IRST Language Modeling Toolkit
Version 5.22.01
USER MANUAL

M. Federico, N. Bertoldi, M. Cettolo
FBK-irst, Trento, Italy

September 15, 2009

1 Introduction

The IRST Language Modeling Toolkit features algorithms and data structures suitable to estimate, store, and access very large LMs. Our software has been integrated into a popular open source SMT decoder called `Moses`[1], and is compatible with LMs created with other tools, such as the SRILM Tooolkit[2]

Acknowledgments. Users of this toolkit might cite in their publications:

> M. Federico, N. Bertoldi, M. Cettolo, *IRSTLM: an Open Source Toolkit for Handling Large Scale Language Models*, Proceedings of Interspeech, Brisbane, Australia, 2008.

References to introductory material on n-gram LMs are given in the appendix.

2 Getting started

Environment Settings Commands and scripts described in this manual are installed under the directories `bin` (and architecture-dependent subdirectories). We assume that directory bin is included in your `PATH` environment variable. You need also to set the environment variable `IRSTLM` to the path of this package. Data sets used in the examples can be found in the `example` directory.

Examples The directory `example` contains two English text files, namely `train.gz` and `test`, which we will use to estimate and evaluate our LM, respectively. In particular, LM evaluation computes both the perplexity and the out-of-vocabulary rate of the test set. Notice that both file are tokenized and contain one sentence per line and sentence boundary symbols. Given a text file, sentence boundary symbols can be added in each line with the script `add-start-end.sh`:

```
$> add-start-end.sh < your-text-file
```

[1] http://www.statmt.org/moses/
[2] http://www.speech.sri.com/projects/srilm

3 Estimating Gigantic LMs

LM estimation starts with the collection of n-grams and their frequency counters. Then, smoothing parameters are estimated for each n-gram level; infrequent n-grams are possibly pruned and, finally, a LM file is created containing n-grams with probabilities and back-off weights. This procedure can be very demanding in terms of memory and time if it applied on huge corpora. We provide here a way to split LM training into smaller and independent steps, that can be easily distributed among independent processes. The procedure relies on a training scripts that makes little use of computer RAM and implements the Witten-Bell smoothing method in an exact way.
Before starting, let us create a working directory under `examples`, as many files will be created:

```
$> mkdir stat
```

The script to generate the LM is:

```
> build-lm.sh -i "gunzip -c train.gz" -n 3  -o train.ilm.gz -k 5
```

where the available options are:

```
-i    Input training file e.g. 'gunzip -c train.gz'
-o    Output gzipped LM, e.g. lm.gz
-k    Number of splits (default 5)
-n    Order of language model (default 3)
-t    Directory for temporary files (default ./stat)
-p    Prune singleton n-grams (default false)
-s    Smoothing: witten-bell (default), kneser-ney, improved-kneser-ney
-b    Include sentence boundary n-grams (optional)
-d    Define subdictionary for n-grams (optional)
-v    Verbose
```

The script splits the estimation procedure into 5 distinct jobs, that are explained in the following section. There are other options that can be used. We recommend for instance to use pruning of singletons to get smaller LM files. Notice that `build-lm.sh` produces a LM file `train.ilm.gz` that is NOT in the final ARPA format, but in an intermediate format called `iARPA`, that is recognized by the `compile-lm` command and by the Moses SMT decoder running with IRSTLM. To convert the file into the standard ARPA format you can use the command:

```
> compile-lm train.ilm.gz --text yes train.lm
```

this will create the proper ARPA file `lm-final`. To create a gzipped file you might also use:

```
> compile-lm train.ilm.gz --text yes /dev/stdout | gzip -c > train.lm.gz
```

In the following sections, we will talk about LM file formats, compiling your LM into a more compact and efficient binary format, and about querying your LM.

