from $\Theta_{i+1}$ to $\Theta^*$, where $\Delta_{i+1} = \Theta^* - \Theta_{i+1}$. This model, which is detailed in the next section, is a Bayesian model that takes into account all of the (approximate) gradient steps taken thus far to estimate generally where the GD algorithm is going to end up, thus yielding the posterior distribution $P(\Delta_{i+1}|\cdot)$.

Supposing that we have the above distributions for $\epsilon_{i+1}$ and $\Delta_{i+1}$ in hand, we can estimate the velocity as follows. The distance to $\Theta^*$ after epoch $i + 1$ ends is $d_{i+1} = ||\Theta^* - \hat{\Theta}_{i+1}|| = ||\Delta_{i+1} + \epsilon_{i+1}||$. Similarly, the distance to $\Theta^*$ after epoch $i$ ends is $d_i = ||\Delta_{i+1} + \epsilon_{i+1} + (\hat{\Theta}_{i+1} - \Theta_i)||$. Since $\hat{\Theta}_{i+1} - \Theta_i$ can be treated as a known quantity upon termination of epoch $i + 1$ and since the distribution of $\Delta_{i+1} + \epsilon_{i+1}$ can be obtained by convolution, the distributions and expected values of $d_i$ and $d_{i+1}$ can be computed. The average velocity during epoch $i + 1$ is then given by $v_{i+1} = (d_{i+1} - d_i)/l_{i+1}$, where $l_{i+1}$ is the observed length of epoch $i + 1$; we can obtain the distribution and expected value of $v_{i+1}$ from the distributions and expected values of $d_i$ and $d_{i+1}$.

In practice, we convolve $P(\Delta_{i+1}|\cdot)$ and $P(\epsilon_{i+1}|\cdot)$ via Monte Carlo sampling, as illustrated in Figures 4.1 and 4.2. Figure 4.1 depicts three samples taken from each posterior.
Figure 4.2 shows how the Monte Carlo samples are used to compute realizations of the various distances, as well as how they are used to estimate the distance traveled during the current epoch. I next describe our high-level statistical model of the GD random walk; MCMC techniques applied to this model yield the samples from the distribution of $\Delta_{i+1}$ that are needed to estimate the velocity.

### 4.5.3 Modeling the GD Walk

The approach outlined thus far requires a model for $\Delta_{i+1}$. At first glance, developing such a model seems nearly as difficult as solving the original optimization problem itself. After all, $\Delta_{i+1}$ represents the difference between the problem’s solution $\Theta^*$ and the set of model parameters $\Theta_{i+1}$. If we actually had a good idea where the value of $\Theta^*$ lay, we could jump there immediately. Indeed, there exists a class of optimization methodologies that use Bayesian methods to model the loss surface using “Kriging”—see, e.g. [115]—with the goal of doing just that.

However, we are asking a lot less of our model. Because we are not trying to use our model for $\Delta_{i+1}$ to immediately jump to $\Theta^*$—we are instead using it in an attempt to determine the relative values of $d_i$ and $d_{i+1}$—we can tolerate a lot of uncertainty and error. In fact, as long as we generally get the direction of $\Delta_{i+1}$ somewhat correct, the model will still be quite useful for determining whether to stop. Even a large error in the magnitude of $\Delta_{i+1}$ will tend to add a similar error term to both $d_{i+1}$ and $d_i$; since we are interested in the velocity $(d_{i+1} - d_i)/l_{i+1}$, these errors tend to cancel out.

Thus, rather than trying to model the loss surface using a method such as Kriging (which can be very difficult), we instead decide to model the walk of the GD algorithm itself. After all, we know that the GD algorithm is making its way towards $\Theta^*$. In standard Bayesian fashion, we first propose a generative, stochastic process that we imagine could be used
to generate the GD walk, and then we fit that process to the observed walk to obtain the posterior distribution. We use a simple Gaussian mixture model (GMM) as the heart of our stochastic process. We assume that the gradient step $Q_i = \Theta_i - \Theta_{i+1}$ is sampled from this GMM, with the slight modification that the distance traveled is assumed to decrease over time, as the GD converges to $\Theta^*$; convergence speed is controlled by a parameter $\beta$.

In this generative process, the OLA error $\epsilon_i$ is still sampled from the posterior OLA error distribution, as described in the previous section. Since the process is recursive, this defines a random walk which converges to $\Theta^*$.

The precise generative process for the model is described as Algorithm 3. For computational tractability, we assume that the mixture components are mutually independent, so that the correlation matrix is diagonal. Thus, besides $\beta$, the key parameters of the model are the vector $\pi$ of mixture probabilities and the $d$-dimensional vectors $\mu_k$ and $\sigma_k^2$ that specify the mean and variance of the $k$th Gaussian mixture component for $1 \leq k \leq q$, where $q$ is the number of mixture components. The prior distributions on the parameters are specified in the first part of Algorithm 3. The fixed hyperparameters of the model are $q$, $\mu_0$, $\sigma_0^2$, and $\alpha$ (as well as the arguments to the initial Beta, Dirichlet, and Inverse Gamma distribution, all of which are equal to 1).

Given this process, we then obtain $P(\Delta_{i+1}|.)$ as follows. First, we use the sequence of gradient steps taken so far, i.e., $\hat{\Theta}_1, \hat{\Theta}_2, \ldots, \hat{\Theta}_{i+1}$, as well as the OLAError(.) distribution (estimated as described in Section 4.4) to generate a candidate set of values for the random walk parameters $\beta, \pi$, and so on using a Gibbs sampler (the details of Gibbs sampler for this model are given in Section 4.7.2). Once this has been done, a sample from $P(\Delta_{i+1}|.)$ may be obtained by by using those candidate parameters to run Algorithm 3 forward stochastically from iteration $i + 1$, with $\Theta_{i+1} = (0, 0, \ldots, 0)$. The value of $\Theta_\infty$ can then be taken as a sample from $P(\Delta_{i+1}|.)$. Note that since the expected distance of each gradient step in
Algorithm 3: Generative Process of Random Walk Model

Gradient attenuation factor, $\beta \sim \text{Beta}(1, 1)$;
Mixture proportion, $\pi \sim \text{Dirichlet}(\alpha)$;

foreach component $k$ do

$\mu_k \sim \text{Normal}(\mu_0, \text{diag}(\sigma^2_0))$;

foreach dimension $d$ do

$\sigma^2_{k,d} \sim \text{InvGamma}(1, 1)$

foreach epoch $i$ do

Mixture-component assignment, $z_i \sim \text{Multinomial}(\pi, 1)$;
Gradient step $Q_i \sim \beta^i \text{Normal}(\mu_{z_i}, \text{diag}(\sigma^2_{z_i}))$;
$\Theta_i = \Theta_{i-1} - Q_i$;
$\epsilon_i \sim \text{OLAError}(i)$;
$\hat{\Theta}_i = \Theta_i - \epsilon_i$;

Algorithm 3 decays exponentially, in practice the simulation converges to $\Delta_{i+1}$ in only a few hundred to a few thousand simulated steps.

4.5.4 Stopping Rules

Suppose that epoch $i + 1$ is underway, and that $t$ time units have elapsed since the start of the epoch. The results in the previous two sections allow us to estimate the velocity $v_{i+1}(t)$ toward $\Theta^*$ for the epoch, assuming that we stop the epoch right now. In what follows, I develop a simple rule of thumb for deciding when to end an epoch. The first step is to understand what a velocity profile during an epoch looks like under our GD random walk model.
Velocity Profiles

We first consider a simple scenario in which there is no startup cost associated with an epoch. Because $\hat{\Theta}_{i+1}$ is merely an estimate of $\Theta_{i+1}$, if the error $\epsilon_{i+1} = \Theta_{i+1} - \hat{\Theta}_{i+1}$ has high variance, then the GD random walk may well go off in the wrong direction, so that the distance to $\Theta^*$ either decreases by only a very small amount, or perhaps even increases during epoch $i + 1$.

In a two-parameter learning problem, for example, suppose that $\Theta_i = \langle 0, 0 \rangle$, the estimate $\hat{\Theta}_{i+1}$ is at $\langle 2, 2 \rangle$, each component of $\epsilon_{i+1}$ has a Normal$(0, \sigma^2)$ distribution, and each component of $\Delta_{i+1} = \Theta^* - \Theta_{i+1}$ has a Normal$(2, 4)$ distribution. Thus, if both $\epsilon_{i+1}$ and $\Delta_{i+1}$ take on their expected values, we move from $\langle 0, 0 \rangle$ to $\langle 2, 2 \rangle$, with $\Theta^*$ at $\langle 4, 4 \rangle$, and so the distance traveled is $\sqrt{8}$. Of course, $\epsilon_{i+1}$ and $\Delta_{i+1}$ will not always take the values $\langle 0, 0 \rangle$ and $\langle 2, 2 \rangle$. The effect of increasing the standard deviation $\sigma$ of the error $\epsilon_{i+1}$—i.e., increasing the variability of our estimate $\hat{\Theta}_{i+1}$—is plotted in Figure 4.5.4 for our example. As $\sigma$ increases, the expected distance traveled in the epoch drops quickly. Conversely, as $\sigma$ decreases, the expected distance traveled increases.

As time passes, more and more data arrive, which causes $\sigma^2$ to drop and the expected distance traveled to increase. This increase, however, tapers off as arriving data items decrease $\sigma$ by ever smaller amounts. Indeed, the standard deviation of a sampling-based estimator generally decreases as $O(m^{-1/2})$, where $m$ is the number of data items processed; this function initially drops steeply but then flattens out. The resulting behavior of expected distance traveled as a function of time is plotted in Figure 4.5.4. The velocity toward $\Theta^*$ at each time point is the slope of this curve. As can be seen, the velocity starts out at its greatest value initially, and then falls over time. This behavior illustrates why SGD is preferred in a single-processor environment where the start-up cost is negligible. In a distributed environment, where there can be a significant start-up cost associated with an
epoch, the curve looks quite different, with a relatively flat segment preceding the rapid increase in distance traveled. This segment corresponds to the period when either there is no data yet, or data is just starting to trickle in.

