Scalable Modified Kneser-Ney Language Model Estimation

Kenneth Heafield[*,†] **Ivan Pouzyrevsky**[‡] **Jonathan H. Clark**[†] **Philipp Koehn**[*]

[*]University of Edinburgh
10 Crichton Street
Edinburgh EH8 9AB, UK

[†]Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213, USA

[‡]Yandex
Zelenograd, bld. 455 fl. 128
Moscow 124498, Russia

heafield@cs.cmu.edu ivan.pouzyrevsky@gmail.com jhclark@cs.cmu.edu pkoehn@inf.ed.ac.uk

Abstract

We present an efficient algorithm to estimate large modified Kneser-Ney models including interpolation. Streaming and sorting enables the algorithm to scale to much larger models by using a fixed amount of RAM and variable amount of disk. Using one machine with 140 GB RAM for 2.8 days, we built an unpruned model on 126 billion tokens. Machine translation experiments with this model show improvement of 0.8 BLEU point over constrained systems for the 2013 Workshop on Machine Translation task in three language pairs. Our algorithm is also faster for small models: we estimated a model on 302 million tokens using 7.7% of the RAM and 14.0% of the wall time taken by SRILM. The code is open source as part of KenLM.

1 Introduction

Relatively low perplexity has made modified Kneser-Ney smoothing (Kneser and Ney, 1995; Chen and Goodman, 1998) a popular choice for language modeling. However, existing estimation methods require either large amounts of RAM (Stolcke, 2002) or machines (Brants et al., 2007). As a result, practitioners have chosen to use less data (Callison-Burch et al., 2012) or simpler smoothing methods (Brants et al., 2007).

Backoff-smoothed n-gram language models (Katz, 1987) assign probability to a word w_n in context w_1^{n-1} according to the recursive equation

$$p(w_n|w_1^{n-1}) = \begin{cases} p(w_n|w_1^{n-1}), \text{ if } w_1^n \text{ was seen} \\ b(w_1^{n-1})p(w_n|w_2^n), \text{ otherwise} \end{cases}$$

The task is to estimate probability p and backoff b from text for each seen entry w_1^n. This paper contributes an efficient multi-pass streaming algorithm using disk and a user-specified amount of RAM.

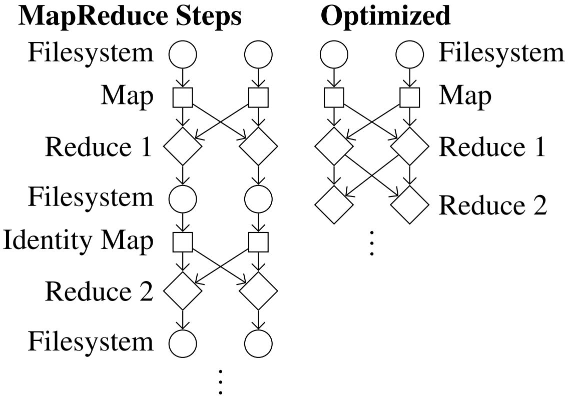

Figure 1: Each MapReduce performs three copies over the network when only one is required. Arrows denote copies over the network (i.e. to and from a *distributed* filesystem). Both options use local disk within each reducer for merge sort.

2 Related Work

Brants et al. (2007) showed how to estimate Kneser-Ney models with a series of five MapReduces (Dean and Ghemawat, 2004). On 31 billion words, estimation took 400 machines for two days. Recently, Google estimated a pruned Kneser-Ney model on 230 billion words (Chelba and Schalkwyk, 2013), though no cost was provided.

Each MapReduce consists of one layer of mappers and an optional layer of reducers. Mappers read from a network filesystem, perform optional processing, and route data to reducers. Reducers process input and write to a network filesystem. Ideally, reducers would send data directly to another layer of reducers, but this is not supported. Their workaround, a series of MapReduces, performs unnecessary copies over the network (Figure 1). In both cases, reducers use local disk.

Writing and reading from the distributed filesystem improves fault tolerance. However, the same level of fault tolerance could be achieved by checkpointing to the network filesystem then only reading in the case of failures. Doing so would enable reducers to start processing without waiting for the network filesystem to write all the data.

Our code currently runs on a single machine while MapReduce targets clusters. Appuswamy et al. (2013) identify several problems with the scale-out approach of distributed computation and put forward several scenarios in which a single machine scale-up approach is more cost effective in terms of both raw performance and performance per dollar.