3.1 Estimating a LM with a Partial Dictionary

We can extract the corpus dictionary sorted by frequency with the command:

```
$> dict -i="gunzip -c train.gz" -o=dict -f=y -sort=no
```

A sub-dictionary can be defined by just taking words occurring at least 5 times:

```
$> (echo DICTIONARY; tail +2 dict | awk '{if ($2>=5) print}') > sdict
```

The LM can be restricted to the defined sub-dictionary with the command `build-lm.sh` by using the option `-d`:

```
> build-lm.sh -i "gunzip -c train.gz" -n 3  -o sublm.gz -k 5 -p -d sdict
```

Notice that, all words outside the sub-dictionary will be mapped to the `<unk>` class, the probability of which will be directly estimated from the corpus statistics. A preferable alternative to this approach is to estimate a large LM and then to filter it according to a list of words (see Filtering a LM).

4 LM File Formats

This toolkit supports three output format of LMs. These formats have the purpose of permitting the use of LMs by external programs. External programs could in principle estimate the LM from an n-gram table before using it, but this would take much more time and memory! So the best thing to do is to first estimate the LM, and then compile it into a binary format that is more compact and that can be quickly loaded and queried by the external program.

4.1 ARPA Format

This format was introduced in DARPA ASR evaluations to exchange LMs. ARPA format is also supported by the SRI LM Toolkit. It is a text format which is rather costly in terms of memory. There is no limit to the size n of n-grams.

4.2 qARPA Format

This extends the ARPA format by including codebooks that quantize probabilities and back-off weights of each n-gram level. This format is created through the command `quantize-lm`.

4.3 iARPA Format

This is an intermediate ARPA format in the sense that each entry of the file does not contain in the first position the full n-gram probability, but just its smoothed frequency, i.e.:

```
...
f(z|x y) x y z bow(x y)
...
```

This format is nevertheless properly managed by the `compile-lm` command in order to generate a binary version or a correct ARPA version.

4.4 Binary Formats

Both ARPA and qARPA formats can be converted into a binary format that allows for space savings on disk and a much quickers upload of the LM file. Binary versions can be created with the command `compile-lm`, that produces files with headers `blmt` or `Qblmt`.

5 LM Pruning

Large LMs files can be pruned in a smart way by means of the command `prune-lm` that removes n-grams for which resorting to the back-off results in a small loss. The syntax is as follows:

```
> prune-lm --threshold=1e-6,1e-6  train.lm.gz  train.plm
```

Thresholds for each n-gram level, up from 2-grams, are based on empirical evidence. Threshold zero results in no pruning. If less thresholds are specified, the right most is applied to the higher levels. Hence, in the above example we could have just specified one threshold, namely `--threshold=1e-6`. The effect of pruning is shown in the following messages of `prune-lm`:

```
1-grams: reading 15059 entries
2-grams: reading 142684 entries
3-grams: reading 293685 entries
done
OOV code is 15058
OOV code is 15058
pruning LM with thresholds:
 1e-06 1e-06
savetxt: train.lm.plm
save: 15059 1-grams
save: 135967 2-grams
save: 185127 3-grams
```

The saved LM table `train.plm` contains about 5% less bigrams, and 37% less trigrams! Notice that the output of prune-lm is an ARPA LM file, while the input can be either an ARPA or binary LM. Notice that quantization must be eventually performed after pruning! In order to measure the loss in accuracy introduced by pruning, perplexity of the resulting LM can per computed (see below).

6 LM Quantization

A language model file in ARPA format, created with the IRST LM toolkit or with other tools, can be quantized and stored in a compact data structure, called language model table. Quantization can be performed by the command:

```
$> quantize-lm  train.lm train.qlm
```

which generates the quantized version `train.qlm` that encodes all probabilities and back-off weights in 8 bits. The output is a modified ARPA format, called qARPA. Notice that quantized LMs reduce memory consumptions at the cost of some loss in performance. Moreover, probabilities of quantized LMs are not supposed to be properly normalized!