**Approximately Optimal Stopping**

For distributed environments, we typically expect the velocity profile to be a unimodal curve that increases from 0 and then eventually decreases to 0, as discussed above. If the epoch is terminated too soon, then insufficient progress will be made; if the epoch runs on too long, then the marginal contribution of the arriving data is small, and time would be better spent on starting up the next epoch. To roughly determine the optimal trade-off, we reason as follows. Suppose at first that we are at some specified distance $z$ from $\Theta^*$, and that we plan to reach $\Theta^*$ in exactly $n$ epochs ($n \geq 1$). Denote by $t_i$ the time spent in the $i$th epoch, by $d_i(t)$ the (expected) distance traveled after the epoch has been underway for $t$ time units ($t \geq 0$), and by $v_i(t)$ the corresponding instantaneous velocity function, i.e., the derivative of $d_i$. The optimization problem is to minimize the total travel time $t_1 + t_2 + \cdots + t_n$ subject to the constraint that $\sum_{i=1}^{n} d_i(t_i) = z$. Since we don’t know exactly what future epochs will look like, we make the simple approximation that, for some function $d$ and some $\beta \in (0, 1]$, the distance function is given by $d_i(t) = \beta^{i-1}d(t)$ for $1 \leq i \leq n$. This model is consistent with the GD random walk model, where we assume that the gradient (and hence distance covered) decreases by a factor of $\beta$ at each epoch. It follows that $v_i(t) = \beta^{i-1}v(t)$, where $v$ is the derivative of $d$. To ensure feasibility, we assume that $\sum_{i=1}^{n} \beta^{i-1}d_{\text{max}} \geq z$, where $d_{\text{max}} = \lim_{t \to \infty} d(t)$. This implies that

$$n \geq \log\left[1 - \left(1 - \beta z\right)/d_{\text{max}}\right]. \quad (4.1)$$
For $t = (t_1, t_2, \ldots, t_n)$, the Lagrangian is given by $L(t, \lambda) = \sum_{i=1}^{n} t_i - \lambda(\sum_{i=1}^{n} d_i(t_i) - z)$. Equating partial derivatives to 0, we find that the optimal times are given by

$$t_i^* = v_i^{-1}(\gamma) = v^{-1}(\gamma/\beta^{i-1}) \quad (4.2)$$

for $1 \leq i \leq n$, where $v^{-1}$ is the inverse of $v$ and $\gamma = 1/\lambda$. Note that $v^{-1}$ is not unique for a unimodal curve. Second order considerations—i.e., the form of the “bordered Hessian” matrix for our constrained optimization problem—imply that $v_i'(t_i^*) \leq 0$, so that we take the inverse point that lies to the right of the point where $v$ achieves its maximum. Thus, to obtain the optimal epoch lengths, we choose $\gamma^*$ as the solution to

$$\sum_{i=1}^{n} \beta^{i-1} d(v^{-1}(\gamma/\beta^{i-1})) = z$$

and then set $t_i^* = v^{-1}(\gamma^*/\beta^{i-1})$ for each $i$. As might be expected, the optimal GD walk spends more time in epochs where the achievable velocities are higher. An outer optimization loop then determines the best value of $n$, subject to (4.1). Note that the important quantity here is $t_1^* = v^{-1}(\gamma^*)$, the stopping time for the current epoch, which has been derived assuming that we will behave optimally from the next epoch onward (and also assuming that the shape, though not the scale, of the velocity profile will look the same at future epochs). At the next epoch we repeat the analysis, using our updated values of $\beta$, $v$, and $d$ from the GD walk model.

A simpler rule of thumb that is much easier to implement can be obtained by considering the case $\beta \approx 1$ (so that the velocity profiles are relatively constant). Making the (admittedly crude) assumption that $\beta = 1$, we see from (4.2) that $t_1^* = \cdots = t_n^*$; denote by $t^*$ this common value. With this simplification, the optimization problem is now to minimize $nt^*$, subject to $nd(t^*) = z$ (where we assume that $z \leq nd_{\text{max}}$). That is, we wish to minimize $zt^*/d(t^*)$. Setting the derivative equal to 0 yields $v(t^*) = d(t^*)/t^*$. That is, the epoch should end when $\bar{v}(t) = d(t)/t$, the average velocity so far, equals the instantaneous
velocity $v(t)$. This is the time point at which the average velocity is maximized (since we have minimized $1/\bar{v}$), and our previous considerations imply $t^*$ will occur sometime after the instantaneous velocity has peaked. Note that this result holds for any value of $n$, so we can avoid the search for optimal $n$. I have found this rule of thumb to be robust, and use it in our implementation.

### 4.6 Hadoop Implementation

While it would clearly be possible to do better than a Hadoop-based implementation in terms of raw performance, our primary goal was a reasonable, scalable high-quality prototype, and Hadoop met those basic needs.

There are four key software components in our implementation: the coordinator, the mappers, the server and the model. The coordinator manages the workflow, assigning data to the workers (mappers) that implement gradient descent epochs. Each mapper computes a gradient over an assigned HDFS data block and sends the $\nabla L(t_j|\Theta_t)$ vector back to the server via HTTP; the server collects these gradients and periodically invokes the model. The model implements Sections 4.4 and 4.5. If the model decides to stop the current epoch, it conveys this to the server, supplying the server with the value $\Theta_{t+1}$. I will now discuss some of the details of our implementation of these components.

#### 4.6.1 The Coordinator

The coordinator is tasked with managing the Hadoop job associated with a submitted parameter-learning problem. This involves making sure that the various database blocks are processed in random order during each epoch. It is widely understood that Hadoop can be slow when used to implement iterative computations (such as GD), the problem being that if each iteration requires starting up a new job, a huge amount of time can be wasted in
the job startup (somewhere between 20 and 30 seconds in our experimental setup). To get around this, rather than scheduling all of the $n$ database blocks in a random order, waiting for an epoch to end, then killing the job and starting a new job, the coordinator instead builds a schedule that is $k$ times larger than $n$ by sampling from $T$ with replacement. So if $T = \{t_1, t_2, t_3\}$, the schedule submitted by the coordinator for processing if $k = 3$ might be:
\[
\langle t_2, t_1, t_3, t_1, t_2, t_3, t_1, t_3, \rangle
\]
This is implemented by overriding the `getSplits()` method of Hadoop’s `InputFormat` class. In this way, when an epoch ends, there is no need to schedule a new job. The same job can continue, and the same stream of random blocks can be used to fuel subsequent epochs. The potential drawback of this approach is that we are approximating a finite-population sampling scheme with an infinite-population one, but as long as the fraction of blocks processed in an epoch is small, this is of little consequence.

### 4.6.2 The Mapper

We use a standard Hadoop mapper to compute $\nabla L(t_j|\Theta_i)$ for each block. At startup, the mapper reads data that are constant for the life of the GD (the number of clusters in a Gaussian Mixture Model, for example) from the Hadoop job description. The current parameter vector $\Theta_i$ is requested via HTTP from the OLA-GD server. Then the mapper begins processing its block. At completion, the mapper reports the value $\nabla L(t_j|\Theta_i)$ to the server when it finishes. As far as Hadoop is concerned, however, the output of the mapper is empty—each mapper communicates its results and status directly with the OLA-GD server.
4.6.3 The OLA-GD Server

The two main responsibilities of the server are responding to messages from the mapper and invoking the model at regular time intervals. The server responds to following types of messages sent by the mapper:

1. **MAP\_START\_TIME**: Before reading an input split, mapper sends this message along with time to indicate that it has started processing the block. The model will use this start time to determine \([t_j]\).

2. **MAP\_DATA**: After computing the gradient over the input block, the mapper sends \(\nabla L(t_j|\Theta_i)\) and \(t_j\) back to the server.

3. **GET\_DATA**: The mapper sends this message as it starts up; the server responds with \(\Theta_i\).

4.6.4 The Model

Periodically, the server invokes the model of Sections 4.4 and 4.5. It sends the data and metadata it has collected to the model, which performs the relevant computations. Based on the results of the statistical computations, the model sends one of the following messages back to the server:

1. **STOP\_EPOCH**: The model provides the server an estimate of the parameter \(\Theta_{i+1}\). The server will then ignore any subsequent **MAP\_DATA** message for the current epoch and will send \(\Theta_{i+1}\) in response to any **GET\_DATA** request.

2. **CONTINUE\_EPOCH**: The server continues with its normal operations.

3. **DONE**: This message indicates that the OLA-GD computation has converged.
4.7 Gibbs Sampling Details

In this section, I describe the Gibbs sampling techniques that we use for the OLA model of Section 4.4 and for the GD walk model of Section 4.5.3.

4.7.1 OLA Sampler

As discussed in Section 4.4.2, the Gibbs sampler proceeds by repeatedly generating each of the random variables \( t_j, x_j, \mu_t, \sigma_t^2 \), and, for \( 1 \leq k \leq d \), the variables \( \mu_k \) and \( \sigma_k^2 \) according to the conditional distribution obtained by holding all other variables constant. These conditional distribution formulas are displayed below for each variable. Note that the formulas for \( t_j \) and \( x_j \) given below only apply to blocks currently being processed; the formulas for the distributions of \( \mu \) and \( \Sigma \), however, depend on the entire set of variables \( \langle x_1, t_1 \rangle, \ldots, \langle x_n, t_n \rangle \). As discussed in Section 4.4.2, the variables \( \langle x_j, t_j \rangle \) for a block that has finished processing are treated as observed constants, and the variables for a block that has not yet begun processing are sampled from the Normal(\( \mu, \Sigma \)) prior distribution.

After enough iterations, the samples will be approximately drawn from the correct posterior distribution. In particular, the \( x_j \) samples will have approximately the correct posterior distribution, namely that of \( \nabla L(t_j|\Theta_t) \), given the data observed so far; these samples are used for determining when to end an epoch, as described in Section 4.5. In some cases, sampling can be done directly, such as when samples need to be drawn from a Normal distribution. In other cases, the distribution is known only up to a normalizing constant or has a very complex density function. In this case we use a rejection-sampling technique: we enclose the density in a finite bounding box such that (i) virtually all of the area of the curve lies within the box and (ii) the peak of the density curve touches the top of the box. Then we generate points uniformly in the box until a point is obtained that lies under the density curve. It is well known [116] that the horizontal coordinate of an accepted point
has (approximately) the desired density.

**Sampling the processing time $t_j$:**

$$t_j \sim \text{Normal} \left( \mu_t + \sigma_t \sum_{k=1}^{d} \frac{(x_{j,k} - \mu_k)\rho_k}{\sigma_k}, \sigma_t^2 \left\{ 1 - \sum_{k=1}^{d} \rho_k^2 \right\}^2 \mid t_j \geq \lfloor t_j \rfloor \right)$$

**Sampling the gradient dimensions $x_{j,k}$:**

$$x_{j,k} \sim \text{Normal} \left( \mu_k + \frac{\sigma_k}{\sigma_t} \rho_k (t_j - \mu_t), (1 - \rho_k^2)\sigma_k^2 \right)$$

**Sampling mean of processing time $\mu_t$:**

$$\mu_t \sim \text{Normal} \left( \frac{\mu_{0,t}\sigma_t^2 \left\{ 1 - \sum_{k=1}^{d} \rho_k^2 \right\}}{n\sigma_{0,t}^2 + \sigma_t^2 \left\{ 1 - \sum_{k=1}^{d} \rho_k^2 \right\}}, \frac{\sigma_{0,t}^2 \sum_{j=1}^{n} \left\{ t_j - \sigma_t \sum_{k=1}^{d} \frac{(x_{j,k} - \mu_k)\rho_k}{\sigma_k} \right\}}{n\sigma_{0,t}^2 + \sigma_t^2 \left\{ 1 - \sum_{k=1}^{d} \rho_k^2 \right\}} \right)$$

**Sampling the mean of the $k$th gradient dimension:**

$$\mu_k \sim \text{Normal} \left( \frac{\mu_{0,k}(1 - \rho_k^2)\sigma_k^2}{n\sigma_{0,k}^2 + (1 - \rho_k^2)\sigma_k^2}, \frac{\sigma_{0,k}^2 \sum_{j=1}^{n} \left\{ x_{j,k} - \frac{\sigma_k}{\sigma_t} \rho_j (t_j - \mu_t) \right\}}{n\sigma_{0,k}^2 + (1 - \rho_k^2)\sigma_k^2} \right)$$
Sampling the variance of the processing time $\sigma_i^2$:

\[
P(\sigma_i^2 | .) \propto \text{InvGamma}(\sigma_i^2 | 1, 1) \times \\
\prod_{j=1}^{n} \text{Normal} \left( t_j | \mu_t + \sigma_t \sum_{k=1}^{d} \frac{(x_{j,k} - \mu_k)\rho_k}{\sigma_k}, \\
\sigma_i^2 \left( 1 - \sum_{k=1}^{d} \rho_k^2 \right) \right)
\]

