Fast Sampling for LDA: Lecture Notes

William W. Cohen

April 3, 2018

1 A naive sampler for LDA

A collapsed Gibbs sampler for LDA spends almost all its time sampling a new topic t for a random variable $Z_{i,d}$, where $Z_{i,d}$ is associated with some position i inside a document d. Let w be the word at position i.

The sampling distribution for $Z_{i,d}$ is influenced by two things. One is the probability (based on other assignments in d) that t appears in d, which might naively and crudely be estimated as just $\Pr(t|d) \propto n_{t|d}$, where $n_{t|d}$ is the number of times topic t has been assigned in d (with the current $z_{i,d}$'s). Less naively, let $z_{i,d}$ be the current topic assignment for that position, and let $\alpha_1, \ldots, \alpha_T$ be the parameters of the Dirichlet we smooth topics with. Let

$$c_{i,d,t} = \begin{cases} 1 & \text{if } z_{i,d} = t \\ 0 & \text{otherwise} \end{cases}$$

A smoothed equation for the influence of other topic assignments in d would be

$$\Pr(Z_{i,d} = t|d) \propto \alpha_t + n_{t|d} - c_{i,d,t} \tag{1}$$

The second influence is the probability that word w is generated by topic t. Let $n_{w|t}$ be the number of times w is assigned to t (in all the $z_{i,d}$'s for the corpus) and let $n_{\cdot|t}$ be the number of times any word is assigned to t. The second part of the sampling equation, after smoothing with parameter β on a vocabulary size V,

$$\Pr(Z_{i,d} = t|w) = \frac{\beta + n_{w|t} - c_{i,d,t}}{\beta V + n_{\cdot|t} - c_{i,d,t}} \tag{2}$$

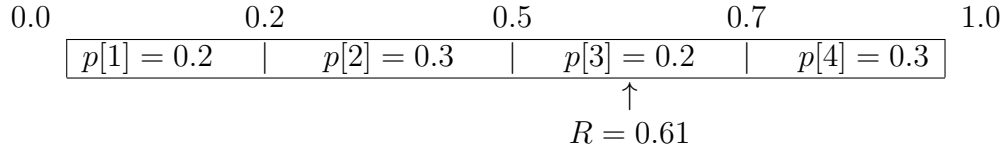


Figure 1: How the naive sampler works. After setting up the array $p[]$ where $p[t] = \Pr(t)$, you draw a random number $r[0, 1]$ and see where R lies in the array. This takes time linear in the number of topics T.

Putting these together we should sample $Z_{i,d}$ by

$$\Pr(Z_{i,d} = t|d, w) \propto (\alpha_t + n_{t|d} - c_{i,d,t}) \cdot \frac{\beta + n_{w|t} - c_{i,d,t}}{\beta V + n_{\cdot|t} - c_{i,d,t}} \tag{3}$$

Starting now I will simplify this equation a little, by dropping the correction term $c_{i,d,t}$ (which in practice is needed to make adjust the counts to only include positions other than i in d) and some other indices that are clear from context, so the simplified distribution I want to sample from is just

$$\Pr(Z = t) \propto (\alpha_t + n_{t|d}) \cdot \frac{\beta + n_{w|t}}{\beta V + n_{\cdot|t}} \tag{4}$$

Code to sample from this is straightforward. We keep the counts n up-to-date by modifying them whenever I flip a Z from one topic to another, and when we need to sample, we run this code, which uses an array p of length T.

Naive Sampler for LDA

1. For $t = 1, \ldots, T$, let $p[t] = (\alpha_t + n_{t|d}) \cdot \frac{\beta + n_{w|t}}{\beta V + n_{\cdot|t}}$
2. Next normalize p so that it is a probability distribution: i.e., compute a normalizer $s = \sum_t p[t]$, and for $t = 1, \ldots, T$, let $p[t] = p[t]/s$. Now p contains the probability distribution you want to sample from.
3. Let $s_{tmp} = 0$, and sample a random number R uniformly from $[0, 1]$.
4. For $t = 1, \ldots, T$
   (a) In each iteration of this loop s_{tmp} is $\sum_{t' \le t} p[t']$. If $s_{tmp} < R \le s_{tmp} + p[t]$ then return t as the sampled topic.
   (b) Otherwise, update $s_{tmp} = s_{tmp} + p[t]$.

2 Discussion

2.1 Complexity analysis

For the naive sampler, the first stage (computing and normalizing the $p[t]$'s) can be visualized as setting up strip of length 1 which is divided into T sections of size $p[1], \ldots, p[T]$. The second stage (picking r and finding out what topic t it corresponds to) can be viewed as "throwing a dart" into the unit interval and scanning through the sections to see where it lies. You can make this a little more efficient, but the key thing to note is that takes time $O(T)$, the number of topics. The time is spent doing two things: computing the $p[t]$'s, and scanning through them to see where r falls. If there are a total of N word positions in the corpus, then, an epoch of sampling takes time $O(NT)$.

