Scaling up LDA

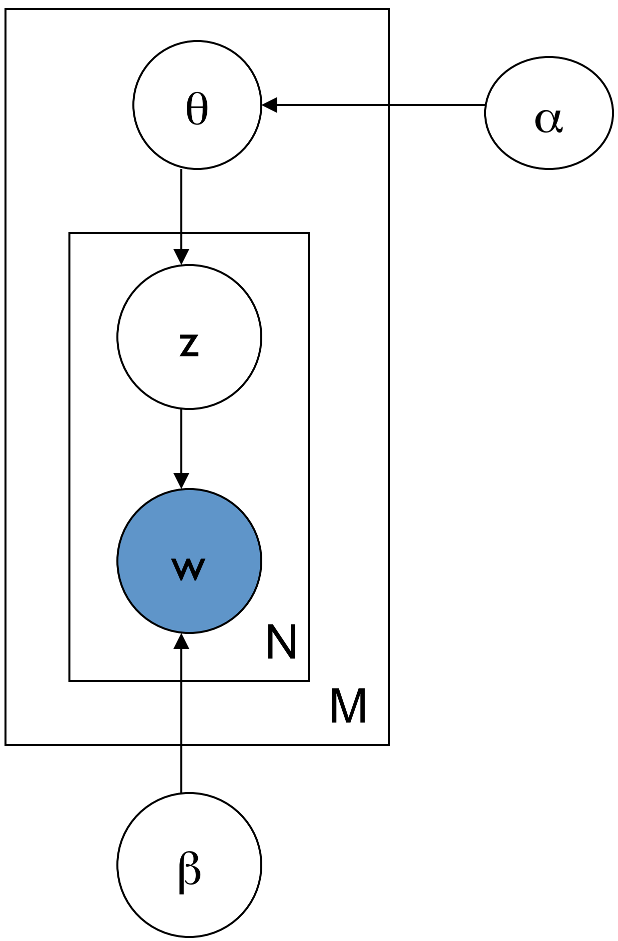
William Cohen

Outline

- LDA/Gibbs algorithm details
- How to speed it up by parallelizing
- How to speed it up by faster sampling
  - Why sampling is key
  - Some sampling ideas for LDA

Review - LDA

- Latent Dirichlet Allocation with Gibbs



- Randomly initialize each $z_{m,n}$
- Repeat for t=1,….
  - For each doc *m,* word *n*
    - Find Pr(z_{mn}=*k*|other z's)
    - Sample z_{mn} according to that distr.

Way way more detail

```
# topic k, docId d, and wordId w are integer indices
#
# x[d][j] = w, index of j-th word in doc d
# z[d][j] = k, index of latent topic of j-th word in doc d
# vocab[w] = string for the word with index w
#
# totalTopicCount[k] = number of words in topic k
# docTopicCount[d][k] = number of words in topic k for document d
# wordTopicCount[w][k] = number of occurrences of word w in topic k
# totalWords = number of words in the corpus
```

```
    # topic k, docId d, and wordId w are integer indices
    #
    # x[d][j] = w, index of j-th word in doc d
    # z[d][j] = k, index of latent topic of j-th word in doc d
    # vocab[w] = string for the word with index w
    #
    # totalTopicCount[k] = number of words in topic k
    # docTopicCount[d][k] = number of words in topic k for document d
    # wordTopicCount[w][k] = number of occurrences of word w in topic k
    # totalWords = number of words in the corpus


def initGibbs(self):
    print '.initializing latent vars'
    self.totalTopicCount = self.topicCounter()
    self.docTopicCount = [self.topicCounter() for d in xrange(len(self.x))]
    self.wordTopicCount = [self.topicCounter() for w in xrange(len(self.vocab))]
    self.z = [[-1 for j in xrange(len(self.x[d]))] for d in xrange(len(self.x))]
    for d in xrange(len(self.x)):
        if (d+1)%self.dstep==0: print '..doc',d+1,'of',len(self.x)
        for j in xrange(len(self.x[d])):
            w = self.x[d][j]
            k = random.randint(0,self.numTopics-1)
            self.z[d][j] = k
            self.docTopicCount[d].add(k, 1)
            self.wordTopicCount[w].add(k, 1)
            self.totalTopicCount.add(k, 1)
    #reasonable parameters
    self.alpha = 1.0/self.numTopics
    self.beta = 1.0/len(self.vocab)
    print "alpha:",self.alpha,"beta:",self.beta
```

```python
def runGibbs(self,maxT):
    for t in xrange(maxT):
        print '.iteration',t+1,'of',maxT
        for d in xrange(len(self.x)):
            if (d+1)%self.dstep==0: print '..doc',d+1,'of',len(self.x)
            for j in xrange(len(self.x[d])):
                k = self.resample(d,j)
                self.flip(d, j, self.z[d][j], k)

def flip(self, d, j, k_old, k_new):
    """update counts to reflect a changed value of z[d][j]"""
    if k_old != k_new:
        w = self.x[d][j]
        self.docTopicCount[d].add(k_old, -1)
        self.docTopicCount[d].add(k_new, +1)
        self.wordTopicCount[w].add(k_old, -1)
        self.wordTopicCount[w].add(k_new, +1)
        self.totalTopicCount.add(k_old, -1)
        self.totalTopicCount.add(k_new, +1)
        self.z[d][j] = k_new
```

```python
def runGibbs(self,maxT):
    for t in xrange(maxT):
        print '.iteration',t+1,'of',maxT
        for d in xrange(len(self.x)):
            if (d+1)%self.dstep==0: print '..doc',d+1,'of',len(self.x)
            for j in xrange(len(self.x[d])):
                k = self.resample(d,j)
                self.flip(d, j, self.z[d][j], k)


def resample(self,d,j):
    """sample a new value of z[d][j]"""
    p = []
    norm = 0.0
    #compute pk = Pr(z_dj=k | everything else)
    for k in xrange(self.numTopics):
        w = self.x[d][j]
        z_dj_equals_k = 1 if self.z[d][j]==k else 0
        #unnormalized chance of picking topic k in doc d
        ak = (self.docTopicCount[d][k] - z_dj_equals_k + self.alpha)
        #unnormalized chance of picking topic k for word w
        bk = ((self.wordTopicCount[w][k] - z_dj_equals_k + self.beta)
              /(self.totalTopicCount[k] - z_dj_equals_k + self.totalWords * self.beta))
        pk = ak*bk
        p.append(pk)
        norm += pk
    #pick randomly from the normalized pk
    sum_p_up_to_k = 0.0
    r = random.random()
    for k in xrange(self.numTopics):
        sum_p_up_to_k += p[k]/norm
        if r < sum_p_up_to_k:
            return k
```

What gets learned…..

```
def phi(self,w,k):
    """weight of word w under topic k"""
    num = (self.wordTopicCount[w][k] + self.beta)
    denom = (self.totalTopicCount[k] + self.totalWords * self.beta)
    return num/denom

def theta(self,d,k):
    """weight of doc unde
    num = (self.docTopicC
    denom = (sum(self.doc
    return num/denom
```

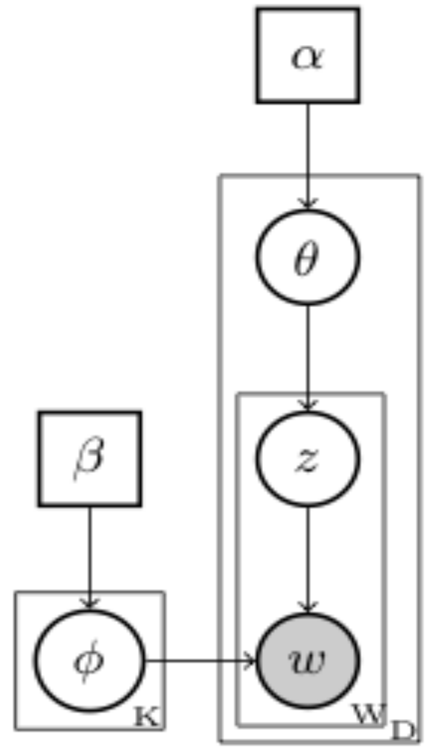


Figure 1: Graphical model for LDA.

