
Optimization of Text Processing for the WordFlare Knowledge Graph

*ECSS Symposium
June 16, 2015*

Robert Sinkovits
San Diego Supercomputer Center

Project overview

The goal of the WordFlare project is to create a tablet-based app to engage K-12 and lifelong learners in exploring language and knowledge. The app is based on a massive thesaurus and features dynamic visualizations of word relationships. Approximately 9% of the content is human-curated, while the other 91% is derived using computational methods executed on XSEDE resources.

ECSS project focused on two tasks

1. Optimization of the Latent Dirichlet Allocation (LDA) algorithm used to discover topics in a body of text
2. Development of a fast method to simultaneously search for large numbers of words in a corpus

Latent Dirichlet Allocation (LDA)

IDEA.org used the GibbsLDA++ implementation of the LDA algorithm, a fast C++ version developed by Xuan-Hieu Phan and Cam-Tu Nguyen. Fortunately, I was able to ignore the impenetrable looking math and focus on just the software

UGH!

$$\begin{aligned} &= \left(\frac{\Gamma(\sum_{i=1}^K \alpha_i)}{\prod_{i=1}^K \Gamma(\alpha_i)} \right)^M \prod_{j \neq m} \frac{\prod_{i=1}^K \Gamma(n_{j,(\cdot)}^i + \alpha_i)}{\Gamma(\sum_{i=1}^K n_{j,(\cdot)}^i + \alpha_i)} \\ &\quad \times \left(\frac{\Gamma(\sum_{r=1}^V \beta_r)}{\prod_{r=1}^V \Gamma(\beta_r)} \right)^K \prod_{i=1}^K \prod_{r \neq v} \Gamma(n_{(\cdot),r}^i + \beta_r) \\ &\quad \times \frac{\prod_{i=1}^K \Gamma(n_{m,(\cdot)}^i + \alpha_i)}{\Gamma(\sum_{i=1}^K n_{m,(\cdot)}^i + \alpha_i)} \prod_{i=1}^K \frac{\Gamma(n_{(\cdot),v}^i + \beta_v)}{\Gamma(\sum_{r=1}^V n_{(\cdot),r}^i + \beta_r)} \\ &\propto \frac{\prod_{i=1}^K \Gamma(n_{m,(\cdot)}^i + \alpha_i)}{\Gamma(\sum_{i=1}^K n_{m,(\cdot)}^i + \alpha_i)} \prod_{i=1}^K \frac{\Gamma(n_{(\cdot),v}^i + \beta_v)}{\Gamma(\sum_{r=1}^V n_{(\cdot),r}^i + \beta_r)} \\ &\propto \prod_{i=1}^K \Gamma(n_{m,(\cdot)}^i + \alpha_i) \prod_{i=1}^K \frac{\Gamma(n_{(\cdot),v}^i + \beta_v)}{\Gamma(\sum_{r=1}^V n_{(\cdot),r}^i + \beta_r)} \dots \end{aligned}$$

Latent Dirichlet Allocation (LDA)

Profiling the code indicated that virtually all of the time was spent in a single method (model::sampling), which was called repeatedly within a pair of nested loops.

```
for (int m=0; m<M; m++) {  
  for (int n=0; n<N; n++) {  
    int topic = sampling(m,n);  
    z[m][n] = topic;  
  }  
}
```

Hmmm... method called
n times in a row with the
same value of m

Latent Dirichlet Allocation (LDA)

sampling(m,n) called repeatedly with same value of m and that only a few elements of nd, nwsum and ndsum are (temporarily) updated

```
int model::sampling(int m, int n) {  
    int topic = z[m][n];  
    int w = ptrndata->docs[m]->words[n];  
    nw[w][topic] -= 1;  
    nd[m][topic] -= 1;  
    nwsum[topic] -= 1;  
    ndsum[m] -= 1;
```

```
    for (int k = 0; k < K; k++) {  
        p[k] = (nw[w][k] + b) / (nwsum[k] + Vb) *  
               (nd[m][k] + a) / (ndsum[m] + Ka);  
    }
```

```
    nw[w][topic] += 1;  
    nd[m][topic] += 1;  
    nwsum[topic] += 1;  
    ndsum[m] += 1;  
    return topic;  
}
```

```
for (int m=0; m<M; m++) {  
    for (int n=0; n<N; n++) {  
        int topic = sampling(m,n);  
        z[m][n] = topic;  
    }  
}
```