Sampling the variances of the $k$th gradient dimension $\sigma_k^2$:

\[
P(\sigma_k^2 | .) \propto (\sigma_k^2)^{-2} \exp(-1/\sigma_k^2) \times \\
\prod_{j=1}^{n} \frac{1}{\sigma_k\sqrt{2\pi(1 - \rho_k^2)}} \times \\
\exp \left( \frac{-(x_{j,k} - \mu_k - \frac{\sigma_k}{\sigma_t} \rho_k (t_j - \mu_t))^2}{2(1 - \rho_k^2)\sigma_k^2} \right)
\]

Sampling the correlation coefficients $\rho_k$: Sampling the correlation coefficient $\rho_k$ between a gradient dimension and the processing time is the most compute-intensive part of the Gibbs sampler. Ignoring for a moment that the resulting covariance matrix must be positive definite (PD), the posterior distribution for correlation coefficient is as follows:

\[
P(\rho_k | .) \propto \text{GenBeta}(\rho_k | 1, 1, 1, 1) \prod_{j=1}^{n} \text{Normal}(x_{j}|\mu, \Sigma) \\
\propto \left( 1 - \sum_{k'=1}^{d} \rho_{k'}^2 \right)^{-\frac{n}{2}} \prod_{j=1}^{n} \exp \left[ -\frac{1}{2} \sum_{k'=1}^{d+1} \left( x_{j,k'} - \mu_{k'} \right) \right] \\
\left( \sum_{k''=1}^{d+1} (x_{j,k''} - \mu_t)(\Sigma^{-1})_{k'',k'} \right) \right].
\]

Note that since $(\rho_k + 1)/2 \propto \text{Beta}(1, 1)$, it follows that $\rho_k \propto \text{GenBeta}(\rho_k | 1, 1, 1, 1)$, where the first two parameters to the Generalized Beta distribution specify the support interval; it is obtained by shifting and scaling the classical Beta distribution function to fit the interval.
Great care has to be taken in designing the rejection algorithm for sampling $\rho_k$. A naive approach encloses each $P(\rho_k|.)$ in a bounding box whose bases extends from $-1$ to $1$, since this is the range of possible values for a correlation coefficient. The algorithm successively generates each correlation using rejection; after all correlations are generated, the resulting $\Sigma$ matrix is tested to see if it is PD; if so, the values $\rho_1, \rho_2, \ldots, \rho_d$ are accepted; otherwise, the procedure is repeated. In practice, the density $P(\rho_k|.)$ is sharply peaked, so that the area under the density curve comprises a very small fraction of the area of the bounding box. Thus very many iterations are required before a sample of a given $\rho_k$ is produced. Compounding the cost is the fact that, as part of the rejection procedure, the density function $P(\rho_k|.)$ must repeatedly be computed for each uniformly generated candidate point in order to check whether the point lies below the density curve. This computation involves computing the matrix inverse $\Sigma^{-1}$, an expensive operation if done using a numerical matrix inversion routine. Since $d$ can be extremely large, the resulting overall costs can be enormous.

To deal with the peakedness of $P(\rho_k|.)$, we note that elementary row and column operations

$$
R_t \leftarrow R_t - R_k \times \frac{\rho_k \sigma_t}{\sigma_k} \quad \text{and} \quad C_t \leftarrow C_t - C_k \times \frac{\rho_k \sigma_t}{\sigma_k}
$$

can be used to transform $\Sigma$ to a matrix

$$
\Sigma' = \text{diag}
\left(
\sigma_1^2, \sigma_2^2, \ldots, \sigma_d^2, \sigma_t^2 (1 - \sum_{k'=1}^{d} \rho_{k'}^2)
\right)
$$

The key observation is that $\Sigma$ is PD if and only if $\Sigma'$ is PD, since the elementary operations do not affect the rank of the matrix. Positive definiteness of $\Sigma'$ in turn requires that its diagonal elements all be strictly positive. Since $\sigma_1^2, \sigma_2^2, \ldots, \sigma_d^2, \sigma_t^2$ will always be positive, it suffices to ensure that $q = 1 - \sum_{k'=1}^{d} \rho_{k'}^2$ is positive, which means that we can significantly narrow the bounding box and hence reduce the number of rejection iterations. Moreover,
we can incrementally maintain the PD property as we generate each $\rho_k$ rather than checking after all $\rho_k$’s have been generated and perhaps having to discard the entire set. Specifically, we generate $\rho_k$ for $k = d, d-1, \ldots, 1$; for each $\rho_k$, the lower and upper limits of the bounding box are $\max(-\xi_k, -1)$ and $\min(\xi_k, 1)$, where $\xi_k = \sqrt{1 - \sum_{l=k+1}^{d-1} \rho_l^2}$.

To handle the second problem, we compute an analytical expression for $\Sigma^{-1}$. We start with the well known general form for block inversion of a matrix based on the Schur complement; see, e.g., [117]. After some algebra, we obtain (in block form)

$$
\Sigma^{-1} = \begin{bmatrix}
D + \frac{\sigma^2}{s_t} G & -\frac{1}{s_t} \left[ \frac{\rho_1 \sigma_1}{\sigma_1}, \ldots, \frac{\rho_d \sigma_d}{\sigma_d} \right]^T

- \frac{1}{s_t} \left[ \frac{\rho_1 \sigma_1}{\sigma_1}, \ldots, \frac{\rho_d \sigma_d}{\sigma_d} \right]

\end{bmatrix}
$$

where all vectors are column vectors, $s_t = \sigma^2_t (1 - \sum_{k=1}^d \rho_k^2)$, $D = \text{diag}(1/\sigma_1^2, 1/\sigma_2^2, \ldots, 1/\sigma_d^2)$, and the entries of the matrix $G$ are given by $G_{r,c} = (\rho_r \rho_c)/(\sigma_r \sigma_c)$.

### 4.7.2 Sampler for GD Walk Model

We now turn to the Gibbs sampler for learning the parameters of the random walk model of Section 4.5.3. To learn this model, we need to be able to sample not only the various parameters of the Gaussian mixture model that underlies the walk as well as the identity of the Gaussian that produced each $\Theta_i$, but we also have to resample the decay parameter $\beta$ as well as the “true” location of each $\Theta_i$ (since we only have access to the sequence of estimates $\hat{\Theta}_1, \hat{\Theta}_2, \hat{\Theta}_3$, and so on, and not the actual $\Theta_i$ values).

**Sampling $\pi$:** This is easy due to Dirichlet-multinomial conjugacy. Specifically, let $y$ be a vector such that $y_k$ is set to $\sum_i I(z_i = k)$. Here, $I$ is the identity function, returning 1 if the boolean argument is true, and zero otherwise. Thus, $y$ counts the prevalence of each mixture component. Then update $\pi$ using:

$$
\pi \sim \text{Dirichlet}(\alpha + y)
$$
**Sampling** $\beta$: The likelihood of a particular value of $\beta$ is proportional to:

$$\text{Beta}(\beta|1,1) \prod_i \text{Normal}(\beta^{-i}(\Theta_{i-1} - \Theta_i) | \mu_{zi}, \text{diag}(\sigma^2_{zi}))$$

This must be sampled using a rejection sampler.

**Sampling** $\Theta_i$: Assume that OLAError($i$) is represented as a set of “particles” (or samples) called $S$. $S$ is going to be generated from the MCMC sampler for the OLA error. Then we simply choose $\Theta_i$ so that $\Theta_i - \hat{\Theta}_i$ is in $S$, where the probability of choosing a given $\Theta_i$ is proportional to:

$$\text{Normal}(\beta^{-i}(\Theta_{i-1} - \Theta_i) | \mu_{zi}, \text{diag}(\sigma^2_{zi}))$$

**Sampling** $z_i$: We sample $z_i \sim \text{Multinomial}(p, 1)$, where $p$ is the vector with one entry for each cluster such that

$$p_k \propto \text{Normal}(\beta^{-i}(\Theta_{i-1} - \Theta_i) | \mu_k, \text{diag}(\sigma^2_k)).$$

**Sampling** $\mu_k$: Due to Normal-Normal conjugacy, this update is easy. $\mu_k$ will have a Normal distribution during Gibbs sampling, with mean

$$\left(\text{diag}(\sigma_0^{-2}) + c \times \text{diag}(\sigma_k^{-2})\right)^{-1} \times$$

$$\left(\text{diag}(\sigma_0^{-2})\mu_0 + c \times \text{diag}(\sigma_k^{-2}) \sum_i I(z_i = k)\Theta_i\right)$$

and variance

$$\left(\text{diag}(\sigma_0^{-2}) + c \times \text{diag}(\sigma_k^{-2})\right)^{-1}.$$  

Here, $c$ is the count $\sum_i I(z_i = k)$.

**Sampling** $\sigma^2_{k,a}$: Due to InverseGamma-Normal conjugacy, this is also easy. $\sigma^2_{k,a}$ will have an InverseGamma distribution, with shape parameter $1 + c/2$ and scale parameter $1 + \sum_i (\Theta_i^{(a)} - \mu_{k,a})^2/2$. $c$ is as defined previously. Note that in contrast to $\mu_k$ where we update
the entire vector at once, here we update each of the \( d \) different \( \sigma_{k,a}^2 \) values associated with \( \sigma_k^2 \) separately.

### 4.8 Experimental Evaluation

#### 4.8.1 Methods and Models Tested

At the highest level, I will compare four distributed GD frameworks on four different distributed learning problems. The frameworks tested are:

1. OLA-GD, as described in this chapter. OLA-GD is run directly on the data stored in HDFS pre-processing.

2. Distributed mini-batch [111]. As described in the original paper, this method accepts iid (independent and identically distributed) samples from some distribution at a number of distributed sites. A local gradient update (that is, a sum of \( \nabla L(t_j|\Theta_i) \) values) is computed at each site. Periodically, these gradient updates are averaged and the model is updated globally, ending the epoch. To implement this on Hadoop, we shuffle the data by sorting according to a pseudo-randomly generated key. This ensures that the various tuples in a block are iid samples from the over population (if one ignores finite population effects). Our actual implementation of the distributed mini-batch framework resembles the OLA-GD implementation described in Section 4.6, including the use of a coordinator to avoid the need to run a new MapReduce job for each epoch, and a server to accumulate and broadcast gradient updates. After some tuning, we arrived at a mini-batch size of 80 HDFS blocks (that is, each mapper in each epoch processes 80 blocks).

3. Distributed SGD [112]. This method runs many SGD computations locally, each of which constantly updates a local version of \( \Theta \) using an epoch size of one tuple. These
local parameter sets are then periodically averaged. To implement this in Hadoop, we first run a shuffle to randomize the data. Again, the implementation resembles our OLA-GD implementation, except that the data in the data set are fully partitioned to the various mappers. Averaging occurs once each mapper has completely processed its subset of the data.

4. Batch GD. This is the simplest framework. Without any pre-processing, the data are fully partitioned and assigned to mappers, each of which computes a sum over $\nabla L(t_j|\Theta_i)$ values for each $t_j$ that it has been assigned. After completion, these sums are averaged and used to compute $\Theta_{i+1}$.