In A Math-ier Notation

```
# topic k, docId d, and wordId w are integer indices
#
# x[d][j] = w, index of j-th word in doc d
# z[d][j] = k, index of latent topic of j-th word in doc d
# vocab[w] = string for the word with index w
#
# totalTopicCount[k] = number of words in topic k
# docTopicCount[d][k] = number of words in topic k for document d
# wordTopicCount[w][k] = number of occurrences of word w in topic k
# totalWords = number of words in the corpus
```

N[,k]*

N[d,k]

M[w,k]

N[,*]=V*

for each document *d* and word position *j* in *d*

- *z[d,j] = k*, a random topic
- *N[d,k]++*
- *W[w,k]++* where *w* = id of *j*-th word in *d*

```
def initGibbs(self):
    print '.initializing latent vars'
    self.totalTopicCount = self.topicCounter()
    self.docTopicCount = [self.topicCounter() for d in xrange(len(self.x))]
    self.wordTopicCount = [self.topicCounter() for w in xrange(len(self.vocab))]
    self.z = [[-1 for j in xrange(len(self.x[d]))] for d in xrange(len(self.x))]
    for d in xrange(len(self.x)):
        if (d+1)%self.dstep==0: print '..doc',d+1,'of',len(self.x)
        for j in xrange(len(self.x[d])):
            w = self.x[d][j]
            k = random.randint(0,self.numTopics-1)
            self.z[d][j] = k
            self.docTopicCount[d].add(k, 1)
            self.wordTopicCount[w].add(k, 1)
            self.totalTopicCount.add(k, 1)
    #reasonable parameters
    self.alpha = 1.0/self.numTopics
    self.beta = 1.0/len(self.vocab)
    print "alpha:",self.alpha,"beta:",self.beta
```

```python
def runGibbs(self,maxT):
    for t in xrange(maxT):
        print '.iteration',t+1,'of',maxT
        for d in xrange(len(self.x)):
            if (d+1)%self.dstep==0: print '..doc',d+1,'of',len(self.x)
            for j in xrange(len(self.x[d])):
                k = self.resample(d,j)
                self.flip(d, j, self.z[d][j], k)
```

for each pass t=1,2,….

for each document *d* and word position *j* in *d*

- *z[d,j]* = *k*, a new random topic
- update *N*, *W* to reflect the new assignment of z:
  - *N[d,k]++; N[d,k']* - - where *k'* is old *z[d,j]*
  - *W[w,k]++;W[w,k']* - - where *w* is *w[d,j]*

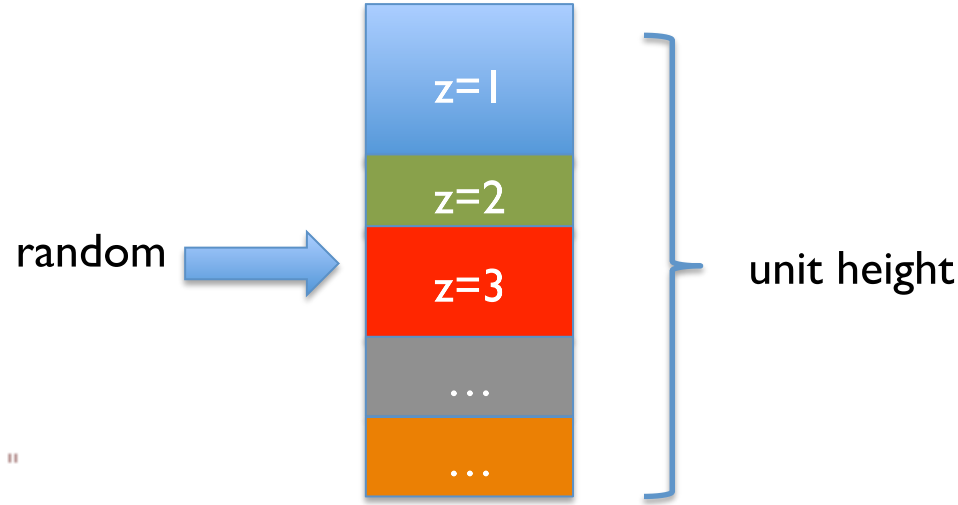
```python
def flip(self, d, j, k_old, k_new):
    """update counts to reflect a changed value of z[d][j]"""
    if k_old != k_new:
        w = self.x[d][j]
        self.docTopicCount[d].add(k_old, -1)
        self.wordTopicCount[w].add(k_old, -1)
        self.wordTopicCount[w].add(k_new, +1)
        self.totalTopicCount.add(k_old, -1)
        self.totalTopicCount.add(k_new, +1)
        self.z[d][j] = k_new
```

$$p(Z_{d,j} = k \mid ..) \propto \Pr(Z_{d,j} = k \mid "d") * \Pr(W_{d,k} = w \mid Z_{d,j} = k, \ldots)$$

$$= \frac{N[k,d] - C_{d,j,k} + \alpha}{Z} \cdot \frac{W[w,k] - C_{d,j,k} + \beta}{(W[*,k] - C_{d,j,k}) + \beta N[*,*]}$$

$$C_{d,j,k} = \begin{cases} 1 & Z_{d,j} = k \\ 0 & \text{else} \end{cases}$$

```
def resample(self,d,j):
    """sample a new value of z[d][j]"""
    p = []
    norm = 0.0
    #compute pk = Pr(z_dj=k | everything else)
    for k in xrange(self.numTopics):
        w = self.x[d][j]
        z_dj_equals_k = 1 if self.z[d][j]==k else 0
        #unnormalized chance of picking topic k in doc d
        ak = (self.docTopicCount[d][k] - z_dj_equals_k + self.alpha)
        #unnormalized chance of picking topic k for word w
        bk = ((self.wordTopicCount[w][k] - z_dj_equals_k + self.beta)
              /(self.totalTopicCount[k] - z_dj_equals_k + self.totalWords * self.beta))
        pk = ak*bk
        p.append(pk)
        norm += pk
    #pick randomly from the normalized pk
    sum_p_up_to_k = 0.0
    r = random.random()
    for k in xrange(self.numTopics):
        sum_p_up_to_k += p[k]/norm
        if r < sum_p_up_to_k:
            return k
```




```
def resample(self,d,j):
    """sample a new value of z[d][j]"""
    p = []
    norm = 0.0
    #compute pk = Pr(z_dj=k | everything else)
    for k in xrange(self.numTopics):
        w = self.x[d][j]
        z_dj_equals_k = 1 if self.z[d][j]==k else 0
        #unnormalized chance of picking topic k in doc d
        ak = (self.docTopicCount[d][k] - z_dj_equals_k + self.alpha)
        #unnormalized chance of picking topic k for word w
        bk = ((self.wordTopicCount[w][k] - z_dj_equals_k + self.beta)
             /(self.totalTopicCount[k] - z_dj_equals_k + self.totalWords * self.beta))
        pk = ak*bk
        p.append(pk)
        norm += pk
    #pick randomly from the normalized pk
    sum_p_up_to_k = 0.0
    r = random.random()
    for k in xrange(self.numTopics):
        sum_p_up_to_k += p[k]/norm
        if r < sum_p_up_to_k:
            return k
```

1. You spend a *lot* of time sampling
2. There's a loop over all topics here in the sampler

Distributed Algorithms for Topic Models

David Newman NEWMAN@UCI.EDU
Arthur Asuncion ASUNCION@ICS.UCI.EDU
Padhraic Smyth SMYTH@ICS.UCI.EDU
Max Welling WELLING@ICS.UCI.EDU
Department of Computer Science
University of California, Irvine
Irvine, CA 92697, USA

JMLR 2009

Observation

- How much does the choice of *z* depend on the other *z's* in the same document?
  - quite a lot
- How much does the choice of *z* depend on the other *z's* in elsewhere in the corpus?
  - maybe not so much
  - depends on Pr(w|t) but that changes slowly
- Can we parallelize Gibbs and still get good results?

