Re-write of Text-NSP

Bjoern Wilmsmann
Ruhr-University, Bochum, Germany
Department of Linguistics

February 12, 2007

Contents

Abstract

Text-NSP (Text Ngram Statistics Package, current version: 1.03) is a Perl module for tokenization and extraction of n-grams (string entities of length n, e.g. character or word sequences) from a corpus.

Furthermore, this module provides several means of statistical analysis of n-gram occurrences and comparing n-gram patterns of different corpora.

The aim of this work is to find out what can be done in order to re-implement the functionality of Text-NSP in a more efficient manner, as well as to re-design the code in a conceptually more elegant way.

keywords:

- Text-NSP
- n-gram processing
- language model
- document model

1 Introduction

Text-NSP (Text Ngram Statistics Package, current version: 1.03) is a Perl module whose purpose mainly consists of tokenization and subsequent extraction of n-grams (string entities of length n, e.g. character or word sequences) from a corpus. Moreover, this module provides several means of statistical analysis of n-gram occurrences and comparing n-gram patterns of different corpora. While the module offers highly useful functionality, the purpose of this work is to evaluate if there is a way of improving the efficiency of the module, especially when dealing with large corpora (200 MB and larger). Moreover, some functionality seems to be distributed among several code locations, therefore making the code less reusable, especially when it comes to using the functionality Text-NSP provides in custom-made programs.

Hence, the aim of this work is trying to re-implement the functionality of Text-NSP, where it seems appropriate, in a more efficient manner, as well as to re-design the code in a conceptually more elegant way. In order to achieve this, I shall dissect the respective components of the module and examine if there is space for both speed and architectural improvements or possible additional features. I shall mainly address the components count.pl, statistic.pl and huge-count.pl (and huge-combine.pl, sort-bigrams.pl and split-data.pl for that matter, as huge-count.pl depends on these), as these contain the core features of Text-NSP (the other scripts, namely combig.pl, kocos.pl and rank.pl, provide supplementary features).

2 Validation

Every output of the new Text-NSP has been checked against the output of an equivalent call on the old Text-NSP by comparing the files using the UNIX command line tool 'diff' in order to make sure that the new program is functionally equivalent to the original software. Moreover, the 'make test' command is used in order to test the correctness of each measure of statistical independence.

3 General Changes

There have been general changes which apply to every re-implemented code section of Text-NSP:

- The first line of each script file now is 'usr/bin/env perl' instead of '/usr/local/bin/perl -w', as this leaves the decision about where to look for the Perl binary to the operating system.
- Source and destination files cannot be supplied implicitly any more, but have to be marked as such by using the '–source' and '–destination' keywords.
- The functionality which is used by several script files or components has been moved from these components to the newly created Support-FunctionLibrary class. For a detailed description of this class (and the other fundamental classes of the new Text-NSP), please have a look at Text-NSP-Class-Diagram.png or Text-NSP-Class-Diagram.pdf in the Docs folder of the Text-NSP archive.
- A new class OptionReader has been created for providing a general reusable interface to command line arguments.
- All functionality related to tokenization is now located in the class Tokenizer in order to be capable of providing this functionality to several scripts (e.g. count.pl and huge.count.pl).
- Whereever deemed appropriate functionality has been moved to classes in order to rearrange the code in a more structured fashion.
- The code has been cleaned up and simplified whereever this has been possible, hopefully making it more legible and comprehensible. Though ometimes which code is legible and which one is not turns out to be a matter of taste, I hope that I did not only meet my personal taste in this matter.

4 count.pl

4.1 What does count.pl do?

count.pl takes a corpus file a directory containing corpus files, URLs and/or a text file containing URLs as input. This input is tokenized by using compiled regular expression for tokens, non-tokens and stop words. Afterwards n-grams of maximum length n and minimum length 1 are counted. The resulting n-gram language model is stored in a file along with some metadata.

