Web-scale Topic Models in Spark: An Asynchronous Parameter Server

Rolf Jagerman
Department of Computer Science, ETH Zürich, Universitätstrasse 6, 8092 Zürich, Switzerland
ROLFJ@student.ethz.ch

Carsten Eickhoff
Department of Computer Science, ETH Zürich, Universitätstrasse 6, 8092 Zürich, Switzerland
CARSTEN.EICKHOFF@inf.ethz.ch

Abstract

In this paper, we train a Latent Dirichlet Allocation (LDA) topic model on the ClueWeb12 data set, a 27-terabyte Web crawl. We extend Spark, a popular tool for performing large-scale data analysis, with an asynchronous parameter server. Such a parameter server provides a distributed and concurrently accessed parameter space for the model. A Metropolis-Hastings based collapsed Gibbs sampler is implemented using this parameter server achieving an amortized $O(1)$ sampling complexity. We compare our implementation to the default Spark implementations and show that it is several orders of magnitude more scalable without sacrificing model quality. A topic model with 1,000 topics is trained on the full ClueWeb12 data set, uncovering some of the prevalent themes that appear on the Web.

1. Introduction

Probabilistic topic models are an immensely useful tool for discovering a set of themes that underlie a text corpus. Each topic is represented as a multinomial probability distribution over a set of words, giving high probability to words that co-occur frequently and small probability to those that do not. For example, a topic about food will assign high probability to words like 'recipe', 'meat', 'spices', etc. and small probability to all other words. Conversely, a topic about jewelry might assign high probability to words like 'gold', 'diamond' and 'ring', while giving small probability to the rest.

Classical inference algorithms for topic models do not scale well to very large data sets. This is unfortunate because, like many other machine learning methods, topic models would benefit from a large amount of training data. When trying to compute a topic model on a Web-scale data set in a distributed setting, we are confronted with two major challenges:

1. How do different processors keep their model synchronized?
2. How do we reduce the computational complexity to manageable levels?

Spark recently emerged as a popular tool for performing large-scale data analysis (Zaharia et al., 2012). It has seen a lot of success in both industry and academia. Unfortunately, Spark is bounded to the typical map-reduce programming paradigm which is inherently synchronous. Inference algorithms for LDA are not easily implemented in such a paradigm since they rely on a large mutable parameter space that is updated asynchronously.

We aim to solve these problems by extending Spark with an asynchronous parameter server. Such a parameter server provides a distributed and concurrently accessed parameter space for the model. A distributed LDA inference algorithm based on LightLDA (Yuan et al., 2015) is implemented in Spark using this parameter server.

The remainder of this paper is structured as follows: Section 2 describes the architecture of the Spark-compatible parameter server. Section 3 provides a short description of the LightLDA algorithm and our implementation of it in Spark. The setup and results of the experiments are described in Section 4. Finally, the paper is concluded in Section 5.

2. Parameter Server

The parameter server is an architecture for distributing large matrices and vectors to multiple machines. It typically offers two basic operations:
We implemented a high-performance parameter server called “Glint” in the Scala programming language that uses the Akka framework. Akka is “a toolkit and runtime for building highly concurrent, distributed, and resilient message-driven applications on the JVM” (LightBend, 2016). It provides high level abstractions that simplify much of the network communication while retaining high performance. We chose to use Scala and Akka to ensure compatibility with Spark, which itself is written in Scala and uses Akka extensively. Although Glint was developed for the purpose of collapsed Gibbs sampling and count tables it can be applied much more broadly and is potentially useful for a variety of large-scale machine learning tasks such as sparse logistic regression (Li et al., 2014) and deep learning (Dean et al., 2012).

The goal of the parameter server is to store a large distributed matrix and provide a user with fast queries and updates to this matrix. In order to achieve this it will partition and distribute the matrix to multiple machines. Each machine only stores a subset of rows. The user interacts with the matrix through the pull and push operations and is unaware of the physical location of the data. By using the parameter server, the user acts solely on a virtual view of the matrix (see Figure 1).

