Supplementary Material (AISTATS 2018): Parallel and Distributed MCMC via Shepherding Distributions

Arkabandhu Chowdhury
Rice University

Chris Jermaine
Rice Univeristy

A LATENT DIRICHLET ALLOCATION

We consider a data parallel implementation of a sampler for Latent Dirichlet Allocation (LDA) [Blei et al., 2003]. Our focus is on examining samplers that are appropriate for a distributed environment where a subset of a large data set is stored on each compute node.

Briefly, LDA is a popular topic model. In LDA, the jth topic ψ_j is a set of word probabilities sampled from a Dirichlet(β) prior, and the topic distribution ρ_d for document d is sampled from a Dirichlet(α) prior. Then, document d is generated by, for each term t in the document, first choosing a topic $z_{d,t} \sim$ Multinomial(ρ_d) and then choosing the word that appears as term t as $w_{d,t} \sim$ Multinomial($\psi_{z_{d,t}}$). The goal is learning an LDA model to infer the unseen model components (in particular, the various ψ_j vectors) from an observed data set.

A.1 DISTRIBUTED LDA

The standard Gibbs sampler for LDA integrates out each ψ_j and ρ_d and instead attempts to obtain a posterior distribution on the number of each dictionary word assigned to each topic. However, the resulting sampler is nearly impossible to parallelize efficiently in a way that guarantees ergodicity. The problem is that after integrating out ψ_j and ρ_d, all of the threads in a distributed compute cluster will need to constantly access and update those word-topic counts. Facilitating this in both a consistent and efficient way is not possible, and so the standard solution is to simply to accept inconsistency and ignore the fact that the resulting chain is non-ergodic.

However, in our experience, such a non-ergodic chain can converge to an inferior model. To ensure ergodicity, in our own distributed LDA Gibbs sampler, we choose not to integrate out each ψ_j and ρ_d, and to instead maintain explicit values for each of these terms. The resulting Gibbs sampler is then quite simple, and nearly all updates are conjugate. It is also easily distributed so as to ensure ergodicity, because ρ_d is local to each document. Assume that the set of documents has been partitioned across the cluster so each machine has a subset of the data. Then, for a document d, ρ_d can be updated locally on the machine on which document d is located, as outlined in Algorithm 1.[1] After this update is performed in parallel around the cluster in an epoch, the number of times each topic was responsible for producing each word in the dictionary is computed using a distributed aggregation, and the results are used to update each ψ_j before the next epoch of the computation begins.

Algorithm 1 Sampler for $\rho_d^{(k)}$ given $\rho_d^{(k-1)}$, $\{\psi_j^{(k)}\}$

```
ρ_d^(k) ← ρ_d^(k-1)
for a ∈ {1...10} do
  for each term t in document d do
    Sample z_{d,t} where Pr[z_{d,t} = γ] ∝ ρ_{d,γ}^(k) ψ_{γ,t}^(k)
  end for
  Compute q where q_j = |{z_{d,t} | z_{d,t} = 1}|
  ρ_d^(k) ~ Dirichlet(α + q)
end for
```

A.2 SHEPHERDED LDA

Since our goal is data parallelism, we will develop an MCMC simulation utilizing the most general formulation of the method of SDs, corresponding to Equation 2 of Section 2 in our paper:

$$f_1(x_{c_1})g(\theta)\prod_{i=2}^{n} f_i(x_{c_i} \mid \theta).$$

Distribution. Assume a compute cluster consists of $n-1$ machines; our shepherded algorithm will maintain $n-1$ shepherded chains and one primary chain. We partition the input data set D into $D_2 \cup D_3 \cup ... \cup D_n$ and locate each D_i on a different machine. The machine holding D_i will be responsible for maintaining the shepherded chain tasked with sampling x_{c_i}, using only the data in D_i. Since x_{c_1} corresponds to the primary chain, there is no D_1 to be operated on by the chain sampling x_{c_1}.

In the case of data-parallel LDA, a sampled x_{c_i} consists of a complete set of word-in-topic probabilities $\{\psi_j\}_{c_i}$,

[1] The ten iterations of the inner loop are chosen somewhat arbitrarily; the idea is to sample until $\rho_d^{(k)}$ is consistent with $\{\psi_j^{(k)}\}$.

maintained locally, learned over D_i. In addition, ρ_d for each $d \in D_i$ is maintained as a local set of auxiliary variables. Hence the machine holding D_i runs a Markov chain that generates samples from $f_{LDA}(\{\psi_j\}_{c_i}|D_i, \alpha, \beta)$, where f_{LDA} corresponds to the PDF for the LDA model.

In addition, the primary chain corresponding to x_{c_1} will be maintained in parallel by all machines, using a standard distributed LDA algorithm (such as the one described in the previous subsection).

