IRST Language Modeling Toolkit
Version 5.60.01
USER MANUAL

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

September 14, 2016

This manual together with source code, executables, examples and regression tests can be accessed in the official website of IRSTLM Toolkit:

http://hlt.fbk.eu/en/irstlm

1 Introduction

This manual illustrates the functionalities of the IRST Language Modeling (LM) toolkit. It should put you quickly in the condition of:

- extracting the dictionary from a corpus
- extracting n-gram statistics from it
- estimating n-gram LMs using different smoothing criteria
- estimating and handling gigantic LMs
- adapting a LM on specific application data
- saving a LM into a compact binary file
- pruning a LM
- reducing LM size through quantization
- querying a LM through a command or script

Among state-of-the-art n-gram smoothing techniques, the IRST LM toolkit features LM very efficient data structures to handle very large LMs and adaptation methods which can be effective when limited task-related data are available.

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.

[1]http://www.statmt.org/moses/

[2]http://www.speech.sri.com/projects/srilm

2 Installation

In order to install the IRSTLM on your machine, please perform the following steps.

2.1 Step 0: preparation of the configuration

```
$> sh regenerate-makefiles.sh [--force]
```

Warning: Run with the ”–force” parameter if you want to recreate all links to the autotools

2.2 Step 1: configuration of the compilation

```
$> ./configure [--prefix=/path/where/to/install]
                          [--enable-caching]
                          [--enable-profiling]
                          [--enable-doc]
                          [--disable-trace]
                          [--disable-debugging]
```

Run the following command to get more details on the compilation options

```
$> configure --help
```

2.3 Step 2: compilation

```
$> make clean
$> make
```

2.4 Step 3: Installation

```
$> make install
```

The IRSTLM library and commands are generated, respectively, under the directories `/path/where/to/install/lib` and `/path/where/to/install/bin`.
If enabled and PdfLatex is installed, this user manual (in pdf) is generated under the directory `/path/where/to/install/doc`.
Although caching is not enabled by default, it is highly recommended to activate through its compilation flag ”`--enable-caching`”.

2.5 Step 4: Environment Settings

Set the environment variable `IRSTLM` to `/path/where/to/install`.
Include the command directory `/path/where/to/install/bin` into your environment variable `PATH`.

3 Getting started

Environment Settings We assume that all steps for installation described in Section 2 have been performed correctly. In particular, the environment variable `IRSTLM` is set , and that environment variable `PATH` includes the command directory `IRSTLM/bin`.
Data sets used in the examples can be found in an archive you can download from the official website of IRSTLM toolkit.

Preparation of Training Data In order to estimate a Language Model, you first need to prepare your training corpus. The corpus just consists of a text. We assume that the text is already preprocessed according to the user needs; this means that lowercasing, uppercasing, tokenization, and any other text transformation has to be performed beforehand with other tools.
You can only decide whether you are interested that IRSTLM toolkit is aware of sentence boundaries, i.e. where a sentence starts and ends. Otherwise, the toolkit considers the corpus as one continuous stream of text, and does not identify sentence splits. The following script adds start and end symbols (`<s>` and `</s>`, respectively, which should be considered reserved symbols, that is used only as delimiters) to all sentences in your training corpus.

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

IRSTLM toolkit does not compute probabilities for cross-sentence n-grams, i.e. any n-gram including the pair `</s> <s>`.

Training our first LM We are now ready to estimate a 3-gram (trigram) LM by running the command:

```
$> tlm -tr="gunzip -c train.gz" -n=3 -lm=wb -te=test
```

which produces the output:

```
n=49984 LP=301734.5406 PP=418.4772517 OOVRate=0.05007602433
```

The output shows the number of words in the test set, the LM log-probability, the LM perplexity and the out-of-vocabulary rate of the test set.
If you need to train and test different language models on the same data, a more efficient way to proceed is to first create an n-gram table of the training data:

```
$> ngt -i="gunzip -c train.gz" -n=3 -o=train.www -b=yes
```