Question

- Can we parallelize Gibbs sampling?
  - formally, no: every choice of z depends on all the other z's
  - Gibbs needs to be sequential
    - just like SGD

What if you try and parallelize?

Split document/term matrix randomly and distribute to p processors .. then run "Approximate Distributed LDA"

Algorithm 1 AD-LDA

repeat
  for each processor p in parallel **do**
    Copy global counts: $N_{wkp} \leftarrow N_{wk}$
    Sample $\mathbf{z}_p$ locally: LDA-Gibbs-Iteration($\mathbf{x}_p, \mathbf{z}_p, N_{kjp}, N_{wkp}, \alpha, \beta$)
  end for
  Synchronize
  Update global counts: $N_{wk} \leftarrow N_{wk} + \sum_p (N_{wkp} - N_{wk})$
until termination criterion satisfied

What if you try and parallelize?

	LDA	AD-LDA
Space	$N+K(D+W)$	$\frac{1}{P}(N+KD)+KW$
Time	NK	$\frac{1}{P}NK+KW+C$

Table 3: Space and time complexity of LDA and AD-LDA.

D=#docs W=#word(types) K=#topics N=words in corpus

	KOS	NIPS	WIKIPEDIA	PUBMED	NEWSGROUPS
D_{train}	3,000	1,500	2,051,929	8,200,000	19500
W	6,906	12,419	120,927	141,043	27,059
N	467,714	2,166,058	344,941,756	737,869,083	2,057,207
D_{test}	430	184	-	-	498

Table 2: Characteristics of data sets used in experiments.

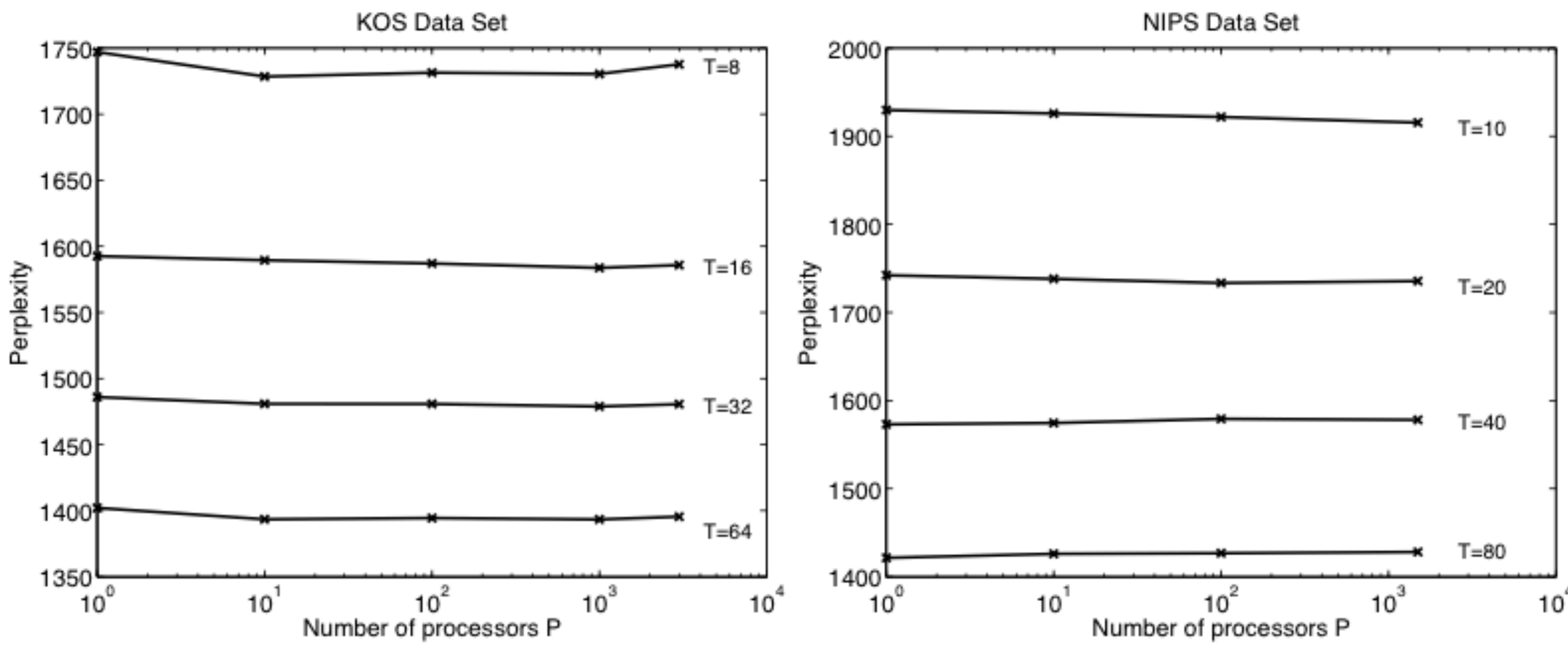
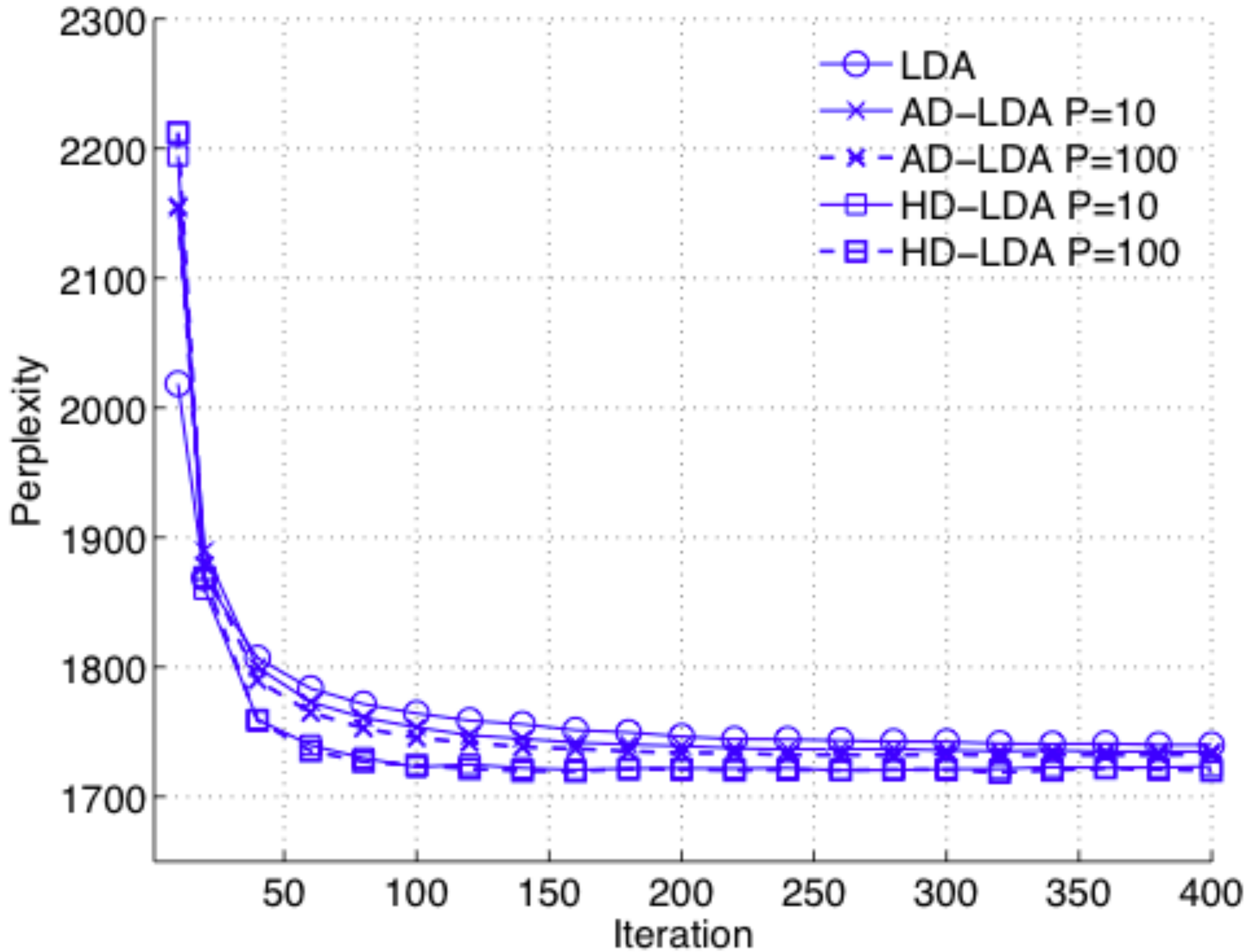


