MSRLM: a scalable language modeling toolkit

Patrick Nguyen, Jianfeng Gao, and Milind Mahajan
{panguyen,jfgao,milindm}@microsoft.com

November 2007

Technical Report
MSR-TR-2007-144

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052
http://www.research.microsoft.com

Abstract

MSRLM is the release of our internal language modeling tool chain used in Microsoft Research. It was used in our submission for NIST MT 2006. The main difference with other freely available tools is that it was designed to scale to large amounts of data. We successfully built a language model on high end hardware on 40 billion words of web data within 8 hours. It only supports a minimal set of features. Large gigaword language models may be consumed in a first pass machine translation decoding without further processing. This document describes the implementation and usage of the tools summarily.

It is our stated goal and hope that this release will be useful to the scientific community. The tool may not be used in a commercial product, or to build models used in a commercial product, or in for any commercial purpose. In addition, we require that you kindly cite this technical report when publishing results derived with this language model tool chain.

This describes the LM tool which is available as:
http://research.microsoft.com/research/downloads/details/78e26f9c-fc9a-44bb-80a7-69324c62df8c/details.aspx

Contents

1 General Description

In this section, we give a rough overview of the tool. Unlike other tools available publicly, our release has the specific charter of provding the ability to build relatively large language models. It may be used to build typically long-span language models on large amounts of data. It provides only the minimal set of features required, namely language model building and evaluation. Compared with other toolkits, we provide just the bare minimum of tools required to build and use language models. Higher level functionality, such as lattice rescoring, is out of the scope of this release.

Again, we provide tools for building n-gram language models on large amounts of data. Once built, we provide the ability to query the language model to get the conditional probability of a word given its history. Our language models are probability measures, i. e., summing up over all words and the unknown class given any history returns one. That is a useful property, for instance, for computing perplexity, and also for linearly *interpolating language models*. This is achieved by computing exact backoff weights. We typically build, store, and serve language models on a single high end machine. We use this tool routinely internally to build gigaword and web language models for machine translation, and we are confident that it will fit the needs of most users in that respect.

We have found the tools to be usually about six times faster than the publicly available CMU toolkit. They also exhibit fewer problems with high vocabulary sizes and may scale to large texts.

1.1 Introduction

Language model building using MSRLM follows a well-known pattern: it consists of collecting the vocabulary, counting up n-grams occuring in some text, and building a large trie. Our implementation follows typical implementations, save for a couple of critical twists, and parallelization points are similar with other LM toolkits. We were careful to use the simplest and more scalable algorithms whenever possible.

For historical reasons, our language model toolkit comes in two parts: lmapp and lmbld. Each part implements radically different data structures. lmapp is used for vocabulary and ngram collection, and modified absolute discounting. lmbld is used for Kneser-Ney smoothing.

1.2 Basic functionality

We provide a small set of core functionality listed below. Note that these tools were used to build language models on 40 billion words of web data, with almost no modification. Up to that amount of data, you should not have a problem with scalability of the tools. To achieve scalability, all of our tools operate on streams of data and output streams of data to the extent possible, and do not require random access to large memory buffers. of data whenever possible, The tools may be extended to larger data sets.

Vocabulary and word frequency: Given a text stream of space delimited words, we provide a list of words and corresponding frequencies in the stream. We ignore words longer than a given threshold. This was implemented with a naïve hash table and scales well up to several millions of words. We do not provide parallelization tools. The word frequency file may be used to truncate the vocabulary by selecting the top most occuring entries, or entries which occur more than a certain cutoff count. *We recommend that you sort the vocabulary file by decreasing occurrence count.*

Counting n-grams: Given a text stream, we collect joint occurrence counts of word sequences w_1^n. To that end, we use a memory buffer and temporary disk space when the buffer overflows. The resulting n-gram file is sorted. The text stream may be arbitrarily long. We provide the option of using compressed files when processing power exceeds disk speed overwhelmingly; that is the case on many-core hardware. When the vocabulary is sorted in lexicographical order, these count files will be compatible with the CMU SLM toolkit.