The command `ngt` reads an input text file, creates an n-gram table of specified size (`-n=3`), and saves it in binary format (`-b=yes`) into a specified output file.
Now, the LM can be estimated and evaluated more quickly:

```
$> tlm -tr=train.www -n=3 -lm=wb -te=test
```

Once estimated, a LM can be also saved in the standard ARPA text format:

```
$> tlm -tr=train.www -n=3 -lm=wb  -o=train.lm
```

or in a binary format

```
$> tlm -tr=train.www -n=3 -lm=wb  -obin=train.blm
```

(Remark: the binary format formerly used by the IRST speech recognizer is still available, through the option `-oasr <filename>`, but is no more supported.)

4 More on ngt

Given a text corpus we can compute its dictionary and word frequencies with the command:

```
$> dict -i="gunzip -c train.gz" -o=train.dict -f=yes
```

For speech recognition applications, it can be often the case to limit the LM dictionary only to the top frequent, let us say, 10K words. We can obtain such a list by:

```
$> dict -i="gunzip -c train.gz" -o=top10k -pr=10000
```

Notice: the list will also include the start/end-sentence symbols.

An alternative pruning strategy is to filter out words occurring less or equal than a specified count. The following example removes the word occurring ≤ 5 times and keeps the top frequent 10K (at most) of the others:

```
$> dict -i="gunzip -c train.gz" -o=top10k5f -pr=10000 -pf=5
```

Statistics about the frequency of words inside a corpus can be gathered through the command `dict` with the option `-curve=yes`, while out-of-vocabulary rate statistics over a test set can be computed with the option `-TestFile=<sample>`. The following example illustrates both features:

```
$> dict -i="gunzip -c train.gz"  -Curve=yes -TestFile=test

**************** DICTIONARY GROWTH CURVE ****************
Freq    Entries Percent       Freq    OOV onTest
>0      15058    100.00%      <1        5.01%
>1      10113     67.16%      <2        7.64%
>2      8057      53.51%      <3        8.89%
>3      6948      46.14%      <4        9.96%
>4      6207      41.22%      <5       10.72%
>5      5644      37.48%      <6       11.67%
>6      5205      34.57%      <7       12.18%
>7      4823      32.03%      <8       12.72%
>8      4523      30.04%      <9       13.47%
>9      4224      28.05%      <10      14.10%
*********************************************************
```

A new n-gram table for the limited dictionary can be computed with `ngt` by specifying the sub-dictionary:

```
$> ngt -i=train.www -sd=top10k -n=3 -o=train.10k.www -b=yes
```

The command replaces all words outside top10K with the special out-of-vocabulary symbol `_unk_`.
Another useful feature of ngt is the merging of two n-gram tables. Assume that we have split our training corpus into files `text-a` and file `text-b` and have computed n-gram tables for both files, we can merge them with the option `-aug`:

```
$> ngt -i="gunzip -c text-a.gz" -n=3 -o=text-a.www -b=yes
$> ngt -i="gunzip -c text-b.gz" -n=3 -o=text-b.www -b=yes
$> ngt -i=text-a.www -aug=text-b.www -n=3 -o=text.www -b=yes
```

Warning: Note that if the concatenation of `text-a.gz` and `text-b.gz` is equal to `train.gz` the resulting n-gram tables `text.www` and `train.www` can slightly differ. This happens because during the construction of each single n-gram table few n-grams are automatically added to make it consistent for further computation.

