How to synthesize a text corpus using `corpusGenerator.exe`

Microsoft

July 17, 2017

This document describes how to use `corpusGenerator.exe` , a component of the `SynthaCorpus` project. Read `user_overview.pdf` and the first part of `developer_overview.pdf` first.

The mathematics and science behind synthesizing a corpus with specified properties is *described elsewhere*.

1 Running `corpusGenerator.exe`

To generate a synthetic corpus for any purpose, the form of the command is:

`./generate_a_corpus.exe <options>`

Use the `-file\_synth\_docs=<filepath>` option to choose the file into which the synthesized corpus will be written. The filepath should have either a `.tsv` or a `.starc` (Simple Text ARChive – see `user_overview.pdf`) suffix. The latter is preferred. If a TSV file is chosen, the output file is in two-column TSV format. The first column contains the text of each document and the second contains a popularity / importance score for the document. Currently `corpusGenerator.exe` always sets it to 1.

To see a list of all available options and brief descriptions of them, run `corpusGenerator.exe` with no arguments.

2 Specifying corpus properties

Each of the following sections describes an attribute of corpora and how you model that attribute using `corpusGenerator.exe` options.

2.1 Corpus size

From the point of view of indexing and query processing, the most appropriate measure of corpus size is the total number of indexable word occurrences. In the traditional IR model, each occurrence of an indexable word results in a posting in the index.

`-synth_postings=<integer>`

The number of postings actually generated will be identical, or very close, to the value supplied.

The `corpusGenerator.exe` system has been designed to generate very large corpora, up to say 100 billion postings. However, the practical limit is set by the amount of RAM on your system. For 100 billion postings you probably need a terabyte of RAM.

2.2 Vocabulary size

The number of distinct words in the corpus determines the size of vocabulary structures used in an indexer or query processor.

`-synth_vocab_size=<integer>`

This parameter is not independent of `-synth_postings`. Obviously the value of this parameter cannot be larger than that of `-synth_postings`.

Because term generation is a random process, the number of distinct words actually generated may only approximately match the value requested.

2.3 Word frequency distribution

The distribution of the number of occurrences of words in a corpus is often said to obey Zipf's law. This law states that, if words are ranked in descending order of occurrence frequency, then the expected frequency of a word is proportional to the rank raised to alpha, a small negative power.

`-zipf_alpha=<float>`

Typically, the value of alpha is around -1.0, however it should not be set at exactly -1.0 due to a property of the mathematics. If you specify this parameter in the absence of the other parameters described in this subsection, then generated occurrences will be modelled in pure Zipfian fashion.

We have observed that there are many real corpora which do not obey Zipf's law. Zipf's law is a continuous function while word occurrences are discrete. This incompatibility shows up in the tail of the distribution, where a high proportion of words occur only once. You can model the singletons separately, by specifying:

`-zipf_tail_perc=<float>`

where the value given is interpreted as a percentage of the vocabulary size. If you use this option, the appropriate proportion of singleton terms will be generated.

We have also observed that the observed frequencies of the few most frequent terms vary considerably from the expected values. For more accurate modeling you can specify the percentages of overall occurrences which are accounted for by the head terms:

`-head_term_percentages=<comma-separated-list-of-floats>`

e.g.

```
head_term_percentages=2.909794,0.963412,0.910098,0.787173,0.606225,
0.591738,0.520460,0.483734,0.481489,0.403257
```

Note that the percentages should be in descending order and should make sense.

Finally, some corpora are not well modeled by Zipf, even in the middle section between the head and the tail. To accommodate this, `corpusGenerator.exe` provides the capability to do piecewise linear modeling of the middle section.