Merging n-gram count files: When collected separately, sorted count files may be merged. We use a hierarchical binary merge-sort which scales in the log of the number of input count files. We use multiple threads. We recommend using on-the-fly merging on compressed n-gram files on multi-core hardware while building language models. A separate merging step is required when the amount of open file handles allowed by the operating system is exceeded.

Building n-grams: Building n-grams requires a vocabulary file, cutoff counts, and a single sorted n-gram count stream. (Lower order n-gram count streams are computed on the fly.)

TCP serving protocol: The best way to interact with our language models from a computer farm is to serve up a central language model and connect to it with the distributed applications. In our current configuration, we typically use one quad-core server which can serve 80 first-pass machine translation processes, at roughly 40% of the CPU capacity on the server.

Smoothing: We provide two smoothing methods: modified absolute discounting (MAD), and Kneser-Ney (KN). KN typically gives superior results, but MAD normally converges to the same performance on large amounts of data. MAD is more scalable.

Perplexity computation: We provide sample code to compute perplexity.

IO format: We made no attempt to optimize the data structure on disk, e.g., quantize counts. Our language models are larger than they could be. There should be no coding artifacts, however.

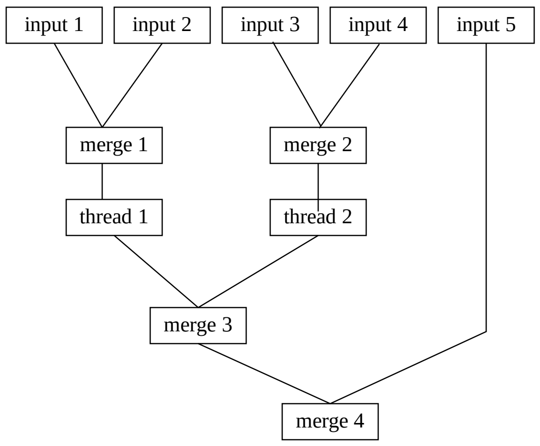

Figure 1: Multi-threaded hierarchical mergesort

2 Technical description

In this section, we give a technical description of the key changes that were implemented in our model.

2.1 Algorithms

Most of our algorithms typically scale up linearly in the amount of data and n-gram order.

Multithread hierarchical mergesort: The architecture of the mergesort is shown on Figure 1. We found this specific emobdiment to work well. It is especially suited when disk I/O is fast. In practice one thread is spawn for each three levels (8-way merge). Multi-threading has not been ported to GCC.

Backsorted trie: Used in Modified Absolute Discounting, the use of a backsorted trie allows us to have maximum scalability at minimal cost. The backsorted trie is shown on Figure 2. We read from a stream of n-grams and write sequentially to $n-1$ output tapes. This includes backoff calculation, which, obviously, is the tricky part.

n-grams are in so-called backsorted order, that is, for w_1^n, the first sort key is w_{n-1}, then w_{n-2}, etc until w_1, and lastly w_n. Therefore, all w_n which have a given w_1^{n-1} will appear contiguously in the file. Moreover, it is trivial to get

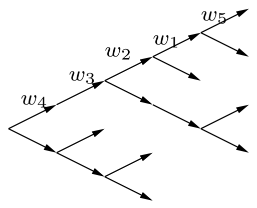

Figure 2: Backsorted trie organization: the last predicted word is at the leaf, otherwise the history is in reverse order

the marginalized counts $c(w_1^{n-1}) = \sum_{w_n} c(w_1^n)$. Going backwards in the n-gram order, we may marginalize both on conditional of a weaker history $c(w_k^n) = \sum_{w_{k-1}} c(w_{k-1}^n)$, and on a the history itself $c(w_k^{n-1}) = \sum_{w_{k-1}} c(w_{k-1}^{n-1})$. Consider the backoff formula at a level k away from the leaf order n:

$$\beta = \frac{1 - \sum_{w_n \in \Omega(w_k^{n-1})} \frac{\tilde{c}(w_k^n)}{c(w_k^{n-1})}}{1 - \sum_{w_n \in \Omega(w_k^{n-1})} \frac{\tilde{c}(w_{k+1}^n)}{c(w_{k+1}^{n-1})}}. \tag{1}$$

Numerators, which count up to w_n, are collected up in memory and accumulated backwards from the farthest node away from the root. Denominators, which count up to w_{n-1}, are accumulated from the n-gram stream. Therefore, the maximum memory required is of the order of bigrams.