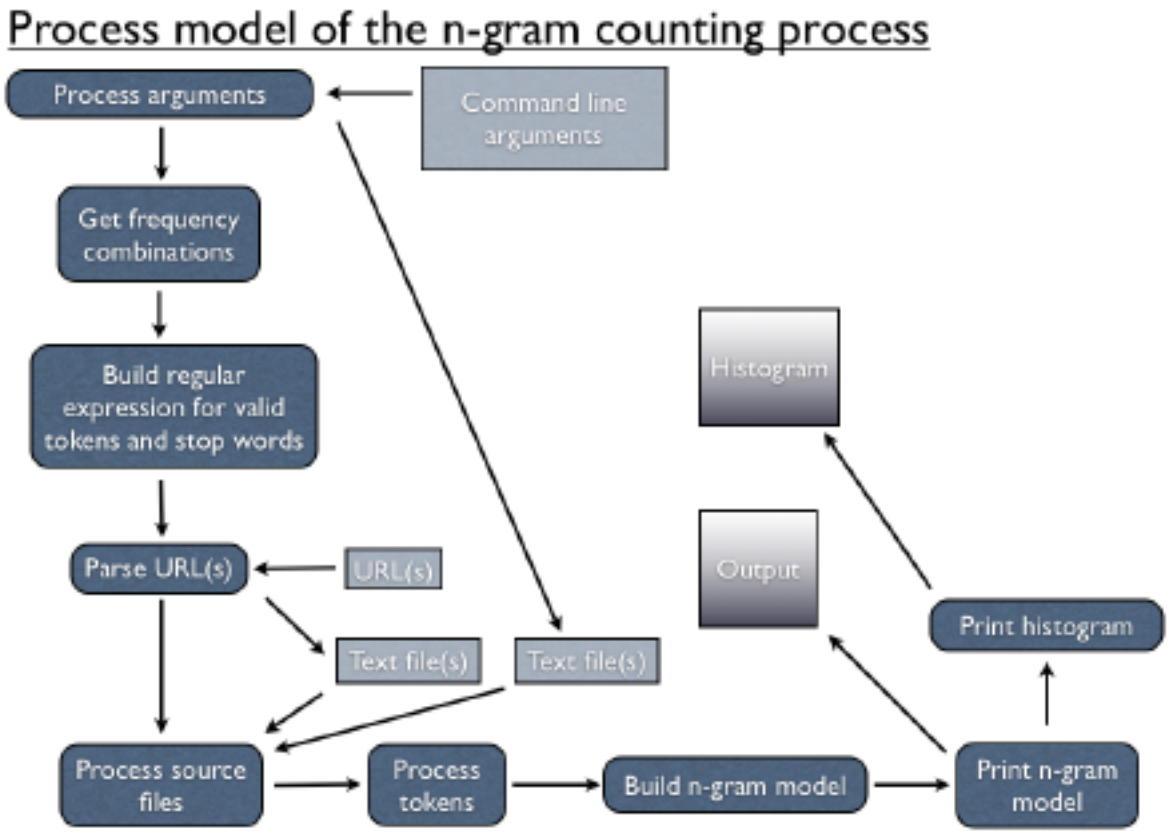
Optionally count.pl considers only user-defined frequency combinations. Moreover count.pl can count n-grams in a window of size m (e.g. a trigram of three tokens can be found in a window of 5, no matter if there are other words in between these tokens).

Finally, count.pl optionally creates a histogram, which contains the frequencies of token frequencies (e.g. 4 tokens with frequency 32, 5 tokens with frequency 43 and so on).

The algorithm for processing n-grams > 1 works like this:

- Token combinations are created for window size - 1 and n-gram size - 1 as additional combinations can be derived from these values (e.g. 0 1 is actually the same template as 1 2 for n = 2 and consequently is applied to an n-gram anyway once window is shifted).
- The window starts at minimum n-gram size and grows until it has reached maximum window size.
- Get token combinations.
- Process combinations in n-gram-sized packages.
- If the current token combination value does not exceed limits of the current window, add this token to the n-gram, otherwise mark this combination as not applicable for the current window size.
- Add the final token of the current window to n-gram, this allows us to have all possible token combinations for the actual window and n-gram size although the combination matrix has been calculated for window size - 1 and n-gram size - 1. This is done for both speed reasons and for avoiding double counts (e.g., combination 1 2 is essentially the same as 0 1 for window = 3 and n-gram = 2, because it will be reached anyway once the window is shifted).
- Calculate frequencies.

4.2 Model



4.3 Improvements

- All functions not specific to this script have been moved to separate classes.
- The function getRegularExpressionTokens of SupportFunctionLibrary is now used for both tokens and non-tokens.
- As mentioned above the tokenizing functionality has been moved to a separate class.
- Frequency combinations are now calculated using the binomial coefficient $\binom{n}{k}$ with n being the n-gram size and k each possible size of an internal n-gram (e.g. 1, 2 and 3 for trigrams).
- N-gram token combinations are also calculated using a binomial coefficient $\binom{n-1}{k-1}$ with n being the window size and k denoting the n-gram

size (see the algorithem described above for a detailed explanation of why n - 1 and k - 1 and not n and k have been used as values).

- Now URLs can be used as a text source, both as single URLs, as well as several URLs via the '–urlfile' option (requires optional LWP packages)
- Now XML/HTML documents are parsed as well (requires HTML::Parser or XML::Parser)
- In order to deal with HTML documents properly, HTML entities are transformed into their ASCII counterparts (requires HTML::Entities)
- The complete functionality of count.pl has been moved to a separate class named Counter for use in huge-count.pl
- Text-NSP now uses locale in count.pl, Tokenizer and Counter for localised tokenization (umlauts, accents etc.).

