Iterative Language Model Estimation: Efficient Data Structure & Algorithms

Bo-June (Paul) Hsu, James Glass

MIT Computer Science and Artificial Intelligence Laboratory
32 Vassar Street, Cambridge, MA 02139, USA
bohsu@mit.edu, glass@mit.edu

Abstract

Despite the availability of better performing techniques, most language models are trained using popular toolkits that do not support perplexity optimization. In this work, we present an efficient data structure and optimized algorithms specifically designed for iterative parameter tuning. With the resulting implementation, we demonstrate the feasibility and effectiveness of such iterative techniques in language model estimation.
Index Terms: language modeling, smoothing, interpolation

1. Introduction

For domains with limited matched training data, many of the most effective techniques for n-gram language model (LM) estimation, such as modified Kneser-Ney smoothing [1] and generalized linear interpolation [2], involve iterative parameter tuning to optimize the development set perplexity. However, due to the lack of support for performing such iterative LM estimation in popular language modeling toolkits, such as the SRI Language Modeling (SRILM) toolkit [3], most work in the field opts for simpler techniques with inferior results.

Previous work on data structures for n-gram models has primarily focused on runtime performance and storage compression [4]. With the availability of the Google Web 1T 5-gram corpus [5], recent research has examined efficient representations for building large-scale LMs [6, 7]. However, these efforts only support simple subpar smoothing and interpolation techniques, few that involve iterative parameter optimization.

In this work, we propose a data structure designed specifically for LM training algorithms with iterative parameter tuning. By taking advantage of the iterative nature of perplexity optimization, we present efficient algorithms for modified Kneser-Ney smoothing, linear interpolation, and perplexity evaluation. Using the resulting MIT Language Modeling (MITLM) toolkit, we demonstrate the efficiency of these iterative algorithms.

2. Data Structure

An n-gram LM represents the probability of a word sequence w_1^N as $p(w_1^N) = \prod_{i=1}^{N} p(w_i|w_1^{i-1}) \approx \prod_{i=1}^{N} p(w_i|h_i)$, where $h_i = w_{i-n+1}^{i-1}$ represents the history for word w_i and n is the model order. Since many n-grams are not observed in the training data, we can smooth the maximum likelihood estimate by distributing probabilities from seen to unseen n-grams and assigning probabilities proportional to the lower-order model to the unseen n-grams. For these cases, $p(w|h) = \alpha(h)p(w|h')$, where $\alpha(h)$ is the backoff weight and h' is the backoff history obtained by removing the earliest word from h. The resulting backoff n-gram LM is commonly represented in the ARPA text file format [8], which stores $p(w|h)$ and $\alpha(h)$ of the observed n-grams and their histories, as shown in Figure 1a.

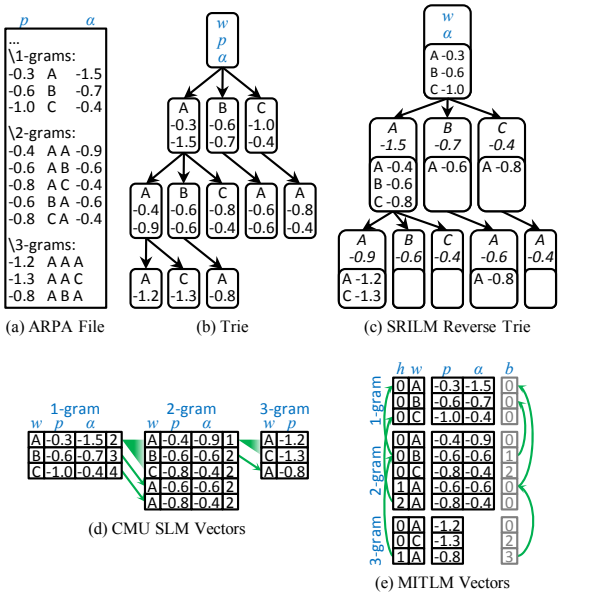

Figure 1: Various n-gram data structures. w: word, p: probability, α: backoff weight, h: history index, b: backoff index.

