Sketchy Inference: Towards Streaming LDA

Jean-Baptiste Tristan
Oracle Labs
jean.baptiste.tristan@oracle.com

Michael L. Wick
Oracle Labs
michael.wick@oracle.com

Joseph Tassarotti
Carnegie Mellon University
jtassaro@andrew.cmu.edu

Abstract

Recent developments in inference algorithms based on stochastic Expectation-maximization or stochastic cellular automata (SCA) have made it possible to employ a variety of randomized data structures that are unavailable to the dominant inference methods in the Bayesian toolkit, including collapsed Gibbs sampling and stochastic variational inference (SVI). Equipped with this recent capability, we make progress towards a *true* streaming inference algorithm for LDA that makes novel use of these random data-structures. We mean "true" in the sense that the algorithm avoids entirely the need for pre-computation, which many current "streaming" variants of latent Dirichlet allocation (LDA) must perform. We find that despite using various randomized data-structures to represent the sufficient statistics, the inference algorithm converges to similar perplexities as more conventional LDA while producing equally interpretable topics.

1 Introduction

Sometimes, we must apply latent variable models to streaming data because, for example, news or social media feeds provide a constant source of new material. Indeed, we may want to keep a topic-model like latent Dirichlet allocation (LDA) up-to-date in response to such a data stream [2].

There are two important considerations when designing a streaming LDA model: model considerations and inference considerations. Model considerations are important for designing technically sound strategies for folding in the new data. Inference considerations are also important because in order for a streaming model to function in the wild, it must cope with the growing volume of data without having to constantly reallocate data-structures.

While there has been some progress towards designing a streaming version of LDA, most of the work has considered the problem from the perspective of the model while ignoring inference considerations [9, 1]. In particular, many approaches overlook the following questions, which are typically answered via a pre-processing step that is not ordinarily possible in a streaming setting:

1. What is the size of the vocabulary V?
2. How to encode words as integers?

There are answers to these questions that avoid pre-processing, but most have limitations. Non-parametric models avoid the need to pre-compute V, but they are much slower than their parametric counterparts. Feature hashing avoids the need to encode the words, but bloats the size of the arrays storing the sufficient statistics. We could make educated guesses about the answers to the aforementioned questions, but then we would inevitably have to re-initialize the data-structures or re-run inference from scratch as the growing volume of data renders our guesses inadequate.

On the other hand, there exist a variety of data-structures specifically suited for addressing these types of questions in the streaming environment. In this paper, we make progress toward a streaming version of LDA by employing randomized data structures to eliminate pre-computing answers to the

31st Conference on Neural Information Processing Systems (NIPS 2017), Long Beach, CA, USA.

two questions above. We employ count-min sketches to compress the sufficient statistics of the model. The count-min sketches avoids the need to know V *a priori* and also directly enables feature-hashing, which addresses the problem of encoding the words into indices.

We evaluate our streaming LDA inference algorithm and find that it achieves similar perplexities as more conventional inference that rely on pre-computation, while incurring little computational overhead and producing good quality topics.

2 Background

2.1 Latent Dirichlet Allocation

Latent Dirichlet Allocation [2] (LDA) is a probabilistic topic model used for unsupervised learning. The model needs to be fitted to some data, an operation referred to as learning or training. The model is defined as follows.

M	(Number of documents)
$\forall m \in \{1..M\}, N_m$	(Length of document m)
V	(Size of the vocabulary)
K	(Number of topics)
$\boldsymbol{\alpha} \in \mathbb{R}^K$	(Hyperparameter controlling documents)
$\boldsymbol{\beta} \in \mathbb{R}^V$	(Hyperparameter controlling topics)
$\forall m \in \{1..M\}, \theta_m \sim Dir(\boldsymbol{\alpha})$	(Distribution of topics in document m)
$\forall k \in \{1..K\}, \phi_k \sim Dir(\boldsymbol{\beta})$	(Distribution of words in topic k)
$\forall m \in \{1..M\}, \forall n \in \{1..N_m\}, z_{mn} \sim Cat(\theta_m)$	(Topic assignment)
$\forall m \in \{1..M\}, \forall n \in \{1..N_m\}, w_{mn} \sim Cat(\phi_{z_{mn}})$	(Corpus content)

There are a variety of different inference algorithms for fitting such a model. One algorithm that has become popular [20, 3, 12] to train LDA on very large datasets is termed SCA [18], which we present in Figure 1.