What's the space complexity—in particular, the space we need to keep in memory? We have T values for $n_{\cdot|t}$. If we have D documents we have TD values of $n_{t|d}$, but since we sample repeatedly in one document, we only need to really keep the ones for the current document in memory: we can store the counts for most of the corpus on disk, and stream through the corpus with each sampling pass, just loading enough information to process one document at a time. So we can get away with using memory for only T of these counts. We also need to keep track of the $z_{i,d}$ values, but by the same argument, we can store these on disk, and only load the $z_{i,d}$'s for the current d in memory. If the maximum document length is fixed to a constant k this is just $O(1)$.

The biggest cost is storing $n_{w|t}$. This needs space VT, so it's like storing T word-based linear classifiers.

This all sounds very efficient—everything is linear in T, N, and V! The problem is that in practice, as corpora grow large, you tend to need more and more topics to model them, so the linear dependency on T for time and space turn out to be a big efficiency problem. So we need to see how this can be made efficient for large numbers of topics.

2.2 Improving the complexity

Can we make this sampling step run faster than $O(T)$?

There is certainly room for improvement in the code. For instance, we can compute $s = \sum_t p[t]$ in the same loop when I compute the $p[t]$'s, and

rather than explicitly normalizing p, you could draw R uniforly from $[0, s]$. That would be faster, but it's still $O(T)$.

Alternatively, we could not bother storing the $p[t]$'s, since what we really use to "look up" t from R are the cumulative sums, $\sum_{t' \leq t} p[t']$. We can also avoid a scan with a binary search, since the cumulative sums will be sorted. Thus even more efficient implementation would be this.

Semi-naive Sampler for LDA

1. Let s be an array of size T
2. For $t = 1, \ldots, T$:
   (a) Let $p_{tmp} = (\alpha_t + n_{t|d}) \cdot \frac{\beta + n_{w|t}}{\beta V + n_{\cdot|t}}$
   (b) Let $s[t] = s[t-1] + p_{tmp}$. (Here we assume $s[0] = 0$).
3. Draw R uniformly from $[0, s[T]]$
4. Find $t : s[t] < R < s[t+1]$ using binary search on s. (Which is sorted).

However, even after switching to a binary search, this is still $O(T)$—we're still spending time $O(T)$ to set up the "strip" that we sample from, even if we've made looking up R inside that strip $O(\log_2 T)$. To set a bigger speedup, we need to somehow incrementally update the "strip" we sample from, and avoid re-computing all the p_t's each time.

3 The SparseLDA sampler

In the naive and semi-naive algorithms, we set up a length-1 strip which is divided into T sections. This is natural, but not the only way to divide things up. For instance, we can take Eq 4 and rewrite it as a three-part sum:

$$\Pr(Z = t) \propto \frac{\alpha_t \beta}{\beta V + n_{\cdot|t}} + \frac{n_{t|d} \beta}{\beta V + n_{\cdot|t}} + \frac{(\alpha_t + n_{t|d}) n_{w|t}}{\beta V + n_{\cdot|t}} \tag{5}$$

Or, to give these parts names, we let $\Pr(Z = t) \propto s_t + r_t + q_t$, where

$$s_t = \frac{\alpha_t \beta}{\beta V + n_{\cdot|t}}$$

$$
\begin{aligned}
r_t &= \frac{n_{t|d}\beta}{\beta V + n_{\cdot|t}} \\
q_t &= \frac{(\alpha_t + n_{t|d})n_{w|t}}{\beta V + n_{\cdot|t}}
\end{aligned}
$$

We could now split each of the T sections into three parts, one for each addend, giving us a strip with $3T$ sections. So sampling from the strip in Figure 1 might now look like this

s_1	r_1	q_1	s_2	r_2	q_2	s_3	r_3	q_3	s_4	r_4	q_4
			↑								

Again, we would need to scan through to see where R falls, but we'd want to return topic t if R falls into any of the sections labeled s_t, r_t, or q_t—in the picture above, for instance, we would return $t = 2$.

