The CMU-Cambridge Statistical Language Modeling Toolkit v2

Contents

If you want to get started making language models as quickly as possible, you should install the toolkit and then read the Typical Use section.

Introduction

Version 1 of the Carnegie Mellon University Statistical Language Modeling toolkit was written by Roni Rosenfeld, and released in 1994. It is available by ftp from here. Here is a excerpt from its README file:

```
                Overview of the CMU SLM Toolkit, Rev 1.0
                ========================================

  The Carnegie Mellon Statistical Language Modeling (CMU SLM) Toolkit
is a set of unix software tools designed to facilitate language
modeling work in the research community.

  Some of the tools are used to process general textual data into:
    - word frequency lists and vocabularies
    - word bigram and trigram counts
    - vocabulary-specific word bigram and trigram counts
    - bigram- and trigram-related statistics
    - various Backoff bigram and trigram language models

  Others use the resulted language models to compute:
    - perplexity
    - Out-Of-Vocabulary (OOV) rate
    - bigram- and trigram-hit ratios
    - distribution of Backoff cases
    - annotation of test data with language scores
```

Version 2 of the toolkit seeks to maintain the structure of version 1, to include all (or very nearly all) of the functionality of version 1, and to provide useful improvements in terms of functionality and efficiency. The key differences between this version and version 1 are described in the next section.

Changes from Version 1

Efficient pre-processing tools

The tools used to generate vocabulariesm, and to process the text stream which is used as training data into a id n-gram file to serve as input to idngram2lm have been completely re-written, in order to increase their efficiency.

All of the tools have been written in C, so there is no longer the reliance on shell scripts and UNIX tools such as sort and awk. The tools now run much faster, due to requiring much less disk I/O, although they do now require more RAM than the tools of version 1.

Multiple discounting strategies

Version 1 of the toolkit allowed only Good-Turing discounting to be used in the construction of the models. Version 2 allows any of the following discounting strategies:

- Good Turing discounting
- Witten Bell discounting
- Absolute discounting
- Linear discounting

Use of n-grams with arbitrary n

The tools in the toolkit are no longer limited to the construction and testing of bigram and trigram language models. As larger corpora, and faster machines with more memory become available, it is becoming more interesting to examine 4-grams, 5-grams, etc. The tools in version 2 of this toolkit enable these models to be constructed and evaluated.

Interactive language model evaluation

The program evallm is used to test the language models produced by the toolkit. Commands to this program are read in from the standard input after the language model has been read, so the user can issue commands interactively, rather than simply from the shell command line. This means that if the user wants to calculate the perplexity of a particular language model with respect to several different texts, the language model only needs to be read once.

Evaluation of ARPA format language models

Version 2 of the toolkit includes the ability to calculate perplexities of ARPA format language models.

Handling of context cues

In version 1, the tags <s>, <p>, and <art> were all hard-wired to represent context cues, and the tag <s> was required to be in the vocabulary. In version 2, one may have any number of context cues (or none at all), and they may be represented by any symbols one chooses. The context cues are a subset of the vocabulary, and are specified in a context cue file.

In order to produce the same behaviour from version 2 as from version 1, the context cues file should contain the following lines:

```
<s>
<p>
<art>
```

Compact data storage

The data structures used to store the n-grams are more compact than those of version 1, with the result that language models construction is a less memory intensive task. For example, for a trigram language model, version 1 required 12 bytes per bigram and 4 bytes per trigram. Version 2 requires only 8 bytes per bigram and 4 bytes per trigram.

Support for `gzip` compression

As well as the `compress` data compression utility used in version 1 of the toolkit, there is now also support for `gzip`.

Confidence interval capping

Confidence interval capping has been omitted from version 2 of the toolkit.