SCA for LDA has the following parameters: I is the number of iterations to perform, M is the number of documents, V is the size of the vocabulary, K is the number of topics, $N[M]$ is an integer array of size M that describes the shape of the data w, α is a parameter that controls how concentrated the distributions of topics per documents should be, β is a parameter that controls how concentrated the distributions of words per topics should be, $w[M][N]$ is ragged array containing the document data (where subarray $w[m]$ has length $N[m]$), $\theta[M][K]$ is an $M \times K$ matrix where $\theta[m][k]$ is the probability of topic k in document m, and $\phi[V][K]$ is a $V \times K$ matrix where $\phi[v][k]$ is the probability of word v in topic k. Each element $w[m][n]$ is a nonnegative integer less than V, indicating which word in the vocabulary is at position n in document m. The matrices θ and ϕ are typically initialized by the caller to randomly chosen distributions of topics for each document and words for each topic; these same arrays serve to deliver "improved" distributions back to the caller.

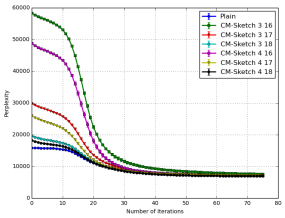
The algorithm uses three local data structures to store various statistics about the model (lines 2–4): $tpd[M][K]$ is a $M \times K$ matrix where $tpd[m][k]$ is the number of times topic k is used in document m, $wpt[V][K]$ is an $V \times K$ matrix where $wpt[v][k]$ is the number of times word v is assigned to topic k, and $wt[K]$ is an array of size K where $wt[k]$ is the total number of time topic k is in use. We actually have two copies of each of these data structures because the algorithm alternates between reading one to write in the other, and vice versa.

The SCA algorithm iterates over the data to compute statistics for the topics (loop starting on line 9 and ending on line 32). The output of SCA are the two probability matrices θ and ϕ, which need to be computed in a post-processing phase that follows the iterative phase. This post-processing phase is similar to the one of a classic collapsed Gibbs sampler. In this post-processing phase, we compute the θ and ϕ distributions as the means of Dirichlet distributions induced by the statistics.

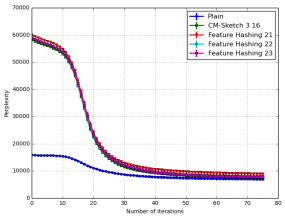
In the iterative phase of SCA, the values of θ and ϕ, which are necessary to compute the topic proportions, are computed on the fly (lines 21 and 22). Unlike the Gibbs algorithm, where in each

```
1:  procedure SCA(int I, int M, int K, int N[M], float α, float β, int w[M][N])
2:      local array int tpd[2][M][K]
3:      local array int wpt[2][H][R_2][K]
4:      local array int wt[2][K]
5:      initialize array tpd[0]                          ▷ Randomly chosen distributions
6:      initialize array wpt[0]                          ▷ Randomly chosen distributions
7:      initialize array wt[0]                           ▷ Randomly chosen distributions
8:      ▷ The main iteration
9:      for i from 0 through (I ÷ 2) − 1 do
10:         for r from 0 through 1 do
11:             clear array tpd[1 − r]                   ▷ Set every element to 0
12:             clear array wpt[1 − r]                   ▷ Set every element to 0
13:             clear array wt[1 − r]                    ▷ Set every element to 0
14:             ▷ Compute new statistics by sampling distributions
15:             ▷   that are computed from old statistics
16:             for all 0 ≤ m < M do
17:                 for all 0 ≤ n < N do
18:                     local array float p[K]
19:                     let v ← w[m][n]
20:                     for all 0 ≤ k < K do
21:                         let θ ← (tpd[r][m][k] + α)/(N[m] + K × α)
22:                         let φ ← (wpt[r][v][k] + β)/(wt[r][k] + V × β)
23:                         p[k] ← θ × φ
24:                     end for
25:                     let z ← sample(p)                ▷ Now 0 ≤ z < K
26:                     increment tpd[1 − r][m][z]       ▷ Increment counters
27:                     increment wpt[1 − r][v][z]
28:                     increment wt[1 − r][z]
29:                 end for
30:             end for
31:         end for
32:     end for
```

Figure 1: Pseudocode for Streaming SCA



(a) Count min sketch evaluation.