Potential
invariants

Latent Dirichlet Allocation (LDA)

Pre-calculate array of values that do not change (much) across successive calls to sampling and update only necessary elements

```
int model::sampling(int m, int n) {
    int topic = z[m][n];
    nd[m][topic] -= 1;
    nwsum[topic] -= 1;
    f1[topic] = (nd[m][topic] + a) /
                ((nwsum[topic] + Vb)*
                 (ndsum[m] - 1.0 + Ka));

    for (int k = 0; k < K; k++) {
        p[k] = (nw[w][k] + b) * f1[k];
    }

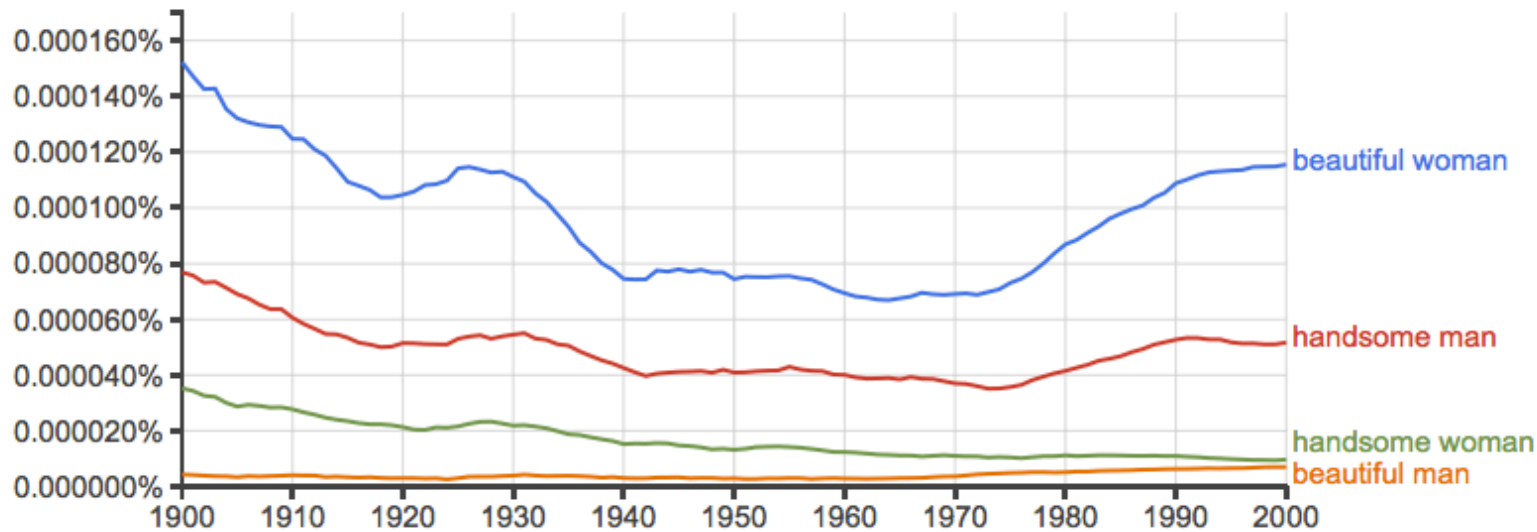
    nd[m][topic] += 1;
    nwsum[topic] += 1;
    f1[topic] = ...;
}
```

```
for (int m=0; m<M; m++) {
    for (int k = 0; k < K; k++) {
        f1[k] = (nd[m][k] + a) /
                ((nwsum[k] + Vb)*
                 (ndsum[m] - 1.0 + Ka));
    }
    for (int n=0; n<N; n++) {
        int topic = sampling(m,n);
        z[m][n] = topic;
    }
}
```

Entire application
is now 1.5-2.2x
faster, depending
on number of topics

Finding words that appear together

Researches from the IDEA.org non-profit want to identify words that are commonly found together. Google solves part of the problem with it's Ngram project and can tell us how often a given word is followed by other words.



<https://books.google.com/ngrams>

Finding words that appear together

What Google can't tell us directly though is how often a given word is preceded by other words.