4.4 Other results

4.4.1 Efficiency of the N-Gram Counting Algorithm

I have tried out several algorithms for counting n-grams within a certain window range so far. However, I have come to the conclusion that the algorithm originally used in Text-NSP is in fact the most effective one, albeit not having been formulated straightforwardly in the original version. This algorithm does not use all possible combinations for $\binom{n}{k}$, as this would include n-grams as well which will be counted once the window is shifted to the right anyway (1, 2 of iteration 1 is essentially the same as 0, 1 of iteration 2 for n = 3 and k = 2). The processing of all possible token sequences of a fixed length k in an iteratively shiftable window of length n needs at most $\binom{n-1}{k-1}$ * floor($\frac{i}{n}$) + i % n iterations with i being the total number of tokens in the input string. The value $\binom{n}{k}$ is equivalent to the number of all possible combinations of length k in a window of length n.

However, under the assumption that the window can be moved only $\binom{n-1}{k-1}$ of these will be needed since all combinations whose value at index 0 is != 0 merely are repetitions of an already processed structure with the values just having been incremented by 1. Hence, these structures will be processed in later iterations anyway. They even must not be processed at all as the same token sequence would be included in the results twice.

Therefore, the algorithm for instance processes the 0, 1 combination for $\binom{3}{2}$ only during the first iteration, that is, when the window still is of size 2. For the later iterations only the combinations 0, 2 and 1, 2 are used because these are completely sufficient for modelling all possible n-grams.

4.4.2 Another Flavour of the N-Gram Counting Algorithm

Apart from these considerations, I have implemented my own variant of the n-gram counting algorithm described above and added it as an experimental option. The differences are as follows:

First of all, all possible windows of size k are written to a hash with the content of a window as the key and the number of occurrences of this window in the corpus as the corresponding value.

Afterwards, the algorithm iterates over the windows just as above. However, it does not increment the value of n-gram frequencies by one, but adds the number of occurrences of the window at hand to the frequency of the n-gram that is currently being dealt with.

In other words, this flavour of the algorithm treats the window contents of a corpus as belonging to equivalence classes, which means that the results which have been calculated for a specific window are the same for all windows of its class and hence can simply be multiplied by the cardinality of the equivalence class.

This variant needs O(n + m * l) time, where n is the number of window tokens in a corpus, m the number of window types and l the number of possible combinations for a window. Considering that the standard flavour of this algorithm takes O(n * l) time, the experimental flavour should perform better, if the number of possible combinations is high and the number of window types in the given corpus is low. The latter should usually be the case with larger corpora.

The algorithm can invoked by calling the count.pl script with the '–algorithm 2' option.

4.4.3 Evaluation of Alternatives to Storing N-Grams as Hashes

- Suffix trees / arrays have been considered as an alternative for storing n-grams. However, the major performance issue with Text-NSP regarding large corpora is not inefficient n-gram counting, but an exponentially increasing memory usage caused by calculation of various n-gram combinations in all available windows. Suffix trees / arrays do not appear to be suitable for increasing the processing speed of counting algorithm, since they improve memory usage by a constant factor at best, if they do not even consume more memory. The current storage method for n-grams occupies n(N - (n - 1)) = O(n * N)) space, while the suffix tree flavour uses O(N|Alphabet|) space. Finally, the suffix array approach consumes O(N) space. A slightly lower memory consumption of n(N - (n - 1)) - N = O(n * N - N) would therefore be traded in for a considerable overhead for building, sorting and searching (counting n-grams takes O(2N - 1) time) the initial suffix tree / array structure.

One has to keep in mind as well that Perl hashes are only dealt with like hashes on the surface, however, their actual implementation is far more efficient than simple hash structures.