(b) Feature hashing evaluation.

Topic A	Topic B	Topic C
band	law	road
album	court	route
released	act	bridge
records	police	highway
rock	case	north
music	states	street
live	united	east
track	legal	west
release	rights	south
guitar	state	state

(c) Example topics.

iteration we have a back-and-forth between two phases, where one reads θ and ϕ in order to update the statistics and the other reads the statistics in order to update θ and ϕ; SCA performs the back-and-forth between two copies of the statistics. Therefore, the number of iterations is halved (line 9), and each iteration has two subiterations (line 10), one that reads $tpd[0]$, $wpt[0]$, and $wt[0]$ in order to write $tpd[1]$, $wpt[1]$, and $wt[1]$, then one that reads $tpd[1]$, $wpt[1]$, and $wt[1]$ in order to write $tpd[0]$, $wpt[0]$, and $wt[0]$.

3 Towards Streaming LDA

In this section we describe how to address the two key challenges for streaming LDA that we mentioned in the introduction. Each subsection addresses one of the two challenges, discusses our proposed solution, and presents associated empirical evaluations.

3.1 Challenge: unknown vocabulary size

In a streaming environment, we cannot pre-compute the vocabulary size V *a priori* and the final size remains unknown throughout the course of inference. Consequently, this makes it difficult to know how large to make the data-structure necessary for the algorithm. For example, the words-per-topic counts (wpt) is typically a $K \times V$ matrix whose size depends directly on V. Other data-structures such as the alias tables for constant-time sampling also depend on the vocabulary size. We could use a non-parametric model or dynamic data-structures to address this problem, but these approaches are computationally expensive [19]. Instead, we employ a count-min sketch.

3.1.1 Count-min sketch

A count-min sketch [4, 13] is a randomized data structure that maintains approximate counts of how many times distinct events have happened in a stream. The count-min sketch works as follows. Assume that we have k hash functions $h_1, \ldots, h_k$, each of which with a range of r bits. The count-min sketch is a $k \times r$ matrix initialized at 0. For each event e in the stream and every hash function k_i, we update the matrix by incrementing the value at index $(i, h_i(e))$.

If we want to query the count-min sketch to know how many times an event e has happened, we compute $h_i(e)$ for every hash function, and then return the minimum of these values. This estimate of the frequency of event e is approximate but this approximation can be made precise by an appropriate choice of the number and the range of the hash functions.

Note that a key property of the count-min sketch is that it is easy to distribute. Indeed, if we split a stream of events e into two substreams e_1 and e_2 and then use two count-min sketches c_1 and c_2 to count the frequency of events for the respective streams, one can estimate the frequency count for the stream e by adding together c_1 and c_2.

3.1.2 Count-min sketch to count words per topics assignments

For each topic, we maintain a CM sketch of word counts. That is, we replace the wpt table with a matrix of dimensions $2 \times X \times K$ where X equals $H \times R_2$. H is the number of hash functions for the CM sketch, R_2 is the range of these hash functions, and the factor of 2 is simply because SCA maintains two data-structures for the sufficient statistics (one for reading, one for writing). A possible concern for such an approach is that distinct words might end up sharing the same counts; however, if T is the total number of words assigned to some topic, the count-min sketch guarantees that the estimate $\hat{v}$ of a true count v for that topic will be such that $v \leq \hat{v} \leq v + T \times \frac{e}{2^{R_2}}$ with probability $1 - e^{-H}$. Because we do not change the hash functions between iterations, the same collisions re-occur in each round. However, as the topic assignments for words change, the effects of these collisions varies across rounds. We show experimentally that H and R_2 can be chosen so that the statistical performance is as good as if we were keeping track of the true counts. Of course, in the streaming setting, we do not know the exact sizes involved when doing this tuning, but as long as the actual sizes do not differ substantially from those used during tuning, we can expect similar performance. In contrast, if we do not use the count-min sketch, *any* underestimation of V will force us to re-allocate the wpt table.