Figure 6: AD-LDA test perplexity versus number of processors up to the limiting case of number of processors equal to number of documents in collection. Left plot shows perplexity for KOS and right plot shows perplexity for NIPS.



LDA
AD−LDA P=10
AD−LDA P=100
HD−LDA P=10
HD−LDA P=100
Perplexity
Iteration
2300
2200
2100
2000
1900
1800
1700
50
100
150
200
250
300
350
400

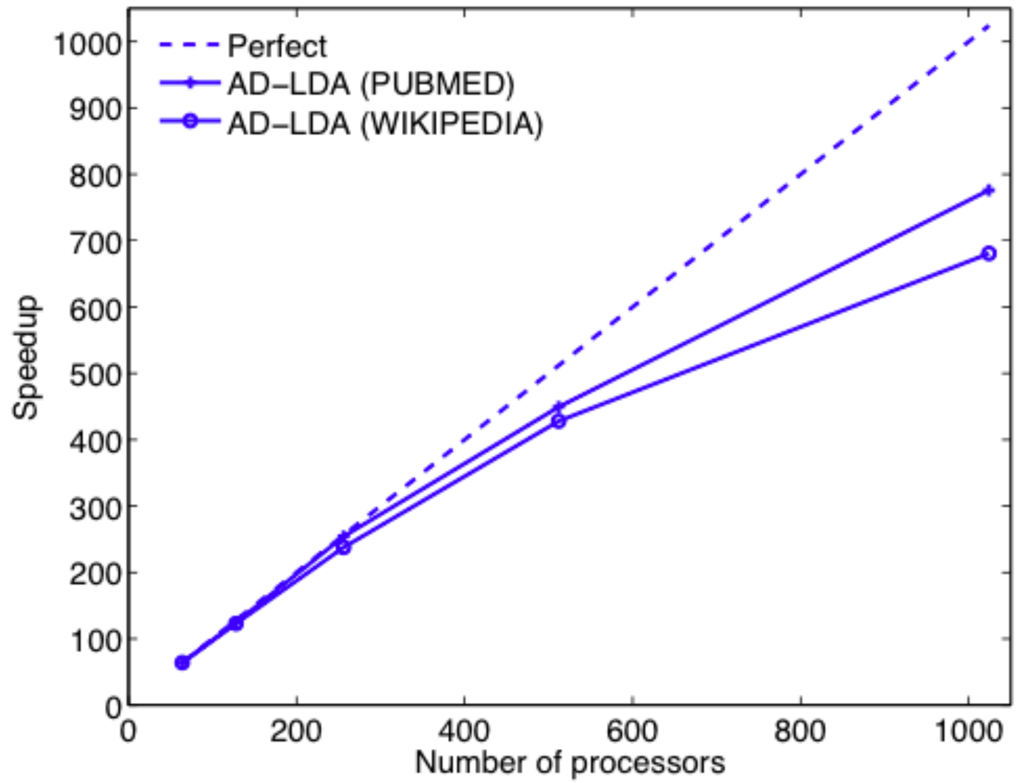


Figure 12: Parallel speedup results for 64 to 1024 processors on multi-million document data sets WIKIPEDIA and PUBMED.

Update *c. 2014*

- Algorithms:
  - Distributed variational EM
  - Asynchronous LDA (AS-LDA)
  - Approximate Distributed LDA (AD-LDA)
  - Ensemble versions of LDA: HLDA, DCM-LDA
- Implementations:
  - GitHub Yahoo_LDA
    - not Hadoop, special-purpose communication code for synchronizing the global counts
    - Alex Smola, Yahoo→CMU
  - Mahout LDA
    - Andy Schlaikjer, CMU→Twitter

Outline

- LDA/Gibbs algorithm details
- How to speed it up by parallelizing
- How to speed it up by faster sampling
  - **Why sampling is key**
  - Some sampling ideas for LDA

Fast Collapsed Gibbs Sampling For Latent Dirichlet Allocation

Ian Porteous
Dept. of Computer Science
University of California, Irvine
Irvine, CA 92697-3425
iporteou@ics.uci.edu

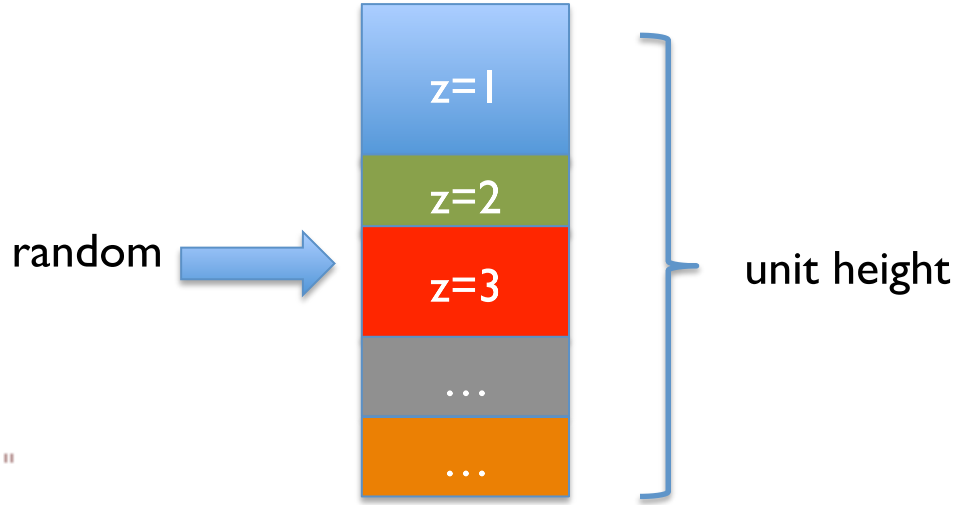
David Newman
Dept. of Computer Science
University of California, Irvine
Irvine, CA 92697-3425
newman@uci.edu

Alexander Ihler
Dept. of Computer Science
University of California, Irvine
Irvine, CA 92697-3425
ihler@ics.uci.edu

Arthur Asuncion
Dept. of Computer Science
University of California, Irvine
Irvine, CA 92697-3425
asuncion@ics.uci.edu

Padhraic Smyth
Dept. of Computer Science
University of California, Irvine
Irvine, CA 92697-3425
smyth@ics.uci.edu