7 LM Compilation

LMs in ARPA, iARPA, and qARPA format can be stored in a compact binary table through the command:

```
$> compile-lm train.lm tran.blm
```

which generates the binary file `train.blm` that can be quickly loaded in memory. If the LM is really very large, `compile-lm` can avoid to create the binary LM directly in memory through the option `-memmap 1`, which exploits the *Memory Mapping* mechanism in order to work as much as possible on disk rather than in RAM.

```
$> compile-lm --memmap 1 train.lm tran.blm
```

This option clearly pays a fee in terms of speed, but is often the only way to proceed. It is also recommended that the hard disk for the LM storage belongs to the computer on which the compilation is performed.
Notice that most of the functionalities of `compile-lm` (see below) apply to binary and quantized models.

8 Filtering a LM

A large LM can be filtered according to a word list through the command:

```
$> compile-lm  train.lm -d list filtered.lm
```

The resulting LM will only contain n-grams inside the provided list of words, with the exception of the 1-gram level, which by default is preserved identical to the original LM. This behavior can be changed by setting the option `--keepunigrams no`. LM filtering can be useful once very large LMs can be specialized in advance to work on a particular portion of language. If the original LM is in binary format and is very large, `compile-lm` can avoid to load it in memory, through the memory mapping option `-memmap 1`.

9 LM Interface

LMs are useful when they can be queried through another application in order to compute perplexity scores or n-gram probabilities. IRSTLM provides two possible interfaces:

- at the command level, through `compile-lm`
- at the c++ library level, mainly through methods of the class `lmtable`

In the following, we will only focus on the command level interface. Details about the c++ library interface will be provided in a future version of this manual.

9.1 Perplexity Computation

To compute the perplexity directly from the LM on disk, we can use the command:

```
$> compile-lm train.lm  --eval=test
   Nw=49984 PP=1064.40 PPwp=589.50 Nbo=39847 Noov=2503 OOV=5.01%
```

Notice that `PPwp` reports the contribution of OOV words to the perplexity. Each OOV word is indeed penalized by dividing the LM probability of the `unk` word by the quantity

`DictionaryUpperBound - SizeOfDictionary`

The OOV penalty can be modify by changing the `DictionaryUpperBound` with the parameter `--dub` (whose default value is set to 10^7).

The perplexity of the pruned LM can be computed with the command:

```
compile-lm train.plm --eval test --dub 10000000
Nw=49984 PP=1019.57 PPwp=564.67 Nbo=42671 Noov=2503 OOV=5.01%
```

Interestingly, a slightly better value is obtained which could be explained by the fact that pruning has removed many unfrequent trigrams and has redistributed their probabilities over more frequent bigrams.

Again, if the LM is in binary format and is very large, `compile-lm` can avoid to load it in memory, through the memory mapping option `-memmap 1`.

9.2 Probability Computations

We can compute as well log-probabilities word-by-word from standard input with the command:

```
$> compile-lm train.lm --score yes < test

> </s>  1 p= NULL
> </s> <s>      1 p= NULL
> </s> <s> <unk>        1 p= -6.130331e+00 bo= 2
> <s> <unk> of  1 p= -3.530050e+00 bo= 2
> <unk> of the  1 p= -1.250671e+00 bo= 1
> of the senate 1 p= -8.805695e+00 bo= 0
> the senate (  1 p= -6.150410e+00 bo= 2
> senate ( <unk>   1 p= -5.547798e+00 bo= 2
....
....
```

the commands reports the currently observed n-gram, including `_unk_` words, a dummy constant frequency 1, the log-probability of the n-gram, and the number of back-offs performed by the LM.
Finally, tracing information with the `--eval` option are shown by setting debug levels from 1 to 4 (`--debug`):

1 reports the back-off level for each word

2 adds the log-prob

3 adds the back-off weight

4 check if probabilities sum up to 1.

10 LM Interpolation

We provide a convenient tool to estimate mixtures of LMs that have been already created in one of the available formats. The tool permits to estimate interpolation weights through the EM algorithm, to compute the perplexity, and to query the interpolated LM. Interpolated LMs are simply defined by a configuration file in the following format:

```
3
0.3 lm-file1
0.3 lm-file2
0.4 lm-file3
```

The first number indicates the number of LMs to be interpolated, then each LM is specified by its weight and its file (either in ARPA or binary format). Notice that you can interpolate LMs with different orders!