Experimental setup We conduct our experiments by fitting LDA on Wikipedia and evaluating on Reuters (RCV1). Our version of Wikipedia contains 6.7 Million documents and has a vocabulary size of 291,561 word-types (after removing stop-words and rare-words as is customary). Words are represented as integers[1], and the CM hash functions are linear congruential generators with seeds drawn from [11]. We run each algorithm distributed over eight machines to fit LDA on Wikipedia and report perplexity on 10k randomly sampled Reuters documents. For the figures, the x-axis is the

[1] See section 3.2 for how such representations can be computed efficiently in the streaming setting.

number of iterations (one pass over the entire data) and the y-axis is the held-out perplexity. Finally, we manually inspect the topics for quality. Unless stated otherwise, we employ this experimental protocol throughout the remainder of the paper.

Results Figure 2a contains the count-min sketch results. We vary both the number of hash-functions $k \in \{3, 4\}$ and the number of bits $r \in \{16, 17, 18\}$. 18 bits span a hash space of similar size to the vocabulary while 17 bits span a space that is about half the size. However, since there are multiple hashes, the total number of bits is larger than the vocabulary. Therefore, we also employ 16-bit hashes, so the count-min sketch takes less space than the original wpt. Although the initial perplexity of the more compressed sketches (fewer hashes, less bits) are initially worse, they all converge to nearly the same perplexity as the "plain" version of SCA[2] that employs standard 2d-arrays for wpt in lieu of sketching. Upon manual inspection, we see that the topics are similar and notice no qualitative difference when compared to vanilla SCA. Due to space, we omit example topics for these configurations of the inference algorithm, but in the last part of this section we show example topics for the final version of our streaming LDA system which also utilizes feature hashing.

Although the algorithm behaves well statistically, there are some computational concerns. In particular, the count-min sketch stores multiple counts per word (one per-hash function) and a distributed algorithm must then communicate the extra counts over the network. Indeed, the average per-iteration run-time for the count-min sketch ranges from 24.8 seconds (3 hashes, 16 bits) to 46.2 seconds (4 hashes, 18 bits) which is slower than the 27.0 seconds for the version where we assume a fixed vocabulary size.

3.2 Challenge: word encodings

For efficient implementations of the learning algorithm, it is critical to encode the words as integers to avoid having to work with strings. Such strings require more space and precludes employing arrays—where we can use indexing to efficiently look-up a word—for the underlying data-structures. Again, we might try to use dynamic data structures and assign codes to words on the fly but this would be prohibitively expensive. Instead, we propose an alternative solution based on feature hashing.

3.2.1 Feature Hashing

When working with text documents, it is customary to associate an integer to each possible word to transform an array of strings into an array of integers. When it is not possible to pre-compute such an encoding, a possibility is to use a hash function to assign words to integers, a method know as feature hashing [6, 17, 15]. This can be implemented very efficiently and alleviates the need to pre-compute an encoding. However, the hash function can introduce collisions (causing two words to share the same encoding). In many machine learning applications, it has been shown that despite such collisions, learning can be done effectively and fast by hashing features. However, most of these applications are supervised classification in which the predicted output label is important and the interpretation of the individual input features are not typically critical.

In contrast, we care about the representations of the input words themselves and it remains to be seen whether learning a topic model is robust to feature hashing, and also how the algorithm should be implemented if we were to use feature hashing. One obvious issue is that we need to use a large enough hash to avoid too many collisions. However, a larger hash means we need to allocate larger data structures and many of the codes will be unused. This is inefficient and wasteful.