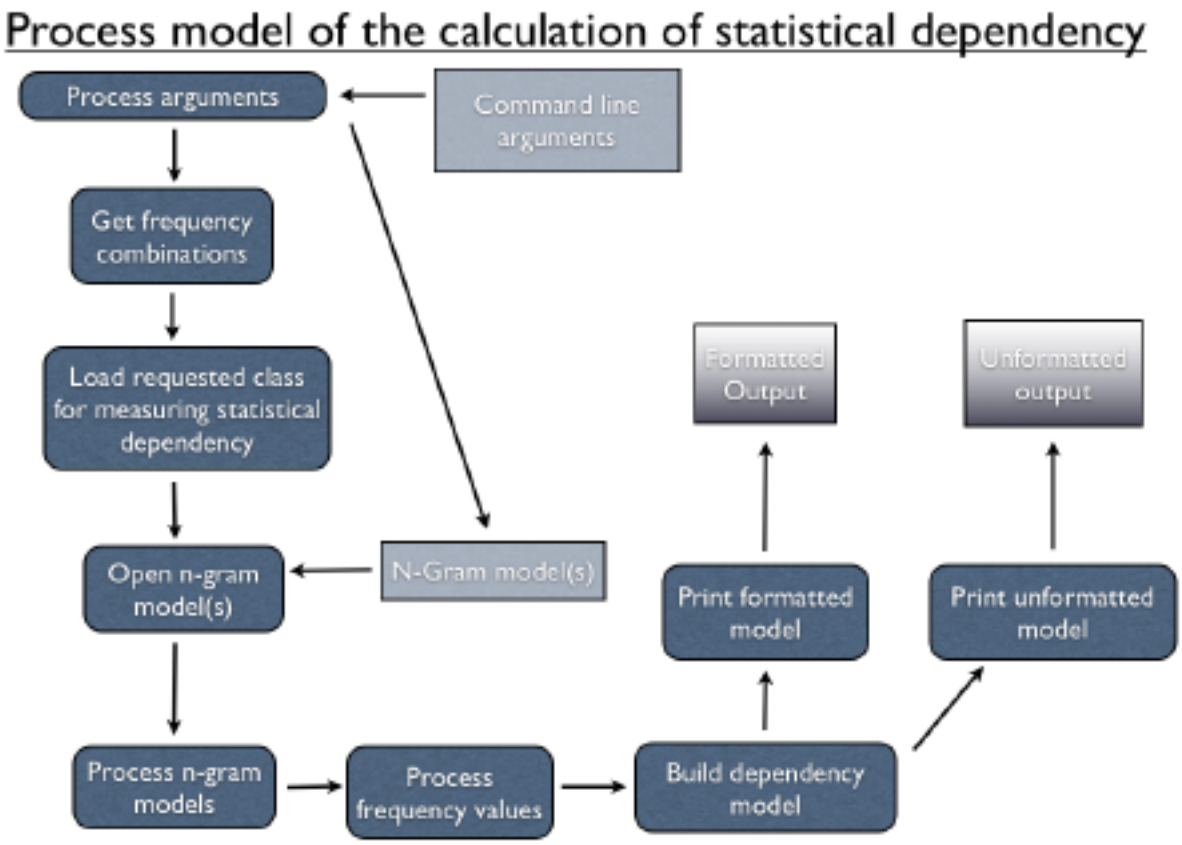
- Another alternative might be Tries: Trie structures would be faster in terms of n-gram retrieval if the average length of an n-gram string is less than the average number of re-hashes needed for storing n-gram in a hash structure.

5 statistic.pl

5.1 What does statistic.pl do?

statistic.pl takes the results of count.pl, that is an n-gram model of a given corpus, as an input and accordingly computes statistical dependence / indepence between terms using the values these terms expose for each of their frequency combinations in the n-gram model. For this purpose a variety of diverse statistical measures for computing dependence both between bigrams and trigrams is available. The basic assumption behind this process is positing a null hypothesis between components of n-grams, that is to say, for instance, if the probability P(w1, w2) for a bigram 'w1 w2' is not higher than the product of the probabilities P(w1)P(w2) for each single component of the bigram, the null hypothesis holds and the terms can be considered to be statistically independent.

5.2 Model



5.3 Improvements

- The 'load frequency combinations from file' option has been removed, as this is pointless if you have to supply the default options anyway in order not to run into an error.
- The error handling when selecting a statistical measure has been streamlined, since now variables are used to make code less repetitive. Moreover, some unnecessary error handling regarding frequency combinations in the main program part has been removed.
- The function isInteger has been simplified and, as well as other functions, moved to the SupportFunctionLibrary.
- From now on only one function will be used for both formatted and unformatted printing.
- The frequency combination index calculation has been re-implemented in a more efficient and straightforward manner.