2.1. Existing Representations

While the ARPA file format serves as a standard cross-implementation representation for backoff n-gram language models, it is inefficient as a runtime data structure. A simple option is to represent the model as a trie, or prefix tree, where each n-gram node stores the word index w, conditional probability $p(w|h)$, and backoff weight $\alpha(h)$ (Figure 1b). To reduce the number of lookups needed to compute n-gram probabilities involving backoff, SRILM maintains the n-gram histories backwards within the trie and stores the conditional probabilities in a dictionary within each n-gram history node (Figure 1c). Although simple to build incrementally, tries demonstrate poor memory locality during node traversals. Instead, the CMU-Cambridge Statistical Language Modeling (CMU SLM) toolkit [9] utilizes a compact vector encoding of the n-gram trie [4] where pointers to the child nodes are encoded with array indices (Figure 1d). The resulting structure not only improves memory locality, but also reduces the memory footprint.

Operations	Examples / Description
$+, -, \times, \div, \cdot, \log, \exp, \sum$	$x = [1, 2, 3]; y = [4, 5, 6];$ $x + y \Rightarrow [5, 7, 9]; \sum(x) \Rightarrow 6$
Indexing	$x[0, 2] \Rightarrow [1, 3]; x[x > 2] \Rightarrow [3]$
$y = \text{bincount}(x)$	$y = \text{binweight}(x, 1)$
$y = \text{binweight}(x, w)$	for i in range(len(x)): $y[x[i]]$ += $w[i]$
$y = \text{binlookup}(x, t, d{=}0)$	for i in range(len(x)): $y[i]$ = ($0 \leq i <$ len(t)) ? $t[x[i]]$: d
$y = \text{binset}(x, m)$	for i in range(len(x)): if $m[i]$: $y[x[i]] = 1$

Figure 2: Common vector operations in LM estimation.

2.2. Ideal Properties

Existing data structures have been primarily designed for efficient n-gram probability evaluations at runtime. However, iterative LM estimation algorithms repeatedly traverse through the n-grams at each level, using features such as counts to compute intermediate values, conditional probabilities, and backoff weights. Thus, an ideal data structure for LM estimation requires simple and fast iterators and allows intermediate values to be associated with each n-gram. Specifically, given that many computations involve statistics from the corresponding history and backoff n-grams, links to these n-grams need to be precomputed for each n-gram to avoid duplicate lookups.

Furthermore, an ideal data structure needs to minimize memory footprint while supporting efficient file serialization and memory-mapped model loading. To optimize the use of hardware resources, it should preserve memory locality for common algorithms, utilize vector instructions, and support multi-core architectures. Finally, to enable the estimation of LMs from extremely large corpora, both the data structure and algorithms have to support parallelization over distributed memory computer clusters.

2.3. MITLM Vectors

In this work, we present a novel data structure designed specifically for LM estimation procedures with iterative parameter tuning. Instead of dynamically building a trie in memory like SRILM, the proposed data structure represents n-grams as a list of vectors, one for each order. Unlike CMU SLM, each element of the vector represents a specific n-gram by storing the target word index and the index of the history n-gram (Figure 1e). Conceptually, we can view this as a compact vector encoding of an n-gram trie with reversed pointers such that each node points to its parent. Unlike standard tries, we can efficiently determine the n-gram words for any node without costly searches.

With a vector representation, traversing over the n-gram structure reduces to pointer increments and preserves memory locality. In addition, data such as count and backoff n-gram index can be easily associated with each n-gram by storing them in corresponding elements of separate data vectors. Furthermore, computations like backoff weights that involve statistics from the history and backoff n-grams can access these values via simple index lookups without costly searches.