Forced back-off

The tool used for evaluating language models allows the user to specify a set of *forced back-off* parameters. There may be items in the vocabulary (especially context cues and the "unknown" symbol) from which we may want to back-off all the time. For example, if we see the word string A <s> B (where <s> is a context cue indicating a sentence boundary), then instead of predicting the probability of `B` based on the full context (P(B | A <s>)), we may wish to disregard the information before the sentence boundary. Therefore we might want to back-off to the bigram distribution P(B | <s>) (*inclusive* forced back-off) or even to the unigram distribution P(B) (*exclusive* forced back-off). Version 2 supports both types of forced back-off for arbitrary vocabulary items.

The `evallm` program allows the user to specify either inclusive or exclusive forced back-off, as well as a list of words from which to enforce back-off.

Installing the Toolkit

For "big-endian" machines (eg those running HP-UX, IRIX, SunOS, Solaris) the installation procedure is simply to change into the `src/` directory and type

```
make install
```

The executables will then be copied into the `bin/` directory, and the library file `SLM2.a` will be copied into the `lib/` directory. For "little-endian" machines (eg those running Ultrix, Linux) the variable

`BYTESWAP_FLAG` will need to be set in the Makefile. This can be done by editing `src/Makefile` directly, so that the line

```
#BYTESWAP_FLAG  = -DSLM_SWAP_BYTES
```

is changed to

```
BYTESWAP_FLAG  = -DSLM_SWAP_BYTES
```

Then the program can be installed as before.

If you are unsure of the "endian-ness" of your machine, then the shell script `endian.sh` should be able to provide some assistance.

In case of problems, then more information can be found by examining `src/Makefile`.

Before building the executables, it might be worth adjusting the value of `STD_MEM` in the file `src/toolkit.h`. This value controls the default amount of memory (in MB) that the programs will attempt to assign for the large buffers used by some of the programs (this value can, of course, be overridden at the command line). The result is that the final process sizes will be a few MB bigger than this value. The more memory that can be grabbed, the faster the programs will run. The default value is 100, but if the machines which the tools will be run on contain less, or much more memory than this, then this value should be adjusted to reflect this.

Terminology and File Formats

Name	Description	Typical file extension
Text stream	An ASCII file containing text. It may or may not have markers to indicate context cues, and white space can be used freely.	`.text`
Word frequency file	An ASCII file containing a list of words, and the number of times that they occurred. This list is not sorted; it will generally be used as the input to `wfreq2vocab`, which does not require sorted input.	`.wfreq`
Word n-gram file	ASCII file containing an **alphabetically sorted** list of n-tuples of words, along with the number of occurrences	`.w3gram`, `.w4gram` etc.
Vocabulary file	ASCII file containing a list of vocabulary words. Comments may also be included - any line beginning `##` is considered a comment. The vocabulary is limited in size to 65535 words.	`.vocab.20K`, `.vocab.60K` etc., depending on the size of the vocabulary.
Context cues file	ASCII file containing the list of words which are to be considered "context cues". These are words which provide useful context information for the n-grams, but which are not to be predicted by the language model. Typical examples would be <s> and <p>, the begin sentence, and begin paragraph tags.	`.ccs`
Id n-gram file	ASCII **or** binary (by default) file containing a **numerically sorted** list of n-tuples of numbers, corresponding to the mapping of the word n-grams relative to the vocabulary. Out of vocabulary (OOV) words are mapped to the number 0.	`.id3gram.bin`, `.id4gram.ascii` etc.
Binary language	Binary file containing all the n-gram counts, together with discounting information and back-off weights. Can be read by `evallm` and used to	`.binlm`

model file	generate word probabilities quickly.	
ARPA language model file	ASCII file containing the language model probabilities in ARPA-standard format.	`.arpa`
Probability stream	ASCII file containing a list of probabilities (one per line). The probabilities correspond the the probability for each word in a specific text stream, with context-cues and OOVs removed.	`.fprobs`
Forced back-off file	ASCII file containing a list of vocabulary words from which to enforce back-off, together with either an 'i' or an 'e' to indicate inclusive or exclusive forced back-off respectively.	`.fblist`

These files may all be written are read by all the tools in compressed or uncompressed mode. Specifically, if a filename is given a `.Z` extension, then it will be read from the specified file via a `zcat` pipe, or written via a `compress` pipe. If a filename is given a `.gz`, it will be read from the specified file via a `gunzip` pipe, or written via a `gzip` pipe. If either of these compression schemes are to be used, then the relevant tools (ie `zcat`, and `compress` or `gzip`) must be available on the system, and pointed to by the path.

