Language Model Tools (v0.1) User's Guide

Stanley F. Chen

October 16, 1998

Contents

1 Overview

These are tools for creating n-gram models in `arpabo` format. N-gram models of any order can be created, and count cutoffs are supported.

The basic tools supplied are as follows:

- `CountNGram` takes text as input and counts the number of occurrences of each n-gram up to some n, outputting binary count files containing these counts.
- `BuildNGram.x` takes count files as input and outputs an n-gram model in text `arpabo` format or a binary `dmp` format. Many different smoothing algorithms are supported, and count cutoffs can be specified. `BuildNGram.x` is a shell script that calls the following programs:
  - `Smooth` takes count files as input and outputs intermediate binary files containing smoothed counts that can be converted into `arpabo` files. It supports many smoothing methods and count cutoffs, and can be used to evaluate the perplexity of text directly.
  - `MakeArpabo` converts the binary files containing smoothed counts produced by `Smooth` into text `arpabo` format.
  - `DumpArpabo` converts a file in `arpabo` format to one in a binary `dmp` format. Files in `dmp` format can be loaded much more quickly ($\sim 10\times$ faster) than `arpabo` files by tools in this toolkit.

- `EvalLM` takes a language model (LM) in `arpabo` or `dmp` format and evaluates the perplexity of text with that model.

2 Location of Programs and Basic Usage Notes

All executables are located in the directory `/net/alf14/usr/sfc/pub/exec`. For all binary executables (*i.e.*, programs without the `.x` suffix), the program in this directory is a shell script that calls the binary of the appropriate architecture.

The actual binaries are located in the directories `/net/alf14/usr/sfc/pub/$arch/bin`, where currently the supported `$arch`'s are `alpha_osf20/osf32/dux40` and `i386_linux2`. To determine the `$arch` value of a machine, use the command

```
/net/alf14/usr/sfc/pub/exec/hosttype.x
```

Access to disks on the machine `alf14` has been restricted to machines in the speech group. Most, but not all, machines in the speech group have access.

To view a list of command-line arguments and flags for any program, run the program with no arguments. For any argument corresponding to an input file, a compressed or uncompressed file can be specified. To have an output file be compressed, specify a filename with the `.gz` suffix.

3 Input Files and Formats

To create a language model, one needs two basic files: a *corpus* of training text, and a *vocabulary*.

3.1 A Corpus

A *corpus* contains the training text used to construct a language model. The basic format for a corpus file is ASCII text where all words/tokens are separated by white space. The token `</s>` signals the end of a sentence. Newlines have no special meaning. Tokens (other than the end-of-sentence token) surrounded by angle brackets and beginning with either a lower-case character or the character '/' followed be a lower-case character are interpreted to be markup tokens and are ignored. Files in this format must have the suffix `.txt` (or `.txt.gz` if compressed).

Another corpus format supported is an executable, such as a shell script, that outputs text in the above format to standard output. Corpora in this format must have the suffix `.x`, `.sh`, or `.csh`.

3.2 A Vocabulary

A *vocabulary* file must contain a list of unique words, one to a line, with no extra spaces on lines or blank lines. A vocabulary *must* contain the following words:

- `<s>`, the beginning-of-sentence token
- `</s>`, the end-of-sentence token
- `<UNK>`, the unknown word

All words not in the vocabulary found in a corpus are mapped to the unknown word.

If disk space is at a premium, the size of the intermediate binary files produced by these tools can be reduced by sorting the vocabulary in frequency order, with the most frequent words listed first. In particular, these tools use a variable-length encoding of word indices, where words with lower indices are coded with fewer bytes.

4 Tool Usage

The language modeling tools will be described with the use of an example. The data files for the example are located in the directory /net/alf14/usr/sfc/pub/data/j015. To step through the example, copy the files in this directory into your current directory.

4.1 Counting N-Grams

The first step in building a language model is counting the n-grams in your training text using CountNGram. The basic usage of this program is:

```
CountNGram [<flags>] <corpus> <vocab> <out-base>
```

The <corpus> argument is the name of the file containing the training text (see Section 3.1). The <vocab> argument is the name of the file containing the vocabulary (see Section 3.2). **NOTE:** for best results, use a full pathname when specifying the vocabulary. The <out-base> argument specifies the base name of all count files to be created; *i.e.*, all files created will begin with this path/prefix. This argument determines which directory count files will be created in, and this directory must have adequate free space.