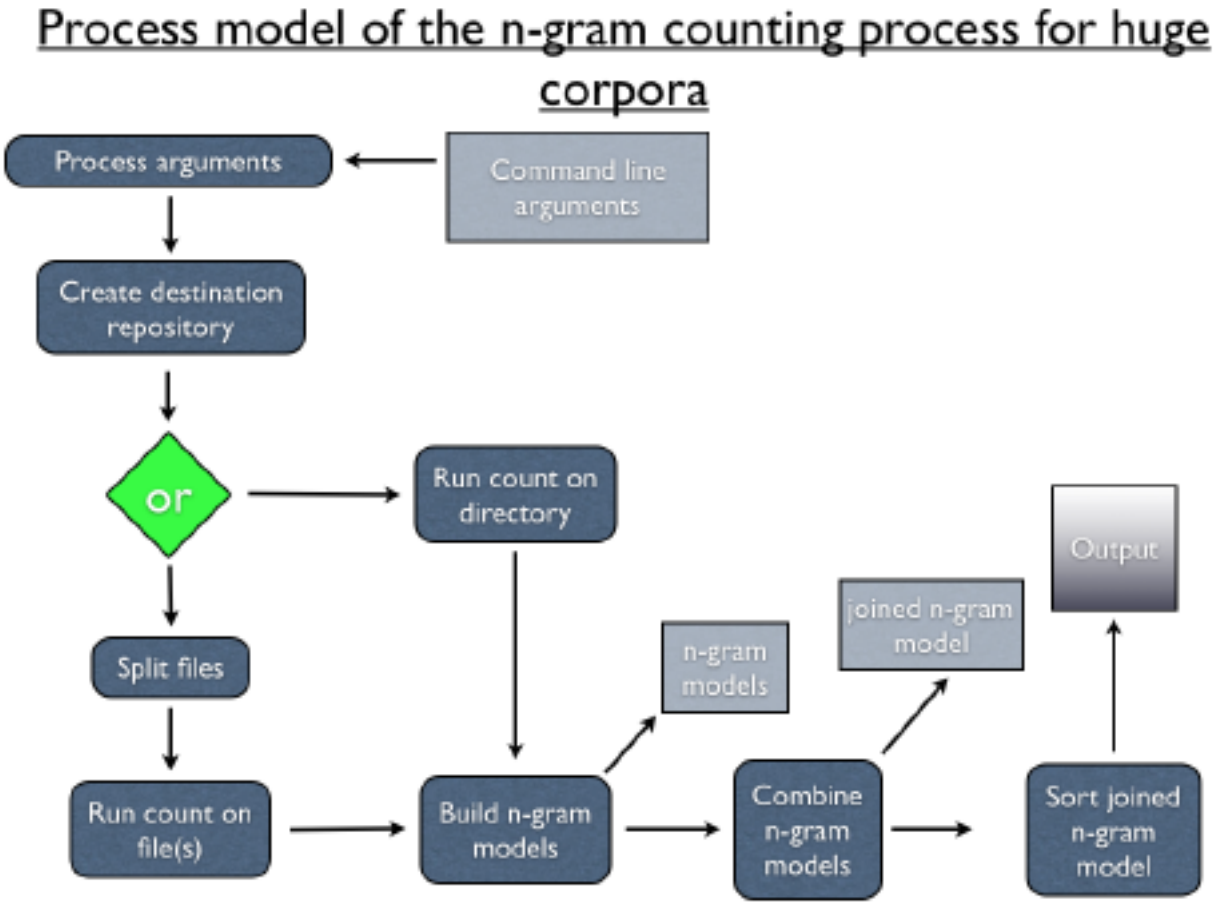
6 huge-count.pl

6.1 What does huge-count.pl do?

huge-count.pl basically delivers the same results as count.pl. However it was created for dealing with huge corpora by dividing the input into several smaller portions which in turn should be more tractable than the original file. For this purpose huge-count.pl makes use of a few helper scripts, namely split-data.pl, count.pl and huge-combine.pl.

Please note that there is a slight variation in the results of huge-count.pl when compare to those of count.pl, if the –newline option is not used. This is caused by the split file being dealt with as completely separate sources during the counting process, which means that the last tokens of one chunk will not be combined with the first tokens of the following chunk as is the case with count.pl, where we simply have to deal with one line following another.

6.2 Model



6.3 Improvements

- system() calls have been replaced to ensure platform independence (before, non-UNIX systems would not run this script).
- The function calling count.pl has been moved to SupportFunctionLibrary and now is called runMultipleCount().
- The code has been simplified, both in huge-count.pl and in the new function runMultipleCount(), since enormous control structures for different variants of calling count.pl are not needed anymore in the object-oriented model.
- The Tokenizer class now is used for the combining process.
- N-gram sorting is not needed anymore, as this is already done by Tokenizer.