If a filename argument is given as `-` then it is assumed to represent either the standard input, or standard output (according to context). Any file read from the standard input is assumed to be uncompressed, and therefore, all desired compression and decompression should take place in a pipe:

```
zcat < abc.Z | abc2xyz | compress > xyz.Z
```

The Tools

Note that in addition to the command line options mentioned, all the tools also support `-help` and `-version`.

text2wfreq

Input : Text stream

Output : List of every word which occurred in the text, along with its number of occurrences.

Notes : Uses a hash-table to provide an efficient method of counting word occurrences. Output list is not sorted (due to "randomness" of the hash-table), but can be easily sorted into the user's desired order by the UNIX `sort` command. In any case, the output does not need to be sorted in order to serve as input for `wfreq2vocab`.

Command Line Syntax:

```
text2wfreq [ -hash 1000000 ]
           [ -verbosity 2 ]
           < .text > .wfreq
```

Higher values for the `-hash` parameter require more memory, but can reduce computation time.

wfreq2vocab

Input : A word unigram file, as produced by `text2wfreq`

Output : A vocabulary file.

Command Line Syntax:

```
wfreq2vocab [ -top 20000 | -gt 10]
            [ -records 1000000 ]
            [ -verbosity 2]
            < .wfreq > .vocab
```

The `-top` parameter allows the user to specify the size of the vocabulary; if the program is called with the command `-top 20000`, then the vocabulary will consist of the most common 20,000 words.

The `-gt` parameter allows the user to specify the number of times that a word must occur to be included in the vocabulary; if the program is called with the command `-gt 10`, then the vocabulary will consist of all the words which occurred more than 10 times.

If neither the `-gt`, nor the `-top` parameters are specified, then the program runs with the default setting of taking the top 20,000 words.

The `-records` parameter allows the user to specify how many of the word and count records to allocate memory for. If the number of words in the input exceeds this number, then the program will fail, but a high number will obviously result in a higher memory requirement.

text2wngram

Input : Text stream

Output : List of every word n-gram which occurred in the text, along with its number of occurrences.

Command Line Syntax:

```
text2wngram [ -n 3 ]
            [ -temp /usr/tmp/ ]
            [ -chars n ]
            [ -words m ]
            [ -gzip | -compress ]
            [ -verbosity 2 ]
            < .text > .wngram
```

The maximum numbers of charactors and words that can be stored in the buffer are given by the `-chars` and `-words` options. The default number of characters and words are chosen so that the memory requirement of the program is approximately that of `STD_MEM`, and the number of charactors is seven times greater than the number of words.

The `-temp` option allows the user to specify where the program should store its temporary files.

text2idngram

Input : Text stream, plus a vocabulary file.

Output : List of every id n-gram which occurred in the text, along with its number of occurrences.

Notes : Maps each word in the text stream to a short integer as soon as it has been read, thus enabling more n-grams to be stored and sorted in memory.

Command Line Syntax:

```
text2idngram -vocab .vocab
          [ -buffer 100 ]
          [ -temp /usr/tmp/ ]
          [ -files 20 ]
          [ -gzip | -compress ]
          [ -n 3 ]
          [ -write_ascii ]
          [ -fof_size 10 ]
          [ -verbosity 2 ]
          < .text > .idngram
```

By default, the id n-gram file is written out as binary file, unless the `-write_ascii` switch is used.

The size of the buffer which is used to store the n-grams can be specified using the `-buffer` parameter. This value is in megabytes, and the default value can be changed from 100 by changing the value of `STD_MEM` in the file `src/toolkit.h` before compiling the toolkit.