The learning problems considered are:

1. A regularized linear regression. If we take $t_j$ to be a vector $x_j$ with a real-valued response $y_j$, then the loss function is

$$\sum_j \left( \left( \sum_k x_{j,k} \times r_k \right) - y_j \right)^2 + \lambda \sum_k r_k^2.$$  

The goal is to learn $\Theta = \langle r_1, r_2, \ldots \rangle$, given the parameter $\lambda$.

2. A support vector machine classifier. This time we have a $+1/ -1$ response, with a loss function

$$\sum_j \max(0, 1 - y_j(\sum_k x_{j,k} \times w_k)) + \lambda \sum_k w_k^2.$$  

The goal is to learn $\Theta = \langle w_1, w_2, \ldots \rangle$, given $\lambda$.

3. A Gaussian mixture model (GMM). The model consists of 10, 100-dimensional Normal or Gaussian distributions, and the parameter set includes $\mu$ and $\Sigma$ for each Gaussian, as well as the set of mixture component probabilities. Given a data set, the task is to minimize the negative log-likelihood of the model given the data.
4. The Matchbox Bayesian recommender system model [113]. This is a quite sophisticated recommender model that allows for rating predictions over (customer, item) pairs where both the customer and item are represented by an arbitrary (typically very high-dimensional) feature vector. The goal is to learn the two matrices that map a customer and an item into the latent feature space where the prediction occurs. Since this is a Bayesian model, we use a GD algorithm to perform a MAP estimate for the matrices, again attempting to minimize the negative log-likelihood of the model given the data. We use a 10-dimensional latent feature space.

Note that the first two problems are quite standard, convex learning problems. In particular, linear regression is often a very easy model to learn. The second two are more difficult, non-convex problems. While Matchbox might be a bit more exotic than the GMM, the GMM in particular is understood to represent a difficult learning problem, where the search space is littered with local optima.

4.8.2 Data, Experimental Platform, Results

Unfortunately, large classification data sets are not readily available, and so for the first three problems we utilize synthetic data.

For the linear regression and SVM learning task, 500GB of individual data points are generated by sampling each dimension from a Uniform$(-50, 50)$ distribution. A mixture of five different regression vectors are generated by sampling the dimensions of each vector from a Normal$(0, 10)$ distribution, subject to the constraint that 80% of the dimensions are assigned a zero value. To generate a particular HDFS data block, a mixture parameter $\theta$ is generated by sampling from a $\theta \sim \text{Dirichlet}(1)$ distribution. Then, the various data points are generated from the five regression vectors by, for each point, sampling the identify of a particular regression vector using a Multinomial$(\theta)$ distribution. By taking the dot product
of the point with the regression vector, the response is generated. In this way, data within a block have a similar distribution, but different blocks have somewhat different distributions.

For the GMM learning task, 500GB of data are generated by sampling directly from the GMM to be learned.

For the Matchbox learning task, we use the MovieLens data set, which contains 10,000,054 ratings for 10,681 movies by 71,567 users. All the movies in the dataset have one of more genres (in all 20) and all the users in the dataset have rated at least 20 movies. This results in about a million parameters to learn. Since this data set is quite small, we replicate it 10 times to produce the data set used in the learning task.

All experiments were conducted on a 11 node c3.xlarge Amazon EC2 cluster. Each node has 64-bit Intel Xeon E5 processors with 14 compute units (i.e., 4 × 3.5 vCPUs), 7.5GB memory, and 280GB SSD disk. Out of this 11 nodes, one machine was kept as the master and 10 machines were used as slaves. We used the Cloudera 4 distribution (CDH4) of Hadoop.

For each learning task/GD framework combination, we use the framework to run the GD algorithm to completion. At the end of each epoch, the value of the loss function is computed. We plot the results in Figure 4.5.

4.8.3 Discussion

We generally find OLA-GD to be the best (or at least most widely applicable) option for running a GD in a distributed environment. It is worth noting that although some of the (complex) learning problems studied here are amenable to highly tuned, specialized algorithms, OLA-GD performs respectably without exploiting deep problem structure. I now discuss some individual points in more detail.

Batch Distributed GD is Slow. It is perhaps not surprising that running a distributed batch
GD is generally the worst choice. For reasons discussed throughout the chapter, a full batch GD spends too much time computing an exact gradient, leading to too few model updates.

**Distributed SGD and Non-Convex Problems.** It was expected that distributed SGD makes little sense for non-convex problems, and our experimental results bear this out. For both Matchbox and the GMM (the two non-convex problems), we were unable to get the distributed SGD to converge to a solution that was anywhere close to the quality of the solution achieved by the other frameworks. As intimated early on in the chapter, running a SGD locally and then averaging periodically makes little sense when the individual SGD runs converge to their own local optima. One might be able to get around this sometimes by averaging more frequently (I averaged after the entire data set was processed), but I feel that distributed SGD should not be considered a natural choice for non-convex problems.

**OLA vs. Distributed SGD.** Across all four problems, OLA-GD achieved a loss value comparable to the other three frameworks, in a time that was noticeably faster than the other three. One method that would seem to compete is a distributed SGD where the data were already sorted (or close to it), used to solve a convex learning problem. In our experiments, I arbitrarily chose to update (average) the model globally only once all of the data has been processed. If the model were updated much more frequently, it is likely that convergence would be even faster.

**OLA vs. Distributed Mini-Batch.** This is the most interesting head-to-head comparison. Distributed mini-batch converges very quickly—in fact, it converges fastest out of the four frameworks tested, *once the data are shuffled*. The downside, of course, is that shuffling our four data sets took anywhere between two and a half hours and 16 hours using Hadoop. This is a long time to wait before distributed mini-batch can be run.

The obvious question that one might ask is: can we simply avoid sorting the input data?
Would that yield the very best algorithm? The data could be left in the blocks that they were originally found in, without shuffling, and then the blocks themselves processed in a random order.

In fact, if one decided to run the distributed mini-batch in this way, one would essentially be running the OLA-GD algorithm. The difficulty however, is that without shuffling, there is no obvious answer to the question: How long each epoch should be run for? The models and algorithms described in this chapter are largely directed at automatically addressing that question in real time, as the data are processed.

Depending upon how the data are grouped, the answer might be very different for the same learning problem over the same data set. Consider the common case where data are grouped into blocks based on their arrival date. Data arriving at the same time are similar, so there is little statistical variation among the data in each block. This is precisely the case in our SVM, linear regression, and GMM experiments, where (by design) the data within a block are somewhat homogeneous, and data across blocks are quite different. In this case, the curve plotting the expected distance traveled towards $\Theta^*$ as a function of time (see Figure 4) will be quite flat, meaning the velocity towards $\Theta^*$ will decrease slowly. Assuming that there is a significant startup cost for an epoch, this implies that it will take a long time for the instantaneous velocity to decrease to the average velocity of the epoch. According to the analysis of Section 4.5.4, this means that the epoch should be run for a relatively long time. On the other hand, perhaps the data set has been shuffled recently, so that the data are nearly randomized. In this case, the curve plotting the expected distance traveled as a function of time will start out steeply and flatten quickly. According to the same analysis, this means that the epoch should be short.

In fact, the shuffling itself explains why the distributed mini-batch converges so quickly. Shuffling allows for a very small epoch duration, which I arrived at via trial and error. With-
out shuffling, epochs will tend to be longer, leading to slower convergence. And without shuffling, one is essentially left with the OLA-GD algorithm, necessitating the machinery developed in this chapter.

**Is the Inspection Paradox a Problem?** It is natural to question whether the bias introduced due to inspection paradox affects the convergence of gradient descent. To investigate this, I used a process similar to the one that I used to generate the linear regression/SVM data, with a couple of changes. First, the identity of the regression vector (a value from 1 through 5) used to generate the data in a block is kept constant, so (in contrast to the prior set of experiments) the contents of the block are uniform. In the heavy skew case, the size of the blocks are exponentially distributed, where the size is correlated with the identity of the regression vector. In the medium skew case, there are two block sizes. In the light skew case, there is a single block size. We might expect the inspection paradox to rear its head, since block size is correlated with block contents, at least in the first two cases.

Given this data, I attempted to learn a linear regression model four different times using the OLA-GD framework. The first three linear regression models were learned over the heavy, medium, and light skew data sets, respectively, but OLA-GD is constrained to have zero correlation between the gradient updates and the block processing time. In the fourth case, the regression model is learned over the heavy skew data, but the OLA-GD is allowed to take into account correlation. The log-loss as a function of time is plotted in Figure 4.6. In the heavy and moderate cases with no correlation, the OLA-GD computation does not converge, but it converges quickly in the heavy skew case when correlations are allowed. Thus, it is easily possible to construct an example where the inspection paradox has an effect on learning efficacy.
4.9 Conclusions

In this chapter, I have described OLA-GD, which is a framework that allows for easy implementation and efficient execution of distributed gradient descent (GD) algorithms. OLA-GD is easy to use, out of the box, and requires no special insights into the structure of the optimization problem at hand. All that a user needs to do to utilize the OLA-GD software for a new learning problem is to provide code to perform the gradient computation.

The OLA-GD framework is based on the observation that at its core, a GD algorithm looks a lot like an aggregate computation. Thus, OLA-GD makes extensive use of Bayesian models that have previously been proposed for generating estimates and confidence bounds when running large-scale distributed aggregate computations. One of the key technical contributions of the OLA-GD approach is that it combines those Bayesian accuracy models with a Bayesian model for the walk being performed by the underlying GD, to develop a set of principled guidelines that allow the OLA-GD software to decide, on-the-fly, when it is time to end one GD epoch and start the next one. This means that OLA-GD can be run on virtually any distributed data set. Whether the data are randomized or not, the OLA-GD software will use these models to “figure out” how to perform the GD so as to maximize the convergence rate. I have implemented OLA-GD and compared it with several other, obvious alternatives, and found that OLA-GD deserves strong consideration as a method of choice for running large-scale distributed GD computations.
Figure 4.1: Two statistical models used in determining the distance traveled toward $\Theta^*$ in epoch $i + 1$. The first is the OLA error model itself, which is used to obtain a distribution over $\varepsilon_{i+1} = \Theta_{i+1} - \hat{\Theta}_{i+1}$. The second is a model for $\Delta_{i+1} = \Theta^* - \Theta_{i+1}$, the distance to the optimal solution. The figure shows three different samples from both of the models. The use of these samples to estimate the expected distance traveled is depicted in Figure 4.2.
Figure 4.2: Using samples from the two models depicted in Figure 4.1 to obtain a posterior distribution on the distance traveled toward $\Theta^*$ in epoch $i + 1$. In each case, $\Theta^*$ is computed as $\hat{\Theta}_{i+1} + \epsilon_{i+1} + \Delta_{i+1}$. From $\Theta^*$ we compute $d_i$, the distance to $\Theta^*$ at the end of the last epoch, and $d_{i+1}$, the distance at the end of the current epoch. The distance traveled toward $\Theta^*$ in epoch $i + 1$ is then $d_i - d_{i+1}$. In this example, the expected distance traveled is $(1.2 + 1.2 + 1.4)/3 = 1.27$. 
Figure 4.3: Expected distance traveled to $\Theta^*$ as a function of the standard deviation of $\epsilon_{i+1}$ in an example scenario.