`-zipf_middle_pieces=<%-separated-list-of-segment-descriptors>`

Each segment is described by 5 parameters:

1. Zipf alpha, specifying the slope of the segment
2. F, rank of the first word in the segment
3. L, rank of the last word in the segment
4. Probability range, proportion of total word occurrences covered by this segment.
5. Cumulative probability, the total probability of occurrence of all the words up to and including rank L.

e.g.

```
-zipf_middle_pieces=-0.5055,11,26,0.0522958906,0.1388696931%
-0.9215,27,62,0.0586029801,0.1974726731%
-0.6543,63,148,0.0723645609,0.2698372341%
-0.7860,149,352,0.0928606095,0.3626978436%
-0.7917,353,838,0.1110285329,0.4737263765%
-1.0073,839,1994,0.1232877705,0.5970141469%
-1.1445,1995,4743,0.1151458519,0.7121599988%
-1.2681,4744,11282,0.0958312508,0.8079912496%
-1.2678,11283,26839,0.0762421134,0.8842333630%
-0.7999,26840,63846,0.0678502872,0.9520836502%
```

2.4 Controlling what words are emitted

Internally, `corpusGenerator.exe` represents words by their rank in the "Zipfian" distribution. This subsection describes options controlling the unique sequence of letters emitted for each of those ranks. In some scenarios, we may be able to use the actual terms from the corpus being emulated, but that's not the case for private corpora, or for scaled-up corpora (because Heaps's law tells us that the vocabulary size grows with the size of the corpus. The option which allows us to specify the method for generating term representations is:

`-synth_term_repn_method=<method_name>`

The methods available are:

base26 The rank of the word is expressed in base 26, where the base 26 digits are 'a' to 'z'.

bubble_babble A representational method invented by Antti Huma to facilitate memorisation of large numbers such as cryptographic keys.

bodo_dave A heuristic method designed to be low cost and cache friendly.

from_tsv Use an existing vocabulary. Use `-synth\_input\_vocab=<file>` to specify the path to the file. The file should be in TSV format with the words in column one. The records should be sorted by descending frequency.

markov-k This method also uses an existing vocabulary whose path is given by `-synth_input_vocab=<file>`. `corpusGenerator.exe` learns an order-k Markov transition matrix for letters and uses it to generate the words. k is the number of characters of context used to determine a transition. If $k = 0$ the transition probabilities are just the raw corpus letter probabilities. If $k = 1$ then the probability of generating the next letter depends upon the preceding letter only. $(0 \leq k \leq 7)$

If implemented exactly as described above the Markov method is unable to generate any transitions which were not observed in the training corpus. This is undesirable because it is likely that many of the unobserved transitions would in fact have been observed if the process which generated the training corpus had been allowed to continue indefinitely. The `corpusGenerator.exe` option:

`-markov_lambda=<probability>`

is provided to smooth the observed transition matrix. Each time a letter is generated there is a probability that the next letter will be generated from the Markov-0 model, i.e. the basic letter probabilities. Smoothing in this way introduces the possibility that we will need to generate transitions from a context which was never observed. For example, in the case of $k = 4$, say we generate "ora" from the order-4 transition matrix, then we choose the next character according to Markov-0. That may give us a four letter context, e.g. "oraq", which has never been observed in the training corpus. To cover this case, the probabilities in empty rows of the order-4 transition matrix are set to the basic letter probabilities.

Because of this flow-on effect the proportion of letters in the synthetic corpus which are generated from the background letter probabilities will be higher than the value supplied for `markov_lambda`.

2.5 Modeling the distribution of document lengths

The `corpusGenerator.exe` system provides four different methods for modeling the distribution of document lengths in a corpus. Please do not supply parameters for more than one of the models. Note that it is the interaction between the chosen model and the overall number of postings which determines exactly how many documents are generated.

2.5.1 Document length model: normal

It is easy to calculate the mean and standard deviation of the document lengths in a real corpus. You can pass those observed values to `corpusGenerator.exe` using:

```
-synth_doc_length=<float>
-synth_doc_length_stdev=<float>
```

However, there are a few issues to keep in mind:

1. The distribution is inevitably left truncated. Lengths sampled from a normal distribution will inevitably generate zero or negative lengths. Rejecting these increases the mean length of the emulated corpus beyond the requested mean. Even if this effect can be compensated for,
2. Observation of many real corpora, shows that the normal distribution is not a good model for document lengths: The shape doesn't correspond and the upper tail is not heavy enough.