The program will also report the frequency of frequency of n-grams, and the corresponding recommended value for the `-spec_num` parameters of `idngram2lm`. The `-fof_size` parameter allows the user to specify the length of this list. A value of 0 will result in no list being displayed.

The `-temp` option allows the user to specify where the program should store its temporary files.

In the case of really huge quantities of data, it may be the case that more temporary files are generated than can be opened at one time by the filing system. In this case, the temporary files will be merged in chunks, and the `-files` parameter can be used to specify how many files are allowed to be open at one time.

ngram2mgram

Input : Either a word n-gram file, or an id n-gram file.

Output : Either a word m-gram file, or an id m-gram file, where m < n.

Command Line Syntax:

```
ngram2mgram -n N -m M
         [ -binary | -ascii | -words ]
         < .ngram > .mgram
```

The `-binary`, `-ascii`, `-words` correspond to the format of the input and output (Note that the output file will be in the same format as the input file). `-ascii` and `-binary` denote id n-gram files, in ASCII and binary formats respectively, and `-words` denotes a word n-gram file.

wngram2idngram

Input : Word n-gram file, plus a vocabulary file.

Output : List of every id n-gram which occurred in the text, along with its number of occurrences, in

either ASCII or binary format.

Note : For this program to be successful, it is important that the vocabulary file is in alphabetical order. If you are using vocabularies generated by the `wfreq2vocab` tool then this should not be an issue, as they will already be alphabetically sorted.

Command Line Syntax:

```
wngram2idngram -vocab .vocab
            [ -buffer 100 ]
            [ -hash 200000 ]
            [ -temp /usr/tmp/ ]
            [ -files 20 ]
            [ -gzip | -compress ]
            [ -verbosity 2 ]
            [ -n 3 ]
            [ -write_ascii ]
            < .wngram > .idngram
```

The size of the buffer which is used to store the n-grams can be specified using the `-buffer` parameter. This value is in megabytes, and the default value can be changed from 100 by changing the value of `STD_MEM` in the file `src/toolkit.h` before compiling the toolkit.

The program will also report the frequency of frequency of n-grams, and the corresponding recommended value for the `-spec_num` parameters of `idngram2lm`. The `-fof_size` parameter allows the user to specify the length of this list. A value of 0 will result in no list being displayed.

Higher values for the `-hash` parameter require more memory, but can reduce computation time.

The `-temp` option allows the user to specify where the program should store its temporary files.

The `-files` parameter is used to specify the number of files which can be open at one time.

idngram2stats

Input : An id n-gram file (in either binary (by default) or ASCII (if specified) mode).

Output : A list of the frequency-of-frequencies for each of the 2-grams, ... , n-grams, which can enable the user to choose appropriate cut-offs, and to specify appropriate memory requirements with the `-spec_num` option in `idngram2lm`.

Command Line Syntax:

```
idngram2stats [ -n 3 ]
            [ -fof_size 50 ]
            [ -verbosity 2 ]
            [ -ascii_input ]
            < .idngram > .stats
```

mergeidngram

Input : A set of id n-gram files (in either binary (by default) or ASCII (if specified) format - note that they should all be in the same format, however).

Output : One id n-gram file (in either binary (by default) or ASCII (if specified) format), containing the merged id n-grams from the input files.

Notes : This utility can also be used to convert id n-gram files between ascii and binary formats.

Command Line Syntax:

```
mergeidngram [ -n 3 ]
             [ -ascii_input ]
             [ -ascii_output ]
             .idngram_1 .idngram_2 ... .idngram_N > .idngram
```

idngram2lm

Input : An id n-gram file (in either binary (by default) or ASCII (if specified) format), a vocabulary file, and (optionally) a context cues file. Additional command line parameters will specify the cutoffs, the discounting strategy and parameters, etc.

Output : A language model, in either binary format (to be read by evallm), or in ARPA format.