Brants et al. (2007) contributed Stupid Backoff, a simpler form of smoothing calculated at runtime from counts. With Stupid Backoff, they scaled to 1.8 trillion tokens. We agree that Stupid Backoff is cheaper to estimate, but contend that this work makes Kneser-Ney smoothing cheap enough.

Another advantage of Stupid Backoff has been that it stores one value, a count, per n-gram instead of probability and backoff. In previous work (Heafield et al., 2012), we showed how to collapse probability and backoff into a single value without changing sentence-level probabilities. However, local scores do change and, like Stupid Backoff, are no longer probabilities.

MSRLM (Nguyen et al., 2007) aims to scalably estimate language models on a single machine. Counting is performed with streaming algorithms similarly to this work. Their parallel merge sort also has the potential to be faster than ours. The biggest difference is that their pipeline delays some computation (part of normalization and all of interpolation) until query time. This means that it cannot produce a standard ARPA file and that more time and memory are required at query time. Moreover, they use memory mapping on entire files and these files may be larger than physical RAM. We have found that, even with mostly-sequential access, memory mapping is slower because the kernel does not explicitly know where to read ahead or write behind. In contrast, we use dedicated threads for reading and writing. Performance comparisons are omitted because we were unable to compile and run MSRLM on recent versions of Linux.

SRILM (Stolcke, 2002) estimates modified Kneser-Ney models by storing n-grams in RAM. It also offers a disk-based pipeline for initial steps (i.e. counting). However, the later steps store all n-grams that survived count pruning in RAM. Without pruning, both options use the same RAM.

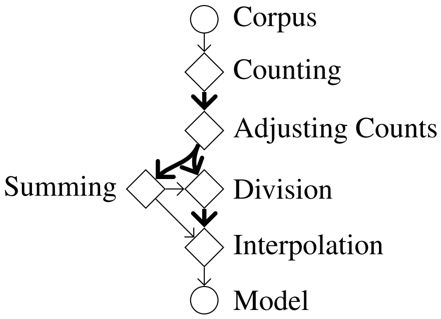

Figure 2: Data flow in the estimation pipeline. Normalization has two threads per order: summing and division. Thick arrows indicate sorting.

IRSTLM (Federico et al., 2008) does not implement modified Kneser-Ney but rather an approximation dubbed “improved Kneser-Ney” (or “modified shift-beta” depending on the version). Estimation is done in RAM. It can also split the corpus into pieces and separately build each piece, introducing further approximation.

3 Estimation Pipeline

Estimation has four streaming passes: counting, adjusting counts, normalization, and interpolation. Data is sorted between passes, three times in total. Figure 2 shows the flow of data.

3.1 Counting

For a language model of order N, this step counts all N-grams (with length exactly N) by streaming through the corpus. Words near the beginning of sentence also form N-grams padded by the marker <s> (possibly repeated multiple times). The end of sentence marker </s> is appended to each sentence and acts like a normal token.

Unpruned N-gram counts are sufficient, so lower-order n-grams ($n < N$) are not counted. Even pruned models require unpruned N-gram counts to compute smoothing statistics.

Vocabulary mapping is done with a hash table.[1] Token strings are written to disk and a 64-bit Mur-

[1] This hash table is the only part of the pipeline that can grow. Users can specify an estimated vocabulary size for memory budgeting. In future work, we plan to support local vocabularies with renumbering.

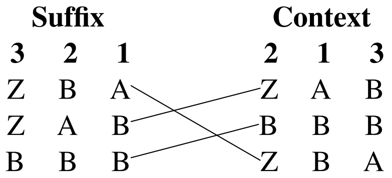

Figure 3: In suffix order, the last word is primary. In context order, the penultimate word is primary.

murHash[2] token identifier is retained in RAM.

Counts are combined in a hash table and spilled to disk when a fixed amount of memory is full. Merge sort also combines identical N-grams (Bitton and DeWitt, 1983).

3.2 Adjusting Counts

The counts c are replaced with adjusted counts a.

$$a(w_1^n) = \begin{cases} c(w_1^n), \text{ if } n = N \text{ or } w_1 = \text{<s>} \\ |v : c(vw_1^n) > 0|, \text{ otherwise} \end{cases}$$

Adjusted counts are computed by streaming through N-grams sorted in suffix order (Figure 3). The algorithm keeps a running total $a(w_i^N)$ for each i and compares consecutive N-grams to decide which adjusted counts to output or increment.