5 More on tlm

Language models have to cope with out-of-vocabulary words, that is internally represented with the word class `_unk_`. In order to compare perplexity of LMs having different vocabulary size it is better to define a conventional dictionary size, or dictionary upper bound size, trough the parameter (`-dub`). In the following example, we compare the perplexity of the full vocabulary LM against the perplexity of the LM estimated over the more frequent 10K-words. In our comparison, we assume a dictionary upper bound of one million words.

```
$>tlm -tr=train.10k.www -n=3 -lm=wb -te=test -dub=1000000
  n=49984 LP=342160.8721 PP=939.5565162 OVVRate=0.07666453265

$>tlm -tr=train.www -n=3 -lm=wb -te=test -dub=1000000
  n=49984 LP=336276.7842 PP=835.2144716 OVVRate=0.05007602433
```

The large difference in perplexity between the two LMs is explained by the significantly higher OOV rate of the 10K-word LM.
N-gram LMs generally apply frequency smoothing techniques, and combine smoothed frequencies according to two main schemes: interpolation and back-off. The toolkit assumes interpolation as default. The back-off scheme is computationally more costly but often provides better performance. It can be activated with the option `-bo=yes`, e.g.:

```
$>tlm -tr=train.10k.www -n=3 -lm=wb -te=test -dub=1000000 -bo=yes
  n=49984 LP=337278.3227 PP=852.1186066 OVVRate=0.07666453265
```

This toolkit implements several frequency smoothing methods, which are specified by the parameter `-lm`. Three methods are particularly recommended:

a) **Modified shift-beta**, also known as "improved kneser-ney smoothing". This smoothing scheme gives top performance when training data is not very sparse but it is more time and memory consuming during the estimation phase:

```
$>tlm -tr=train.www -n=3 -lm=msb -te=test -dub=1000000 -bo=yes
  n=49984 LP=321877.3411 PP=626.1609806 OVVRate=0.05007602433
```

b) **Witten Bell smoothing**. This is an excellent smoothing method which works well in every data condition and is much less time and memory consuming:

```
$> tlm -tr=train.www -n=3 -lm=wb -te=test -dub=1000000  -bo=yes
  n=49984 LP=331577.2279 PP=760.2652095 OVVRate=0.05007602433
```

c) **Shift-beta smoothing**. This smoothing method is a simpler and cheaper version of the Modified shift-beta method and works sometimes better than Witten-Bell method:

```
$> tlm -tr=train.www -n=3 -lm=sb -te=test -dub=1000000  -bo=yes
  n=49984 LP=334724.5032 PP=809.6750442 OVVRate=0.05007602433
```

Moreover, the non linear smoothing parameter β can be specified with the option `-beta`:

```
$> tlm -tr=train.www -n=3 -lm=sb -beta=0.001 -te=test -dub=1000000
       -bo=yes
  n=49984 LP=449339.8282 PP=8019.836058 OVVRate=0.05007602433
```

This could be helpful in case we need to use language models with very limited frequency smoothing.

Limited Vocabulary

Using an n-gram table with a fixed or limited dictionary will cause some performance degradation, as LM smoothing statistics result slightly distorted. A valid alternative is to estimate the LM on the full dictionary of the training corpus and to use a limited dictionary just when saving the LM on a file. This can be achieved with the option `-d` (or `-dictionary`):

```
$> tlm -tr=train.www -n=3 -lm=msb -bo=y -te=test -o=train.lm -d=top10k
```

6 LM Adaptation

Language model adaptation can be applied when little training data is given for the task at hand, but much more data from other less related sources is available. `tlm` supports two adaptation methods.

6.1 Minimum Discriminative Information Adaptation

MDI adaptation is used when domain related data is very little but enough to estimate a unigram LM. Basically, the n-gram probs of a general purpose (background) LM are scaled so that they match the target unigram distribution.
Relevant parameters:

- `-ar=value`: the adaptation `rate`, a real number ranging from 0 (=no adaptation) to 1 (=strong adaptation).
- `-ad=file`: the adaptation file, either a text or a unigram table.
- `-ao=y`: open vocabulary mode, which must be set if the adaptation file might contain new words to be added to the basic dictionary.