2.1. Data Storage

The parameter servers store their data in main memory. Since fast random updates are necessary it uses a dense representation. Partial matrices are represented by two-dimensional arrays which are stored in row-major order. Vectors are represented by single-dimensional arrays.

Since the system is implemented in Scala, we use the Java Virtual Machine (JVM) to run it. Internally, the JVM can store arrays of primitives as contiguous blocks of memory. By carefully implementing the parameter server to use the appropriate specialized primitives, we prevent boxing and thus garbage collection overhead. This is especially important since the parameter server should have high throughput and cannot afford long ‘stop-the-world’ garbage collection routines, which would effectively stop all user code to run garbage collection (Oracle, 2016). For large JVM heap sizes this could take an unacceptable amount of time, making the parameter server seem unresponsive.

2.2. Partitioning

The parameter server partitions matrices row-wise in a cyclical fashion. This means that the first row will be stored on the first parameter server, the second row on the second parameter server, etc. Such a partitioning scheme is easy to implement and efficient.

2.3. Pull action

Whenever a user wants to retrieve certain entries from the matrix they will call the pull method. This method triggers a pull request with a specific set of row and column indices that should be retrieved. The request is split up into smaller requests based on the partitioning of the matrix such that there will be at most one request per parameter server. Akka provides an ‘at-most-once’ guarantee on message delivery. This is problematic because it is impossible to know whether a message sent to a parameter server is lost or just takes a long time to compute. However, since pull requests do not modify the state of the parameter server, we can safely retry the request multiple times until a successful response is received. To prevent flooding the parameter server with too many requests, we use an exponential backoff timeout mechanism. Every time a request times out, the timeout for the next request is increased exponentially. If after a specified number of retries there is still no response, we consider the pull operation as failed and let the user know.

2.4. Push action

In contrast to a pull request, a push requests will modify the state on the parameter servers. This means we cannot naively resend requests on timeout because if we were to accidentally process a push request twice it would result in a wrong state on the parameter server. For this reason we want to ensure ‘exactly-once’ message delivery.

A hand-shaking protocol was created to guarantee ‘exactly-once’ delivery on push requests with minimal overhead.

---

[1] https://github.com/rjagerman/glint
2.5. Application to LDA

We wish to use the parameter server to compute a large Latent Dirichlet Allocation (LDA) model on a large data set. Since we will use collapsed Gibbs sampling, the parameter server will act as a count table and either increment or decrement certain entries. In this scenario, a push implementation that aggregates values through addition is ideal. Addition is both commutative and associative which means the order of operations is irrelevant. This eliminates the need for complex locking schemes and conflict resolution that is typical in most key-value storage systems. It also allows us to implement the system very efficiently using atomic operations.

3. Distributed LDA

We implemented the LightLDA algorithm (Yuan et al., 2015) in Spark using our implementation of the parameter server. LightLDA is a Collapsed Gibbs sampling (Griffiths & Steyvers, 2004) algorithm that uses a Metropolis-Hastings approximation (Metropolis et al., 1953). Collapsed Gibbs sampling for LDA is a Markov Chain Monte-Carlo type algorithm that assigns a topic \( z \in \{1 \ldots K\} \) to every token in the corpus. It then repeatedly re-samples the topic assignments \( z \). To do this, we need to keep track of the statistics \( n_k \), \( n_{wk} \) and \( n_{dk} \):

- \( n_k \): Number of times any word was assigned topic \( k \)
- \( n_{wk} \): Number of times word \( w \) was assigned topic \( k \)
- \( n_{dk} \): Number of times a token in document \( d \) was assigned topic \( k \)

It is clear that the document-topic counts \( n_{dk} \) are document-specific and thus local to the data and need not be shared across workers. However, the word-topic counts \( n_{wk} \) and topic counts \( n_k \) are global and require sharing. The parameter server provides a shared interface to these values in the form of a distributed matrix storing \( n_{wk} \), and a distributed vector storing \( n_k \).