Command Line Syntax:

```
idngram2lm -idngram .idngram
           -vocab .vocab
           -arpa .arpa | -binary .binlm
         [ -context .ccs ]
         [ -calc_mem | -buffer 100 | -spec_num y ... z ]
         [ -vocab_type 1 ]
         [ -oov_fraction 0.5 ]
         [ -linear | -absolute | -good_turing | -witten_bell ]
         [ -disc_ranges 1 7 7 ]
         [ -cutoffs 0 ... 0 ]
         [ -min_unicount 0 ]
         [ -zeroton_fraction 1.0 ]
         [ -ascii_input | -bin_input ]
         [ -n 3 ]
         [ -verbosity 2 ]
         [ -four_byte_counts ]
         [ -two_byte_bo_weights
            [ -min_bo_weight -3.2 ] [ -max_bo_weight 2.5 ]
            [ -out_of_range_bo_weights 10000 ] ]
```

The `-context` parameter allows the user to specify a file containing a list of words within the vocabulary which will serve as context cues (for example, markers which indicate the beginnings of sentences and paragraphs).

`-calc_mem, -buffer` and `-spec_num x y ... z` are options to dictate how it is decided how much memory should be allocated for the n-gram counts data structure. `-calc_mem` demands that the id n-gram file should be read twice, so that we can accurately calculate the amount of memory required. `-buffer` allows the user to specify an amount of memory to grab, and divides this memory equally between the 2,3, ..., n-gram tables. `-spec_num` allows the user to specify exactly how many 2-grams, 3-grams, ... , and n-grams will need to be stored. The default is `-buffer` `STD_MEM`.

The toolkit provides for three types of vocabulary, which each handle out-of-vocabulary (OOV) words in different ways, and which are specified using the `-vocab_type` flag.

A *closed vocabulary* (`-vocab_type 0`) model does not make any provision for OOVs. Any such words

which appear in either the training or test data will cause an error. This type of model might be used in a command/control environment where the vocabulary is restricted to the number of commands that the system understands, and we can therefore guarantee that no OOVs will occur in the training or test data.

An *open vocabulary* model allows for OOVs to occur; out of vocabulary words are all mapped to the same symbol. Two types of open vocabulary model are implemented in the toolkit. The first type (`-vocab_type 1`) treats this symbol the same way as any other word in the vocabulary. The second type (`-vocab_type 2`) of open vocabulary model is to cover situations where no OOVs occurred in the training data, but we wish to allow for the situation where they could occur in the test data. This situation could occur, for example, if we have a limited amount of training data, and we choose a vocabulary which provides 100% coverage of the training set. In this case, an arbitrary proportion of the discount probability mass (specified by the `-oov_fraction` option) is reserved for OOV words.

The discounting strategy and its parameters are specified by the `-linear`, `-absolute`, `-good_turing` and `-witten_bell` options. With Good Turing discounting, one can also specify the range over which discounting occurs, using the `-disc_ranges` option.

The user can specify the cutoffs for the 2-grams, 3-grams, ..., n-grams by using the `-cutoffs` parameter. A cutoff of *K* means that > n-grams occurring *K* or fewer times are discarded. If the parameter is omitted, then all the cutoffs are set to zero.

The `-zeroton_fraction` option specifies that P(zeroton) (the unigram probability assigned to a vocabulary word that did not occurred at all in the training data) will be at least that fraction of P(singleton) (the probability assigned to a vocabulary word that occurred exactly once in the training data).

By default, the n-gram counts are stored in two bytes by use of a count table (this allows the counts to exceed 65535, while keeping the data structures used to store the model compact). However, if more than 65535 **distinct** counts need to be stored (very unlikely, unless constructing 4-gram or higher language models using Good-Turing discounting), the -four_byte_counts option will need to be used.

The floating point values of the back-off weights may be stored as two-byte integers, by using the `-two_byte_alphas` switch. This will introduce slight rounding errors, and so should only be used if memory is short. The `-min_alpha`, `-max_alpha` and `-out_of_range_alphas` are parameters used by the functions for using two-byte alphas. Their values should only be altered if the program instructs it. For further details, see the comments in the source file `src/two_byte_alphas.c`.

binlm2arpa

Input : A binary format language model, as generated by `idngram2lm`.