Smoothing statistics are also collected. For each length n, it collects the number $t_{n,k}$ of n-grams with adjusted count $k \in [1, 4]$.

$$t_{n,k} = |\{w_1^n : a(w_1^n) = k\}|$$

These are used to compute closed-form estimates (Chen and Goodman, 1998) of discounts $D_n(k)$

$$D_n(k) = k - \frac{(k+1)t_{n,1}t_{n,k+1}}{(t_{n,1} + 2t_{n,2})t_{n,k}}$$

for $k \in [1, 3]$. Other cases are $D_n(0) = 0$ and $D_n(k) = D_n(3)$ for $k \geq 3$. Less formally, counts 0 (unknown) through 2 have special discounts.

3.3 Normalization

Normalization computes pseudo probability u

$$u(w_n|w_1^{n-1}) = \frac{a(w_1^n) - D_n(a(w_1^n))}{\sum_x a(w_1^{n-1}x)}$$

and backoff b

$$b(w_1^{n-1}) = \frac{\sum_{i=1}^{3} D_n(i)|\{x : a(w_1^{n-1}x) = i\}|}{\sum_x a(w_1^{n-1}x)}$$

The difficulty lies in computing denominator $\sum_x a(w_1^{n-1}x)$ for all w_1^{n-1}. For this, we sort in context order (Figure 3) so that, for every w_1^{n-1}, the entries $w_1^{n-1}x$ are consecutive. One pass collects both the denominator and backoff[3] terms $|\{x : a(w_1^{n-1}x) = i\}|$ for $i \in [1, 3]$.

A problem arises in that denominator $\sum_x a(w_1^{n-1}x)$ is known only after streaming through all $w_1^{n-1}x$, but is needed immediately to compute each $u(w_n|w_1^{n-1})$. One option is to buffer in memory, taking $O(N|\text{vocabulary}|)$ space since each order is run independently in parallel. Instead, we use two threads for each order. The sum thread reads ahead to compute $\sum_x a(w_1^{n-1}x)$ and $b(w_1^{n-1})$ then places these in a secondary stream. The divide thread reads the input and the secondary stream then writes records of the form

$$(w_1^n, u(w_n|w_1^{n-1}), b(w_1^{n-1})) \qquad (1)$$

The secondary stream is short so that data read by the sum thread will likely be cached when read by the divide thread. This sort of optimization is not possible with most MapReduce implementations.

Because normalization streams through $w_1^{n-1}x$ in context order, the backoffs $b(w_1^{n-1})$ are computed in suffix order. This will be useful later (§3.5), so backoffs are written to secondary files (one for each order) as bare values without keys.

3.4 Interpolation

Chen and Goodman (1998) found that perplexity improves when the various orders within the same model are interpolated. The interpolation step computes final probability p according to the recursive equation

$$p(w_n|w_1^{n-1}) = u(w_n|w_1^{n-1}) + b(w_1^{n-1})p(w_n|w_2^{n-1}) \qquad (2)$$

Recursion terminates when unigrams are interpolated with the uniform distribution

$$p(w_n) = u(w_n) + b(\epsilon)\frac{1}{|\text{vocabulary}|}$$

where ϵ denotes the empty string. The unknown word counts as part of the vocabulary and has count zero,[4] so its probability is $b(\epsilon)/|\text{vocabulary}|$.

[2] https://code.google.com/p/smhasher/

[3] Sums and counts are done with exact integer arithmetic. Thus, every floating-point value generated by our toolkit is the result of $O(N)$ floating-point operations. SRILM has numerical precision issues because it uses $O(N|\text{vocabulary}|)$ floating-point operations to compute backoff.

[4] SRILM implements "another hack" that computes $p_{\text{SRILM}}(w_n) = u(w_n)$ and $p_{\text{SRILM}}(\text{<unk>}) = b(\epsilon)$ whenever $p(\text{<unk>}) < 3 \times 10^{-6}$, as it usually is. We implement both and suspect their motivation was numerical precision.

Probabilities are computed by streaming in suffix lexicographic order: w_n appears before w_{n-1}^n, which in turn appears before w_{n-2}^n. In this way, $p(w_n)$ is computed before it is needed to compute $p(w_n|w_{n-1})$, and so on. This is implemented by jointly iterating through N streams, one for each length of n-gram. The relevant pseudo probability $u(w_n|w_1^{n-1})$ and backoff $b(w_1^{n-1})$ appear in the input records (Equation 1).