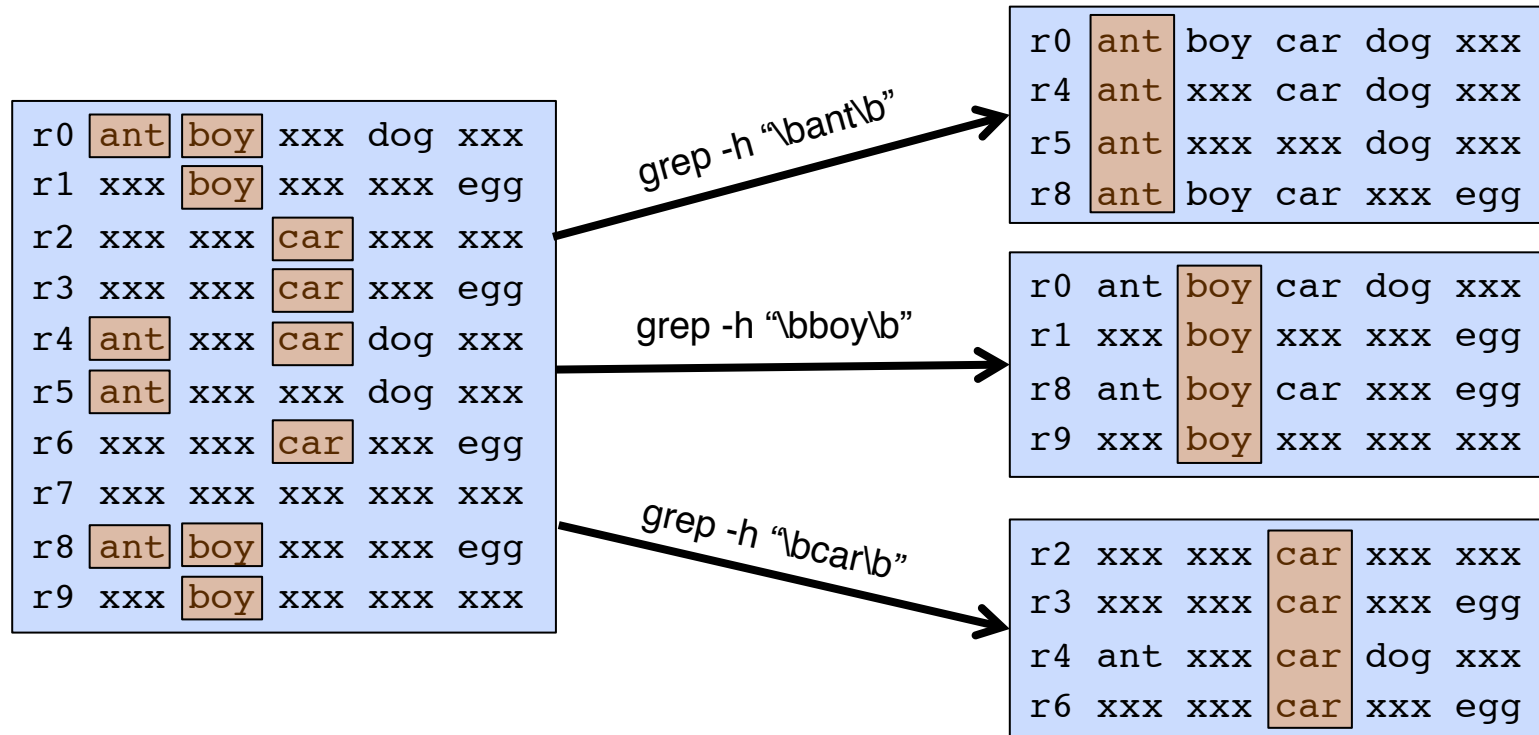
- It's easy to learn that “handsome” is followed frequently by “man”
- It's much harder to find out which words frequently precede “man”

To do this, we'll search through the Ngram data for all occurrences of a word. In addition, we want to do this for a very large list of words, $O(100K-1M)$, rather than just a few words.

The first step in this project is to use grep or some other tool to generate results that we can then subject to further analysis.