Max Welling
Dept. of Computer Science
University of California, Irvine
Irvine, CA 92697-3425
welling@ics.uci.edu




```
def resample(self,d,j):
    """sample a new value of z[d][j]"""
    p = []
    norm = 0.0
    #compute pk = Pr(z_dj=k | everything else)
    for k in xrange(self.numTopics):
        w = self.x[d][j]
        z_dj_equals_k = 1 if self.z[d][j]==k else 0
        #unnormalized chance of picking topic k in doc d
        ak = (self.docTopicCount[d][k] - z_dj_equals_k + self.alpha)
        #unnormalized chance of picking topic k for word w
        bk = ((self.wordTopicCount[w][k] - z_dj_equals_k + self.beta)
              /(self.totalTopicCount[k] - z_dj_equals_k + self.totalWords * self.beta))
        pk = ak*bk
        p.append(pk)
        norm += pk
    #pick randomly from the normalized pk
    sum_p_up_to_k = 0.0
    r = random.random()
    for k in xrange(self.numTopics):
        sum_p_up_to_k += p[k]/norm
        if r < sum_p_up_to_k:
            return k
```

$$p(z_{ij} = k | \mathbf{z}^{\neg ij}, \mathbf{x}, \alpha, \beta) = \frac{1}{Z} a_{kj} b_{wk} \quad (1)$$

where

$$a_{kj} = N_{kj}^{\neg ij} + \alpha \qquad b_{wk} = \frac{N_{wk}^{\neg ij} + \beta}{N_k^{\neg ij} + W\beta},$$

Z is the normalization constant

$$Z = \sum_k a_{kj} b_{wk},$$

```
def resample(self,d,j):
    """sample a new value
    p = []
    norm = 0.0
    #compute pk = Pr(z_dj=k | everything else)
    for k in xrange(self.numTopics):
        w = self.x[d][j]
        z_dj_equals_k = 1 if self.z[d][j]==k else 0
        #unnormalized chance of picking topic k in doc d
        ak = (self.docTopicCount[d][k] - z_dj_equals_k + self.alpha)
        #unnormalized chance of picking topic k for word w
        bk = ((self.wordTopicCount[w][k] - z_dj_equals_k + self.beta)
              /(self.totalTopicCount[k] - z_dj_equals_k + self.totalWords * self.beta))
        pk = ak*bk
        p.append(pk)
        norm += pk
    #pick randomly from the normalized pk
    sum_p_up_to_k = 0.0
    r = random.random()
    for k in xrange(self.numTopics):
        sum_p_up_to_k += p[k]/norm
        if r < sum_p_up_to_k:
            return k
```

Algorithm 3.1: LDA GIBBS SAMPLING($\mathbf{z}, \mathbf{x}$)

for $i \leftarrow 1$ **to** N
  do
    $u \leftarrow$ draw from Uniform$[0, 1]$
    for $k \leftarrow 1$ **to** K
      do
        $P[k] \leftarrow P[k-1] + \frac{(N_{kj}^{\neg ij} + \alpha)(N_{x_{ij}k}^{\neg ij} + \beta)}{(N_{k}^{\neg ij} + W\beta)}$
    for $k \leftarrow 1$ **to** K
      do
        if $u < P[k]/P[K]$
          then $z_{ij} = k, stop$

Running total of P(z=k|…) or P(z<=k)

Discussion….

- Where do you spend your time?
  - sampling the *z*'s
  - each sampling step involves a loop over *all topics*
  - number of topics *grows with corpus size*
  - this seems wasteful
    - even with many topics, words are often only assigned to a *few* different topics
      - low frequency words appear < K times … and there are lots and lots of them!
      - even frequent words are not in every topic

Results from Porteus et al

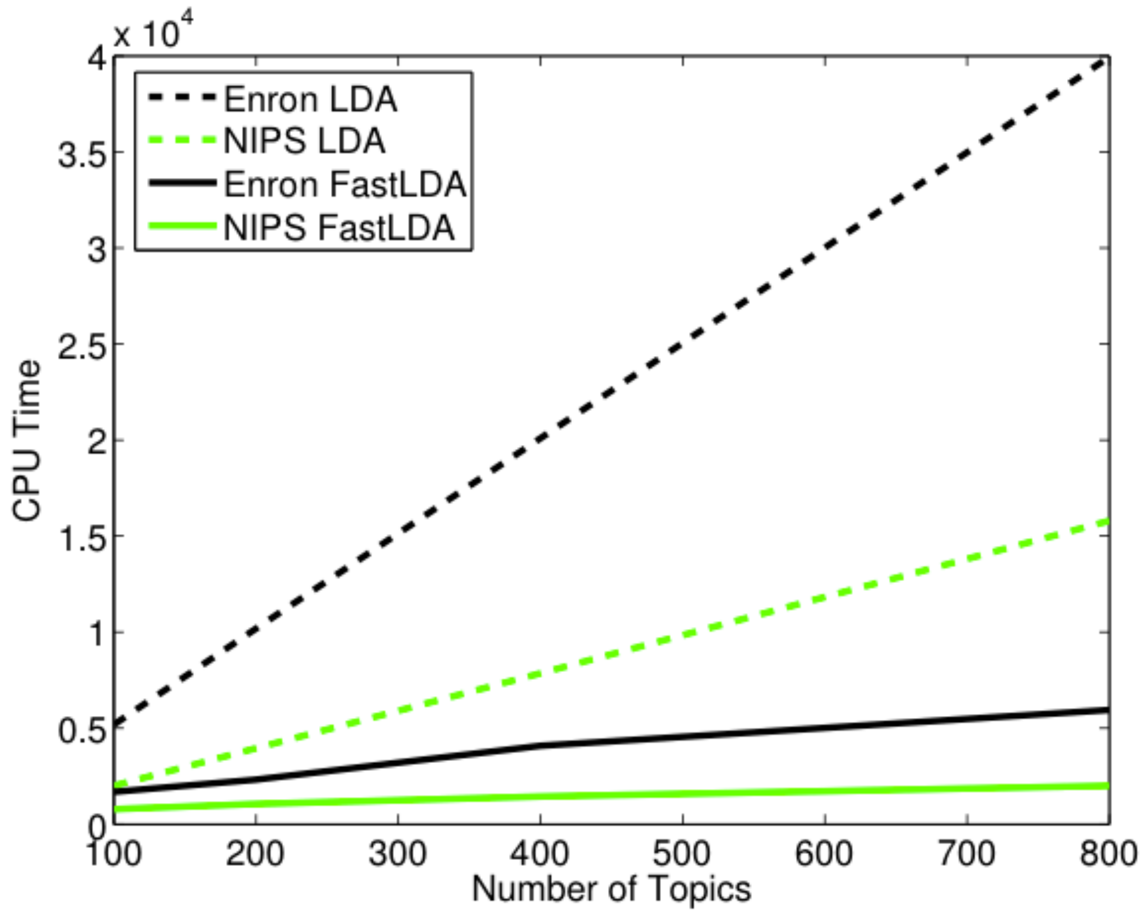


Figure 5: CPU time for LDA and FastLDA, as a function of the number of topics K for NIPS and Enron data sets.