As an example, we apply MDI adaptation on the “adapt” file:

```
$> tlm -tr=train.www -lm=wb -n=3 -te=test -dub=1000000 -ad=adapt -ar=0.8 -ao=yes
   n=49984 LP=326327.8053 PP=684.470312 OVVRate=0.04193341869
```

Warning: modified shift-beta smoothing cannot be applied in open vocabulary mode (`-ao=yes`). If this is the case, you should either change smoothing method or simply add the adaptation text to the background LM (use `-aug` parameter of `ngt`). In general, this solution should provide better performance.

```
$> ngt -i=train.www -aug=adapt -o=train-adapt.www -n=3 -b=yes
$> tlm -tr=train-adapt.www -lm=msb -n=3 -te=test -dub=1000000 -ad=adapt -ar=0.8
  n=49984 LP=312276.1746 PP=516.7311396 OVVRate=0.04193341869
```

6.2 Mixture Adaptation

Mixture adaptation is useful when you have enough training data to estimate a bigram or trigram LM and you also have data collections from other domains.
Relevant parameters:

- `-lm=mix` : specifies mixture smoothing method
- `-slmi=<filename>`: specifies filename with information about LMs to combine.

In the example directory, the file `sublmi` contains the following lines:

```
2
-slm=msb -str=adapt -sp=0
-slm=msb -str=train.www -sp=0
```

This means that we use train a mixture model on the `adapt` data set and combine it with the train data. For each data set the desired smoothing method is specified (disregard the parameter `-sp`). The file used for adaptation is the one in FIRST position.

```
$> tlm -tr=train.www -lm=mix -slmi=sublm -n=3 -te=test -dub=1000000
  n=49984 LP=307199.3273 PP=466.8244383 OVVRate=0.04193341869
```

Warning: for computational reasons it is expected that the n-gram table specified by `-tr` contains AT LEAST the n-grams of the last table specified in the slmi file, i.e. `train.www` in the example. Faster computations are achieved by putting the largest dataset as the last sub-model in the list and the union of all data sets as training file.
It is also IMPORTANT that a large `-dub` value is specified so that probabilities of sub-LMs can be correctly computed in case of out-of-vocabulary words.

7 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 discuss on LM file formats, on compiling LMs into a more compact and efficient binary format, and on querying LMs.

7.1 Estimating a LM with a Partial Dictionary

A sub-dictionary can be defined by just taking words occurring more than 5 times (`-pf=5`) and at most the top frequent 5000 words (`-pr=5000`):

```
$>dict -i="gunzip -c train.gz" -o=sdict -pr=5000 -pf=5
```

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  train.ilm.gz -k 5 -p -d sdict
```

Notice that all words outside the sub-dictionary will be mapped into 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).

8 LM File Formats

The 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.

8.1 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.

8.2 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`.

8.3 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.

8.4 Binary Formats

Both ARPA and qARPA formats can be converted into a binary format that allows for space savings on disk and a much quicker upload of the LM file. Binary versions can be created with the command `compile-lm`, that produces files with headers `blmt` or `Qblmt`.
Moreover, for an even faster access you can store the ngrams in an inverted order (see Section 11.1); the files with inverted-ordered ngrams have headers `blmtI` or `QblmtI`.

9 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. IRSTLM toolkit implements a method similar to the Weighted Difference Method described in the paper *Scalable Backoff Language Models* by Seymore and Rosenfeld.
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`:

10 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.

11 LM Compilation

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

```
$> compile-lm train.lm train.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 train.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.
By default, the command uses the directory "/tmp" for storing intermediate results. For huge LMs, the temporary files can grow dramatically causing a "disk full" system error. It is possible to explicitly set the directory used for temporary computation through the parameter "–tmpdir".

```
$> compile-lm --tmpdir=<mytmpdir> train.lm train.blm
```

11.1 Inverted order of ngrams

For a faster access, the ngrams can be stored in inverted order with the following two commands:

```
$> sort-lm.pl -inv -ilm train.lm -olm train.inv.lm
$> compile-lm train.inv.lm train.inv.blm --invert yes
```

Warning: The following pipeline is no more allowed!!

```
$> cat train.lm | sort-lm.pl -inv | \
   compile-lm /dev/stdin train.inv.blm --invert yes
```

12 Filtering a LM

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

```
$> compile-lm  train.lm --filter 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`.

13 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.

13.1 Perplexity Computation

Assume we have estimated and saved the following LM:

```
$> tlm -tr=train.www -n=3 -lm=wb -te=test -o=train.lm -ps=no
 n=49984 LP=308057.0419 PP=474.9041687 OVVRate=0.05007602433
```

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=38071 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=1019.69 PPwp=564.73 Nbo=39907 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.
Notice that `PPwp` reports the perplexity with a fixed dictionary upper-bound of 10 million words. Indeed:

```
$> tlm -tr=train.www -n=3 -lm=wb -te=test -o=train.lm -ps=no -dub=10000000
n=49984 LP=348396.8632 PP=1064.401254 OVVRate=0.05007602433
```

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`.

By enabling the option “`--sentence yes`”, `compile-lm` computes perplexity and related figures (OOV rate, number of backoffs, etc.) for each input sentence. The end of a sentence is identified by a given symbol (`</s>` by default).

```
$> compile-lm train.plm --eval test --dub 10000000 --sentence yes
```

```
%% sent_Nw=1 sent_PP=23.22 sent_PPwp=0.00 sent_Nbo=0 sent_Noov=0 sent_OOV=0.00%
%% sent_Nw=8 sent_PP=7489.50 sent_PPwp=7356.27 sent_Nbo=7 sent_Noov=2 sent_OOV=25.00%
%% sent_Nw=9 sent_PP=1231.44 sent_PPwp=0.00 sent_Nbo=14 sent_Noov=0 sent_OOV=0.00%
%% sent_Nw=6 sent_PP=27759.10 sent_PPwp=25867.42 sent_Nbo=19 sent_Noov=1 sent_OOV=16.67%
.....
%% sent_Nw=5 sent_PP=378.38 sent_PPwp=0.00 sent_Nbo=39893 sent_Noov=0 sent_OOV=0.00%
%% sent_Nw=15 sent_PP=4300.44 sent_PPwp=2831.89 sent_Nbo=39907 sent_Noov=1 sent_OOV=6.67%
%% Nw=49984 PP=1019.69 PPwp=564.73 Nbo=39907 Noov=2503 OOV=5.01%
```

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.

13.2 Probability Computations

Word-by-word log-probabilities can be computed as well from standard input with the command:

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

```
> </s>  1 p= NULL
> <s> <unk>     1 p= NULL
> <s> <unk> of  1 p= -3.530047e+00 bo= 2
> <unk> of the  1 p= -1.250668e+00 bo= 1
> of the senate 1 p= -1.170901e+01 bo= 1
> the senate (  1 p= -5.457265e+00 bo= 2
> senate ( <unk>        1 p= -2.166440e+01 bo= 2
....
....
```

the command 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.

Warning: All cross-sentence n-grams are skipped. The 1-grams with the sentence start symbol are also skipped. In a n-grams all words before the sentence start symbol are removed. For n-grams, whose size is smaller than the LM order, probability is not computed, but a `NULL` value is returned.

14 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.
Data used in those examples can be found in the directory `example/interpolateLM/`, which represents the relative path for all the parameters of the referred commands.
Interpolated LMs are 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.init` (with arbitrary weights), new weights can be estimated through Expectation-Maximization on some text sample `test` by running the command:

```
$> interpolate-lm lmlist.init --learn test
```

New weights will be written in the updated configuration file, called by default `lmlist.init.out`. You can also specify the name of the updated configuration file as follows:

```
$> interpolate-lm lmlist.init --learn test lmlist.final
```

Similarly to `compile-lm`, interpolated LMs can be queried through the option `--score`

```
$> interpolate-lm lmlist.final --score yes < test
```

and can return the perplexity of a given input text (“`--eval text-file`”), optionally at sentence level by enabling the option “`--sentence yes`”,

```
$> interpolate-lm lmlist.final --eval test
$> interpolate-lm lmlist.final --eval test --sentence yes
```

If there are binary LMs in the list, `interpolate-lm` can avoid to load them in memory through the memory mapping option `-memmap 1`.
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)
--sentence [yes|no] (compute perplexity at sentence level (identified
                    through the end symbol)
--memmap 1          use memory map to read a binary LM
```

15 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`):

```
$> 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 7, but some computation is performed in parallel (instead of sequentially) distributing the tasks on several machines.

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

16 Class and Chunk LMs

IRSTLM toolkit allows the use of class and chunk LMs, and a special handling of input tokens which are concatenation of $N \geq 1$ fields separated by the character #, e.g.