Figure 4.4: Expected distance traveled to $\Theta^*$ as a function of time in an example scenario.
Figure 4.5: Loss versus time for the various learning task/GD framework combinations.
Figure 4.6: Effect of data skew with (and without) correlation in the OLA model.
Chapter 5

Topic Models Over Spoken Language

5.1 Introduction

In text processing, a topic is defined as a probability distribution over sets of words or phrases and each document in the corpus is drawn from mixture of these topics [28]. A topic model for a corpus specifies the set of topics, as well as the proportion in which they are present in any given document. For example, if a topic model were learned over a set of machine learning papers published in the last few years, one of the learned topics might put high probability on sets of words having to do with learning over graphs (“edge”, “vertex”, “SVD”, “diameter”), one might favor sets of words having to do with kernel methods (“SVM”, “Gaussian processes”, “hyperplane”), one might favor words associated with text processing (“LDA”, “topic”, “TF-IDF”), and so on. The topic model might further identify that 80% of the words in a specified document come from the graph topic, 10% from the kernel topic, and 10% from the text topic.

There are many good reasons for learning a topic model over a corpus. Often, topic modeling is used as a methodology for feature selection. A topic model is used to map each of the corpus’ documents to a feature vector in “topic space”, where the $i$th entry in the vector corresponds to the importance of the $i$th learned topic in the document. Then, important learning tasks—document classification, clustering, outlier detection, recommendation, and so on—can be performed on the resulting, topic-based representations of the documents. In this way, topic modeling serves as an alternative to other vector-based
representations, such as mapping each dictionary word to a dimension and using the word’s TF-IDF value to position the document in the resulting vector space. In practice, performing learning tasks in topic space works well because the position of the document in topic space is semantically meaningful; documents that are close together presumably cover the same subject matter.

5.1.1 Topics Over Spoken Language

The recent interest in topic models has been driven by the explosion of electronic, text-based data that are available for analysis. From web pages to emails to microblogs, text-based data are everywhere. However, not all electronically-available natural language corpora are text-based. In this chapter, I consider the problem of learning topic models over spoken language. Our work is motivated by our involvement with the Spoken Web (also called the World Wide Telecomm Web), which allows users in rural India to post farming-related questions and responses to an audio forum using mobile phones [29, 30, 31]. More conventional, text-based social media are not convenient for Spoken Web users for many reasons, from lack of literacy to lack of SmartPhone or Internet access. We would like to learn topic models over users’ audio posts, so that we can subsequently perform learning tasks on the data.

Although the Spoken Web serves as our motivating application, the utility of learning topic models over spoken language is not limited to this setting. A large amount of natural language data is available in audio format. For example, the BBC has archived 75 years worth of television programs—more than one million in all. Building topic models to facilitate mining of this data would potentially be of great value.
5.1.2 Moving Beyond Text

Given the potential usefulness of topic modeling for spoken language, it is natural to ask: Are there significant technical challenges as we move beyond electronic text, or is it a simple matter of applying a speech-to-text software to the audio document, and then learning a standard topic model over the resulting document? I argue that the situation may not be simple at all. The difficulty is that even modern speech-to-text software is notoriously inaccurate, especially if recordings are made under non-optimal circumstances. Such inaccuracy can have an adverse effect on the quality of a topic model that is learned. The situation is particularly poor for non-English-language speech. In our motivating application to the Spoken Web, the spoken text we wish to analyze are in Gujarati, for example.

5.1.3 Taking Into Account Transcription Errors

In this chapter, I propose a new topic model that leverages the statistical algorithms used in most modern speech-to-text software. These algorithms typically consider all possible paths through a lattice, where each node in the lattice is a possible transcription of a spoken phrase. The most likely path is returned as the final transcription. It is easy to modify such a software to return not a single transcription but rather, for each phrase, a weighted set of possible transcriptions instead of just the most likely one. For example, imagine that a document from an audio corpus contains the sentence, “The cow jumped over the moon.” The speech-to-text software would break the file into a sequence of phrases. Each phrase is represented by a weighted set of possible values. For example, we might have:
The idea is to exploit the additional information contained in such $\Psi$ vectors. Specifically, I develop an alternative version of the popular LDA [33] topic model called the *spoken topic model*, or STM for short. This model uses a Bayesian interpretation of each $\Psi$ vector that takes into account the vector’s explicit uncertainty description (the phrases and weights) in a principled fashion. I derive a Markov Chain Monte Carlo algorithm that learns this spoken topic model from an audio document corpus. Also, I show that the resulting model learns a more useful set of topics than would be learned by simply applying standard learning methods to the final output of the speech-to-text software.

Importantly, our methods are applicable to any domain for which it would be useful to build a topic model over data that explicitly represents uncertainties. For example, LDA-like models have been applied to problems in computer vision [118]. Our methods could potentially be applied to computer-vision problems in which, during a pre-processing stage, segments of the image are explicitly labeled with numbers that represent uncertainty in the segment’s visual features or associated semantic labels.

### 5.2 The Spoken Topic Model

In this section, I detail the generative process underlying the spoken topic model. The STM generative process is a relatively straightforward modification of the LDA generative process. The key difference is that the ultimate output of the process is not a document corpus as in LDA, but rather a sequence of sets of *weighted possible phrases*, or WPPs for...
short, as in the sequence $\Psi_1, \Psi_2, \Psi_3$ shown in Section 5.1.

### 5.2.1 A Brief Introduction to LDA

In case the reader does not have a basic familiarity with LDA, I give a very brief overview here. For a more detailed introduction, the first two sections of the original LDA paper [33] should suffice.

The idea behind LDA is quite simple. LDA imagines that a corpus of $M$ documents is produced by a set of $K$ latent topics. The $k$th latent topic $\phi_k$ is a vector of word probabilities that has been produced by sampling from a Dirichlet distribution. To generate document $i$ using LDA, a topic proportion vector $\theta_i$ is first sampled from another Dirichlet distribution. After generating $\theta_i$, each of the $N_i$ words in the document are generated by first sampling a topic according to the discrete probability distribution encoded by $\theta_i$ and then using the selected topic to generate a word according to probabilities in the corresponding $\phi_k$ vector. Thus $\theta_i$ controls the (random) fraction of the words in document $i$ that are generated by each of the topics.

Given this assumed generative model, the actual corpus at hand is then used to estimate or make inferences about (unknown) LDA parameters—also called latent variables—such as $\theta_i$ and $\phi_k$. Note, for example, that $\theta_i$ corresponds precisely to the topic feature vector for document $i$ as discussed in Section 5.1, and so is of great interest for a variety of learning tasks.

### 5.2.2 The STM Generative Process

Before I describe the STM generative process in detail, it is useful to describe the three key differences between the STM generative process and the generative process underlying LDA:
**A document’s words are not visible.** The most significant difference between LDA and STM is that, in the STM generative process, the actual sequence of words in a document is typically not visible, and so this sequence is treated as a latent variable to be estimated. All that is visible to the inference process is the output of the speech-to-text software. This software takes as input an audio signal corresponding to a sequence of words and, as described in Section 5.1, produces as output a sequence of sets of WPPs. Denote by $\Psi_{i,j}$ the set of WPPs corresponding to the $j$th phrase in document $i$, so that $\Psi_{i,j}$ is a set of possible word sequences, where each possible sequence is labeled with a weight indicating the likelihood (according to the speech-to-text software) that the sequence is the correct transcription. Using $W_{i,j}$ to denote the words in the corresponding, actual phrase, we assume that $\Psi_{i,j}$ is sampled from a “ToSpeechToText($W_{i,j}$)” distribution. One of the most novel aspects of our approach is that we impose no apriori assumptions on the ToSpeechToText(.) distribution, other than assuming that the weights recorded in the $\Psi_{i,j}$ generated by the distribution are, in fact, good approximations to the true posterior distribution of $W_{ij}$. I discuss this assumption in detail in Section 5.3.

**Variable number of words per document.** Rather than document $i$ having a fixed number of words that is chosen prior to generation as in LDA, the STM process assumes that document $i$ instead has a variable number of words and a fixed number of phrases. We denote the number of words in the $j$th phrase in document $i$ by $l_{i,j}$ and the total number of phrases by $N_i$. This modeling choice reflects the fact that each alternative in a typical WPP output by the speech-to-text software has a fixed (or almost fixed) number of sounds or syllables, but the number of words in the alternative will vary, e.g., “the moon” versus “cocoon” in our previous example. Because the correct number of words in a phrase is unavailable from examination of corresponding WPP, we treat it as a latent variable to be estimated.
One topic per phrase. Under the STM generative process, all of the words in a phrase are generated using the same topic. This is not only a plausible assumption, but more importantly it will make inference significantly more efficient. If we do not assume a single topic for a phrase of length \( l \), then there are \( K^l \) possible topic assignments for such a phrase, and it seems difficult to avoid enumerating all of these possibilities during inference. For reasonable values of \( K \) between 20 and 1000 and of \( l \) between 2 and 5, the inference problem is likely to be computationally infeasible.

In light of the foregoing considerations, I define the generative process for STM as Algorithm 4.

**Algorithm 4: Generative process for STM**

```
foreach \( k \in \{1...K\} \) do
    \( \phi_k \sim \text{Dirichlet}(\alpha); \)

foreach \( i \in \{1...M\} \) do
    \( \theta_i \sim \text{Dirichlet}(\beta); \)

foreach \( j \in \{1...N_i\} \) do
    \( l_{i,j} \sim \text{Poisson}(\lambda); \)
    \( z_{i,j} \sim \text{Discrete}(\theta_i); \)
    \( W_{i,j} \leftarrow \langle \rangle; \)

foreach \( \ell \in \{1...l_{i,j}\} \) do
    \( w \sim \text{Discrete}(\phi_{z_{i,j}}); \)
    \( W_{i,j} \leftarrow W_{i,j} \cdot \langle w \rangle; \)
    \( \Psi_{i,j} \sim \text{ToSpeechToText}(W_{i,j}); \)
```

In this process, \( \alpha, \beta, \) and \( \lambda \) are hyperparameters that we define a priori. The generation
process comprises two main steps. Steps 1 and 2 generate the set of topics, and Steps 3–12 generate the document corpus. To generate a document, the STM process first generates the topic proportion vector \( \theta_i \) in Step 4, and then generates the sequence of phrases in Steps 5–11. To generate a phrase, the STM process first (in Step 6) generates a phrase length, denoted by \( l_{i,j} \), and then (in Step 7) the identity \( z_{i,j} \) of the phrase’s topic. Next, in Steps 8–11, the words in the phrase are generated, in sequence; the “•” symbol denotes the concatenation operation, which takes the new word \( w \) and appends it to the current phrase. Finally, the process of the phrase being spoken, converted into an audio signal, and then being converted back into text is modeled using the ToSpeechToText(\( \cdot \)) distribution in Step 12. The output of the ToSpeechToText(\( \cdot \)) distribution is a set of WPPs, denoted by \( \Psi_{i,j} \). One of the most novel aspects of our approach is that we impose no apriori assumptions on the ToSpeechToText(\( \cdot \)) distribution, other than assuming that the weights recorded in the \( \Psi_{i,j} \) generated by the distribution are, in fact, good approximations to the true posterior distribution of \( W_{i,j} \).

In general, the ultimate output of the STM generative process is the set of \( \Psi_{i,j} \) values; everything else—each \( \theta_i, \phi_k, l_{i,j}, z_{i,j}, \) and \( W_{i,j} \)—is a latent variable whose value must be estimated.