Word search using grep

Linux grep utility is probably the fastest way to search for a word or small number of words in a document



Word search using grep

Once we start searching for a large number of words in a document, grep can be inefficient. Need to read the entire file and parse lines for each execution of grep. Effort scales as $O(N_{\text{words}})$

r0	ant	boy	xxx	dog	xxx
r1	xxx	boy	xxx	xxx	egg
r2	xxx	xxx	car	xxx	xxx
r3	xxx	xxx	car	xxx	egg
r4	ant	xxx	car	dog	xxx
r5	ant	xxx	xxx	dog	xxx
r6	xxx	xxx	car	xxx	egg
r7	xxx	xxx	xxx	xxx	xxx
r8	ant	boy	xxx	xxx	egg
r9	xxx	boy	xxx	xxx	xxx

```
Read line r0, search for "ant"
Read line r1, search for "ant"
Read line r2, search for "ant"
...
Read line r0, search for "boy"
Read line r1, search for "boy"
Read line r2, search for "boy"
...
Read line r0, search for "car"
Read line r1, search for "car"
Read line r2, search for "car"
...
```

Word search using hashes

When searching for large number of words in a document, it's better to read each line of the document once and then search for all words in one fell swoop. Here's our new approach

```
Read list of words (vocabulary) that will be searched for in corpus
Generate a hash from the word list
```

```
Loop over records in corpus
```

```
    Loop over words in record
```

```
        If word is found in vocabulary write record to word.txt
```

There are a few complications not shown (e.g. I/O buffering), but overall pretty simple and doable in ~ 70 lines of Perl code

Word search using hashes

When searching for large number of words in a document, it's better to read each line of the document once and then search for all words.

```
hash = (ant:1, boy:1, car:1, dog:1, egg:1)
```

```
1st line of document → (ant, boy, xxx, dog, yyy)
```

```
Is "ant" found in hash?
```

```
Yes: write "ant boy xxx dog yyy" to ant.txt
```

```
Is "boy" found in hash?
```

```
Yes: write "ant boy xxx dog yyy" to boy.txt
```

```
Is "xxx" found in hash?
```

```
No: Do nothing
```

```
Is "dog" found in hash?
```

```
Yes: write "ant boy xxx dog yyy" to dog.txt
```

```
Is "yyy" found in hash?
```

```
No: Do nothing
```

```
2nd line of document → (xxx, boy, yyy, zzz, egg)
```

```
Is "xxx" found in hash
```

```
No: Do nothing
```

```
Is "boy" found in hash
```

```
Yes: write "xxx boy yyy zzz egg" to boy.txt
```

Use hash for fast $O(1)$ lookup time

Read and parse each line from document only once. Work amortized over entire word list

Benchmarks

All timings obtained using single core on Gordon compute node, with data read from and written to SSD. Search done against Google 2-gram data for words starting with “mo” (319 million records). Timings do not include time to copy Google ngram data from parallel file system (/oasis) to SSD and results back to /oasis (~ 1 minute).

Word list	# words	time (hh:mm)	grep time (hh:mm)	speedup
10K ¹	100	00:43	0:31	0.72x
10K ¹	1000	00:43	5:10	7.2x
10K ¹	10,000	00:46	51:40 ³	67x
Wiktionary ²	404,248	1:22	2088:37 ³	1528x

(1) 10K word list provided by IDEA

(2) Wiktionary data limited to words containing just letters and numbers

(3) Estimates

Benchmarks – Perl vs. Ruby

The programmers at IDEA.org have backgrounds in Ruby and C++, but not Perl. To help them effectively use and manage our software solution, we re-implemented in Ruby. Although the Ruby version is a little slower, the benefits of being able to work in a familiar language outweigh this performance hit.

Corpus = 2-gram “mo” data, Word list = Wiktionary (404,248 words)

Language	time (hh:mm)	Relative speed
Perl	1:22	1
Ruby	1:41	0.81

Managing large file counts

Now that you have 400,000 files, what do you do with them?

- Output too large to store on NFS
- Large file count will overwhelm Lustre parallel file system

Solution:

- Write files to SSD
- Create archive (tar -cf results.tar results)
- Move archive to parallel file system
- Extract files as needed (tar -xf results.tar results/word.txt)

Gotchas

- **Do not** deflate entire archive in Lustre (tar -xf results.tar)!
- Delete extracted files when no longer needed

Summary

- Modest improvements to the GibbsLDA++ code, 1.5x to 2.2x speedup depending on number of topics. Potential to have a large impact since the LDA algorithm is so heavily used in the text mining and linguistics communities
- Game changing $\sim 1000x$ speedup in identifying Ngram records containing specific words. Computing needs now $O(10^3)$ rather than $O(10^6)$ SU.
- ... But, XSEDE resources still needed due to large data volumes and large file counts $O(10^5)$ to $O(10^6)$