7 sort-ngrams.pl

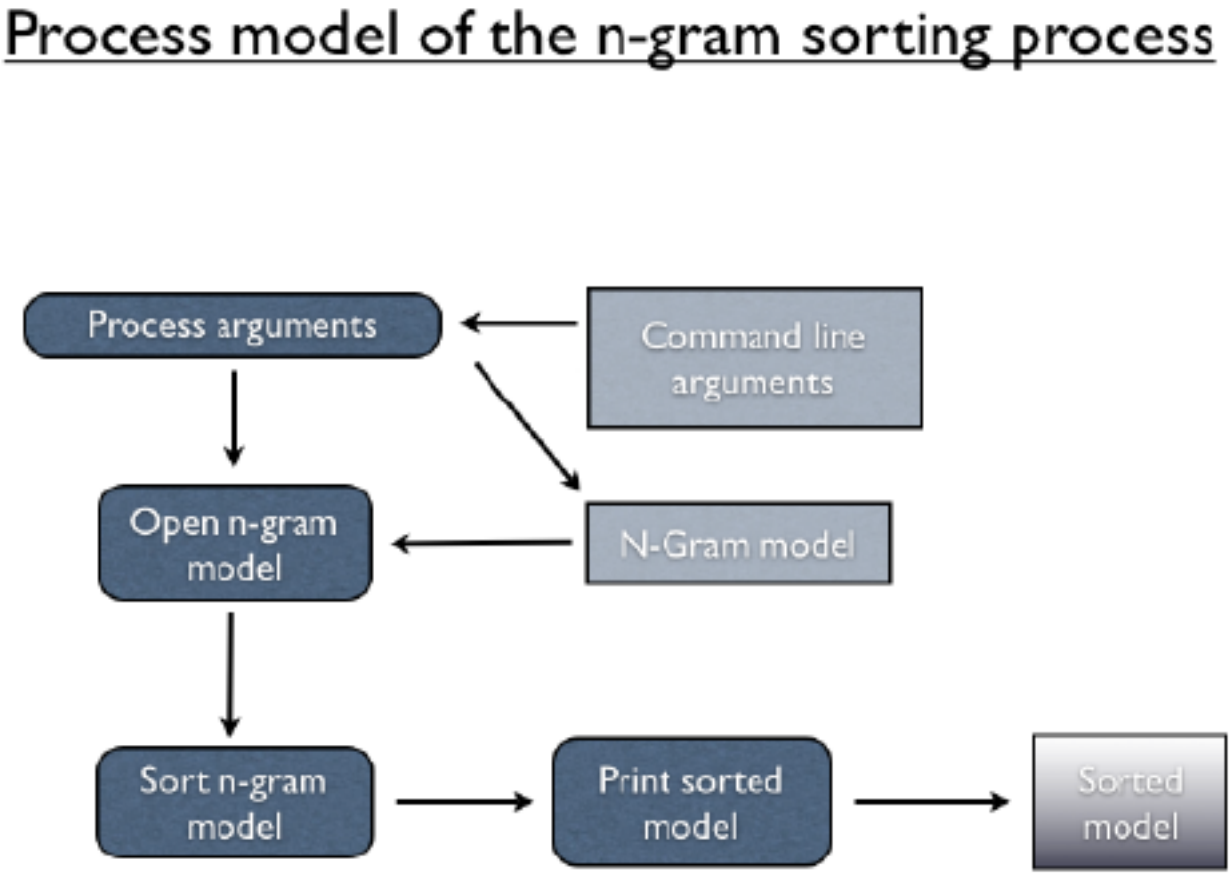
7.1 What does sort-ngrams.pl do?

The original sort-bigrams.pl simply took an unsorted bigram model in the usual TextNSP format as used by count.pl and statistic.pl (the ones created by count.pl and statistic.pl are already sorted, this script basically is a helper script for processing the files for example created by the additional script combig.pl, which prints out the models in no particular ordering). The new sort-ngrams.pl by and large does the same, however not only for bigrams but for any kind of n-gram, no matter which value n assumes.

The functionality of this script has become obsolete during the re-implementation process, since n-gram sorting now is done implicitly by the Tokenizer class.

However, I decided to include this script in the distribution as well, because of the results given by combig.pl. Furthemore, it might turn out to be quite useful for other applications, too.