### 5.3 To Speech And Back Again

Before I discuss the problem of inference for STM, it is necessary to discuss in detail the nature of the ToSpeechToText(\( \cdot \)) function. In order to generate \( \Psi_{i,j} \) in Step 12 of the STM generative process, we sample from a variable whose distribution is characterized by the probability density function (PDF) ToSpeechToText(\( \Psi|W \)). This PDF takes into account the entire process for generating \( \Psi \) from \( W \): a speaker verbalizes \( W \), the resulting sound is converted into a waveform that is transmitted over wire or air to the speech-to-text software,
and then the speech-to-text software processes the waveform and emits $\Psi$. It is quite natural to try and model all of the possible errors incurred in this sequence of events as a stochastic process, which in turn is embodied by the ToSpeechToText($\Psi|W$) function.

We immediately encounter a serious problem, however, in that the ToSpeechToText(.) PDF is necessary for inference, but we are unlikely to have direct access to it in realistic application scenarios. I.e., we can postulate that such a function exists, but obtaining the function when it is time to perform inference is another matter entirely. A reasonable way around this problem starts with the observation that, when we wish to evaluate ToSpeechToText($\Psi|W$), we do have access to the output $\Psi$ itself. Recall that $\Psi$ consists of a set pairs of the form $(\omega, R)$ where $\omega$ is a weight and $R$ is a phrase. Presumably, the developers of the speech-to-text software have devoted considerable energy towards making the output of the software meaningful. Assuming that this is the case, we can reasonably view $\omega$ as a valid degree of belief that $R$ is in fact the actual phrase that was spoken. This view suggests a Bayesian interpretation of the process of generating value $\Psi$ from a stochastic process having a ToSpeechToText(.) distribution. Specifically, we assume that if $\Psi \sim \text{ToSpeechToText}(W)$, then

$$P(W|\Psi) = \begin{cases} \omega & \text{if } (\omega, R) \in \Psi \text{ and } W = R \\ 0 & \text{otherwise} \end{cases}$$

(5.1)

In other words, we assume that the stochastic process embodied by ToSpeechToText($W$) produces as its output an encoding of the posterior distribution over its input $W$, given the output. It follows that we have access to $P(W|\Psi)$, and so we can define the conditional PDF

$$\text{ToSpeechToText}(\Psi|W) = P(\Psi|W) = \frac{P(W|\Psi)P(\Psi)}{P(W)}$$

(5.2)

For example, given $\Psi = (\Psi_1, \Psi_2, \Psi_3)$ as in Section 5.1, we would take $P(\text{“bow wow”}|\Psi) = \ldots$
0.4, \text{P}(\text{“the cow”} | \Psi) = 0.2, \text{and P}(\text{“a sow”} | \Psi) = 0.0 \text{ for purposes of inference. Similarly,}
if we had a prior belief that, for a given audio document, the phrase “the cow” would be uttered with probability 0.1 at any instant and that the chance of obtaining \Psi for an arbitrary phrase is 0.3, then \text{P}(\Psi | \text{“the cow”}) = 0.2 \cdot 0.3/0.1 = 0.6.

This may all seem a bit abstract, and so it is useful to illustrate the general principle at work in a simpler scenario. Imagine that we have a random process that takes as input an unseen integer n, and then generates a number x where x \sim \text{Poisson}(2). We know that \text{P}(x = 0) = 0.14, \text{P}(x = 1) = 0.27, \text{P}(x = 2) = 0.27, \text{P}(x = 3) = 0.18, and so on. After generating x, the process computes n' = n + x, and then outputs the set

\[ S = \{(0.14, n'), (0.27, n' - 1), (0.27, n' - 2), (0.18, n' - 3), \ldots\}. \]

Observing only the set S, one could correctly say that \text{P}(x = n'|S) = 0.14, \text{P}(x = n' - 1|S) = 0.27, \text{P}(x = n'|S) = 0.27, \text{and so on. Thus, this stochastic process does in fact produce, as its output, a set S that encodes the posterior distribution of n given S. This is perfectly analogous to our assumed situation with the ToSpeechToText(.) process.}

I close this section by drawing attention to a couple of assumptions that are implicit in our approach. First, I assume that the real-world mechanism that generates the observed WPPs is such that the WPPs are, to a good approximation, mutually statistically independent. Second, the weights in each \Psi_{i,j} that result are, in fact, meaningful in that they correspond roughly to the true probabilities of correct transcription for the speech-to-text software. Specifically, if we fix an arbitrary finite sequence of (i, j) pairs \((i_1, j_1), (i_2, j_2), \ldots, (i_m, j_m)\), where 1 \leq i_l \leq M and 1 \leq j_l \leq N_i \text{ for } l \in \{1, 2, \ldots, m\}, \text{then the total number of times that } R_{i_l,j_l} = W_{i_l,j_l} \text{ should (approximately) follow a Poisson-binomial distribution with parameter vector } p = (\omega_{i_1,j_1}, \omega_{i_2,j_2}, \ldots, \omega_{i_m,j_m}). \text{ In practice, the STM model appears to be somewhat robust to deviations from the foregoing assumptions, but systematic violations can lead to poor learning accuracy. I give a detailed example of this phenomenon in the}
5.4 Inference

As discussed previously, the overall goal of our analysis is to use the output of a speech-to-text program that has been applied to a corpus of audio files to estimate the parameters of our STM generative model. In this section I develop a Monte Carlo procedure for this task.

5.4.1 Markov Chain Monte Carlo

I derive a Gibbs sampler to infer the posterior PDF

\[
P\left(\{\phi_k\}_{k=1}^K, \{\theta_i, \{l_{i,j}, z_{i,j}, W_{i,j}\}_{j=1}^{N_i}\}_{i=1}^M, \alpha, \beta, \lambda, \{\{\Psi_{i,j}\}_{j=1}^{N_i}\}_{i=1}^M\right)
\]

There are many options for performing this sort of Bayesian inference; I choose a Gibbs sampler for its simplicity and ease of implementation. The usual concern with Gibbs sampling—speed—does not seem to be a huge issue in our setting because (1) most of the priors used are conjugate and (2) in those cases where the priors are not conjugate, the variables are discrete. Thus, I have no need to apply complicated (and computationally expensive) Monte Carlo methods such as rejection sampling to implement our Gibbs sampler. If speed really is an issue, then an approximate variational algorithm [119] can be derived.

Briefly, a Gibbs sampler is a Markov Chain Monte Carlo method that operates by continually maintaining candidate values for each of the variables to be inferred. The sampler repeatedly updates each of the variables for a sufficient number of iterations, until the chain has “mixed”. After this period, referred to as the “burn in,” one can take a snapshot of all of the current values to obtain a sample from the true posterior distribution. Often, the key challenge in applying a Gibbs sampler is deriving the update rule for each variable. To update a variable \(X\) during Gibbs sampling, it is necessary to sample from the distribution
\( P(X|\Theta - \{X\}) \), where the set \( \Theta \) contains all of the observed data values as well as all of the candidate values for the latent variables.

### 5.4.2 The Joint Density Function

To develop a Gibbs sampler for the above posterior PDF, I first write down an explicit formula for the full joint distribution of all of the variables:

\[
P(\{\phi_k\}_{k=1}^K, \{\theta_i\}, \{l_{i,j}, z_{i,j}, W_{i,j}\}_{i=1}^N) \times \{\{\Psi_{i,j}\}_{j=1}^M\}_{i=1}^M | \alpha, \beta, \lambda)
\]

\[
\propto \left( \prod_k \text{Dirichlet}(\phi_k | \alpha) \right) \times \left( \prod_i \text{Dirichlet}(\theta_i | \beta) \right) \times \\
\left( \prod_{i,j} \text{Poisson}(l_{i,j} | \lambda) \right) \times \left( \prod_{i,j} \text{Discrete}(z_{i,j} | \theta_i) \right) \times \\
\left( \prod_{i,j} \text{Multinomial}(W_{i,j} | l_{i,j}, \phi_{z_{i,j}}) \right) \times \left( \prod_{i,j} \text{ToSpeechToText}(\Psi_{i,j} | W_{i,j}) \right) 
\]

The close link between \( W_{i,j} \) and \( l_{i,j} \) is potentially problematic for our Gibbs sampler. Because the number of words in \( W_{i,j} \) must equal \( l_{i,j} \), it will not be possible to sample these values independently. E.g., given a current value for \( W_{i,j} \), there is only one possible value for \( l_{i,j} \), so if I fix the value of the former, I cannot change the value of the latter. I therefore remove all of the \( l_{i,j} \) parameters from the sampling process by marginalizing them away, thereby removing the need to maintain them explicitly. Doing this and re-arranging terms yields:

\[
P(\{\phi_k\}_{k=1}^K, \{\theta_i\}, \{z_{i,j}, W_{i,j}\}_{j=1}^N) \times \{\{\Psi_{i,j}\}_{j=1}^M\}_{i=1}^M | \alpha, \beta, \lambda)
\]

\[
\propto \left( \prod_k \text{Dirichlet}(\phi_k | \alpha) \right) \times \left( \prod_i \text{Dirichlet}(\theta_i | \beta) \right) \times \\
\left( \prod_{i,j} \text{Discrete}(z_{i,j} | \theta_i) \right) \times \\
\left( \prod_{i,j} \sum_{l=1}^\infty \text{Multinomial}(l_{i,j} | l, \phi_{z_{i,j}}) \times \text{Poisson}(l | \lambda) \times \text{ToSpeechToText}(\Psi_{i,j} | W_{i,j}) \right) 
\]
5.4.3 Update Samplers

In this section, I use $\Theta$ to denote the complete set of data, including all observed data, hyperparameters, and latent variables, so

$$\Theta = \{\alpha, \beta, \lambda\} \cup \{\phi_k\}_{k=1}^K \cup \{\theta_i\}_{i=1}^M \cup \{z_{i,j}, W_{i,j}\}_{i,j=1}^{M,N_i}.$$  

Let $z_i$ be the vector whose $k$th entry is the number of phrases in document $i$ having topic $k$ and $w_k$ be the vector whose $m$th entry is the number of $(i, j)$ pairs such that (1) $z_{i,j} = k$ and (2) word $m$ appears in $W_{i,j}$.

The equations for each of the individual update samplers for Gibbs sampling are as follows:

$$\theta_i \sim \text{Dirichlet}(\beta + z_i)$$

$$\phi_k \sim \text{Dirichlet}(\alpha + w_k)$$

$$P(z_{i,j} | \Theta - \{z_{i,j}\}) \propto \text{Discrete}(z_{i,j} | \theta_i) \times \text{Multinomial}(W_{i,j} | \lambda \times P(W_{i,j} | \phi_{z_{i,j}}))$$

$$P(W_{i,j} | \Theta - \{W_{i,j}\}) \propto \text{Poisson}(|W_{i,j}| | \lambda) \times \frac{P(W_{i,j} | \Psi_{i,j})}{P(W_{i,j})} \times \text{Multinomial}(W_{i,j} | \lambda \times P(W_{i,j} | \phi_{z_{i,j}}))$$

**Updating the word assignments**

Evaluating each of the constituent terms for $P(W_{i,j} | \Theta - \{W_{i,j}\})$ is straightforward, except for $P(W_{i,j})$. This term measures our prior belief that a random phrase from a random document would contain exactly the set of words in $W_{i,j}$.