LightLDA factorizes the original collapsed Gibbs distribution into two parts: a document proposal distribution \( P_d \) and a word proposal distribution \( P_w \) (see Equation 1). The LightLDA paper provides methods to draw from \( P_d \) and \( P_w \) in amortized constant time. This is important because sampling billions of tokens is computationally infeasible if every sampling step would use \( O(K) \) operations, where \( K \) is a potentially large number of topics. LightLDA provides corresponding acceptance probabilities \( \pi_d \) and \( \pi_w \). Algorithm 1 describes the distributed LightLDA method.

\[
P(z = k) \propto \left( \frac{n_{dk} - dw}{p_d} + \alpha \right) \frac{n_{wk} - dw + \beta}{p_w} \frac{n_k - dw + V\beta}{p_k} \tag{1}
\]

**Algorithm 1** The distributed LightLDA algorithm.

```plaintext
for \( d \in D \) in parallel do
    for \( (w, z_{old}) \in d \) do
        \( z_{new} \leftarrow z_{old} \)
        for \( i = 1 \ldots mh\text{-steps} \) do
            \( z_{proposal} \sim P_w(z) \)
            \( z_{new} \leftarrow z_{proposal} \) with probability \( \pi_w \)
            \( z_{proposal} \sim P_d(z) \)
            \( z_{new} \leftarrow k_{proposal} \) with probability \( \pi_d \)
        end for
        Replace \( (w, z_{old}) \) with \( (w, z_{new}) \) in \( d \)
        Update counts \( n_k \), \( n_{wk} \) and \( n_{dk} \) asynchronously
    end for
end for
```
Table 1. Measurements on perplexity, runtime and shuffle write size for our implementation, Spark EM LDA and Spark Online LDA using subsets of ClueWeb12 B13. We vary either the data set size (2.5% - 10%) or the number of topics (20 - 80). The best result per experiment is highlighted in bold.

4. Experiments

We compare our system to two existing LDA algorithms in Spark: The variational EM algorithm and the online algorithm on small subsets of ClueWeb12. We vary either the number of topics or the size of the data set to measure how the different systems scale with those variables. Additionally, we have used our system to train an LDA model with 1,000 topics on the full 27-terabyte ClueWeb12 data set, something that was not possible with the default Spark implementations. The system was run on a computing cluster with 30 nodes, which account for a cluster-wide total of 480 CPU cores and 3.7TB RAM. The nodes are interconnected over 10Gb/s ethernet.

Table 1 shows the results of the experiments when compared against the default Spark implementations. We observe that the perplexity is roughly equal for all algorithms. However, our implementation has a significantly smaller runtime. The default Spark algorithms have difficulty scaling to the full ClueWeb12 data set due to an increase in runtime and/or shuffle write. Our implementation was able to compute an LDA model on the full ClueWeb12 data set in roughly 80 hours. See Figure 3 for the perplexity of the model as it is training. It converges to a perplexity of roughly 4,250. We have made the final 1,000-topic LDA model publicly available in CSV format.

5. Conclusion

In this paper, we described a Spark-compatible asynchronous parameter server. It is used to compute an LDA topic model on a Web-scale data set: ClueWeb12. We conclude our work by revisiting the challenges that were presented in the introduction: How do we keep the model synchronized across multiple processors? And how do we reduce the computational complexity to manageable levels? The first challenge is solved using a parameter server, which provides a distributed and concurrently accessed parameter space for the model. The second challenge is solved by using the LightLDA algorithm, which reduces the sampling complexity to $O(1)$.

By implementing a Spark-compatible parameter server we are able to extend Spark beyond the synchronous map-reduce paradigm. This enables us to implement a highly efficient and distributed collapsed Gibbs sampler for LDA in Spark, something that was previously not possible. Finally, we present several prospects for future work:

- Use the parameter server to implement various other algorithms in Spark such as support vector machines, logistic regression, deep learning, etc.
- Improve the parameter server by achieving better reliability. For example, instantaneous failover can be achieved by implementing distributed hash tables and replicating the model to multiple nodes.
- Investigate sparse representations of matrices and vectors on the parameter servers. This would grant additional scalability, as it would reduce both memory usage and network communication overhead.

http://cake.da.inf.ethz.ch/clueweb-topicmodels/
References


LightBend. Akka, 2016. URL \url{http://akka.io/}.