3.2.2 Sketches enable feature-hashing

Since we cannot pre-compute an encoding of words and do not want to use a dynamic data structure for wpt, we can use a hash function to map words as strings into R_1-bits integer. An obvious solution could be to compute a hash as a table at runtime: for every word, look up the table to see if the word has been assigned a code yet, and if not, choose the next available code. If no codes are available, then reuse an existing one. The advantage of such a method is that it guarantees that all the codes are used, which avoid a waste of space for wpt. However, in the context of a parallel/distributed implementation in which different threads/nodes see different data, the consequence is that we need

[2] SCA is a sufficient baseline because previous work cited in this paper has already established it as a strong performer in terms of perplexity.

to synchronize the table creation. This can become quite complicated since two nodes could assign to the same code two different words (or different codes to the same word), which would need to be resolved. One could imagine allowing collisions but that would potentially lead to bad collisions between highly frequent words.

Employing a count-min sketch to represent wpt enables interesting possibilities. First, assuming that we use a count-min sketch with l hash functions of R_2 bits, we could choose the hash to be the concatenation of the l R_2-bits codes. That is, we use the count-min sketch directly for feature hashing. This is an effective possibility, but it has the disadvantage of using potentially too many bits to represent a word since it uses $l \times R_2$ bits.

Another possibility is to use a classic hash from string to R_1-bits in which we choose R_1 freely. Such an approach would usually not be viable since it would require to set $V = 2^{R_1}$ for the number of columns in wpt which would be unreasonably large and would likely make poor use of the memory space since many indices would end up not being used. However, since we are using a count-min sketch to represent wpt, we do not have to worry about such issues. Indeed the count-min sketch has its own set of hash functions that map R_1-bit representations into R_2-bit representations. Assuming that the true size of the vocabulary is V, then the probability of a collision is $1 - e^{-\frac{V^2}{2^{R_1+1}}}$. We also show experimentally that learning is largely unaffected by the feature hashing with the right configuration. We also note that going through R_1 bits, where R_1 is less than $l \times R_2$ does not seem to be problematic. Note that if we also use the alias method to sample from a discrete distribution in constant time, then we need to keep a hash table from all the codes that we actually use to their corresponding alias table.

Result We present the results in Figure 2b employing the same experimental design as Section 3.1.2. We vary the number of bits in the feature-hash $R_1 \in \{21, 22, 23\}$, while fixing the parameters of the count-min sketch to $k = 3, R_2 = 16$. We employ both a "plain" version of SCA that employs conventional data structures, and a count-min sketch version as baselines. Varying the size of the feature-hash affects the initial perplexities, but again, they all converge to similar final values. We begin noticing some difference at $R_2 = 21$ and may start seeing more significant differences at lower values. Upon inspection, we find that the topics are of similar quality to the baseline models and we present a few topics in Figure 2c.

Implementation Modifying SCA to support feature hashing and the count-min sketch is fairly simple. The read of wpt on line 22 and the write of wpt on line 27 are replaced by the read and write procedures of the count-min sketch, respectively. Note that the input data w on line 1 is not of type int anymore but rather of type string. Consequently, the data needs to be hashed before the main iteration starts on line 9.

4 Conclusions

In this paper we made novel use of count-min sketch and feature-hashing data-structures to make progress toward streaming LDA by eliminating pre-computation steps for calculating the vocabulary size and the word encodings. We found that despite compressing the sufficient statistics with these data-structures, we could still learn good quality topic models.

References

[1] Hesam Amoualian, Marianne Clausel, Eric Gaussier, and Massih-Reza Amini. Streaming-lda: A copula-based approach to modeling topic dependencies in document streams. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pages 695–704, New York, NY, USA, 2016. ACM.

[2] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. Latent dirichlet allocation. *J. Mach. Learn. Res.*, 3:993–1022, March 2003.

[3] Jianfei Chen, Kaiwei Li, Jun Zhu, and Wenguang Chen. Warplda: A cache efficient o(1) algorithm for latent dirichlet allocation. *Proc. VLDB Endow.*, 9(10):744–755, June 2016.