3.5 Joining

The last task is to unite $b(w_1^n)$ computed in §3.3 with $p(w_n|w_1^{n-1})$ computed in §3.4 for storage in the model. We note that interpolation (Equation 2) used the different backoff $b(w_1^{n-1})$ and so $b(w_1^n)$ is not immediately available. However, the backoff values were saved in suffix order (§3.3) and interpolation produces probabilities in suffix order. During the same streaming pass as interpolation, we merge the two streams.[5] Suffix order is also convenient because the popular reverse trie data structure can be built in the same pass.[6]

4 Sorting

Much work has been done on efficient disk-based merge sort. Particularly important is arity, the number of blocks that are merged at once. Low arity leads to more passes while high arity incurs more disk seeks. Abello and Vitter (1999) modeled these costs and derived an optimal strategy: use fixed-size read buffers (one for each block being merged) and set arity to the number of buffers that fit in RAM. The optimal buffer size is hardware-dependent; we use 64 MB by default. To overcome the operating system limit on file handles, multiple blocks are stored in the same file.

To further reduce the costs of merge sort, we implemented pipelining (Dementiev et al., 2008). If there is enough RAM, input is lazily merged and streamed to the algorithm. Output is cut into blocks, sorted in the next step's desired order, and then written to disk. These optimizations eliminate up to two copies to disk if enough RAM is available. Input, the algorithm, block sorting, and output are all threads on a chain of producer-consumer queues. Therefore, computation and disk operations happen simultaneously.

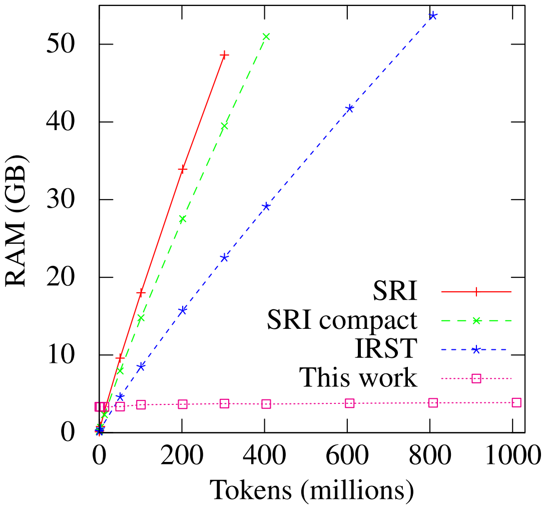

Figure 4: Peak virtual memory usage.

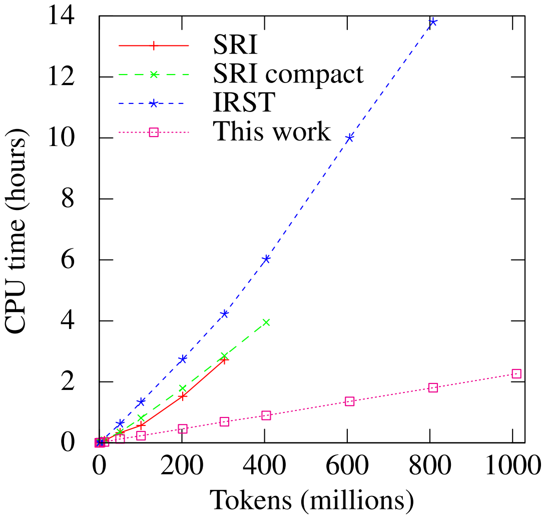

Figure 5: CPU usage (system plus user).

Each n-gram record is an array of n vocabulary identifiers (4 bytes each) and an 8-byte count or probability and backoff. At peak, records are stored twice on disk because lazy merge sort is not easily amenable to overwriting the input file. Additional costs are the secondary backoff file (4 bytes per backoff) and the vocabulary in plaintext.

5 Experiments

Experiments use ClueWeb09.[7] After spam filtering (Cormack et al., 2011), removing markup, selecting English, splitting sentences (Koehn, 2005), deduplicating, tokenizing (Koehn et al., 2007), and truecasing, 126 billion tokens remained.

[5] Backoffs only exist if the n-gram is the context of some $n+1$-gram, so merging skips n-grams that are not contexts.

[6] With quantization (Whittaker and Raj, 2001), the quantizer is trained in a first pass and applied in a second pass.