Output : An ARPA format language model.

Command Line Syntax:

```
binlm2arpa -binary .binlm
           -arpa .arpa
         [ -verbosity 2 ]
```

evallm

Input : A binary or ARPA format language model, as generated by `idngram2lm`. In addition, one may also specify a text stream to be used to compute the perplexity of the language model. The ARPA format language model does not contain information as to which words are context cues, so if an ARPA format lanaguage model is used, then a context cues file may be specified as well.

Output : The program can run in one of two modes.

- compute-PP - Output is the perplexity of the language model with respect to the input text stream.
- validate - Output is confirmation or denial that the sum of the probabilities of each of the words in the context supplied by the user sums to one.

Command Line Syntax:

```
evallm [ -binary .binlm |
         -arpa .arpa [ -context .ccs ] ]
```

Notes: `evallm` can receive and process commands interactively. When it is run, it loads the language model specified at the command line, and waits for instructions from the user. The user may specify one of the following commands:

- **`perplexity`**
  Computes the perplexity of a given text. May optionally specify words from which to force back-off.

  Syntax:

  ```
  perplexity -text .text
          [ -probs .fprobs ]
          [ -oovs .oov_file ]
          [ -annotate .annotation_file ]
          [ -backoff_from_unk_inc | -backoff_from_unk_exc ]
          [ -backoff_from_ccs_inc | -backoff_from_ccs_exc ]
          [ -backoff_from_list .fblist ]
          [ -include_unks ]
  ```

  If the `-probs` parameter is specified, then each individual word probability will be written out to the specified probability stream file.
  If the `-oovs` parameter is specified, then any out-of-vocabulary (OOV) words which are encountered in the test set will be written out to the specified file.
  If the `-annotate` parameter is used, then an annotation file will be created, containing information on the probability of each word in the test set according to the language model, as well as the back-off class for each event. The back-off classes can be interpreted as follows: Assume we have a trigram language model, and are trying to predict P(C | A B). Then back-off class "3" means that the trigram "A B C" is contained in the model, and the probability was predicted based on that trigram. "3-2" and "3x2" mean that the model backed-off and predicted the probability based on the bigram "B C"; "3-2" means that the context "A B" was found (so a back-off weight was applied), "3x2" means that the context "A B" was not found.
  To force back-off from all unknown words, use the `-backoff_from_unk_inc` or `-backoff_from_unk_exc` flag (the difference being the difference between inclusive or exclusive forced back-off). To force back-off from all context-cues, use the `-backoff_from_ccs_inc` or `-backoff_from_ccs_inc` flag. One can also specify a list of words from which to back-off, by storing this list in a forced back-off list file and using the `-backoff_from_list` switch.
  `-include_unks` results in a perplexity calculation in which the probability estimates for the unkown word are included.
- **`validate`**
  Calculate the sum of the probabilities of all the words in the vocabulary given the context

specified by the user.

Syntax:

```
validate [ -backoff_from_unk_inc | -backoff_from_unk_exc ]
         [ -backoff_from_ccs_inc | -backoff_from_ccs_exc ]
         [ -backoff_from_list .fblist ]
           word1 word2 ... word_(n-1)
```

Where n is the n in n-gram.

- **`help`**
  Displays a help message.

  Syntax:

  ```
  help
  ```

- **`quit`**
  Exits the program.

  Syntax:

  ```
  quit
  ```

Since the commands are read from standard input, a command file can be piped into it directly, thus removing the need for the program to run interactively:

```
echo "perplexity -text b.text" | evallm -binary a.binlm
```

interpolate

Input : Files containing probability streams, as generated by the `-probs` option of the `perplexity` command of `evallm`. Alternatively these probabilities could be generated from a seperate piece of code, which assigns word probabilities according to some other language model, for example a cache-based LM. This probability stream can then be linearly interpolated with one from a standard n-gram model using this tool.

