BigML Assignment 2: Streaming Phrase Finding

Due February 12, 2015 via Autolab

Out Jan 29, 2015

You can work in groups. However, no written notes can be shared, or taken during group discussions. You may ask clarifying questions on Piazza. However, under no circumstances should you reveal any part of the answer publicly on Piazza or any other public website. The intention of this policy is to facilitate learning, not circumvent it. Any incidents of plagiarism will be handled in accordance with CMU's Policy on Academic Integrity.

- Did you receive any help whatsoever from anyone in solving this assignment? Yes / No.
- If you answered *yes*, give full details: _____ (e.g. *Jane explained to me what is asked in Question 3.4*)
- Did you give any help whatsoever to anyone in solving this assignment? Yes / No.
- If you answered *yes*, give full details: _____ (e.g. *I pointed Joe to section 2.3 to help him with Question 2*).

1 Important Note

This assignment is the first of two that use the phrase finding algorithm we discussed in class (see Tomokiyo and Hurst, 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**.

Yipei Wang (yipeiw@cmu.edu) and Rahul Goutam (rgoutam@cs.cmu.edu) are the contact TAs for this homework. Please post questions on piazza.

2 Streaming Phrase Finding

This assignment is based on the method in *A language model approach to keyphrase extraction*, Tomokiyo and Hurst, 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 1960-1969. The background corpus is those books published in the years 1970-1999. 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 - it's 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 all the counts for one bigram into 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. So, we will 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 $x \to Bx, x \to Cx, xy \to Cxy, xy \to Bxy$
   - 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 with

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

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 *war*, and all unigrams appearing in those bigrams.

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

You should ignore the bigrams with a stop word inside since we are interested in the informative phrases. Please use the stop word list given on the course wiki page.

4 Autolab Implementation Details

Following the first two steps in the algorithm described in the Section 2, you should implement the Aggregate.java class, and merge the counts using:

```
cat bigram.txt | sort -k1 | java -Xmx128m Aggregate 1 > bigram_processed.txt
cat unigram.txt | sort -k1 | java -Xmx128m Aggregate 0 > unigram_processed.txt
```

The only argument in the Aggregate.java function is the identifier for n-gram: 1 means the e being processed is the bigram counts, while 0 denotes the unigram setup. According to the third step in the phrase finding algorithm, you should implement and run the MessageGenerator.java class, using the following command:

```
cat bigram_processed.txt | java -Xmx128m MessageGenerator > message.txt
```

Next, you will need to write the MessageUnigramCombiner.java class that follows the step 4 in the algorithm described in Section 2. The java class must be able to run using the following command:

```
cat message.txt unigram_processed.txt | sort -k1,1 | \
java -Xmx128m MessageUnigramCombiner > message_unigram.txt
```

Finally, in order to generate the final results, you need to write the PhraseGenerator.java, which follows the step 5 in the algorithm, and should be able to run using:

```
cat message_unigram.txt bigram_processed.txt | sort -k1,2 | \
java -Xmx128m PhraseGenerator
```

Your PhraseGenerator code should write to stdout the top-20 phrases sorted by by total score. To be more specific, it must print out the phrases and the scores in this format: ¡phrase¿¡total score¿¡phraseness score¿¡informativeness score¿ ... where total score is the sum of phraseness score and informativeness score. You should tar the following items into hw2.tar and submit to the homework assignment via Autolab:

1. Aggregate.java
2. MessageGenerator.java
3. MessageUnigramCombiner.java
4. PhraseGenerator.java

and all other auxiliary functions you have written

5 Deliverables

Compress your files using "tar -cvf hw2.tar *.java report.pdf". Do NOT include any subfolders in your tar package.

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.

6 Report

1. The top 20 phrases (sorted by total score) from the full data set and the war data set.(3 points)

2. Please use year from 1970-1979, 1980-1989 and 1990-1999 as foreground separately, and compare the top 20 phrases from the war data. Do they give you any insights into events or trends? Your answer must be within 50 words.(3 points)

3. 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?Your answer must be within 30 words. (2 points)

4. Consider the workflow discussed in class:

   ```
   java CountForNB train.dat > eventCounts.dat \\
   java CountsByWord eventCounts.dat | sort | java CollectRecords > words.dat \\
   requestWordCounts test.dat \
   cat words.dat | sort | java answerWordCountRequests \ \\
   cat test.dat| sort | testNBUsingRequests\\
   ```

   and also consider a corpus with these documents in class ?breakfast?:
   likes toast
   Jane likes toast and jam
   Joe burnt his toast
   and these in class ?dinner?:
   Joe likes steak
   Mary likes steak and ale
   and finally the test document:
   Jane ordered eggs and toast
   Part 1. Answer the questions below (5 points):

   (a) What are the entries in eventCounts.dat associated with the words "toast", "likes", and "steak"?

(b) What are the entries in words.dat associated with the words "toast", "likes", and "steak"?

(c) What is the output of requestWordCounts on the test corpus? Please write key values pairs as "key//value" so we can see the different parts easily.

(d) What is the output of answerWordCountRequests on the test corpus?

(e) What is the input to testNBUsingRequests?

Part 2. Suppose there are K classes, V distinct words in the training corpus, and N tokens in the test corpus. Answer the questions below (7 points):

(a) The number of integers that are stored in eventCounts.dat.

(b) The number of key-value pairs that are stored in eventCounts.dat.

(c) The number of integers that are stored in words.dat.

(d) The number of key-values pairs that are stored in words.dat.

(e) The number of key-value pairs output by requestWordCounts.

(f) The number of key-value pairs read as input by answerWordCountRequests.

(g) The number of key-value pairs produced as output by answerWordCountRequests.

5. Answer the questions in the collaboration policy on page 1.

7 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
```