Lookup cost in the backsorted trie: Let us now compare worst case lookup costs in the backsorted structure vs the forward sorted structure. Let us expand the Katz formula for lookup:

$$p(w_n|w_1^{n-1} = \begin{cases} \hat{p}(w_n|w_1^{n-1}) & \text{if } w_1^n \text{ is in the trie, else} \\ \beta(w_1^{\hat{n}-1}) \begin{cases} \hat{p}(w_n|w_2^{n-1}) & \text{if } w_2^n \text{ is in the trie, else} \\ \beta(w_2^{\hat{n}-1}) \begin{cases} \hat{p}(w_n|w_3^{n-1}) & \text{if } w_3^n \text{ is in the trie, else} \\ \ldots \end{cases} \end{cases} \end{cases} \tag{2}$$

and:

$$\hat{\beta}(w_1^n) = \begin{cases} \beta(w_1^k) & \text{if } w_1^k \text{ is in the trie, else} \\ & 1. \end{cases} \tag{3}$$

Let us define $\hat{k}$ and $\hat{h}$, the highest order k for which there exists a $p(w_n|w_k^n)$ and $\beta(w_k^{n-1})$ respectively. The conditional probability is:

$$p(w_1^n) = \left(\prod_{k=\hat{h}}^{n-1} \beta(w_k^{n-1})\right)\hat{p}(w_n|w_{\hat{k}}^{n-1}). \tag{4}$$

Note that we are guaranteed that all $\beta(w_k^{n-1})$ exist for $k \geq \hat{h}$. Also, we know that $\hat{h} + \hat{k} \leq n$.

The cost of looking up entries is dominated by how many times we have to find a conditional word entry given a history. This is tpyically done with a binary search. We use a slightly different variation. Instead, we use the fact that word IDs are sorted in decreasing order of unigram frequency. We assume that this is correlated with conditional probability given any history. First, we start reading a few entries and search linearly for a few entries. If found, this will bypass random access to successors, and will also make it faster for words will low word IDs, and slower for all others. Then, we use a biased binary search where we do not cut each interval in half, but rather, make the lower half (associated to lower word IDs) smaller than high half. This will make looking for words with lower IDs faster, and finding higher word IDs slower. In addition, when given a sorted n-gram array, we share the common prefix or suffix to avoid lookup twice.

Let us describe how looking up probabilities naïvely in the forward trie structure. First, we start looking for w_1^n, performing $n-1$ binary lookups in the worst case, the last one of which fails to find w_n. (In practice, unigrams are indexed directly.) After the search, we would have collected $\beta(w_1^{n-1})$ if present. Then, we weaken the history to w_2^{n-1}, and perform $n-2$ searches, starting from w_2 onwards. So, in the worst case, we have collected the backoff weights, and performed $\frac{(n-1)(n-2)}{2}$, or $O(n^2)$.

In the backsorted trie, we pursue two search branches. Let us first assume that we have built a backsorted trie of order $n+1$. This may be done with the same code and setting infinite cutoffs for order $n+1$. We start with w_{n-1} and perform exactly $n-\hat{k}$ searches, by successively strengthening the history. We will then have collected $c(w_n|w_{\hat{k}}^n)$. Then, we must find $c(w_{\hat{k}}^{n-1})$. This is guaranteed to be in the trie, and it is found with $n-\hat{k}-1$ searches. Up to now, we have performed exactly $n-2\hat{k}-1$ searches. At that stage, we are at the node associated with $w_{\hat{k}}^n$. We need to find the backoff history, *starting from that point* upwards in the trie. There are at most $\hat{k}$ because we have backed off $\hat{k}$ times. In other words, in the second branch of searches ending at w_{n-1}, we may not go down more than $n-1$ times. Therefore, in the worst case, we have performed $O(2n)$. Therefore, searching in the backsorted trie becomes $O(n)$ faster than in the forward trie in the worst case. For $n = 5$ they should be roughly equal. In practice, we found the backsorted version to be significantly faster.