The most important command line flags are:

-n <val> This specifies the longest n-grams to count, *e.g.*, -n 3 is appropriate for a trigram model (and is the default). A set of count files can be used to build n-gram models of that order or below.

-mem <MB> This specifies how many MB of memory the program will use; the default is 100 MB. The larger this value, the faster the program will run. Use a value comfortably lower than the total amount of memory in the machine.

Run CountNGram with no arguments to view a listing of all flags.

To give an example, the command

```
/net/alf14/usr/sfc/pub/exec/CountNGram j015d.train.txt j015d.voc j015e
```

will create the count files

```
j015e.count.1, j015e.count.2, j015e.count.3, j015e.count.check
```

4.2 Creating a Language Model

To build an n-gram language model from the count files produced by CountNGram, use the BuildNGram.x shell script. The basic usage of this program is:

```
BuildNGram.x [<flags>] <count-base> <out-base>
```

The <count-base> argument is the base name of the input count files, and should be the same as the last argument used in the corresponding run of CountNGram. The <out-base> argument specifies the base name of all files to be created; *i.e.*, all files created will begin with this path/prefix. This argument determines which directory the language model file will be created in, and this directory must have adequate free space. Most of these created files will be deleted, except for the output language model <out-base>.dmp. To create a language model in arpabo format, see the -arpabo flag below.

To view the command line flags of BuildNGram.x, run Smooth with no arguments. The flags of these two programs are identical except that the -arpabo flag is treated differently. The most important command line flags are:

-n <val> This specifies the order of the n-gram model to construct; *e.g.*, -n 3 is appropriate for a trigram model (and is the default).

-alg <smooth-alg> This specifies the type of smoothing to use. The suggested choices are `kneser-ney-mod-fix` if you don't have appropriate held-out data to optimize smoothing parameters with, and `kneser-ney-mod` if you do. To specify a held-out set, use the `-heldout` flag.

-heldout <corpus> This specifies the held-out corpus to use to optimize the parameters of the chosen smoothing technique, if applicable.

-cutoffs <count-cutoffs> This specifies the count cutoffs to use. For example, for a trigram model the flag `-cutoffs 0,1,2` would specify using all unigrams, ignoring bigrams with one count or less, and ignoring trigrams with two counts or less.

-arpabo Use this flag to create the arpabo file `<out-base>.arpabo.gz` as well as the `dmp`-format LM file.

To give an example, the command

```
/net/alf14/usr/sfc/pub/exec/BuildNGram.x \
    -alg kneser-ney-mod -heldout j015d.heldout.txt -cutoffs 0,0,1 j015e j015e
```

will create the language model file `j015e.dmp`. It will be a trigram model smoothed with modified Kneser-Ney smoothing and excludes trigrams with only one count. The smoothing parameters will be chosen to optimize the likelihood of the held-out data `j015d.heldout.txt`.

4.3 Evaluating a Language Model

To evaluate the perplexity of a language model on some text, use the command

```
EvalLM [<flags>] <corpus> <LM-file>
```

The `<corpus>` argument is the name of the file containing the text to be evaluated; it should be in the format given in Section 3.1. The `<LM-file>` argument is the name of the file containing the language model to be evaluated; it can be in `arpabo` or `dmp` format, compressed or uncompressed, and must have the appropriate suffix. **NOTE:** files in `dmp` format load much faster than the corresponding `arpabo` files.

The most important command line flag is:

-n <val> This specifies the order of the n-gram model to evaluate; *e.g.*, `-n 3` is appropriate for a trigram model (and is the default). This value can be lower than the actual order of the language model file specified; in this case only the lower-order subset of the language model is used. **NOTE:** setting this value lower than the actual order of an n-gram model can yield very poor performance depending on the smoothing method used to create the model.

Run `EvalLM` with no arguments to view a listing of all flags.

To give an example, the command

```
/net/alf14/usr/sfc/pub/exec/EvalLM j015d.heldout.txt j015e.dmp
```

will output the perplexity of the trigram model `j015e.dmp` on the heldout set `j015d.heldout.txt`.

4.4 Resource Consumption

This section will be completed on demand.

A Undocumented Features

There are many undocumented features of this toolkit, which will be documented on demand. These features include:

- support for linearly interpolating several language models
- support for class n-gram models
- support for cache models

B Known Bugs

- The binary `dmp` format used by these tools is not compatible with the Sphinx-III language model `DMP` format.