Results

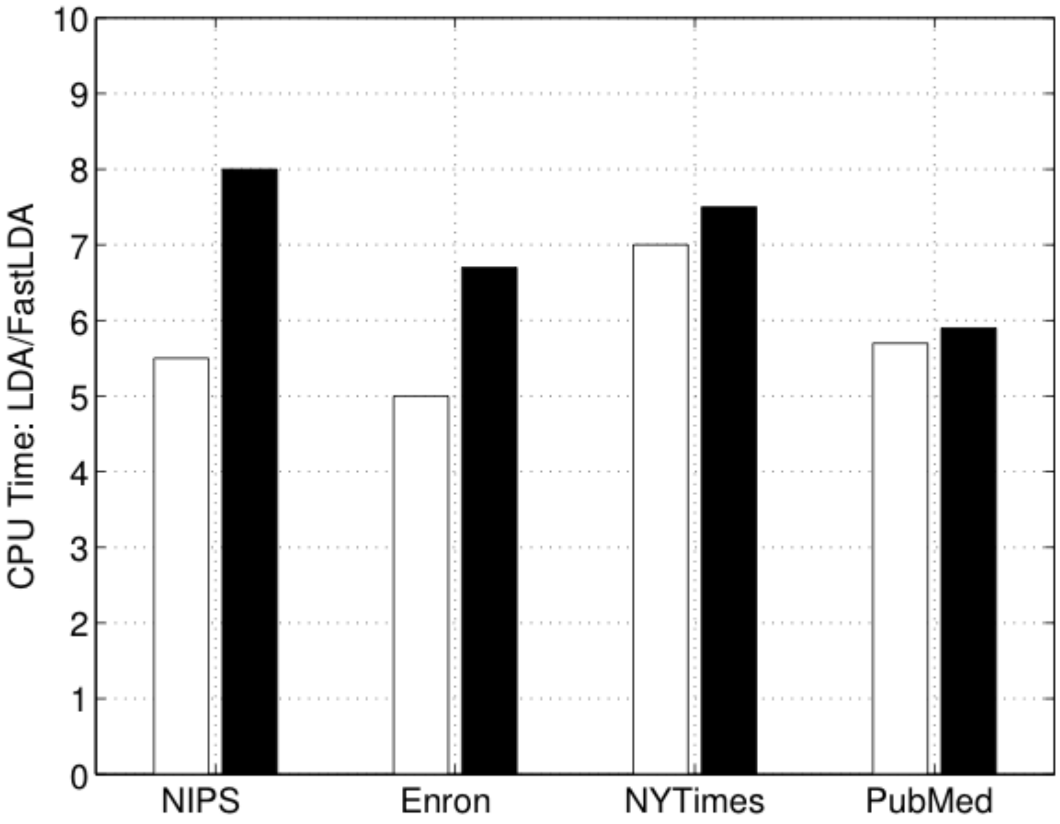


Figure 7: Speedup of FastLDA over LDA for the four corpora. Bars show: NIPS $K = 400, 800$, Enron $K = 400, 800$, NYTimes $K = 800, 1600$ and PubMed $K = 2000, 4000$. $\alpha = 2/K$ for all runs.

Outline

- LDA/Gibbs algorithm details
- How to speed it up by parallelizing
- How to speed it up by faster sampling
  - Why sampling is key
  - Some sampling ideas for LDA
    - **The Mimno/McCallum decomposition**

Efficient Methods for Topic Model Inference on Streaming Document Collections

Limin Yao, David Mimno, and Andrew McCallum
Department of Computer Science
University of Massachusetts, Amherst
{lmyao, mimno, mccallum}@cs.umass.edu

KDD 09

$$P(z = t|w) \quad \propto \quad (\alpha_t + n_{t|d}) \frac{\beta + n_{w|t}}{\beta V + n_{\cdot|t}}.$$

$$P(z = t|w) \propto \frac{\alpha_t \beta}{\beta V + n_{\cdot|t}} + \frac{n_{t|d} \beta}{\beta V + n_{\cdot|t}} + \frac{(\alpha_t + n_{t|d}) n_{w|t}}{\beta V + n_{\cdot|t}}.$$

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

- If $U<s$:
  - lookup U on line segment with tic-marks at $\alpha_1\beta/(\beta V + n_{.|1})$, $\alpha_2\beta/(\beta V + n_{.|2})$, ...
- If $s<U<r$:
  - lookup U on line segment for r

Only need to check t such that $n_{t|d}>0$

$z=s+r+q$

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

- If $U<s$:
  - lookup U on line segment with tic-marks at $\alpha_1\beta/(\beta V + n_{.|1})$, $\alpha_2\beta/(\beta V + n_{.|2})$, ...
- If $s<U<s+r$:
  - lookup U on line segment for r
- If $s+r<U$:
  - lookup U on line segment for q

$z=s+r+q$

$$r = \sum_t \frac{n_{t|d}\beta}{\beta V + n_{.|t}}$$

$$q = \sum_t \frac{(\alpha_t + n_{t|d})n_{w|t}}{\beta V + n_{.|t}}.$$

Only need to check t such that $n_{w|t}>0$

Only need to check occasionally (< 10% of the time)

Only need to check t such that $n_{t|d}>0$

Only need to check t such that $n_{w|t}>0$

$z=s+r+q$

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

Only need to **store** (and maintain) total words per topic and α's, β, V

Trick; count up $n_{t|d}$ for d when you start working on d and update incrementally

Only need to **store** $n_{t|d}$ for current d

Need to **store** $n_{w|t}$ for each word, topic pair …???

$z = s + r + q$

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

$$q = \sum_t \left[\frac{\alpha_t + n_{t|d}}{\beta V + n_{\cdot|t}} \times n_{w|t} \right].$$

1. Precompute, for each t, $\frac{\alpha_t + n_{t|d}}{\beta V + n_{\cdot|t}}$

2. Quickly find t's such that $n_{w|t}$ *is large for w*

z=s+r+q

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

$$r = \sum_t \frac{n_{t|d}\beta}{\beta V + n_{\cdot|t}}$$

$$q = \sum_t \frac{(\alpha_t + n_{t|d}) n_{w|t}}{\beta V + n_{\cdot|t}}.$$

Most (>90%) of the time and space is here…

store $n_{w|t}$ for each word, topic pair …???

$$q = \sum_t \left[\frac{\alpha_t + n_{t|d}}{\beta V + n_{\cdot|t}} \times n_{w|t} \right].$$

1. Precompute, for each t, $\frac{\alpha_t + n_{t|d}}{\beta V + n_{\cdot|t}}$

2. Quickly find t's such that $n_{w|t}$ is large for w

- map w to an int array
  - no larger than frequency w
  - no larger than #topics
- encode *(t,n)* as a bit vector
  - *n* in the high-order bits
  - *t* in the low-order bits
- keep ints sorted in descending order

Most (>90%) of the time and space is here…

store $n_{w|t}$ for each word, topic pair …???