Why the forward sorted trie is not feasible: Building language models using a forward trie is done by induction, by building the unigram structure, then the bigram structure, then trigram, etc. Consider the problem of building an ngram level n when the $n-1$ structure was built. Again, the problem lies in backoff calculation. The problem is that the numerator and denominator in eq. (1) may not be both available at the same time. Note that the summation has to be done over the seen mass of w_1^{n-1}. While building all histories under the w_1 branch, A pointer on the w_2 starts in the beginning of the $(n-1)$-gram structure and is incremented. At the end of processing w_1, this pointer will be at the end of the lower order structure. If there are V words in the vocabulary, the $(n-1)$-gram structure must be traversed sequentially V times, that is, every time we get a new w_1. To fix ideas, V is typically of the order of 10^4 to 10^7. For large texts, if the n-gram structure does not fit entirely into memory, this becomes quickly prohibitively slow.

2.2 Network protocol

Language models are typically large. To ensure best performance for first pass decoding, it is best to offload them on a machine different from the ones which runs the decoding processes. In practice, during NIST evaluations, we use a single server machine for our Gigaword language models.

To that end, we use a TCP/IP network layer to access language models remotely. To minimize network traffic, the server supports upstream and downstream compression. The protocol is depicted on Figure 3. Each client prepares a bulk of ngram entries for which it needs conditional probabilities. Each ngram has a length (number of words in the history plus one), and the list of words in the history followed by the word for which the conditional probability is requested. Then, the client:

1. Sorts in appropriate order (forward order for KN, backward for MAD), to enable server optimizations and improve compression ratio.
2. Decides if it wants to compress its request. If not, it just sends an int32 indicating the size in bytes of the bulk request, then sends bytes immediately thereafter. If it decides to compress:
    (a) It sends the negated uncompressed byte length, announcing compression and how many bytes need to be allocated in the srever.

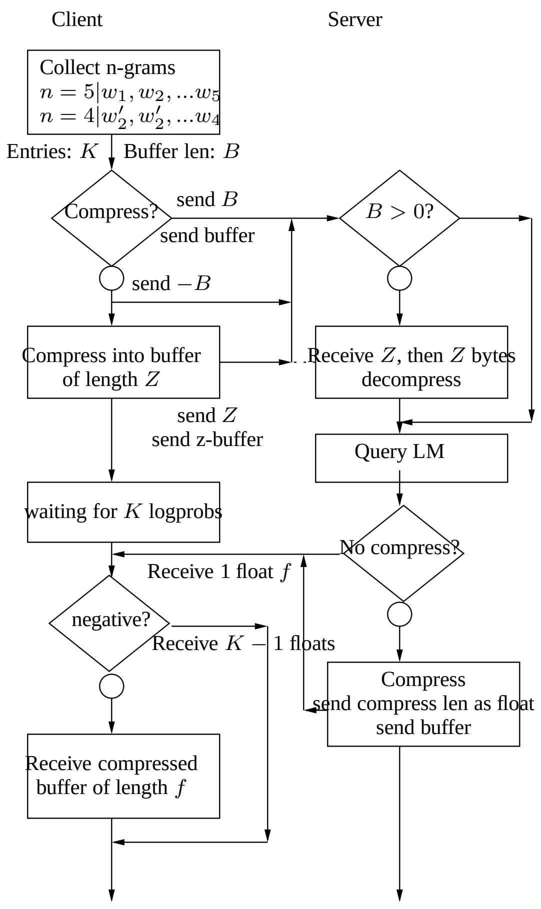

Figure 3: Flow of the network protocol

   (b) It compresses its bulk request as one batch. It then sends an int32 indicating the length of the compressed buffer, then the compressed buffer follows.

3. The server is expected to return as many probabilities as there were ngrams present in the bulk request. Both client and server know what this number is and it needs not be transmitted.

4. The server may decide to compress its output. If not compressing, it will just send log probabilities in IEEE single precision (float) format.

5. If compressing, it will compress its reply, and send the length of its output in bytes, as a single precision float number. A positive float number (> 1) therefore announces compressed output, since log probabilities *may not be greater than one*.