Output : An optimal set of interpolation weights for these probability streams, and (optionally) a probability stream corresponding to the linear combination of all the input streams, according to the optimal weights. The optimal weights are calculated using the expectation maximisation (EM) algorithm.

Command Line Syntax :

```
interpolate +[-] model1.fprobs +[-] model2.fprobs ...
       [ -test_all | -test_first n | -test_last n | -cv ]
       [ -tag .tags ]
       [ -captions .captions ]
       [ -in_lambdas .lambdas ]
       [ -out_lambdas .lambdas ]
       [ -stop_ratio 0.999 ]
       [ -probs .fprobs ]
       [ -max_probs 6000000 ]
```

The probability stream filenames are prefaced with a `+` (or a `+-` to indicate that the weighting of that model should be fixed).

There are a range of options to determine which part of the data is used to calculate the weights, and which is used to test them. One can test the perplexity of the interpolated model based on all the data, using the `-test_all` option, in which case a set of lambdas must also be specified with the `-lambda` option (ie the lambdas are pre-specified, and not calculated by the program). One can specify that the first or last n items are the test set by use of the `-test_first n` or `-test_last n` options. Or one can perform two-way cross validation using the `-cv` option. If none of these are specified then the whole of the data is used for weight estimation.

By default, the initial interpolation weights are fixed as 1/number_of_models, but alternative values can be stored in a file and used via the `-in_lambdas` option.

The `-probs` switch allows the user to specify a filename in which to store the combined probability stream. The optimal lambdas can also be stored in a file by use of the `-out_lambdas` command.

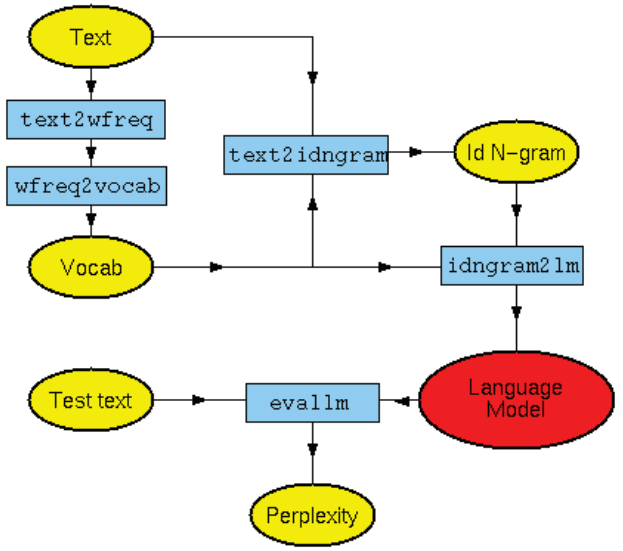
The program stops when the ratio of the test-set perplexity between two successive iterations is above the value specified in the `-stop_ratio` option.

The data can be partitioned into different classes (with optimisation being performed seperately on each class) using the `-tags` parameter. The tags file will contain an integer for each item in the probability streams corresponding to the class that the item belongs to. A file specified using the `-captions` option will allow the user to attach names to each of the classes. There should be one line in the captions file for each tag, with each line corresponding to the name of the tag.

The amount of memory allocated to store the probability streams is dictated by the `-max_probs` option, which indicates the maximum number of probabilities allowed in one stream.

Note: For an example use and output of a previous version of this program (with slightly different syntax), see Appendix B of **R. Rosenfeld** *Adaptive Statistical Language Modeling: A Statistical Approach* PhD Thesis, School of Computer Science, Carnegie Mellon University, April 1994. Published as Techical Report CMU-CS-94-138

Typical Usage



Given a large corpus of text in a file `a.text`, but no specified vocabulary

- Compute the word unigram counts

  ```
  cat a.text | text2wfreq > a.wfreq
  ```

- Convert the word unigram counts into a vocabulary consisting of the 20,000 most common words

  ```
  cat a.wfreq | wfreq2vocab -top 20000 > a.vocab
  ```