I consider two methods for evaluating $P(W_{i,j})$. The first approach tries to evaluate $P(W_{i,j})$ purely according to the STM generative model. Unfortunately, evaluating this probability naively is difficult, since under STM the words in $W_{i,j}$ are effectively sampled from a multinomial distribution with parameter $\phi_{z_{i,j}}$, and $\phi_{z_{i,j}}$ is bound to show a greater affinity for some words than for others, leading to a high likelihood of phrases containing

...
repeated words. If the candidate phrases in \( \Psi_{i,j} \) contain different numbers of repeated words, I must evaluate the following expression exactly when computing \( P(W_{i,j}|\Theta - \{W_{i,j}\}) \):

\[
P(W_{i,j}) = \text{Poisson}(\|W_{i,j}\| \mid \lambda) \times \int_{\phi} \text{Dirichlet}(\phi|\alpha) \text{ Multinomial}(W_{i,j} \mid \|W_{i,j}\|, \phi) \, d\phi.
\]

To avoid this difficulty, we can impose the assumption that all words in a candidate phrase in \( \Psi_{i,j} \) appear only once, as is almost always the case in practice. This simplifies the situation considerably. Assuming that \( \alpha \) is symmetric, the STM generative process itself does not favor any particular word over another. Then, if each possible value for \( W_{i,j} \) has no duplicates, the integral in the above expression must evaluate to \( c \times \left( \frac{D}{\|W_{i,j}\|} \right)^{-1} \) where \( D \) is the size of the dictionary and \( c \) is some normalizing constant that is the same for each possible value of \( W_{i,j} \). The reason is simple: given that the phrase \( W_{i,j} \) must contain a unique set of words, and given that each word is equally likely to be chosen, the probability of choosing exactly the words in \( W_{i,j} \) is \( \left( \frac{D}{\|W_{i,j}\|} \right)^{-1} \). Thus

\[
P(W_{i,j}) \propto \frac{\text{Poisson}(\|W_{i,j}\| \mid \lambda)}{\left( \frac{D}{\|W_{i,j}\|} \right)^{-1}}
\]

Plugging this into the expression for \( P(W_{i,j}|\Theta - \{W_{i,j}\}) \), the Poisson terms cancel:

\[
P(W_{i,j}|\Theta - \{W_{i,j}\}) \propto P(W_{i,j}|\Psi_{i,j}) \times \left( \frac{D}{\|W_{i,j}\|} \right)^{\frac{1}{2}} \times \text{Multinomial}(W_{i,j} \mid \|W_{i,j}\|, \phi_{z_{i,j}}).
\]

A concern with the above approach is that usually all words are not equally likely. I therefore consider the following alternative, empirical-Bayes approach [104, Sec. 5.25]. Estimate the probability associated with a word \( w \) by computing \( P(w \in W_{i,j}|\Psi_{i,j}) \) for each pair \( i, j \). This is easily done by summing the weights of the WPPs in \( \Psi_{i,j} \) in which \( w \) appears. Then estimate \( P(w) \) as being proportional to \( \sum_{i,j} P(w \in W_{i,j}|\Psi_{i,j}) \). Finally, use \( \prod_{w \in W_{i,j}} P(w) \) rather than \( \left( \frac{D}{\|W_{i,j}\|} \right)^{-1} \) when evaluating \( P(W_{i,j}) \). This empirical approach has
several expected benefits, including counteracting systematic biases present in the speech-to-text software. Indeed, using the empirical prior, if the speech-to-text software tends to (incorrectly) favor word \( w \), it will be assigned a high \( P(w) \) value, which has the effect of lowering the probability that a WPP containing \( w \) will be selected as being the “correct” one.

For either prior, we update \( W_{i,j} \) by first assigning a weight to each phrase that appears in \( \Psi_{i,j} \) using the above formula, and then select one of these phrases at random, where the probability of selecting a given phrase is proportional to its weight. Note that we do not need to consider any phrase \( R \) that does not appear in \( \Psi_{i,j} \), because \( P(R | \Psi_{i,j}) = 0 \) by assumption.

I close this section by noting that the prior distribution over the number of words in a phrase turns out to have no influence on the inference process. In every case, the \( \text{Poisson}(.) \) term is either a constant with respect to the value being sampled, or else, in the case of the update of \( W_{i,j} \), the term cancels out due to our Bayesian interpretation of the ToSpeech-ToText(.) function. Also, we note that the computationally expensive methods such as rejection sampling is not required to implement our Gibbs sampler because (1) most of the priors used are conjugate and (2) in those cases where the priors are not conjugate, the variables are discrete. If speed really is an issue, then an approximate variational algorithm [119] can be derived.

### 5.5 Experiments

In this section, I detail four experiments aimed at determining whether the STM variant on LDA has value for learning topic models over spoken text. Specifically, I apply both STM and LDA to the output of a speech-to-text software, and then see which topic model performs better on a variety of classification tasks. I stress that our goal is not to determine
whether STM can be used to build the best classifier over spoken text. In fact, our results will show that for highest accuracy, it is sometimes best not to use a topic model—or at least, that it is best to directly build a classifier using the set of phrases selected as being most likely by the STM model. This is not surprising, given that topic modeling is in essence a lossy data reduction. Instead, I begin with the assumption that one has decided to use topic modeling as a pre-processing step; given this, our goal is to determine whether the STM method does in fact build a model that is of higher quality than LDA.

5.5.1 Experimental Setup

Data Sets. I used three data sets in our experiments. The first data set is “synthetic” in the sense that it is produced via a process that is meant to simulate the output of a speech-to-text software. The benefit of simulating the speech-to-text software (as opposed to using an actual speech-to-text program) is that we can control the accuracy of the output. The second and third data sets were produced by using the CMU Sphinx4 speech-to-text software plus the VoxForge language and acoustic model to process a number of TedTalks (see www.ted.com/talks) and Open Yale courses (see oyc.yale.edu).

The first data set was created using the ubiquitous 20 Newsgroups data set. I assigned the classification label of ‘+1’ to articles coming from comp.* and ‘-1’ to those from sci.* newsgroup. To simulate the output of a speech-to-text software, I first split an input document into “phrases” of three consecutive words. I then generated a WPP for a phrase $W$ comprising words $w_1, w_2, \text{ and } w_3$ as follows. For $i$ in 1...3, I randomly chose a word $w_i'$ from the set $\{w_{i,j} \mid j \in \{1...10\}\}$, where $w_{i,j}$ denotes the $j$th closest word to $w_i$ according to phonetic edit distance* and the probability of selecting a particular $w_{i,j}$ is

*The phonetic edit distances are computed using the ARPAbet representation of the words in the CMU pronunciation dictionary.
proportional to $\frac{1}{j}$. The resulting alternative phrase $w'_1 \cdot w'_2 \cdot w'_3$ is likely to contain words that are phonetically close to $W$. After creating four alternative phrases in this manner, I generated the WPP weights for the phrases as a sample from a Dirichlet($\alpha$) distribution. Here $\alpha_k = \gamma^{ED_k}$, where $\gamma$ is a specified parameter and $ED_k$ is the edit distance between the $k$th candidate phrase and $W$.

This data-generation methodology allowed us to vary the accuracy of the simulated speech-to-text software by modifying $\gamma$, thereby modeling, for example, inaccuracies introduced due to background noise or poor audio quality. A smaller $\gamma$ value simulates a more accurate software, since only those phrases with words close to $W$ will have large weights. For example, here are the alternative WPPs output for the phrase “sound like there” for two different $\gamma$ values:

<table>
<thead>
<tr>
<th>Low precision ($\gamma = 0.99$)</th>
<th>High precision ($\gamma = 0.5$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>around lock bear 0.36</td>
<td>sound like there 0.86</td>
</tr>
<tr>
<td>sound like there 0.24</td>
<td>ounder lock bare 0.14</td>
</tr>
<tr>
<td>wound psych bare 0.16</td>
<td>mound bike bear 6.19E-4</td>
</tr>
<tr>
<td>around lyme dare 0.12</td>
<td>around lock bare 7.82E-7</td>
</tr>
<tr>
<td>mound psych bare 0.11</td>
<td>sounder bike bare 3.2E-11</td>
</tr>
</tbody>
</table>

Three versions of the first data set were created; a low precision version ($\gamma = 0.99$), a medium precision version ($\gamma = 0.8$), and a high precision version ($\gamma = 0.5$).

The second data set was created by selecting 36 TedTalks with tags “psychology” or “happiness”, and 37 with tags “climate change” or “environment”. I then used the Sphinx software to perform speech-to-text processing on the 73 talks. I first modified the software so that when a pause occurs in the input audio (or after relatively small number of audio frames have been transcribed), the software will output the ten most likely transcriptions, along with their scores. The scores are then converted to probabilities using the $n$-best
<table>
<thead>
<tr>
<th>Approach</th>
<th>Prior</th>
<th>Precision</th>
<th>Low precision ($\gamma = 0.99$)</th>
<th>Med precision ($\gamma = 0.8$)</th>
<th>High precision ($\gamma = 0.5$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SVM</td>
<td></td>
<td>STM</td>
<td>MLE</td>
<td>STM</td>
<td>MLE</td>
</tr>
<tr>
<td></td>
<td>Uniform</td>
<td>84.69</td>
<td>81.56</td>
<td>81.9</td>
<td>84.91</td>
</tr>
<tr>
<td></td>
<td>Empirical</td>
<td>85.22</td>
<td>83.74</td>
<td>83.68</td>
<td>85.82</td>
</tr>
<tr>
<td>SVM</td>
<td></td>
<td>Words Only</td>
<td>85.96</td>
<td>83.74</td>
<td>83.68</td>
</tr>
<tr>
<td>Naive Bayes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Uniform</td>
<td>80.1</td>
<td>79.3</td>
<td>77.4</td>
<td>80.2</td>
</tr>
<tr>
<td></td>
<td>Empirical</td>
<td>79.37</td>
<td>79.56</td>
<td>74.25</td>
<td>81.55</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Words+topic</td>
<td>81.33</td>
<td>79.64</td>
<td>78.58</td>
</tr>
</tbody>
</table>

Figure 5.1: Average accuracy for synthetic data
method [120], and a set of WPPs are created. The speech-to-text program is then reset, and the transcription process continues on. Although “chunking” the input in this way tended to discount the language model used by the speech-to-text software and increase the importance of the acoustic model, it gave the best results, in that it maximized the chance that the true phrase (or at least a large number of the words in the true phrase) would appear somewhere in the generated WPPs. Without chunking, all of the alternative transcriptions tended to be very similar to one another, and were mostly wrong. Chunking does tend to produce phrases flow poorly and lack continuity, but since we are performing bag-of-words classification where there is no notion of word ordering, this approach was the most obvious choice.

To actually create the corpus, I grouped ten consecutive phrases of each TedTalk to create a document—thus each document is around 50 words in length, and comprises a snippet of one of the TedTalks. In this way, I create 773 different documents. Documents created from a talk on “psychology” or “happiness” are labeled with a +1 and those from “climate change” or “environment” are labeled with a −1.

The third dataset was created using the same procedure as the second dataset, but for four Open Yale courses, namely ‘Financial Market’ (econ 252), ‘Introduction to Psychology’ (psyc 110), ‘The American Revolution’ (hist 116) and ‘Introduction to the New Testament History and Literature’ (rlst 152). Documents that are created using the same course are given the same classification label. Unlike the TedTalks dataset, audio files belonging to a given course have exactly one speaker. Moreover, Open Yale courses have less spontaneity than TedTalks. Finally, the current Voxforge model is well-suited for short voice commands, but is not powerful enough for continuous speaker-independent dictation. These factors, along with the respective scores assigned by Sphinx4, imply that the second dataset serves as better evaluation method for speaker-independent, continu-
uous speech, whereas the third dataset serves as better evaluation method for single-speaker, read speech.