Another simpler protocol was implemented. In that case, unicode strings are sent. The number of ngram is sent as a 32bit integer. Then each word is sent as a 32bit integer representing its length including zero terminator, then the string including the zero terminator. It is the responsibility of the client and servers to agree *a priori* which protocol should be used.

3 Tool reference

3.1 lmapp: generic options

lmapp welcomes generic options listed in Table 1. These options are available by prefixing a double-dash, for instance:

```
lmapp --printcfg true
```

will print some compilation configuration information and exit the program. Standard I/O is turned off by default to facilitate debugging output. For best performance while piping, for instance an ngram counting to a compression program, it is better to turn that option off. Binary stdio is turned off by default. While piping text containing end of line characters, the operating system might decide to translate them and prepend a carriage return at each occurence. This would corrupt binary files. Also, the operating system might declare an end of stream if it encounters either ˆZ or ˆD. This special processing may be explicitly turned off by setting this to true. All options are found in `lmapp/ngcore/cfggen.xml`.

Option	Type	default	Description
task	Task	LMCOUNT	What to do, cf Table 2
printcfg	bool	false	Print configuration info and exit.
crtdbgbrk	bool	false	(In VS debug builds only) Raise exception to inter debugger
bufstdio	bool	false	Allow default buffering of standard input and output
binstdio	bool	false	Change mode of stdio to binary, e.g., turns off ˆM and ˆZ processing.
include	string	`""`	include a configuration file

Table 1: lmapp: generic options

Include files. For common configuration options to be shared across multiple commands, it is sometimes convenient to set options in a file, and include that file as a configuration. For instance, the previous example may be achieved by creating a file, say `a.cfg`, and calling:

```
lmapp --include a.cfg
```

The file `a.cfg` would contain:

```
printcfg true
```

Specifying arrays. lmapp sometimes allows arrays to be specified from the command line. For instance, cutoff frequencies are expected to be an array of integers. The name of the option is `--app.lmbuild.coff`. There are three was of specifying cuttoffs of, say, "0 2 4". The first would be to specify a space delimited string as a *single* argument, as:

```
  lmapp --app.lmbuild.coff "0 2 4"
```

The second way would be to specify each as an option:

```
  lmapp --app.lmbuild.coff0 0 --app.lmbuild.coff1 2 --app.lmbuild.coff2 4
```

The last way is to create a file, say, `coff.txt`, and call:

```
  lmapp --app.lmbuild.coff @coff.txt
```

The file `coff.txt` would contain:

```
0
2
4
```

Value	Description
LMWFREQ	Collect word frequencies
LMWFREQMERGE	Rarely used. Merge word freqs
LMCOUNT	Collecting n-gram counts
LMCOUNTMERGE	Merging multiple count files
LMBUILD	Build backsorted MAD LM
LMSERVER	Serving backsorted MAD LM
LMEVAL	Eval probabilities and perplexity

Table 2: Task types for lmapp's `--task` option.

Option	Type	Default	Description
text	string	-	filename of text or '-' for stdin
maxwlen	size_t	80	maximum word length

Table 3: Options prefixed by `--app.wfreq.`

Special filenames. There is a special filename defined as the dash character (`-`). When used in a write context, it means standard output, and when used in a read context, it means standard input.

3.2 lmapp: vocabulary collection

The first step in building language models consists of vocabulary collection. The vocabulary is the finite length list of words which may predicted by the language model. That is done by applying some strategy to a word frequency file. The strategy is to either take the first V words which have highest frequencies, or to take all words which we have seen more than C times. We do not provide tools to implement this policy. The word frequency is collected with lmapp, as follows:

```
echo Some training text text here | \
   lmapp --task LMWFREQ --app.lmwfreq.wfreq - --bufstdio true > wfreq.txt
```

The vocabulary may be recovered as:

```
cat wfreq.txt | sort -nr -k2 | head -10 > vocab.txt
```

For reasons highlighted before, *we recommend that the words in the vocabulary be sorted in reverse frequency*. Options are shown in Table 3.