- Generate a binary id 3-gram of the training text, based on this vocabulary

  ```
  cat a.text | text2idngram -vocab a.vocab > a.idngram
  ```

- Convert the idngram into a binary format language model

  ```
  idngram2lm -idngram a.idngram -vocab a.vocab -binary a.binlm
  ```

- Compute the perplexity of the language model, with respect to some test text `b.text`

  ```
  evallm -binary a.binlm
  Reading in language model from file a.binlm
  Done.
  evallm : perplexity -text b.text
  Computing perplexity of the language model with respect
  to the text b.text
  Perplexity = 128.15, Entropy = 7.00 bits
  Computation based on 8842804 words.
  Number of 3-grams hit = 6806674 (76.97%)
  Number of 2-grams hit = 1766798 (19.98%)
  Number of 1-grams hit = 269332 (3.05%)
  1218322 OOVs (12.11%) and 576763 context cues were removed from the calculation.
  ```

```
      evallm : quit
```

Alternatively, some of these processes can be piped together:

```
cat a.text | text2wfreq | wfreq2vocab -top 20000 > a.vocab
cat a.text | text2idngram -vocab a.vocab | \
   idngram2lm -vocab a.vocab -idngram - \
   -binary a.binlm -spec_num 5000000 15000000
echo "perplexity -text b.text" | evallm -binary a.binlm
```

Discounting Strategies

Discounting is the process of replacing the original counts with modified counts so as to redistribute the probability mass from the more commonly observed events to the less frequent and unseen events. If the actual number of occurrences of an event E (such as a bigram or trigram occurrence) is $c(E)$, then the modified count is $d(c(E))c(E)$, where $d(c(E))$ is known as the discount ratio.

Good Turing discounting

Good Turing discounting defines $d(r) = (r+1)n(r+1) / rn(r)$ where $n(r)$ is the number of events which occur r times.

The discounting is only applied to counts which occur fewer than K times, where typically K is chosen to be around 7. This is the "discounting range" which is specified using the `-disc_ranges` parameter of the `idngram2lm` program.

For further details see "*Estimation of Probabilities from Sparse Data for the Language Model Component of a Speech Recognizer*", **Slava M. Katz**, in "IEEE Transactions on Acoustics, Speech and Signal Processing", volume ASSP-35, pages 400-401, March 1987.

Witten Bell discounting

The discounting scheme which we refer to here as "Witten Bell discounting" is that which is referred to as type C in "*The Zero-Frequency Problem: Estimating the Probabilities of Novel Events in Adaptive Text Compression*", **Ian H. Witten and Timothy C. Bell**, in "IEEE Transactions on Information Theory, Vol 37, No. 4, July 1991".

The discounting ratio is not dependent on the event's count, but on t, the number of types which followed the particular context. It defines $d(r,t) = n/(n + t)$, where n is the size of the training set in words. This is equivalent to setting $P(w \mid h) = c / (n + t)$ (where w is a word, h is the history and c is the number of occurrences of w in the context h), for events that have been seen, and $P(w \mid h) = t / (n + t)$ for unseen events.

Absolute discounting

Absolute discounting defines $d(r) = (r-b)/r$. Typically $b=n(1)/(n(1)+2n(2))$. The discounting is applied to all counts.

This is, of course, equivalent to simply subtracting the constant b from each count.

Linear discounting

Linear discounting defines $d(r) = 1 - (n(1)/C)$, where C is the total number of events. The discounting

is applied to all counts.

For further details of both linear and absolute discounting, see "*On structuring probabilistic dependencies in stochastic language modeling*", **H. Ney, U. Essen and R. Kneser** in "Computer Speech and Language", volume 8(1), pages 1-28, 1994.

Up-to-date Information

The latest news on updates, bug fixes etc. can be found here.

Feedback

Any comments, questions or bug reports concerning the toolkit should be addressed to Philip Clarkson.

Philip Clarkson - prc14@eng.cam.ac.uk