**Testing procedure.** For each data set, I applied STM to learn a model with 100 topics (i.e. $K = 100$). I randomly selected 20% of the data to hold back as test data, and then trained both an SVM and a naive Bayes classifier over the remaining 80%, using off-the-shelf tools SVM$^{light}$, SVM$^{multiclass}$, and Weka 3. The SVM parameters were chosen carefully to obtain the best classifier possible.

For STM, the classifiers are trained in three different ways:

- **words only**: using the version of the document that the STM model determines is most likely (that is, using the values that maximize $P(W_{i,j})$ for each $i, j$), and I simply train the classifier on the vector of word counts for the document.

- **topic coeff**: using only the topic weight vector $\theta$.

- **words+topic**: using both the words and the topic weight vectors.

<table>
<thead>
<tr>
<th>Approach</th>
<th>STM</th>
<th>MLE</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>STM</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Prior</td>
<td>Uniform</td>
<td>Empirical</td>
</tr>
<tr>
<td>SVM</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Words Only</td>
<td>62.89</td>
<td>62.79</td>
</tr>
<tr>
<td>Topic coeff</td>
<td>55.8</td>
<td>55.6</td>
</tr>
<tr>
<td>Words+topic</td>
<td>62.88</td>
<td>62.7</td>
</tr>
<tr>
<td>Naive Bayes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Words Only</td>
<td>58.65</td>
<td>58.99</td>
</tr>
<tr>
<td>Topic coeff</td>
<td>52.25</td>
<td>50.89</td>
</tr>
<tr>
<td>Words+topic</td>
<td>58.95</td>
<td>57.53</td>
</tr>
</tbody>
</table>

Figure 5.2: Average accuracy for TedTalks data
I learned two different STM models; one using a *uniform* prior over phrases, and a second using an *empirical prior*. These models correspond to the two approaches for evaluating $P(W_{i,j})$ when updating word assignments, as discussed in the previous section.

For comparison and as a sanity check, I also used a maximum-likelihood-based approach, denoted as *MLE*. For each phrase, I chose the WPP that was weighted most highly; this approach is referred to as the MLE approach, and is equivalent to simply using the usual output of the speech-to-text software directly. Using the most-likely WPPs, I again learn three models: the “words only” model, the “topic coeff” model, and the “words+topic” model. Each resulting classifier is then tested on the 20% of the data that were held back.

This whole process is repeated 21 times, where each time a new model is learned and a different set of data are held back for testing. The average accuracy (that is, the fraction of the predicted labels that are correct) is computed over the 21 repetitions.

**Experiments.** I ran four experiments. In the first experiment, I used three 20 newsgroup “synthetic” datasets, each generated using different $\gamma$ to represent a different level of pre-
<table>
<thead>
<tr>
<th>Approach</th>
<th>STM</th>
<th>MLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prior</td>
<td>Uniform</td>
<td>Empirical</td>
</tr>
<tr>
<td>SVM</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Words Only</td>
<td>24.59</td>
<td>24.84</td>
</tr>
<tr>
<td>Topic coeff</td>
<td>85.89</td>
<td>89.07</td>
</tr>
<tr>
<td>Words+topic</td>
<td>25.08</td>
<td>25.1</td>
</tr>
<tr>
<td>Naive Bayes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Words Only</td>
<td>87.81</td>
<td>88.11</td>
</tr>
<tr>
<td>Topic coeff</td>
<td>68.66</td>
<td>61.88</td>
</tr>
<tr>
<td>Words+topic</td>
<td>81.6</td>
<td>81.16</td>
</tr>
</tbody>
</table>

Figure 5.4: Average accuracy for Yale data (4 classes)

cision of speech-to-text. For second and third experiments, I used the TedTalks and Open Yale courses dataset respectively, and used binary classification to evaluate STM. Whereas in third experiment I used only two courses (econ252 and psyc110), in the fourth experiment I used all four courses in the Open Yale dataset, each having a different classification label.

The average accuracy over 21 runs for these experiments is given in tabular form in Figures 1–4. To statistically compare the average accuracies of the three topic models—STM with uniform prior, STM with empirical prior, and MLE—for a given choice of experiment, dataset, and classifier, I ran the Tukey HSD (Honest Significant Difference) test using a significance level of 0.01. If an average accuracy value is significantly higher than the other two values in its row, then it appears as the unique boldface entry. If the highest two values are not significantly different from each other but are both significantly higher than the lowest value, then the two “winners” are both displayed in boldface. If the middle value is significantly different from both the highest and the lowest values, then it appears in italics. Finally, if none of the values can be distinguished from each other, then they all appear in...
5.5.2 Discussion

I begin with the most obvious conclusion: the only three cases where the MLE-based method is significantly better than one of the STM methods is for the parameter “topic co-eff” for first experiment with high precision, and for parameters “words only”, “words+topic” for the third experiment. However, even in these mentioned cases, the other STM method performs at least as well as (or better than) the MLE-based method. In every other case, both STM methods either significantly outperformed the MLE-based method or were equally as good.

In the experiments over synthetic data, the improvement in accuracy using STM was generally greater than for the real data (as I will discuss in more detail in a moment). But I point out that our goal was not to build a classifier with significantly higher accuracy: our goal was to build a topic model that is of higher quality, and these results seem to confirm that the STM model is of higher quality. This is particularly pronounced when considering the Naive Bayes results for models that are constructed using only the topic coefficients. Given the relative simplicity and lack of robustness of Naive Bayes, one might expect that Naive Bayes is most sensitive to a poor choice of features (in this case, topics). In every case over the real data, building the Naive Bayes classifier using one of the STM models outperformed the MLE-based LDA, often quite significantly. For the 2-class Yale data, STM with a uniform prior obtained 83% accuracy, whereas LDA obtained less than 80% accuracy. On the TedTalks data, STM with a uniform prior acheived 52% accuracy, whereas MLE was worse than 50%—the level of a random classifier. And on the 4-class Yale data, STM with a uniform prior achieved nearly 69% accuracy, which is significantly better than the 59.6% accuracy obtained via LDA.
I close by again noting that the improvement in accuracy for the real data sets, while often statistically significant, does not approach the improvement in accuracy observed for the synthetic data. I conjecture that this is related to the way in which I extract the alternate phrases from the speech-to-text engine, i.e. using the \(n\)-best list [120]. Even using chunking, in many cases, the output phrases contained in the \(n\)-best list are quite similar to each other, and differ only in the last word or two: “the cow jumped over the moon”, “the cow jumped over too soon”, “the cow jumped over cocoon”, and so on. This seems to provide the STM model with less information about the possible alternatives than in the synthetic data than I might like. As a future work, I plan to investigate other approaches to generating the alternative phrases, such as random sampling over word-lattices to generate the WPPs. I conjecture that this could provide a broader set of possible phrases, increasing the likelihood that one of them is close to the correct phrase, which might increase the performance of the STM model even more when compared to a simple LDA model.

I close by examining the low precision, synthetic results, which is the one case where the STM-based classifiers were not always superior. In the “topic coeff” case, the MLE-based LDA did a better job than either, STM-based model, though the STM-based model outperformed using the other two options. An explanation for why the STM-based model did better is that the MLE-based approach selected the correct phrase only 58,355 times (out of 289,601 total phrases), whereas the STM-based approach using a uniform prior selected the correct phrase 128,720 times (in every case—not just this one—the STM approach was able to substantially boost the number of correct words compared to the MLE approach). So the set of words it had to work with is far more accurate.

It is more difficult to explain why in the “topic coeff” case, STM did not outperform the LDA-based approach over the low precision, synthetic data. One explanation is that while the accuracy of the selected phrases is much higher under STM, it is simply not high
enough (less than 50%) to result in a better topic model. Still, even in this case, the best SVM classifier uses the output of the STM model (for naive Bayes, the best classifier does not use the individual words, since naive Bayes tends to do poorly with so many features).
Chapter 6

Conclusions

In this thesis I have summarized the three research projects I worked on during my PhD studies, namely, Online aggregation for large MapReduce jobs, Distributed gradient descent using online aggregation and Topic models over spoken language. In the first project I developed a bayesian model that outputted robust statistical online estimates with confidence bounds for aggregate query on MapReduce. In the second project I build on my first work to develop a framework for performing fast distributed gradient descent on MapReduce. In the third project I developed a bayesian model that extended Latent Dirichlet Allocation for audio data.
## Appendix A

### Notation

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X$</td>
<td>random variable (upper case italics)</td>
</tr>
<tr>
<td>$\mathbf{x}$</td>
<td>vector (lower case bold)</td>
</tr>
<tr>
<td>$\mathbf{X}$</td>
<td>matrix (upper case bold)</td>
</tr>
<tr>
<td>$x$</td>
<td>datapoint or temporary variables of an algorithm (lower case italics)</td>
</tr>
<tr>
<td>$\mathcal{X}$</td>
<td>constant (upper case calligraphic)</td>
</tr>
<tr>
<td>$\mathcal{D}$</td>
<td>number of dimensions of data</td>
</tr>
<tr>
<td>$\mathcal{N}$</td>
<td>number of training examples</td>
</tr>
<tr>
<td>$\mathcal{M}$</td>
<td>number of machines</td>
</tr>
<tr>
<td>$\mathcal{P}$</td>
<td>total number of mappers</td>
</tr>
<tr>
<td>$\lfloor x \rfloor$</td>
<td>Floor of value $x$</td>
</tr>
<tr>
<td>$\mathbb{I}(x)$</td>
<td>Indicator function, $\mathbb{I}(x) = 1$ if $x$ is true else $0$</td>
</tr>
<tr>
<td>$\arg \max_x f(x)$</td>
<td>The value of $x$ that maximizes the function $f$</td>
</tr>
<tr>
<td>$</td>
<td>x</td>
</tr>
<tr>
<td>$\nabla$</td>
<td>Vector of first derivates</td>
</tr>
<tr>
<td>$\nabla^2$</td>
<td>Hessian matrix of second derivates</td>
</tr>
<tr>
<td>$</td>
<td>\mathbf{M}</td>
</tr>
<tr>
<td>$\mathbf{M}^{-1}$</td>
<td>inverse of matrix $\mathbf{M}$</td>
</tr>
</tbody>
</table>
\( M^T \) transpose of matrix \( M \)

\( diag(x) \) diagonal matrix obtained from the vector \( x \)

\( d(x_1, x_2) \) euclidean distance between points \( x_1 \) and \( x_2 \)

\( X \sim \text{Normal}(\mu, \sigma) \) the random variable \( X \) is distributed according to Normal distribution with mean \( \mu \) and standard deviation \( \sigma \)

\( \mathbb{E}[X] \) Expected value of the random variable \( X \)

\( \mu_X \) mean of the random variable \( X \)

\( \mu_\Phi \) mean of the posterior \( \Phi \)

\( \sigma_X \) standard deviation of the random variable \( X \)

\( \sigma_\Phi \) standard deviation of the posterior \( \Phi \)

\( \Sigma \) covariance matrix

\( \Theta \) the parameter to be minimized or maximized using gradient descent algorithm

\( t \) datapoint (or tuple) from HDFS for gradient descent algorithm

\( t \) dataset in HDFS for gradient descent algorithm

\( L \) loss function

\( V \) set of observed data

\( U \) set of unobserved data

\( Z \) set of observed and unobserved data
Bibliography


141


