BigML Assignment 3: Streaming Phrase Finding

Due Tuesday, February 14, 2012 via Blackboard

Out February 7, 2012

1 Important Note

This assignment is the first of two that use the phrase finding algorithm we discussed in class (see Tomokiyo and Hearst, 2003, http://dl.acm.org/citation.cfm?id=1119287). You will be expected to reuse the code you develop for this assignment for a future assignment, and **you are expected to use Java for this assignment**.

Alona Fyshe (afyshe@cs.cmu.edu) is the contact TA for this homework. Please post clarification questions to the Google Group:
`machine-learning-with-large-datasets-10-605-in-spring-2012`

2 Streaming Phrase Finding

This assignment is based on the method in *A language model approach to keyphrase extraction*, Tomokiyo and Hearst, 2003, http://dl.acm.org/citation.cfm?id=1119287 . The basic goal is to pick out word sequences in a corpus that are meaningful as phrases (in the paper's words, have high phraseness), and over-represented in the corpus (have high informativeness). Phraseness and informativeness are computed based on some simple frequency features, which we will denote as

C(x,y)	frequency of the phrase x y in the corpus of interest
B(x,y)	analogous for the background corpus
C(x), C(y)	frequency of a word x or y in the corpus of interest
B(x), B(y)	analogous for the background corpus

You also need a handful of constants, like the number of words and bigrams in each corpus, and the vocabulary sizes. For this assignment, the *corpus of interest* (foreground corpus) is those books published in the years 1990-1999. The background corpus is those books published in the years 1960-1989. We've done some of the preprocessing for you (for example, we've thrown out words and phrases with non-letter characters, normalized the

words to their lowercase variants, and aggregated the counts by decade). As before, there is a smaller test set for debugging your code.

We suggest using a simple data structure that associates a set of attribute-value pairs with a phrase. Once you have this data structure - for instance like the examples below - its easy to compute phraseness and informativeness in a streaming setting:

Key	Value
united states	Cxy=104324,Bxy=214442,Cx=321313,Cy=424134,Bx=23141,By=444141
white house	Cxy=4343003,Bxy=43322,Cx=2344233,Cy=786677,Bx=235661,By=1056773

Consider the first row in the example above. Cxy denotes the C(x,y) count for the phrase `united states` in the foreground corpus. Cx is the count for `united` and Cy is the count for `states` in the foreground corpus. Its easy enough to tally up these counts over the corpus. The tricky part is getting these counts in one place.

As in assignment 2, we're going to assume the unigrams won't fit in memory; all Java commands must be followed with -Xmx128m. We must implement a stream and sort algorithm to request count, fulfill the requests, and aggregate the responses. Here's a general outline of the algorithm for aggregating the counts (refer also to the lecture slides from January 31):

1. Compute counter files mapping $x \rightarrow Bx, x \rightarrow Cx, xy \rightarrow Cxy, xy \rightarrow Bxy$

   - This is mostly done, but you will need to aggregate together the counts for the different decades in the background corpus.

2. Gather together the background and foreground counts for each unigram to create one data structure with key `x` and value `Bx=some number,Cx=some number`. Do the same for the bigrams.

3. Stream through all the phrases and for each phrase x y, create two messages–one asking for the unigram frequencies for x, and one asking for the frequencies of y. Each message is just a pair consisting of the query word and the phrase making the request: so the phrase united states will generate the messages

   ```
   united,united states
   states,united states
   ```

4. To deliver the messages, sort them in with the unigram frequency file. Use a secondary key so the attribute-value pairs come first, and the messages come last (as described in lecture). While scanning through the unigram file, you will do something like this:

   (a) for each distinct key x (e.g. `united`)

i. read the attribute-value pairs for x (e.g. `Bx=14342,Cx=24123`)

ii. for each message from step two addressed to x from a phrase of the form xz (e.g. `united states`):