Option	Type	Default	Description
vocab	string		filename of vocabulary file
text	string	-	filename of text or '-' for stdin
order	int	5	order of count (n)
bufsiz	int	50	buffer memory size
backsort	bool	false	backsorted (for MAD)
count_file	string	full.ngc	output filename or '-' for stdout
padunk	bool	false	pad the input text with unk tokens (for small files)
temp	string	.	temp directory
tempbase	string	tmpcount	temp basename for tmp files (for parallel runs)
compress_tmp	bool	false	compress the temp count files
final_merge	bool	true	merge temp files before exiting

Table 4: Options prefixed by `--app.lmcount.`

3.3 ngram count collection

Once we have decided what the vocabulary is going to be, we will collect n-gram counts. Options are shown in Table 4. The most useful are vocab, text, order, bufsiz, backsort, and count_file. The vocabulary was produced in the last subsection. The text is the same text as before. The bufsiz specifies, in MB, how much temporary memory should be used to sort the ngram entries. It should be set to the largest available quantity available on the machine.

The option padunk is used to ensure that all lower-order ngram entries will be present by marginalizing, by inserting a number of tokens with UNK id at the end of the word stream. While collecting ngrams on the same machine, be sure to change either temp or tempbase to make sure that temp files will not get the identity.

The example command line is:

```
echo Some training text text here | \
lmapp --task LMCOUNT \
   --app.lmcount.vocab       vocab.txt \
   --app.lmcount.order       5 \
   --app.lmcount.bufsiz      128 \
   --app.lmcount.backsort    true \
   --app.lmcount.text        - \
   --app.lmcount.count_file  ngc.bin \
   --app.lmcount.padunk      true \
   --binstdio                true \
```

Option	Type	Default	Description
incount	string		backsorted count file
order	size_t	5	order of the ngram
lm	string		output lm (output)
disc	string		binary discount parameters (output)
vocab	string		vocabulary file
coff	vector(int)		vector of cutoff counts for all orders. First must be zero.
discbin	size_t	4	bins of counts for discount

Table 5: Options prefixed by `--app.lmbuild.`

```
   --bufstdio                true
```

It will produce a file called `ngc.bin`. Once this is done, we no longer need the input text.

3.4 lmapp: modified absolute discounting

The modified absolute discouting code is the more scalable way of building language models. Invoking LM building is done as follows:

```
lmapp --task LMBUILD \
  --app.lmbuild.order          5 \
  --app.lmbuild.vocab          vocab.txt \
  --app.lmbuild.incount        ngc.bin \
  --app.lmbuild.lm             lm \
  --app.lmbuild.disc           lm.disc \
  --app.lmbuild.coff           "0 1 1 2 2"
```

It will produce files `lm` and `lm.disc`. Notice that the count cutoffs are specified for all orders including unigrams, but for unigrams it must always be zero.

3.5 lmapp: serving the language model

The binary language model is now ready to be used by clients. To set up the language model on a server port, the following command line is a prototypical example:

```
lmapp --task LMSERVER \
  --app.lmserver.order    5           \
  --app.lmserver.lm       eg.blm      \
  --app.lmserver.disc     eg.blm.disc \
  --app.lmserver.vocab    eg.vocab    \
  --app.lmserver.binipc   true        \
  --app.lmserver.bulkipc  true        \
  --app.lmserver.port     9350
```

3.6 lmbld: creating count files

Because of its complex structure, Kneser-Ney requires multiple types of information, and backsorting will not help. Therefore, multiple lower-order count files, in forward and backward order, must be created before efficient and scalable creation of language models may be performed. First, a forward count file is created by lmapp:

```
echo Some training text text text here | \
lmapp --task LMCOUNT \
  --app.lmcount.vocab       vocab.txt \
  --app.lmcount.order       5 \
  --app.lmcount.bufsiz      128 \
  --app.lmcount.backsort    false \
  --app.lmcount.text        - \
  --app.lmcount.count_file  - \
  --app.lmcount.padunk      true \
  --binstdio                true \
  --bufstdio                true > n.id
```

This count file must be first sorted in backwards order (sorted by w_n, then w_{n-1}, etc up to w_1). This is done with:

```
lmbld --bsid -idngm n.id -bs_idmgm n.bs.id -n 5 -m 5 \
  -temp n.tmp. -buffer 128
```