[7] http://lemurproject.org/clueweb09/

1	2	3	4	5
393	3,775	17,629	39,919	59,794

Table 1: Counts of unique n-grams (in millions) for the 5 orders in the large LM.

5.1 Estimation Comparison

We estimated unpruned language models in binary format on sentences randomly sampled from ClueWeb09. SRILM and IRSTLM were run until the test machine ran out of RAM (64 GB). For our code, the memory limit was set to 3.5 GB because larger limits did not improve performance on this small data. Results are in Figures 4 and 5. Our code used an average of 1.34–1.87 CPUs, so wall time is better than suggested in Figure 5 despite using disk. Other toolkits are single-threaded. SRILM's partial disk pipeline is not shown; it used the same RAM and took more time. IRSTLM's splitting approximation took 2.5 times as much CPU and about one-third the memory (for a 3-way split) compared with normal IRSTLM.

For 302 million tokens, our toolkit used 25.4% of SRILM's CPU time, 14.0% of the wall time, and 7.7% of the RAM. Compared with IRSTLM, our toolkit used 16.4% of the CPU time, 9.0% of the wall time, and 16.6% of the RAM.

5.2 Scaling

We built an unpruned model (Table 1) on 126 billion tokens. Estimation used a machine with 140 GB RAM and six hard drives in a RAID5 configuration (sustained read: 405 MB/s). It took 123 GB RAM, 2.8 days wall time, and 5.4 CPU days. A summary of Google's results from 2007 on different data and hardware appears in §2.

We then used this language model as an additional feature in unconstrained Czech-English, French-English, and Spanish-English submissions to the 2013 Workshop on Machine Translation.[8] Our baseline is the University of Edinburgh's phrase-based Moses (Koehn et al., 2007) submission (Durrani et al., 2013), which used all constrained data specified by the evaluation (7 billion tokens of English). It placed first by BLEU (Papineni et al., 2002) among constrained submissions in each language pair we consider.

Source	Baseline	Large
Czech	27.4	28.2
French	32.6	33.4
Spanish	31.8	32.6

Table 2: Uncased BLEU results from the 2013 Workshop on Machine Translation.

In order to translate, the large model was quantized (Whittaker and Raj, 2001) to 10 bits and compressed to 643 GB with KenLM (Heafield, 2011) then copied to a machine with 1 TB RAM. Better compression methods (Guthrie and Hepple, 2010; Talbot and Osborne, 2007) and distributed language models (Brants et al., 2007) could reduce hardware requirements. Feature weights were retuned with PRO (Hopkins and May, 2011) for Czech-English and batch MIRA (Cherry and Foster, 2012) for French-English and Spanish-English because these worked best for the baseline. Uncased BLEU scores on the 2013 test set are shown in Table 2. The improvement is remarkably consistent at 0.8 BLEU point in each language pair.

6 Conclusion

Our open-source (LGPL) estimation code is available from `kheafield.com/code/kenlm/` and should prove useful to the community. Sorting makes it scalable; efficient merge sort makes it fast. In future work, we plan to extend to the Common Crawl corpus and improve parallelism.

Acknowledgements

Miles Osborne preprocessed ClueWeb09. Mohammed Mediani contributed to early designs. Jianfeng Gao clarified how MSRLM operates. This work used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number OCI-1053575. We used Stampede and Trestles under allocation TG-CCR110017. System administrators from the Texas Advanced Computing Center (TACC) at The University of Texas at Austin made configuration changes on our request. This work made use of the resources provided by the Edinburgh Compute and Data Facility (`http://www.ecdf.ed.ac.uk/`). The ECDF is partially supported by the eDIKT initiative (`http://www.edikt.org.uk/`). The research leading to these results has received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement 287658 (EU BRIDGE).

[8] http://statmt.org/wmt13/

References

James M. Abello and Jeffrey Scott Vitter, editors. 1999. *External memory algorithms*. American Mathematical Society, Boston, MA, USA.

Raja Appuswamy, Christos Gkantsidis, Dushyanth Narayanan, Orion Hodson, and Antony Rowstron. 2013. Nobody ever got fired for buying a cluster. Technical Report MSR-TR-2013-2, Microsoft Research.

Dina Bitton and David J DeWitt. 1983. Duplicate record elimination in large data files. *ACM Transactions on database systems (TODS)*, 8(2):255–265.

Thorsten Brants, Ashok C. Popat, Peng Xu, Franz J. Och, and Jeffrey Dean. 2007. Large language models in machine translation. In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Language Learning*, pages 858–867, June.