2.5.2 Document length model: gamma

Using a negative binomial distribution to model document lengths has an intuitive appeal. After each word is generated, there is a probability that the document will end. The repeated application of this probability results in a distribution of document lengths. Unfortunately, our observation of document lengths in a dozen different corpora suggests that modeling with a gamma distribution achieves greater accuracy. Accordingly, the following options are provided:

```
-synth_dl_gamma_shape=<float>
-synth_dl_gamma_scale=<float>
```

E.g.

```
-synth_dl_gamma_shape=4.5640 -synth_dl_gamma_scale=0.5789
```

2.5.3 Document length model: piecewise

The gamma distribution is unimodal, which means that it is unable to model bimodal or multimodal distributions. It was hoped that a piecewise linear approach might work here too, but it turns out to be difficult to avoid either an excessive number of segments or inaccurate fitting.

```
-synth_dl_segments=<length_segment_list>
```

The length segment list must comprise the number of `<length_segment>`s followed by a colon and by the specified number of `<length_segment>`s, separated by semicolons. A `<length_segment>` consists of a comma separated pair of numbers. E.g.

```
-synth_dl_segments=4:1,0.333333;10,0.500000;200,0.6666667;5000,1.000000
```

2.5.4 Document length model: scaled histogram

Text corpora are often mixtures of documents of different types, each with their own length distribution. This may result in a distribution with multiple maxima. Some documents in a corpus may be system generated (e.g system generated emails) to a fixed length, possibly resulting in a spike in the document length distribution. The most accurate way to emulate the length distribution for a corpus is to use the length histogram of the base corpus and scale it to match the number of documents required in the emulated corpus. `corpusGenerator.exe` provides the following to achieve this:

```
-synth_dl_read_histo=<filepath>
```

where the argument is the path of a histogram file in TSV format with length in column one, and frequency in column two. Such a file may be created by `corpusPropertyExtractor.exe`.

3 Checklist for manually requesting a synthetic corpus

Do the following:

1. Specify corpus size.
2. Specify vocabulary size.
3. Specify the term frequency distribution.
4. Choose a term representation method and specify the appropriate options.
5. Choose a model of document length and specify appropriate parameters.
6. Run `corpusGenerator.exe`

Please ignore line breaks in the following example. They are inserted only for legibility reasons.

```
./generate_a_corpus.exe
-synth_postings=380977156
-synth_vocab_size=6415147
-zipf_tail_perc=59.141497
-head_term_percentages=1.416744,1.265595,1.092416,0.909492,
0.899532,0.854723,0.633177,0.611741,0.594657,0.466848
-file_synth_docs=QBASH.forward
-zipf_middle_pieces=-0.9883,11,38,0.0585174928,0.1459667440%
-0.5406,39,131,0.0805264398,0.2264931838%
-0.6663,132,451,0.1315660958,0.3580592795%
-0.8667,452,1557,0.1792479075,0.5373071870%
-1.0788,1558,5370,0.1839448426,0.7212520296%
-1.4283,5371,18520,0.1373040120,0.8585560416%
-1.6676,18521,63879,0.0690803493,0.9276363909%
-1.5528,63880,220328,0.0310007879,0.9586371788%
-1.3264,220329,759940,0.0175953201,0.9762324988%
-0.8873,759941,2621133,0.0138088621,0.9900413609%
-synth_dl_read_histo=..../QBASH.doclenhist
-synth_term_repn_method=markov-5e
-synth_input_vocab=..../vocab.tsv
-markov_lambda=0.0001
```

4 Overview of `corpusGenerator.exe` internals and their memory requiements

`corpusGenerator.exe` 's first step is to process the parameter list. Currently, it is the user's responsibility to ensure the consistency of the parameter values. Maybe, when time permits, we will add logic to reject problematic combinations.