Given an initial configuration file `lmlist.start` (with arbitrary weights), new weights can be estimated through Expectation-Maximization on some text sample `text-file` by running the command: [lm-list-file.out]

```
> interpolate-lm lmlist.start --learn text-file  lmlist.final
```

New weights will be written in the updated configuration file `lmlist.final`. The full list of options is:

```
--learn text-file   learn optimal interpolation for text-file
--order n           order of n-grams used in --learn (optional)
--eval text-file    compute perplexity on text-file
--dub dict-size     dictionary upper bound (default 10^7)
--score [yes|no]    compute log-probs of n-grams from stdin
--debug [1-3]       verbose output for --eval option (see compile-lm)
--memmap 1          use memory map to read a binary LM
```

Interpolated LMs can be queried through the option `--score`, similarly to `compile-lm`.

If there are binary LMs in the list, `interpolate-lm` can avoid to load them in memory through the memory mapping option `-memmap 1`.

11 Parallel Computation

This package provides facilities to build a gigantic LM in parallel in order to reduce computation time. The script implementing this feature is based on the `SUN Grid Engine` software[3].
To apply the parallel computation run the following script (instead of `build-lm.sh`):

[3] http://www.sun.com/software/gridware

```
> build-lm-qsub.sh -i "gunzip -c train.gz" -n 3  -o train.ilm.gz -k 5
```

Besides the options of `build-lm.sh`, parameters for the SGE manager can be provided through the following one:

```
-q      parameters for qsub, e.g. "-q <queue>", "-l <resources>"
```

The script performs the same *split-and merge* policy described in Section 6, but some computation is performed in parallel (instead of sequentially) distributing the tasks on several machines.

A Reference Material

The following books contain basic introductions to statistical language modeling:

- *Spoken Dialogues with Computers*, by Renato DeMori, chapter 7.
- *Speech and Language Processing*, by Dan Jurafsky and Jim Martin, chapter 6.
- *Foundations of Statistical Natural Language Processing*, by C. Manning and H. Schuetze.
- *Statistical Methods for Speech Recognition*, by Frederick Jelinek.
- *Spoken Language Processing*, by Huang, Acero and Hon.

B Release Notes

B.1 Version 3.2

- Quantization of probabilities
- Efficient run-time data structure for LM querying
- Dismissal of MT output format

B.2 Version 4.2

- Distinction between open source and internal Irstlm tools
- More memory efficient versions of binarization and quantization commands
- Memory mapping of run-time LM
- Scripts and data structures for the estimation and handling of gigantic LMs
- Integration of IRSTLM into Moses Decoder

B.3 Version 5.00

- Fixed bug in the documentation
- General script `build-lm.sh` for the estimation of large LMs.
- Management of iARPA file format.
- Bug fixes
- Estimation of LM over a partial dictionary.

B.4 Version 5.04

- Parallel estimation of gigantic LM through SGE
- Better management of sub dictionary with build-lm.sh
- Minor bug fixes

B.5 Version 5.05

- (Optional) computation of OOV penalty in terms of single OOV word instead of OOV class
- Extended use of OOV penalty to the standard input LM scores of compile-lm.
- Minor bug fixes

B.6 Version 5.10

- Extended ngt to compute statistics for approximated Kneser-Ney smoothing
- New implementation of approximated Kneser-Ney smoothing method
- Minor bug fixes
- More to be added here ....

B.7 Version 5.20

- Improved tracing of back-offs
- Added command prune-lm (thanks to Fabio Brugnara)
- Extended lprob function to supply back-off weight/level information
- Improved back-off handling of OOV words with quantized LM
- Added more debug modalities to compile-lm
- Fixed minor bugs in regression tests
- Updated documentation

B.8 Version 5.21

- Addition of interpolate-lm
- Added LM filtering to compile-lm
- Improved regression tests
- Integration of interpolated LMs in Moses
- Extended tests on compilers and platforms
- Improved documentation with website

B.9 Version 5.22

- Use of AutoConf/AutoMake toolkit compilation and installation