This by itself isn't helpful—but there's also no reason these sections need to be ordered by topic. We could just as well lay out the strip with all the s's first, ordered by topic, then the r's, and then the q's:

s_1	s_2	s_3	s_4	r_1	r_2	r_3	r_4	q_1	q_2	q_3	q_4

"Throwing a dart" at this strip and finding out where it lies can be done as follows. First, compute

$$
\begin{aligned}
s &= \sum_t \frac{\alpha_t \beta}{\beta V + n_{\cdot|t}} \\
r &= \sum_t \frac{n_{t|d}\beta}{\beta V + n_{\cdot|t}} \\
q &= \sum_t \frac{(\alpha_t + n_{t|d})n_{w|t}}{\beta V + n_{\cdot|t}}
\end{aligned}
$$

These markers divide the strip above into three main sections, which separate the s's, the r's, and the q's. Now draw R uniformly from $[0, s+r+q]$. One of three things will happen. (1) If $R < s$, then we will have hit one the sections labeled s_t. (2) If $s \leq Rq$, we hit one of the r_t sections. (3) otherwise, we hit one of the q_t sections. For typical smoothing parameters we hit one of the q_t's about 90% of the time.

In every one of these cases, we still need to look up the topic t within the section, but in some cases we can be a little more clever about how we search for the topic t we hit, and/or how we maintain the data structures we use to sample from. Let's first see how to do the sampling steps.

Case 1. If $R < s$ (we've hit one of the "smoothing" buckets) we recompute the s_i's and do linear search to find t, as in the naive algorithm. Not pretty, but this case doesn't happen too often.

Case 2. If $s < R \leq r$ we've hit one of the "document count" buckets. Here we can speed things up *if the length of the document k_d is less than the number of topics T*, because we only have to bother with topics for which $n_{t|d} > 0$: if $n_{t|d} = 0$ we don't even need to compute r_t. Notice that for a large corpora (where we need a large T) containing many short documents, it will usually be the case that $k_d << T$.

Case 3. If $R > r$ we've hit one of the "topic word count" buckets, and we can speed things up by skipping topics for which $n_{w|t} = 0$. Fortunately, most words in a corpus are rare, so there will be many words that don't even occur T times: so again, usually there are *lots of topics with $n_{w|t} = 0$ which will not need to be checked.* Experimentally, a few epochs, even fairly frequent words tend to be placed in only a few different topics.

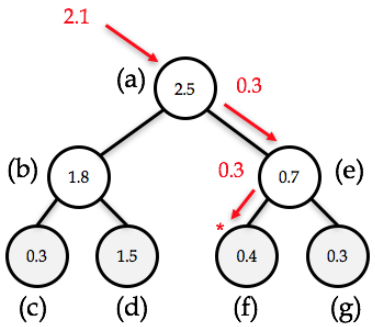
Now lets consider the bookkeeping we need to maintain the data structures needed for sampling. For case 1, it's easy to keep s and the s_t's up-to-date, since they don't change when we resample. For case 2, the $n_{t|d}$'s can be loaded once when you start sampling document d, and the s_t's computed at that point. They can then be updated as you sample, along with q.

The trickier thing to maintain is q and the q_t's, which depend on the expensive-to-store $n_{w|t}$ counts. In SparseLDA [1] the $n_{w|t}$ counts are kept in an unusual data structure. Every word w is associated with a list of pairs $(n_{w|t}, t)$. The length of this list is no more than the frequency of w in the corpus, and also no more than T. For example, $T =$ and the word "aardvark" appeared three times in the corpus, and had been assigned once to topic 2 and twice to topic 3, the list for "aardvark" might be

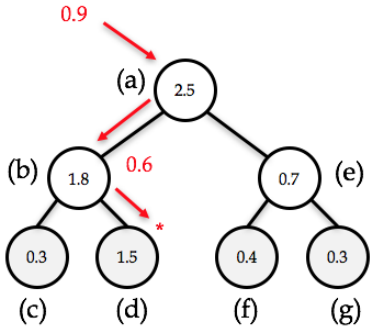
$$(2, 3), (1, 2), (0, 4)$$

This is the list that will be scanned through when looking for t's that are associated some q_t'. The list is also kept in descending order by $n_{w|t}$ (as shown here) which makes it easy to find not just the topics with non-zero counts, but also the topics with *large* counts. After a few epochs, the topic counts are often skewed, so if you start scanning from the front of the list, you will often reach a value larger than R quickly.

When a count changes you need to scan the list to make an update, and the list is then bubble-sorted to restore the order. The bubble-sort is quite



v	*R*	*condition*	*action*
a	2.1	R > a.left.val	R = R - a.left.val; v = v.right
e	0.3	R < a.left.val	v = v.left
f	0.3	v.isLeaf	return v.topic



v	*R*	*condition*	*action*
a	0.9	R<a.left.val	v = v.left
b	0.9	R>a.left.val	R = R - a.left.val; v = v.right
f	0.6	v.isLeaf	return v.topic

Figure 2: An F+ Tree sampler, with traces for two samples: sampling $R = 2.1$ samples node (f) and sampling $R = 0.9$ brings you to node (d).

efficient when only a few counts have changed. Finally, the pairs are actually stored as integers, where the count is in the high-order bits and the topic t in the low-order bits, which also makes the sort efficient (and keeps the space small).