$$q = \sum_t \frac{(\alpha_t + n_{t|d}) n_{w|t}}{\beta V + n_{\cdot|t}}.$$

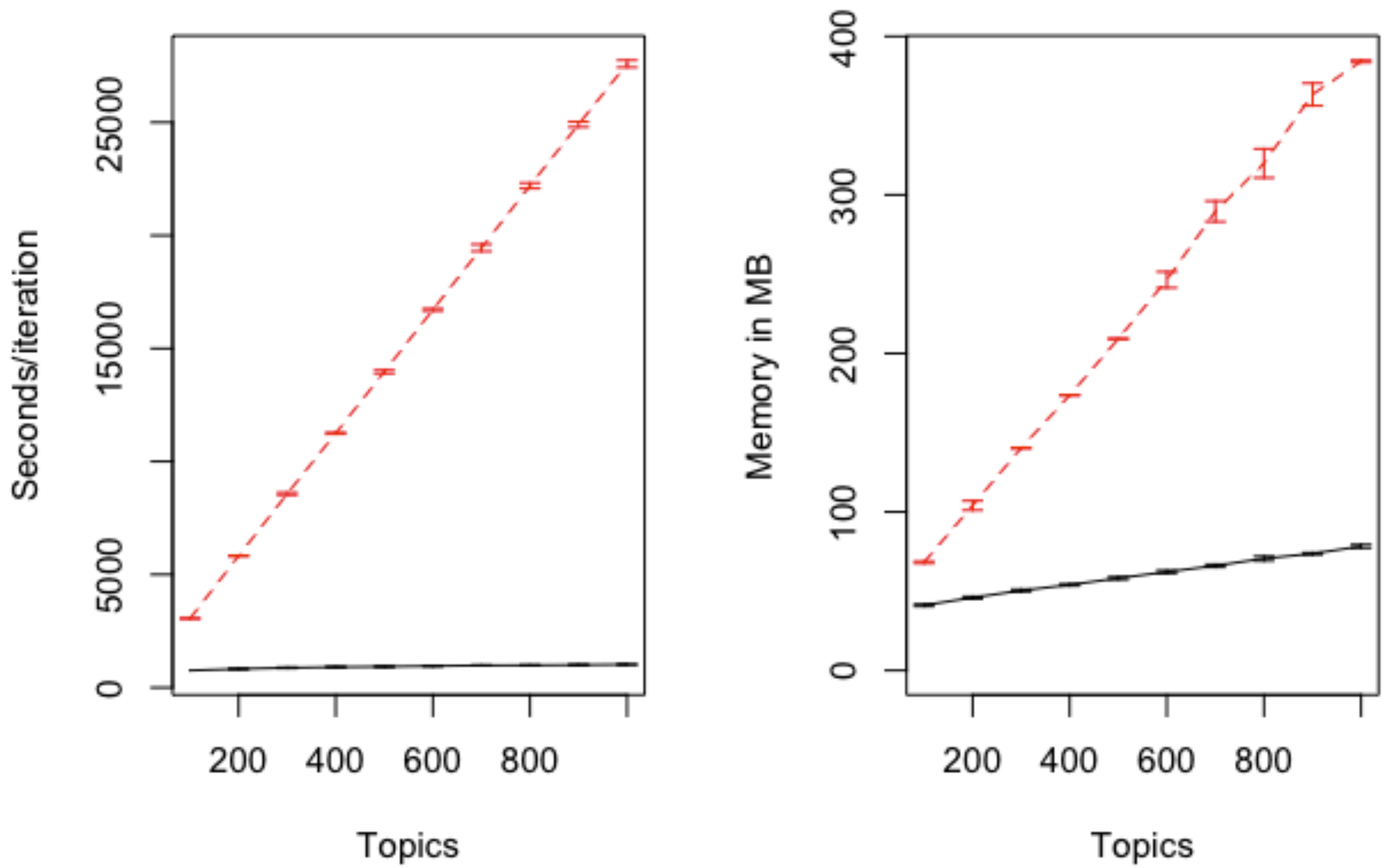


Figure 2: A comparison of time and space efficiency between standard Gibbs sampling (dashed red lines) and the SparseLDA algorithm and data structure presented in this paper (solid black lines). Error bars show the standard deviation over five runs.

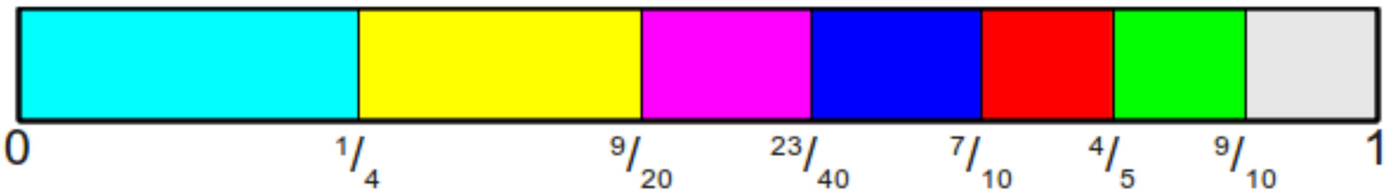
Outline

- LDA/Gibbs algorithm details
- How to speed it up by parallelizing
- How to speed it up by faster sampling
  - Why sampling is key
  - Some sampling ideas for LDA
    - The Mimno/McCallum decomposition (SparseLDA)
    - **Alias tables** (Walker 1977; Li, Ahmed, Ravi, Smola KDD 2014)

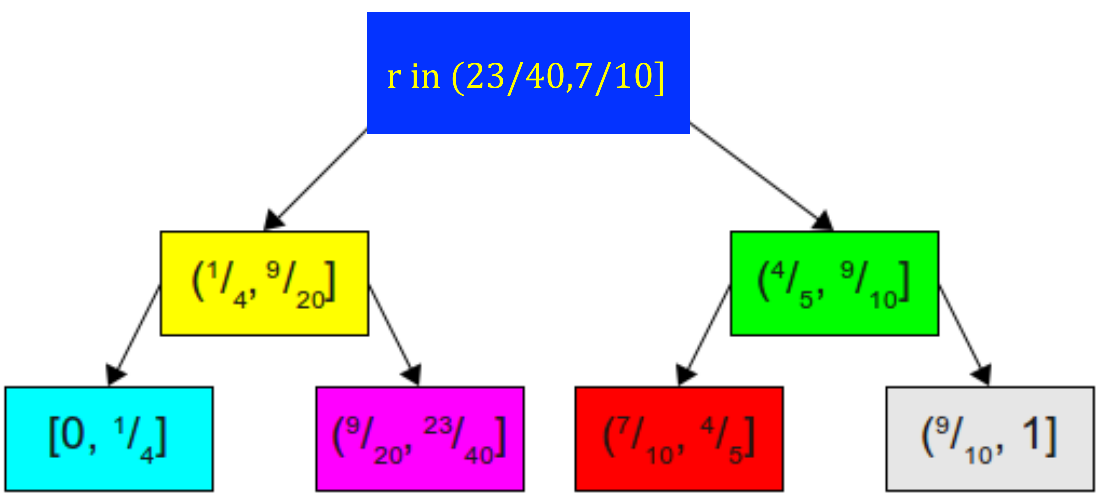
Alias tables

O(K)

Basic problem: how can we sample from a biased coin quickly?



If the distribution changes slowly maybe we can do some preprocessing and then sample multiple times. Proof of concept: generate *r~uniform* and use a binary tree

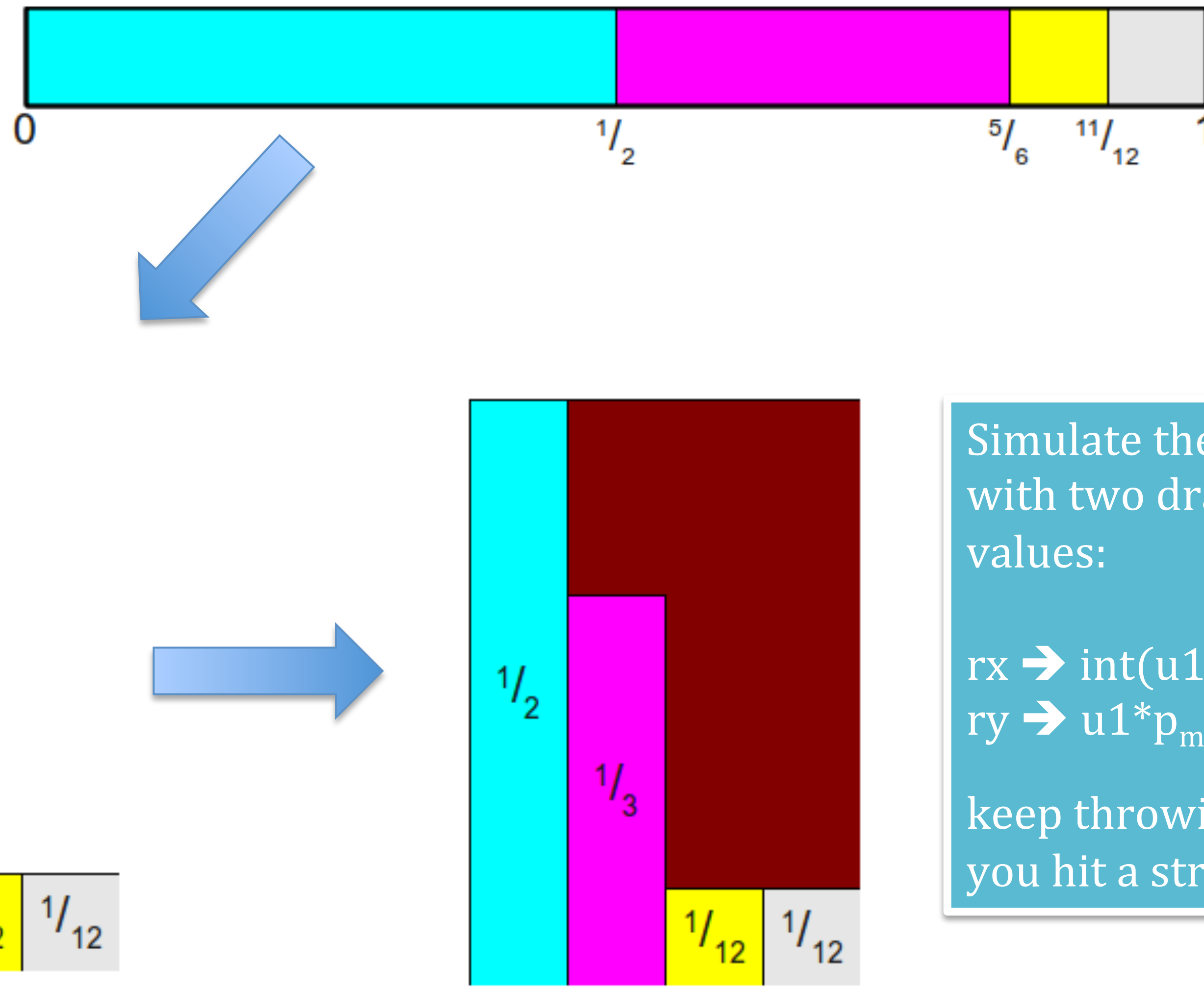


O(log2K)

http://www.keithschwarz.com/darts-dice-coins/

Alias tables

Another idea…



Simulate the dart with two drawn values:

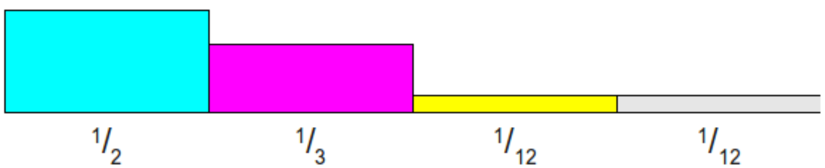
rx → int(u1*K)
ry → u1*p_{max}

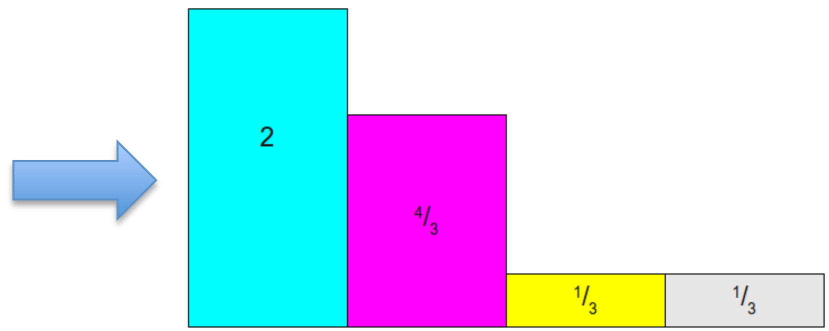
keep throwing till you hit a stripe

http://www.keithschwarz.com/darts-dice-coins/

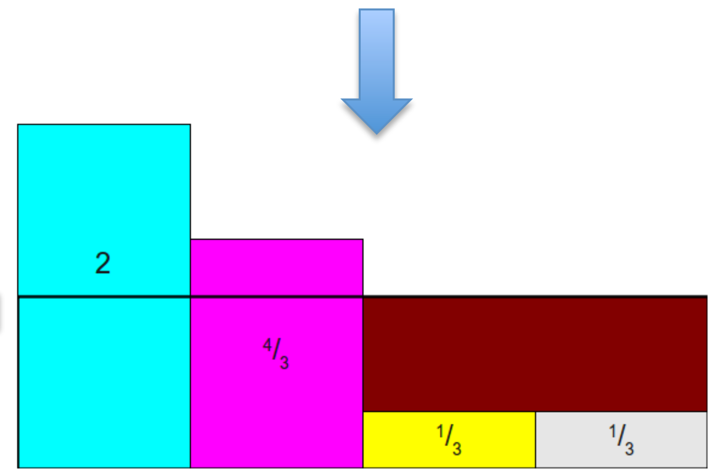
Alias tables

An even more clever idea: minimize the brown space (where the dart "misses") by sizing the rectangle's height to the *average* probability, not the *maximum* probability, and cutting and pasting a bit.





You can always do this using only **two** colors in each column of the final *alias table* and the dart **never misses!**



mathematically speaking…

http://www.keithschwarz.com/darts-dice-coins/

Reducing the Sampling Complexity of Topic Models

KDD 2014

Aaron Q. Li
CMU Language Technologies
Pittsburgh, PA
aaronli@cmu.edu

Amr Ahmed
Google Strategic Technologies
Mountain View, CA
amra@google.com

Sujith Ravi
Google Strategic Technologies
Mountain View, CA
sravi@google.com

Alexander J. Smola
CMU MLD and Google ST
Pittsburgh PA
alex@smola.org

Key ideas

- use variant of Mimno/McCallum decomposition

$$P(z=t|w) \propto \frac{\alpha_t \beta}{\beta V + n_{\cdot|t}} + \frac{n_{t|d}\beta}{\beta V + n_{\cdot|t}} + \frac{(\alpha_t + n_{t|d})n_{w|t}}{\beta V + n_{\cdot|t}}.$$

- Use alias tables to sample from the dense parts

- Since the alias table gradually goes stale, use Metropolis-Hastings sampling instead of Gibbs

Reducing the Sampling Complexity of Topic Models

KDD 2014

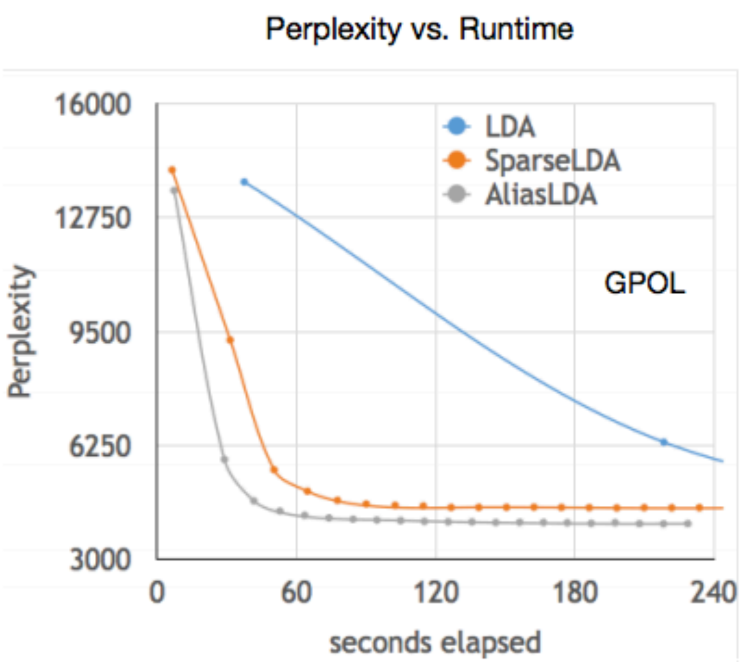
StationaryMetropolisHastings(p, q, n)

if no initial state exists **then** $i \sim q(i)$
for $l = 1$ **to** n **do**
  Draw $j \sim q(j)$
  if RandUnif(1) $< \min\left(1, \frac{p(j)q(i)}{p(i)q(j)}\right)$ **then**
    $i \leftarrow j$
  end if **else** the dart missed
end for
return i

- *q* is stale, easy-to-draw from distribution
- *p* is updated distribution
- computing ratios $p(i)/q(i)$ is cheap
- usually the ratio is close to one

Reducing the Sampling Complexity of Topic Models

KDD 2014



Dataset	V	L	D	T	L/V	L/D
RedState	12,272	321,699	2,045	231	26.21	157
GPOL	73,444	2,638,750	14,377	1,596	35.9	183
Enron	28,099	6,125,138	36,999	2,860	218	165
PubMedSmall	106,797	35,980,539	546,665	2,002	337	66
NYTimes	101,636	98,607,383	297,253	2,497	970	331

Table 1: Datasets and their statistics. V: vocabulary size; L: total number of training tokens, D: number of training documents; T: number of test documents. L/V is the average number occurrences of a word. L/D is the average document length.