Shepherding Methodology. The question then becomes, how to shepherd the $n-1$ chains that are machine-local? This is a crucial question, because all of the chains are executed independently over different subsets of the data. Unless they are shepherded effectively, each chain is going to learn a different set of topics from a different subset of data, none of which is likely to be of high quality with respect to the entire data set.

We choose to shepherd the hyperparameter β controlling the generation of the per-topic word probability, so the shepherding parameter $\theta = \langle \beta'_1, \beta'_2, ... \rangle$, with one shepherded vector per topic. In this way, if a shepherded chain discovers that word w is important to topic j in its own subset of the data, it can cause the value of $\beta'_{j,w}$ to increase, which will tend to increase the importance of word w in topic j for *all* of the shepherded chains. We choose a Gamma(a, b) prior on each $\beta'_{t,w}$. It is easy to develop a rejection sampler that is able to sample from $P(\beta'_{j,w}|\{\psi_j\}_{c_i}$ forall $i \geq 2)$ efficiently.

Given this, the shepherded algorithm works as follows. In an epoch, each machine performs one or more rounds of sampling of its local $\{\psi_j\}_{c_i}, \{\rho_d | d \in D_i\}$ values. Also in an epoch, all machines perform one iteration of the sampler for the primary chain. At the end of the epoch, a distributed aggregation is then used to update θ, as well as the state of the primary chain. An epoch completes with an attempt to swap the primary chain with one of the shepherded chains, as we discuss now.

Swapping. An epoch ends by attempting to swap the identity of the primary chain with the shepherded chain named by c_i, for a randomly selected $i \neq 1$. The swap is accepted with probability

$$\frac{f_1(x_{c_i}) f_i(x_{c_1}|\theta)}{f_1(x_{c_1}) f_i(x_{c_i}|\theta)}$$

where:

$$\begin{aligned} f_1(x_{c_i}) &\propto f_{LDA}(D|\{\psi_j\}_{c_i}\alpha, \beta) \\ f_i(x_{c_1}|\theta) &\propto f_{LDA}(D_i|\{\psi_j\}_{c_1}, \alpha, \{\beta'_j\}) \\ f_1(x_{c_1}) &\propto f_{LDA}(D|\{\psi_j\}_{c_1}, \alpha, \beta) \\ f_i(x_{c_i}|\theta) &\propto f_{LDA}(D_i|\{\psi_j\}_{c_i}, \alpha, \{\beta'_j\}) \end{aligned}$$

Note that computing each of these four terms is a perplexity computation. If the swap is accepted, c_i and c_1 exchange values and the old c_i becomes the primary chain.

A.3 EVALUATION

To evaluate the shepherded LDA algorithm, we consider the case where we have distributed an LDA computation over a cluster of ten machines. We consider two options for running a distributed LDA computation. In the first, we run the distributed Gibbs sampler described in this section. In the second, we run the shepherded sampler. As before, we measure the LLH achieved by the sampler as a function of the amount of computation performed.

Again, this forces us to consider exactly how we measure the amount of computation performed in a fair way. After some thought, we decided to measure this as a function of the number of distributed aggregations performed. That is, in the case of the "vanilla" distributed LDA, Algorithm 1 is run over each document in the corpus, and then a distributed aggregation is performed to collect the statistics necessary to re-sample all of the ψ_j values. Thus, we have one aggregation each time that Algorithm 1 is run over the entire corpus.

Shepherded LDA, in contrast, performs a variety of different computations in each epoch. Algorithm 1 is run over each document in the corpus twice: once for the primary chain, and once for one of the shepherded chains. Then, the primary chain must perform a distributed aggregation to update each ψ_j associated with the primary chain. Next, each shepherded chain must update its own ψ values, and those are used to update the shepherding parameters; this is a second distributed aggregation. Finally, the perplexity computation required to check for a swap requires a third distributed aggregation.[2] Given this, in our experiments, when comparing the distributed Gibbs sampler with shepherded LDA, we allow the Gibbs sampler to run for three epochs corresponding to every epoch of the shepherded sampler.

We ran the two samplers on two different data sets: the 20 Newsgroups data[3], and a corpus of Wikipedia articles[4]. For both data sets, we used 100 topics and a dictionary size of 10,000 words.

References

D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent dirichlet allocation. *JMLR*, 3(Jan):993–1022, 2003.

[2]Note that in practice, there is a lot freedom to vary this sequence. For example, one may decide to attempt a swap or to update the primary chain only periodically, favoring updates to the shepherded chains. Exploring these options is left to future work.

[3]`kdd.ics.uci.edu/databases/20newsgroups/20newsgroups.html`

[4]`dumps.wikimedia.org/enwiki/`