The main advantage that SparseLDA gives you, is, as the name suggests, sparsity. Most of the time you only need to check some of the many topics T. You also need less storage: since the size of each $n_{w|t}$ list is bounded by both T and the frequency of w, the total size of all these lists is bounded by N. So space has been decreased from $O(VT)$ to $O(\min(N, VT))$.

4 Tree-based samplers

The data structures SparseLDA uses are efficient but ad hoc. We will describe here a new data structure, which can be thought of as an improved version of the Semi-naive Sampler. The Semi-naive sampler computed an array of cumulative scores—i.e., $s[t] = \sum_{t'<t} p[t']$–and then picked $R\tilde{[}0, s[T]]$, and used binary search to find a t such that $s[t-1] < R < s[t]$. The improvement will be a way to incrementally maintain the cumulative scores using a binary tree.

The tree structure we use is called an $F+$ *tree* [2]. It is a binary tree with leaves labeled with pairs (t, q_t), where t is a topic index, and q_t is a weight associated with t. The topic indices are all in order, left to right. Our goal is to sample a leaf node in time $O(\log_2 T)$, where

$$\Pr(v \text{ associated with } t \text{ is sampled}) \propto q_t$$

The trick, as shown in Figure 2, is store at each internal node v a value $v.val$ which is the sum of the q_t's of all the leaves below it. We can do this by compute the values bottom-up, leaves-to-root, and always setting

$$\texttt{v.val} = \texttt{v.left.val} + \texttt{v.right.val}$$

To sample, we look at the root of the tree—let's call that v_0—and draw a number R uniformly from $[0, v_0.\texttt{val}]$. We then call the recursive function below, `TreeSample(`v_0, R`)`.

`TreeSample(`v, R`)`.

1. If v is a leaf node, return `v.topic`.
2. If v is an internal node and $R \leq$ `v.left.val`, then
   - return `TreeSample(v.left,`R`)`.
3. If v is an internal node and $R >$ `v.left.val`, then
   - return `TreeSample(v.right,`$R -$ `v.left.val)`.

The nice thing about an $F+$ tree is that it is also possible to update the sampler in $O(\log_2 T)$ time after one of q_t changes: when you make a change,

you only need to update the v.`val` fields for the nodes between the leaf node for t and the root.

A detailed description of a specific, very efficient heap-like implementation of an $F+$ tree can be found in [2], where the tree is stored in a array of size $2T$.

5 LDA with a Tree Sampler

SparseLDA [1] formulates topic sampling as a heirarchical sampling process, where you first decide whether use one of the the "smoothing", "document count", or "topic word count" buckets, and then sample within the appropriate group of buckets. It obtains sparsity speedups for the topic word buckets (especially for rare words) and speedups for document count buckets (especially for short documents).

Another heirarchical sampling scheme is outlined in [2]. The LDA sampling formula is re-written as

$$\begin{aligned}\Pr(Z=t) &\propto (\alpha_t + n_{t|d}) \cdot \frac{\beta + n_{w|t}}{\beta V + n_{\cdot|t}} \\ &= \beta(\frac{\alpha_t + n_{t|d}}{\beta V + n_{\cdot|t}}) + n_{w|t}(\frac{n_{t|d} + \alpha_t}{\beta V + n_{\cdot|t}})\end{aligned}$$

You then sample hierarchically as follows. First let

$$\begin{aligned}a_t &= \beta(\frac{\alpha_t + n_{t|d}}{\beta V + n_{\cdot|t}}) \\ b_t &= n_{w|t}(\frac{n_{t|d} + \alpha_t}{\beta V + n_{\cdot|t}}) \\ a &= \sum_t q_t \\ b &= \sum_t r_t\end{aligned}$$

Then draw $R\tilde{[}0, a+b]$ and (1) if $R < a$, sample from the "smoothing and document count buckets" defined by the a_t's and (2) otherwise, sample from the "word count buckets" defined by the b_t's. The a_t's are now dense, but when you perform a sampling step, only two values will change, so [2] suggest using an $F+$ tree sampler. For the word count buckets, a semi-naive binary-search is used instead.

References

[1] David Mimno, Matthew D Hoffman, and David M Blei. Sparse stochastic inference for latent dirichlet allocation. In *Proceedings of the 29th International Coference on International Conference on Machine Learning*, pages 1515–1522. Omnipress, 2012.

[2] Hsiang-Fu Yu, Cho-Jui Hsieh, Hyokun Yun, SVN Vishwanathan, and Inderjit S Dhillon. A scalable asynchronous distributed algorithm for topic modeling. In *Proceedings of the 24th International Conference on World Wide Web*, pages 1340–1350. International World Wide Web Conferences Steering Committee, 2015.