3. Algorithms

Once the data structure has been constructed, most LM estimation algorithms can be decomposed into simple vector operations (Figure 2) on the n-gram counts and history/backoff indices without referring to the word indices. For algorithms involving iterative estimation of model parameters, intermediate values can often be cached across iterations to reduce computation. As the model parameters are tuned to minimize an objective function such as the development set perplexity, we can further improve the performance by masking the operations to the subset of n-gram values that affect the result during the iterative optimization. Lastly, in iterative procedures, it may be worthwhile to first compute sufficient statistics that summarize the full data to reduce the work involved per iteration. In the following sections, we will describe example applications of the above techniques to a few common LM estimation approaches.

1.	$\tilde{c} = \textit{highestOrder}\ ?\ c : \text{bincount}(b^+)$
2.	$n = \text{bincount}(\tilde{c}); Y = \frac{n[1]}{n[1]+2n[2]}$ for i in (1,2,3): $f[i] = i - (i+1)Y\frac{n[i+1]}{n[i]}$
3.	$c_h = 1/\text{binweight}(h, \tilde{c})$
4.	$d = \text{binlookup}(\tilde{c}, f, f[3])$
5.	$\alpha = \text{bincount}(h, d) \times c_h$
6.	$p = (\tilde{c} - d) \times c_h[h] + \alpha[h] \times p^-[b]$

Figure 3: Modified Kneser-Ney smoothing. b: backoff indices, b^+: higher order backoff indices, h: history indices, c: n-gram counts, $\tilde{c}$: modified counts, c_h: inverse history counts, n: count of counts, f: discount factors, d: n-gram discounts, α: backoff weights, p: probabilities, p^-: lower order probabilities.

3.1. Modified Kneser-Ney Smoothing

Modified Kneser-Ney smoothing [1] is one of the best performing techniques for n-gram LM estimation. Unlike the original Kneser-Ney smoothing [10] where a constant is subtracted from each count, modified Kneser-Ney subtracts a different value depending on the actual count. Specifically, it assigns probability $p(w|h) = \frac{\tilde{c}(hw) - D(\tilde{c}(hw))}{\sum_w \tilde{c}(hw)} + \alpha(h)p(w|h')$, where $\tilde{c}(hw)$ is a modified n-gram count, $D(\cdot)$ is the discount function, and $\alpha(h)$ is the backoff weight satisfying $\sum_w p(w|h) = 1$.

As shown in Figure 3, we can compute all probabilities and backoff weights simultaneously using efficient vector operations. For example, the Kneser-Ney counts $\tilde{c}(hw)$, defined by the number of unique n-grams with hw as their backoffs, can be computed using a bincount operation on the higher order backoff indices b^+. Each higher order n-gram contributes a count of 1 to its backoff n-gram, as desired. With aligned vectors, we can also compute the backoff probabilities for the current order by simply indexing the lower order probability vector p^- by the backoff indices b and multiplying it by the backoff weights.

Instead of estimating discount factors $f[i] = D(i)$ from count statistics, we can also tune them to minimize the development set perplexity, which has been observed to improve performance [1]. In each iteration of the parameter optimization, it is sufficient to repeat steps 4–6, as the inverse history counts $c_h(h) = \frac{1}{\sum_w \tilde{c}(hw)}$ are independent of the parameters f. Thus, by caching c_h in the n-gram vector structure, we significantly reduce the amount of computation within each iteration.

3.2. Linear Interpolation

Linear interpolation [11] is the most popular algorithm for merging multiple n-gram LMs. To create a static backoff n-gram LM from component LMs, SRILM interpolates the component probabilities for the union of observed n-grams, using backoff probabilities if necessary, and computes the backoff weights to normalize the model. Figure 4 contains an efficient vector implementation of the linear interpolation algorithm.

1. for i in $[1,2]$: $p_i, \alpha_i = \text{loadlm}(\textit{lm}_i, \textit{model})$
2. for i in $[1,2]$: $z = (p_i == 0); p_i[z] = \alpha_i[h[z]] \times p_i^-[b[z]]$
3. $p = p_1 \times w_1 + p_2 \times w_2$
4. $\alpha = (1 - \text{binweight}(h,p))/(1 - \text{binweight}(h,p^-[b]))$

Figure 4: Linear interpolation. LMs are loaded into a common n-gram structure such they share common history and backoff indices, h and b. p_i, p_i^-, α_i: probabilities, lower order probabilities, and backoff weights of i^{th} LM; w_1, w_2: interpolation weights; p, α: interpolated probabilities and backoff weights.