7.2 Model



7.3 Improvements

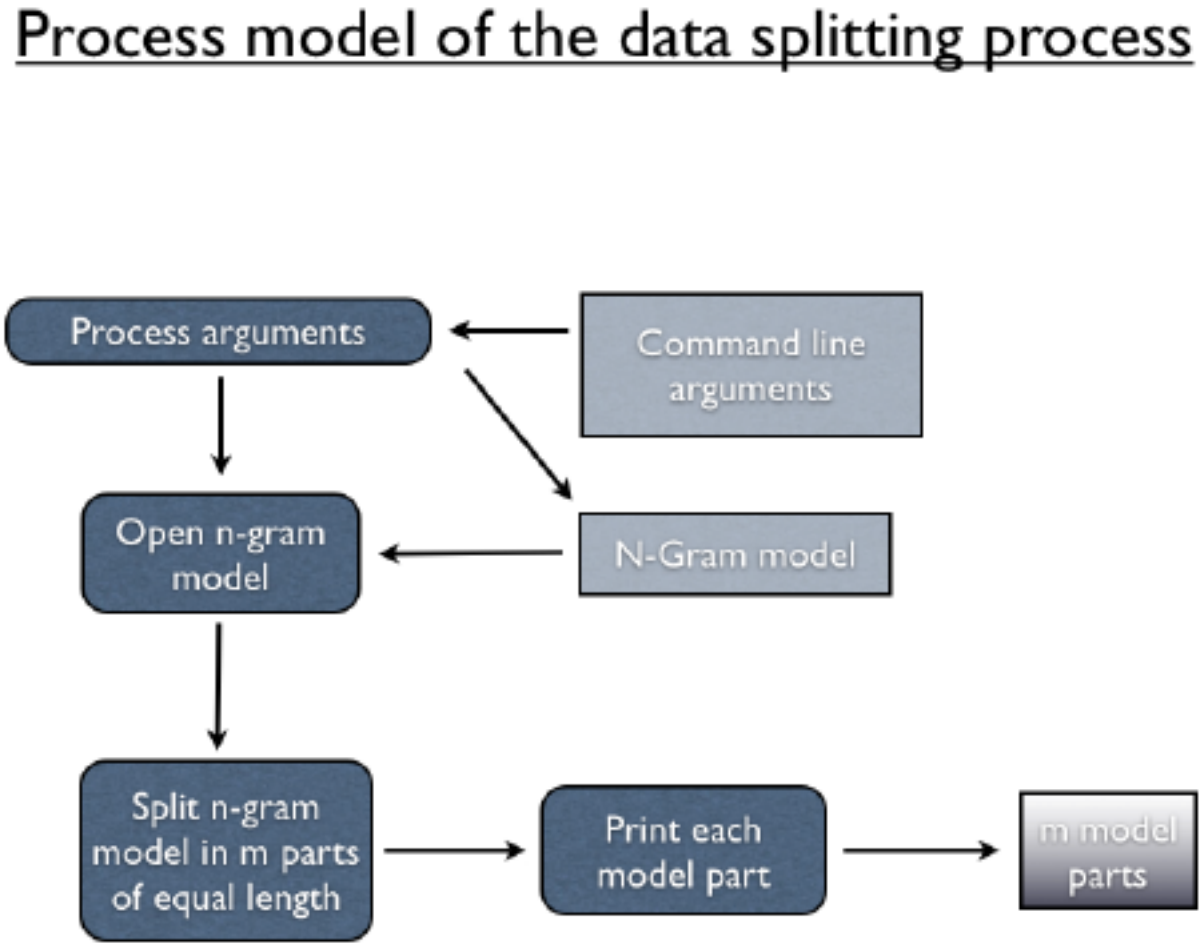
- The functionality has been moved to the NgramSorter class
- The functionality has been implemented for n-grams with n larger than 2, too.
- The algorithm and script structure have been simplified, now only lines will be re-arranged for sorting, no juggling around a whole bunch of values any more.

8 split-data.pl

8.1 What does split-data.pl do?

split-data.pl takes a monolithic file and splits it into several files of equal length for more efficient processing in terms of the huge-count.pl programm part.

8.2 Model



8.3 Improvements

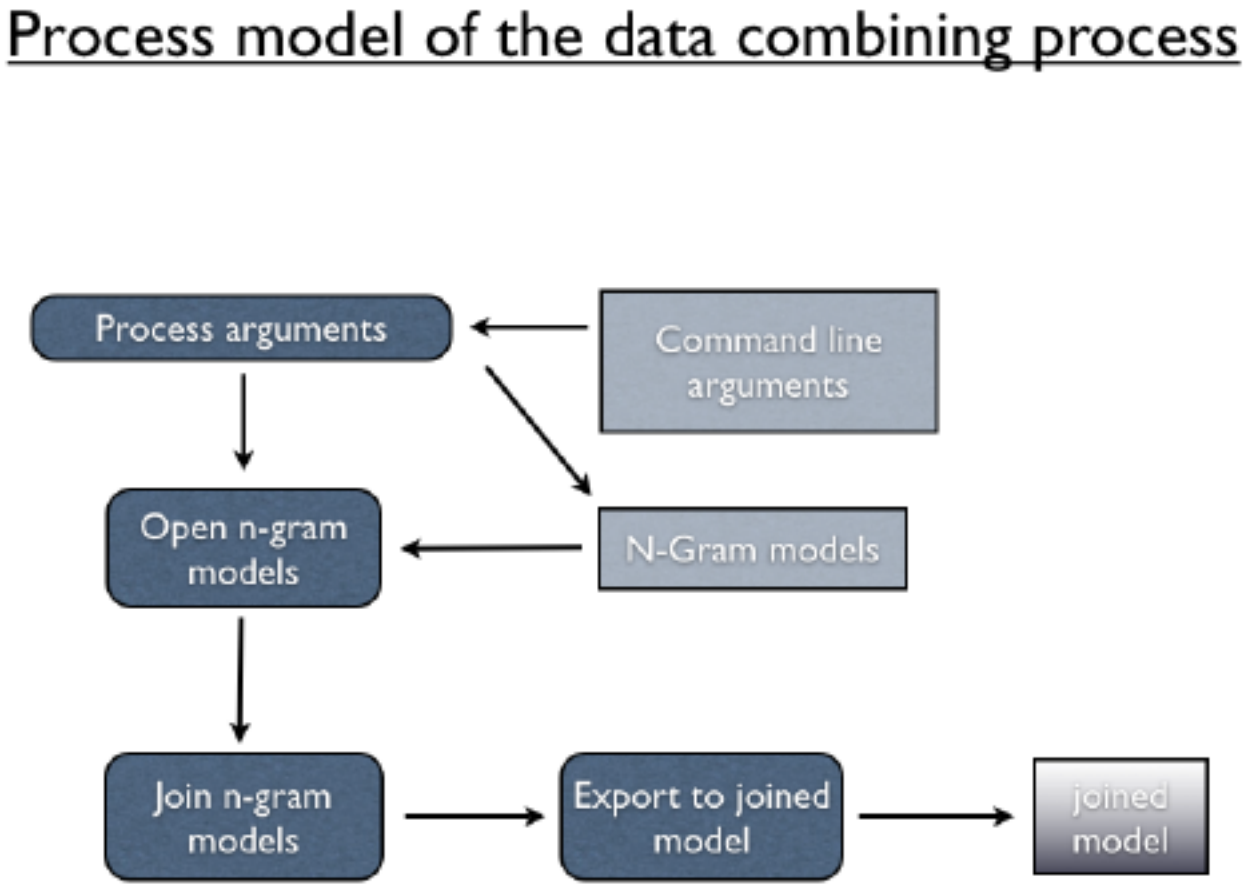
- The functionality has been moved to the DataSplitter class