```
   word#lemma#part-of-speech#word-class
```

The processing is guided by the format of the file passed to Moses or `compile-lm`: if it contains just the LM, either in textual or binary format, it is treated as usual; otherwise, it is supposed to have the following format:

```
LMMACRO <lmmacroSize> <selectedField> <collapse>
<lmfilename>
<mapfilename>
```

where:

```
 LMMACRO is a reserved keyword
 <lmmacroSize> is a positive integer
 <selectedField> is an integer >=-1
 <collapse> is a boolean value (true, false)
 <lmfilename> is a file containing a LM (format compatible with IRSTLM)
 <mapfilename> is an (optional) file with a (one|many)-to-one map
```

The various cases are discussed with examples in the following. Data used in those examples can be found in the directory `example/chunkLM/` which represents the relative path for all the parameters of the referred commands. Note that texts with different tokens (words, POS, word#POS pairs...) used either as input or for training LMs are all derived from the same multifield texts in order to allow direct comparison of results.

16.1 Field selection

The simplest case is that of the LM in `<lmfilename>` referring just to one specific field of the input tokens. In this case, it is possible to specify the field to be selected before querying the LM through the integer `<selectedField>` (0 for the first filed, 1 for the second...). With the value -1, no selection is applied and the LM is queried with n-grams of whole strings. The other parameters are set as:

```
 <lmmacroSize> : set to the size of the LM in <lmfilename>
 <collapse>    : false
```

The third line optionally reserved to `<mapfilename>` does not exist.

Examples:

16.1.a) selection of the second field:

```
$> compile-lm --eval test/test.w-micro cfgfile/cfg.2ndfield
%% Nw=126 PP=2.68 PPwp=0.00 Nbo=0 Noov=0 OOV=0.00%
```

16.1.b) selection of the first field:

```
$> compile-lm --eval test/test.w-micro cfgfile/cfg.1stfield
%% Nw=126 PP=9.71 PPwp=0.00 Nbo=76 Noov=0 OOV=0.00%
```

The result of the latter case is identical to that obtained with the standard configuration involving just words:

16.1.c) usual case on words:

```
$> compile-lm --eval test/test.w lm/train.en.blm
%% Nw=126 PP=9.71 PPwp=0.00 Nbo=76 Noov=0 OOV=0.00%
```

16.2 Class LMs

Warning: In this context, it assumes an important role the parameter `<lmmacroSize>`: it defines the size of the n-gram before the collapsing operation, that is the number of microtags of the actually processed sequence. `<lmmacroSize>` should be large enough to ensure that after the collapsing operation, the resulting n-gram of chunks is at least of the size of the LM to be queried (the `<lmfilename>`). As an example, assuming `<lmmacroSize>=6`, `<selectedField>=1`, `<collapse>=true` and 3 the size of the chunk LM, the following input

```
 on#PP average#NP( 30#NP+ -#NP+ 40#NP+ cm#NP)
```

will yield to query the LM with just the bigram `(PP,NP)`, instead of a more informative trigram; for this particular case, the value 6 for `<lmmacroSize>` is not enough. On the other side, for efficiency reasons, it cannot be set to an unlimited valued. A reasonable value could derive from the average number of microtags per chunk (2-3), which means setting `<lmmacroSize>` to two-three times the size of the LM in `<lmfilename>`. Examples:

16.3.a) second field, micro→macro map, collapse:

```
$> compile-lm --eval test/test.w-micro cfgfile/cfg.2ndfld-map-cllps
%% Nw=126 PP=1.84 PPwp=0.00 Nbo=0 Noov=0 OOV=0.00%