1. $c_p, c_\alpha, N, N_{oov} = \text{loadevalcorpus}(\textit{corpus}, \textit{lm})$
2. $p, \alpha = \text{estimatelm}(\textit{lm}, \textit{params})$
3. $\textit{perplexity} = \exp[-\frac{1}{N}(c_p \cdot \log p + c_\alpha \cdot \log \alpha)]$

Figure 5: Perplexity evaluation. Sufficient statistics c_p and c_α are computed as the contribution each LM probability and backoff weight makes towards the corpus perplexity. N and N_{oov} are the total counts of in-vocabulary and out-of-vocabulary words in the corpus, respectively.

The interpolation weights are typically chosen to minimize the development set perplexity using an iterative algorithm, such as Expectation-Maximization (EM) [12] or numerical optimization techniques [13]. Since computing the development set perplexity typically only involves a small subset of n-gram probabilities and backoff weights, we can pre-compute Boolean masks m_p and m_α to represent this subset. Because the backoff weight computation (step 4) involves other n-gram probabilities, we need to extend the probability mask to these n-grams. Specifically, computing binweight(h,p) with mask m_α requires the probabilities of n-grams whose histories are in m_α. Thus, we need to extend the mask via m_h = binlookup(h, m_α), m_p |=m_h. Likewise, to compute binweight$(h, p^-[b])$, we shall extend the mask of the lower order probability vector using m_{p^-} |= binset(b, m_h). Using these masks, we can now substantially reduce the computation per iteration in steps 3 and 4 by limiting it to only the masked n-grams. Similar techniques can be applied to smoothing and other algorithms.

3.3. Perplexity Evaluation

The perplexity of a LM evaluated on a corpus w_1^N is defined as $\textit{perplexity} = \exp[-\frac{1}{N}\log\prod_{i=1}^{N} p(w_i|h_i)]$. Given that not all n-grams in the corpus may be observed in the LM, we may need to rely on backoff probabilities to compute $p(w_i|h_i)$.

Finding an n-gram from word indices is a costly procedure. Instead of performing this lookup in every iteration of perplexity optimization, we can pre-compute sufficient statistics that describe the evaluation corpus, as they remain constant across iterations. Specifically, since the corpus probability can be decomposed into the product of observed $p(w|h)$'s and $\alpha(h)$'s, we can represent the corpus compactly using the count each appears in the product, or $c_{p(w|h)}$ and $c_{\alpha(h)}$. Thus, after updating the LM probabilities and backoff weights in each iteration, we can evaluate the LM perplexity efficiently using vector dot products, as shown in step 3 of Figure 5. The counts can also serve as the n-gram masks for efficient LM estimation.

4. Implementation

Using the data structure and algorithms described in the previous sections, we have implemented the MITLM toolkit, supporting various iterative LM estimation algorithms, including modified Kneser-Ney smoothing [1], linear interpolation [11], count merging [14], and generalized linear interpolation [2]. Model parameters are tuned to minimize development set perplexity using numerical optimization techniques such as Powell's method [13] and L-BFGS-B [15].

For efficiency, core data structures and vector operations are implemented in C++. For ease of development and experimentation, high level algorithms are written in Python with NumPy and SciPy packages.[1] As vector operations often generate intermediate values, we expanded these operations in C to reduce the memory requirement and improve performance. The current implementation does not yet support parallelism.

Dataset	Words	1-grams	2-grams	3-grams
BNDev	3,573,908	56,156	778,556	1,973,886
BNTest	13,931,084	110,079	2,028,988	6,158,630
BNTrain	131,668,976	355,995	9,153,440	37,884,316
NYT96	156,879,556	319,279	10,939,278	38,566,120

Table 1: Summary of evaluation datasets.