9 huge-combine.pl

9.1 What does huge-combine.pl do?

huge-combine.pl takes several n-gram model files as created by count.pl and merges them into one model. The algorithm used before was limited to bigrams. The current algorithm is more flexible, as it can process n-grams of any length.

9.2 Model



9.3 Improvements

- The functionality has been moved to the Combiner class
- huge-combine.pl now allows for other n-grams than just bigrams.
- The combining algorithm has been simplified, it now works similar to n-gram sorting, as only the relevant components will be re-calculated.
- The Tokenizer class is now used during the combining process.

10 Time::StopWatch

StopWatch.pm is a Perl module developed by me for basic measuring of time elapsed during a given process or algorithm. Its functionality does not inherently belong to Text-NSP. However, it is used for generating some

debugging information during execution of count.pl, which is why I have included it in this package as well.

11 Class Diagrams

- http://www.topicalizer.com/files/TextNSP/models/Text-NSP-Class-Diagram.png
- http://www.topicalizer.com/files/TextNSP/models/Measurements-Class-Diagram.png

12 Downloads

Current version:
http://www.topicalizer.com/files/TextNSP/Text-NSP-1.10.tar.gz

Old versions:

- http://www.topicalizer.com/files/TextNSP/Text-NSP-1.04.tar.gz
- http://www.topicalizer.com/files/TextNSP/Text-NSP-1.05.tar.gz
- http://www.topicalizer.com/files/TextNSP/Text-NSP-1.06.tar.gz
- http://www.topicalizer.com/files/TextNSP/Text-NSP-1.07.tar.gz
- http://www.topicalizer.com/files/TextNSP/Text-NSP-1.08.tar.gz
- http://www.topicalizer.com/files/TextNSP/Text-NSP-1.09.tar.gz

13 Resources for Text-NSP

- http://ngram.sourceforge.net/
- http://search.cpan.org/dist/Text-NSP/Docs/FAQ.pod
- http://www.d.umn.edu/~tpederse/Pubs/aaai96-cmpl.pdf
- http://www.d.umn.edu/~tpederse/Pubs/cicling2003-2.pdf
- http://www.d.umn.edu/~tpederse/Pubs/scsug96.pdf

14 Resources for Optimising Perl Code

- http://www.ccl4.org/~nick/P/Fast_Enough/
- http://www.perl.com/pub/a/2005/12/21/a_timely_start.html
- http://www.perl.org/tpc/1998/Perl_Language_and_Modules/Efficient%20Perl/handout.html

15 Suffix Tree Resources

- http://acl.ldc.upenn.edu/J/J01/J01-1001.pdf
- http://www.milab.is.tsukuba.ac.jp/~myama/tfdf/index.html
- http://search.cpan.org/~gray/Tree-Suffix-0.14/lib/Tree/Suffix.pm

16 Tries in Perl

- http://search.cpan.org/~hammond/data-trie-0.01/Trie.pm
- http://search.cpan.org/~avif/Tree-Trie-1.2/Trie.pm