Then, we produce a foward and backward count for each lower order counts by marginalizing:

```
J=5
for i in 5 4 3 2 1; do
  lmbld --ngm2mgm -ngram n.bs.$J.id -mgram n.bs.$i.id -n $J \
            -m $i -vocab vocab.txt -u;
  lmbld --bsid -idngm n.bs.$i.id -bs_idmgm n.fs.bs.$i.id -n $$i \
            -m $i -temp n.tmp. -buffer 128;
  J=$i;
done
```

Note that filenames follow a convention which is required by lmbld. The first command in the loop marginalizes down, and counts types instead of tokens (i.e. unique histories). The second command resorts in backward order. This step is the dominating step for language model building with Kneser-Ney. In general, KN LMs may not be built as efficiently as MAD language models. They exhibit significantly superior performance when the size of the training text is reduced.

3.7 lmbld: Knesery-Ney

Finally, the KN language model may be built:

```
lmbld --bld -idngm n.5.id -lm lm.bin -voc vocab.txt -n 5 -temp n. -ex
```

We also provide a simpler, faster method available for smaller sizes of training data. We can describe it upon request, otherwise, please read the code, when the option "-ex" is removed. lmbld does not support cutoffs.

4 Running the tools

4.1 Compiling the code

Under Visual Studio There is one solution file for each of lmapp and lmbld. Compiling the code from each of these solution files will produce binaries located in the bin directory, in a subdirectory called either Release or Debug, and in that directory another subdirectory called x64 or win32, depending whether a 32bit version or 64bit version was requested. 32bit language models are limited to a file size of 2GB, memory limits (e.g. counting) of 2GB, and generally 32bit counts. Changing the cfggen.xml will not rebuild the source files: in that cae, calling cfggen.pl must be done by hand.

Makefile and GCC Each source directory contains a Makefile. We have ported the software to a recent version of GCC. Multi-threading was not ported to GCC. We have not tested the software thoroughly in that configuration, although it passes the regression test provided in the "recipe" directory.

4.2 Running the recipe

We have provided a simple regression test in the directory called recipe. Again, a Makefile builds the language models, runs the server, and tests perplexity. The source code is used as sample text.

Preamble The preamble contains definitions. Some are configurable. For instance, it is possible to switch between 64bit and 32bit by changing the "DARCH" definition. Definitions may be overriden on the fly, for instance:

```
make DMODE=Debug X_TXT="cat essay.txt|\
  perl -wnle ’print \"<s> $_ </s>\"’"
```

will instead use the debug version of the tools, and use essay.txt as the training text.

Mnemonic rules and xall To run the recipe end to end, please use the xall rule. The dependency graph is explicitly described by the makefile, so as to assist the user in understand what input and outputs are produced by each command, and to aid readability. Therefore making the eval_present rule is equivalent. For ease of understanding, however, we have provided mnemonics to break down the process into sub-steps described above. The mnemonics are vocab, count, lm, and eval_present. Vocabulary collection is the same for KN smoothing and MAD. Counting in MAD is done in backsorted mode, whereas counting in KN is done in forward mode, and then all lower-order backsorted ngram count files are generated. LM building and perplexity test are duplicated for each branch. For instance, the language model building is achieved by the "lm" rule, and results in eg.blm, and eg.lm being produced.

For convenience, we have provided read-text and read-test-text as examples. Also, the "clean" rule will kill the server and delete files to come back to the clean, released state.

make xall. Running make xall in the recipe directory will produce all intermediate eg.* files. It will launch the server and run the perplexity tests as well. To detail what each step would do, please use the make -n dry run capability, for instance,

```
make -n eg.lm
```

will show what commands have to be launched to build a MAD language model, from the current state. If starting from the clean state, it will show what needs to be built from scratch. If the command make eg.ngc was issued (successfully) immediately preceding, then it will show a single command.

5 Conclusion

We are delighted to be able to make this tool available and sincerly hope that it will be useful to the scientific community. We have done our best to document the tool, but we are understand that this documentation may be improved. Such as it is, the release is a “research” prototype, it is not intended – and may not – be used in a production environment. If you become aware of deficiencies in this release, or have any comment, please let us know. If there is a pressing need to extend the toolkit expressed by the community, we will do our best to address it.