5. Experiments

To demonstrate the capabilities of the MITLM toolkit, we will evaluate the runtime performance of select LM smoothing and interpolation experiments on the broadcast news domain. Specifically, the target corpus consists of the Broadcast News corpus [16] from 1996, where we designated the first month as the development set (BNDev) and used the remaining five months as the test set (BNTest). For training data, we use the Broadcast News data from 1992 to 1995 (BNTrain), along with the New York Times articles [17] from 1996 (NYT96). Table 1 summarizes all the evaluation data.

5.1. LM Smoothing

In [1], Chen and Goodman conclusively showed that modified Kneser-Ney smoothing achieves better performance with tuned discount parameters, especially in the presence of mismatch between the training and test data. However, perhaps because popular language modeling toolkits, such as SRILM, do not directly support parameter tuning, most subsequent publications using this smoothing algorithm estimated the discount parameters from count statistics instead.

Using a 2.66 GHz CPU, estimating a trigram LM using modified Kneser-Ney smoothing with fixed parameters on the BNTrain corpus takes about 4.6 minutes using SRILM, with a final memory usage of 3.2 GB. With MITLM, the performance improves considerably to 3.0 minutes and 1.9 GB. Excluding time spent in file I/O, the difference is even more dramatic with SRILM and MITLM spending 122 and 3.8 seconds, respectively. By using a vector representation with precomputed backoff indices, MITLM improves the efficiency of LM smoothing by more than an order of magnitude.

Instead of using fixed values, we can also tune the 9 modified Kneser-Ney discount parameters to minimize the development set perplexity using Powell's method. For this dataset, applying masking to compute only those n-grams needed for perplexity evaluation reduces the vector computation by up to a factor of 26x, enabling the 240 iterations of the parameter optimization process to complete in only 3.1 minutes and 2.2 GB of memory.

[1] `http://www.scipy.org/`

Technique	Time (min)	Memory (GB)	Perplexity
LI	8.3	5.4	125.8
CM	9.8	7.2	125.5
GLI($\log(c+1)$)	16.8	7.4	124.7

Table 2: Training time, memory usage, and test set perplexity for various interpolation techniques on BNTrain and NYT96.

With its mostly sequential memory accesses, MITLM also improves memory locality over SRILM. For example, although both toolkits significantly exceed the available 8GB of memory when estimating a trigram model on the full 1.1 billion words of New York Times articles from 1994 to 2004 [17], MITLM manages to finish in 30 minutes, whereas SRILM took over 24 hours due to disk thrashing. By designing the data structure for iterative paramater optimization, we can significantly improve the runtime performance and enable previously impractical parameter tuning algorithms.

5.2. LM Interpolation

SRILM supports building a static backoff LM from the linear interpolation (LI) of component LMs. It also provides a separate script for tuning the interpolation weights to minimize the development set perplexity using Expectation-Maximization. Using SRILM, we were able to tune and interpolate BNTrain and NYT96 into a single LM in 19.5 minutes and 3.6 GB via 18 iterations of the EM algorithm. By integrating the tuning process into the estimation procedure, MITLM interpolates the same LMs via 10 iterations of L-BFGS-B in 8.3 minutes and 5.4 GB, spending only 1.8 minutes on parameter optimization. Since SRILM optimizes the parameters for dynamic interpolation, it uses less memory than MITLM, but generally yields suboptimal results for static interpolation.

Extending beyond the capabilities of current publicly available language modeling toolkits, Table 2 lists the performance and computation times of various LM interpolation techniques. By adjusting the interpolation weight as a function of features derived from the n-gram history, count merging (CM) [14] and generalized linear interpolation (GLI) [2] improve the test set performance with only a moderate increase in computational resources. In fact, both interpolation techniques using MITLM complete in less time than linear interpolation using SRILM.