[4] Graham Cormode and S. Muthukrishnan. An improved data stream summary: The count-min sketch and its applications. *J. Algorithms*, 55(1):58–75, April 2005.

[5] Philippe Flajolet. Approximate counting: A detailed analysis. *BIT*, 25(1):113–134, June 1985.

[6] George Forman and Evan Kirshenbaum. Extremely fast text feature extraction for classification and indexing. In *Proceedings of the 17th ACM Conference on Information and Knowledge Management*, CIKM '08, pages 1221–1230, New York, NY, USA, 2008. ACM.

[7] Yang Gao, Jianfei Chen, and Jun Zhu. Streaming gibbs sampling for LDA model. *CoRR*, abs/1601.01142, 2016.

[8] Thomas L. Griffiths and Mark Steyvers. Finding scientific topics. *PNAS*, 100(1):5228–5235, April 2004.

[9] Matthew Hoffman, Francis R. Bach, and David M. Blei. Online learning for latent dirichlet allocation. In J. D. Lafferty, C. K. I. Williams, J. Shawe-Taylor, R. S. Zemel, and A. Culotta, editors, *Advances in Neural Information Processing Systems 23*, pages 856–864. Curran Associates, Inc., 2010.

[10] Matthew D. Hoffman, David M. Blei, Chong Wang, and John Paisley. Stochastic variational inference. *Journal of Machine Learning Research*, 14:1303–1347, 2013.

[11] Pierre L'Ecuyer. Tables of linear congruential generators of different sizes and good lattice structure. *Math. Comput.*, 68(225):249–260, 1999.

[12] Kaiwei Li, Jianfei Chen, Wenguang Chen, and Jun Zhu. Saberlda: Sparsity-aware learning of topic models on gpus. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 497–509, New York, NY, USA, 2017. ACM.

[13] Sergiy Matusevych, Alex Smola, and Amr Ahmed. Hokusai | sketching streams in real time. In *Proceedings of the 28th International Conference on Conference on Uncertainty in Artificial Intelligence (UAI)*, 2012.

[14] Robert Morris. Counting large numbers of events in small registers. *Commun. ACM*, 21(10):840–842, October 1978.

[15] Qinfeng Shi, James Petterson, Gideon Dror, John Langford, Alex Smola, Alex Strehl, and Vishy Vishwanathan. Hash kernels. In David van Dyk and Max Welling, editors, *Proceedings of the Twelth International Conference on Artificial Intelligence and Statistics*, volume 5 of *Proceedings of Machine Learning Research*, pages 496–503, Hilton Clearwater Beach Resort, Clearwater Beach, Florida USA, 16–18 Apr 2009. PMLR.

[16] Guy L. Steele, Jr. and Jean-Baptiste Tristan. Adding approximate counters. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '16, pages 15:1–15:12, New York, NY, USA, 2016. ACM.

[17] Kilian Weinberger, Anirban Dasgupta, John Langford, Alex Smola, and Josh Attenberg. Feature hashing for large scale multitask learning. In *Proceedings of the 26th Annual International Conference on Machine Learning*, ICML '09, pages 1113–1120, New York, NY, USA, 2009. ACM.

[18] Manzil Zaheer, Michael Wick, Jean-Baptiste Tristan, Alex Smola, and Guy Steele. Exponential stochastic cellular automata for massively parallel inference. In Arthur Gretton and Christian C. Robert, editors, *Proceedings of the 19th International Conference on Artificial Intelligence and Statistics*, volume 51 of *Proceedings of Machine Learning Research*, pages 966–975, Cadiz, Spain, 09–11 May 2016. PMLR.

[19] Ke Zhai and Jordan L. Boyd-graber. Online latent dirichlet allocation with infinite vocabulary. In Sanjoy Dasgupta and David Mcallester, editors, *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, volume 28, pages 561–569. JMLR Workshop and Conference Proceedings, 2013.

[20] Bo Zhao, Hucheng Zhou, Guoqiang Li, and Yihua Huang. Zenlda: An efficient and scalable topic model training system on distributed data-parallel platform. *CoRR*, abs/1511.00440, 2015.