   A. send a message to xz with the attribute-value list at the content. (i.e. write out a pair with key xz and value of the attribute-value list (eg, `united states, Bx=14342,Cx=24123`)

iii. for each message from step two addressed to x from a phrase of the form zx (e.g. `fly united`):

   A. send a message to zx with the attribute-value list at the content. (i.e. write out a pair with key zx and value of the attribute-value list (eg, `united states, By=14342,Cy=24123`)
   (***Notice the change from x to y in the value list***)

5. Now you've created two new data structures for each bigram, one with `Bx=some number,Cx=some number` and one with `By=some number,Cy=some number` . To deliver these messages, sort them in with the bigram frequency file that contains `Bxy=some number,Cxy=some number` , and merge the data structures.

Now you have all of the counts in the same data structure, and can compute the phraseness and informativeness scores as described in the paper. Recall the definition of point wise KL Divergence:

$$\delta_w(p||q) = p(w)log\left(\frac{p(w)}{q(w)}\right) \tag{1}$$

Phraseness is point wise KL Divergence

$$p = p_{fg}(x \wedge y) \quad q = p_{fg}(x)p_{fg}(y). \tag{2}$$

Informativeness is point wise KL Divergence with

$$p = p_{fg}(x \wedge y) \quad q = p_{bg}(x \wedge y) \tag{3}$$

Where p_{fg} is the probability of an event under the foreground corpus (corpus of interest), and p_{bg} is the probability of an event under the background corpus. The phrase score is just the sum of phraseness and informativeness. Please use the natural logarithm for this assignment. The paper uses a more complex smoothing algorithm, but you should just use add one smoothing, as in Assignments 1 & 2.

3 The Data

We are using the google books corpus. We've done some preprocessing of the data, and created two sets of data files. This is unsupervised learning, so there is no test set.

The data has the following format:

`<text>\t<decade>\t<count>`

where `text` is either a bigram or a unigram, and count is the number of times that text occurred in a book in the given decade. The smaller dataset is all bigrams that contain the word *apple*, and all unigrams appearing in those bigrams. This subsample was chosen because Apple Computer became a more popular company in the 90s, and thus the word Apple took on a different meaning. You should be able to see that trend in the phrase statistics you calculate for the smaller dataset.

The data appears at `/afs/cs.cmu.edu/project/bigML/phrases/`. Files with the word *apple* in the name are the subsampled datasets.

4 Deliverables

Submit a compressed archive (zip, tar, etc) of your code. Your code should print out the phrases and the scores in this format:

`<phrase>\t<total score>\t<phraseness score>\t<informativeness score>`

where total score is the sum of phraseness score and informativeness score.

With it, include a makefile so that

- The command *make demo* will compile counts from scratch (i.e. perform all of the steps outlined above) using the apple corpus on afs, and write to stdout the top 20 phrases sorted by by total score.

In addition to your code and a makefile, please include a pdf document with:

1. The top 20 phrases (sorted by total score) from the full data set and the apple data set.
2. Answers to the following questions:
   (a) What do you notice about the phrases ranked highest in your results for the two data sets? Do they give you any insights into what was going on in the 90s?
   (b) Are there any downfalls you see to using the total phrase score? For example, are there some phrases that are ranked high even though you don't think they should be? Why are they ranked so high?
   (c) How could you improve upon the total score proposed by Tomokiyo and Hurst?

BONUS question:

1. Implement the improvement you outlined for question 2 (c), and show the results. How is your method better? Are there any further improvements to be made?

5 Marking breakdown

- Code correctness and makefile functionality [**60 points**].
- 1 [**10 points**]
- 2a [**10 points**]
- 2b [**10 points**]
- 2c [**10 points**]
- Bonus question: [**15 points**]

6 Hints/Good to know

Unix sort can handle very large files. This is helpful when you need to collect together the counts for a particular key. To ensure sort behaves properly, you should set the following environment variable:

```
LC_ALL='C'
```

Get it to sort using tabs by setting this flag:

```
-t $'\t'
```

And, when files are large it may be a good idea to tell sort where to store its temporary files:

```
-T /some/dir
```