6. Conclusion & Future Work

By designing the data structure and algorithms for iterative LM estimation, we have implemented an efficient language modeling toolkit that supports parameter optimization for modified Kneser-Ney smoothing and various LM interpolation techniques. Compared with the popular SRILM toolkit, the resulting MITLM toolkit improves the running time of the modified Kneser-Ney algorithm by over 30x and reduces the memory usage by 40%. Furthermore, it enables the efficient optimization of parameters for advanced interpolation techniques, instead of relying on empirical trial and error approaches [14, 18].

For future work, we plan to explore additional iterative LM estimation techniques and conduct more detailed perplexity and recognition evaluations. We would also like to further optimize the memory usage and parallelize the implementation of MITLM to support large-scale LM estimation.

With the simple vector representation of n-grams, we invite researchers to implement future language modeling techniques using MITLM. We hope that with the public release of this toolkit, future research will also consider more effective language model estimation techniques that are previously impractical.

Availability The latest version of MITLM and accompanying documentation are available as open source software at `http://www.sls.csail.mit.edu/mitlm/`.

7. Acknowledgments

We would like to thank Lee Hetherington for the helpful discussions and the reviewers for their constructive feedback. This research is supported in part by the T-Party Project, a joint research program between MIT and Quanta Computer Inc.

8. References

[1] S. Chen and J. Goodman, "An empirical study of smoothing techniques for language modeling," in *Technical Report TR-10-98*. Computer Science Group, Harvard University, 1998.

[2] B. Hsu, "Generalized linear interpolation of language models," in *Proc. ASRU*, 2007.

[3] A. Stolcke, "SRILM – An extensible language modeling toolkit," in *Proc. ICSLP*, 2002.

[4] E. Whittaker and B. Raj, "Quantization-based language model compression," in *Proc. Interspeech*, 2001.

[5] T. Brants and A. Franz, "Web 1T 5-gram version 1," Linguistic Data Consortium, Philadelphia, 2006.

[6] M. Federico and M. Cettolo, "Efficient handling of n-gram language models for statistical machine translation," in *Workshop on Statistical Machine Translation*, 2007.

[7] P. Nguyen, J. Gao, and M. Mahajan, "MSRLM: A scalable language modeling toolkit," in *MSR-TR-2007-144*. Microsoft, Inc., 2007.

[8] A. Stolcke, "SRILM man pages: ngram-format," 2004, http://www.speech.sri.com/projects/srilm/manpages/ngram-format.html.

[9] P. Clarkson and R. Rosenfeld, "Statistical language modeling using the CMU-Cambridge toolkit," in *Proc. Eurospeech*, 1997.

[10] R. Kneser and H. Ney, "Improved backing-off for m-gram language modeling," in *Proc. ICASSP*, 1995.

[11] F. Jelinek and R. Mercer, "Interpolated estimation of Markov source parameters from sparse data," in *Proc. Workshop on Pattern Recognition in Practice*, 1980.

[12] A. Dempster, N. Laird, and D. Rubin, "Maximum likelihood from incomplete data via the EM algorithm," *Journal of the Royal Statistical Society B*, vol. 39, no. 1, pp. 1–38, 1977.

[13] W. Press, S. Teukolsky, W. Vetterling, and B. Flannery, *Numerical Recipes*, 3rd ed. Cambridge University Press, September 2007.

[14] M. Bacchiani, M. Riley, B. Roark, and R. Sproat, "MAP adaptation of stochastic grammars," *Computer Speech & Language*, vol. 20, no. 1, pp. 41–68, 2006.

[15] R. Byrd, P. Lu, and J. Nocedal, "A limited memory algorithm for bound constrained optimization," *SIAM Journal on Scientific and Statistical Computing*, vol. 16, no. 5, pp. 1190–1208, 1995.

[16] D. Graff and J. Alabiso, "1996 English Broadcast News transcripts (HUB4)," Linguistic Data Consortium, Philadelphia, 1997.

[17] D. Graff, "English Gigaword," Linguistic Data Consortium, Philadelphia, 2003.

[18] J. Gauvain, L. Lamel, and G. Adda, "The LIMSI Broadcast News transcription system," *Speech Communication*, vol. 37, no. 1-2, pp. 89–108, 2002.