Chris Callison-Burch, Philipp Koehn, Christof Monz, Matt Post, Radu Soricut, and Lucia Specia. 2012. Findings of the 2012 workshop on statistical machine translation. In *Proceedings of the Seventh Workshop on Statistical Machine Translation*, pages 10–51, Montréal, Canada, June. Association for Computational Linguistics.

Ciprian Chelba and Johan Schalkwyk, 2013. *Empirical Exploration of Language Modeling for the google.com Query Stream as Applied to Mobile Voice Search*, pages 197–229. Springer, New York.

Stanley Chen and Joshua Goodman. 1998. An empirical study of smoothing techniques for language modeling. Technical Report TR-10-98, Harvard University, August.

Colin Cherry and George Foster. 2012. Batch tuning strategies for statistical machine translation. In *Proceedings of the 2012 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 427–436. Association for Computational Linguistics.

Gordon V Cormack, Mark D Smucker, and Charles LA Clarke. 2011. Efficient and effective spam filtering and re-ranking for large web datasets. *Information retrieval*, 14(5):441–465.

Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified data processing on large clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, San Francisco, CA, USA, 12.

Roman Dementiev, Lutz Kettner, and Peter Sanders. 2008. STXXL: standard template library for XXL data sets. *Software: Practice and Experience*, 38(6):589–637.

Nadir Durrani, Barry Haddow, Kenneth Heafield, and Philipp Koehn. 2013. Edinburgh's machine translation systems for European language pairs. In *Proceedings of the ACL 2013 Eighth Workshop on Statistical Machine Translation*, Sofia, Bulgaria, August.

Marcello Federico, Nicola Bertoldi, and Mauro Cettolo. 2008. IRSTLM: an open source toolkit for handling large scale language models. In *Proceedings of Interspeech*, Brisbane, Australia.

David Guthrie and Mark Hepple. 2010. Storing the web in memory: Space efficient language models with constant time retrieval. In *Proceedings of EMNLP 2010*, Los Angeles, CA.

Kenneth Heafield, Philipp Koehn, and Alon Lavie. 2012. Language model rest costs and space-efficient storage. In *Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, Jeju Island, Korea.

Kenneth Heafield. 2011. KenLM: Faster and smaller language model queries. In *Proceedings of the Sixth Workshop on Statistical Machine Translation*, Edinburgh, UK, July. Association for Computational Linguistics.

Mark Hopkins and Jonathan May. 2011. Tuning as ranking. In *Proceedings of the 2011 Conference on Empirical Methods in Natural Language Processing*, pages 1352–1362, Edinburgh, Scotland, July.

Slava Katz. 1987. Estimation of probabilities from sparse data for the language model component of a speech recognizer. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, ASSP-35(3):400–401, March.

Reinhard Kneser and Hermann Ney. 1995. Improved backing-off for m-gram language modeling. In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 181–184.

Philipp Koehn, Hieu Hoang, Alexandra Birch, Chris Callison-Burch, Marcello Federico, Nicola Bertoldi, Brooke Cowan, Wade Shen, Christine Moran, Richard Zens, Chris Dyer, Ondřej Bojar, Alexandra Constantin, and Evan Herbst. 2007. Moses: Open source toolkit for statistical machine translation. In *Annual Meeting of the Association for Computational Linguistics (ACL)*, Prague, Czech Republic, June.

Philipp Koehn. 2005. Europarl: A parallel corpus for statistical machine translation. In *Proceedings of MT Summit*.

Patrick Nguyen, Jianfeng Gao, and Milind Mahajan. 2007. MSRLM: a scalable language modeling toolkit. Technical Report MSR-TR-2007-144, Microsoft Research.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. BLEU: A method for automatic evalution of machine translation. In *Proceedings 40th Annual Meeting of the Association for Computational Linguistics*, pages 311–318, Philadelphia, PA, July.

Andreas Stolcke. 2002. SRILM - an extensible language modeling toolkit. In *Proceedings of the Seventh International Conference on Spoken Language Processing*, pages 901–904.

David Talbot and Miles Osborne. 2007. Randomised language modelling for statistical machine translation. In *Proceedings of ACL*, pages 512–519, Prague, Czech Republic.

Edward Whittaker and Bhiksha Raj. 2001. Quantization-based language model compression. In *Proceedings of Eurospeech*, pages 33–36, Aalborg, Denmark, December.