$> compile-lm --eval test/test.w-micro cfgfile/cfg.2ndfld-map-cllps -d=1
%% Nw=126 PP=1.83774013 ... OOV=0.00% logPr=-33.29979642
```

16.3.b) whole token, micro→macro map, collapse:

```
$> compile-lm --eval test/test.micro cfgfile/cfg.token-map-cllps
%% Nw=126 PP=1.84 PPwp=0.00 Nbo=0 Noov=0 OOV=0.00%
```

16.3.c) whole token, micro→macro map, NO collapse:

```
$> compile-lm --eval test/test.micro cfgfile/cfg.token-map
%% Nw=126 PP=16.40 PPwp=0.00 Nbo=0 Noov=0 OOV=0.00%
```

Note that the configuration (16.3.c) gives the same result of that in example (16.2.b), as they are equivalent.

16.3.d) As an actual example related to the "warning" note reported above, the following configuration with usual LM:

```
$> compile-lm --eval test/test.chunk lm/train.macro.blm -d=1
Nw=73 PP=2.85754443 ... OOV=0.00000000% logPr=-33.28748842
```

not necessarily yields the same log-likelihood (`logPr`) nor the same perplexity (`PP`) of case (16.3.a). In fact, concerning `PP`, the length of the input sequence is definitely different (126 tokens before collapsing, 73 after that). Even the `logPr` is different (-33.29979642 vs. -33.28748842) because in (16.3.a) some 6-grams (`<lmmacroSize>` is set to 6) after collapsing reduce to n-grams of size less than 3 (the size of lm/train.macro.blm). By setting `<lmmacroSize>` to a larger value (e.g. 8), the same `logPr` will be computed.

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.

The following papers describe the IRST LM toolkit:

- Efficient data structures to handle huge language models:

  Marcello Federico and Mauro Cettolo, *Efficient Handling of N-gram Language Models for Statistical Machine Translation*, In Proc. of the Second Workshop on Statistical Machine Translation, pp. 88–95, ACL, Prague, Czech Republic, 2007.

- Language Model quantization:

  Marcello Federico and Nicola Bertoldi, *How Many Bits Are Needed To Store Probabilities for Phrase-Based Translation?*, In Proc. of the Workshop on Statistical Machine Translation. pp. 94-101, NAACL, New York City, NY, 2006.

- Language Model adaptation with mixtures:

  Marcello Federico and Nicola Bertoldi, *Broadcast news LM adaptation over time*, Computer Speech and Language. 18(4): pp. 417-435, October, 2004.

- Language Model adaptation with MDI:

  Marcello Federico, *Efficient LM Adaptation through MDI Estimation.* In Proc. of Eurospeech, Budapest, Hungary, 1999.

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

- Extended documentation with ShiftBeta smoothing.
- Smoothing parameter of ShiftBeta can be set manually.
- Robust handling for smoothing parameters of ModifiedShiftBeta.
- Fixed probability checks in TLM.
- 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

B.10 Version 5.30

- Support for a safe management of LMs with a total amount of n-grams larger than 250 million
- Use of a new parameter to specify a directory for temporary computation because the default ("/tmp") could be too small
- Improved a safer method of concatenation of gzipped sub lms
- Improved management of log files

B.11 Version 5.40

- Merging of internal-only tlm code into the public version
- Updated documentation into the public version
- Included documentation into the public version

B.12 Version 5.50

- **5.50.01**
  - binary saving directly with tlm
  - speed improvement through
    - caching of probability and states of ngrams in the LM interface
    - storing of ngrams in inverted order
- **5.50.02**
  - optional creation of documentation
  - improved documentation
  - optional computation of the perplexity at sentence-level

B.13 Version 5.60

- **5.60.01**
  - handling of class/chunk LMs with both compile-lm and interpolate-lm
  - improved pruning strategy to handle with sentence-start symbols
  - improved documentation and examples