Next, a table of term representations is created. The length of the table corresponds to the vocabulary size. This table must remain allocated until the corpus has been generated. If the maximum term length is 15 bytes, then 16 bytes are required per table entry and the size of a table for a vocabulary size of 100 million entries would be about **1.6GB**. Depending upon the method selected for generating term representations, large amounts of memory may be needed temporarily during term generation. Markov methods with large k are particularly demanding, as they require a large transition matrix (currently stored in dense form) and a large hashtable. The latter is required to ensure that the same word is not generated more than once. Total space required by `markov-5` with a vocabulary of 100 million is approximately **5.0GB (temporary)**.

Step three is to generate a document table from a histogram of document lengths into a linear array of the lengths. Document table entries include a 40 bit offset within the entire corpus and a 24-bit word

count. The length histogram is generated for normal, gamma and piecewise methods, and otherwise read in and (possibly) scaled. The document table is shuffled in place to remove the inherent sorting by length. Entries include the offset within the entire corpus If the target corpus comprises a billion documents then the shuffled document table will require about **8.0GB**. This memory is required until all the terms have been generated and shuffled.

Since April 2017 the algorithm for allocating terms (both compound and simple) to positions in documents has been as follows. Note that the only compound terms supported at this stage are n-grams.

Step four is to allocate a term occurrence array with an integer element (word rank) for every word occurrence in the corpus. If there are 100 billion such occurrences, then the amount of memory needed for this will be approximately **400GB**.

In Step five the compound terms are assigned to randomly chosen documents, reducing the word count in the document table entry, placing the termids in the term occurrence array, and reducing occurrence frequency of the component words. For example, if the 3-gram "geneva motor show" occurs 500 times, we randomly assign the 3-grams to documents, including flags to indicate that they should be kept intact, reduce the word count of each chosen document by three, and subtract 500 from the occurrence frequencies of "geneva", "motor" and "show".

Once compound terms are dealt with, in Step six, another large array, **400GB** for 100 billion term occurrences, is filled with word ranks, as per the word frequency distribution. This array is shuffled in place. and then scanned, assigning each word rank encountered to a document, and entering it in the appropriate position in the term occurrences array. The space required by this additional array is regrettable but it was introduced to avoid short documents being filled with stop words. Twenty occurrences of the equivalent of "the" is unremarkable in a document of 1000 words, but is ridiculous in a document of length 20.

Term assignment for both words and compounds proceeds as follows: An element of the document table corresponding to a non-full document is chosen at random. If that document has insufficient room, another random choice is made. To avoid the need for retries when allocating words to documents, a pointer is retained to the highest numbered doctable entry which is not yet full. When a document fills, it is swapped with the highest non-full one and the pointer is adjusted. Random selections are made among the non-full documents only. All the full ones sit at the top of the table. Note that compound terms are assigned first to reduce the chance of failed allocations of terms into documents with few unallocated postions.

Step seven uses the document table to mark the last word occurrence of each document with a flag. Once this is done the memory used by the document lengths array can be freed.

In Step eight, each document is independently shuffled, taking care not to break up or reorder Ngrams.

Finally, in Step nine the array of term occurrences with flagged document ends is used in combination with the term representations table to write the output file.

4.1 Reducing memory requirements?

As may be seen from the above, the memory requirements for generating a corpus with tens of billions of postings are considerable. The array of term occurrences normally dominates memory demand. The current `corpusGenerator.exe` algorithm uses a Knuth shuffle which generates random accesses within that array, meaning that that array should be fully resident. A significant reduction in memory requirements would require substantial changes to algorithims.

Note however, that the hypothetical corpus discussed above is a very large one (100 billion word occurrences). A corpus of 10 billion postings should be reasonably easily generated on a server with 128GB of RAM.