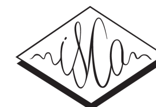


Unary Data Structures for Language Models

Jeffrey Sorensen, Cyril Allauzen

Google, Inc., 76 Ninth Avenue, New York, NY 10011
{sorenj,allauzen}@google.com

Abstract

Language models are important components of speech recognition and machine translation systems. Trained on billions of words, and consisting of billions of parameters, language models often are the single largest components of these systems. There have been many proposed techniques to reduce the storage requirements for language models. A technique based upon pointer-free compact storage of ordinal trees shows compression competitive with the best proposed systems, while retaining the full finite state structure, and without using computationally expensive block compression schemes or lossy quantization techniques.

Index Terms: n-gram language models, unary data structures

1. Introduction

Models of language constitute one of the largest components of contemporary speech recognition and machine translation systems. Typically, language models are based upon n-gram models which provide probability estimates for seeing words following a partial history of preceding words, or an n-gram context.

$$P\{w_t|w_{t-1},\ldots,w_1\} \approx P\{\underbrace{w_t}_{\text{future}}|\underbrace{w_{t-1}\ldots w_{t-n+1}}_{\text{context}}\} \quad (1)$$

By assuming the language being modeled is an n-th order Markov process, we make tractable the number of parameters that comprise the model. Even so, language models still contain massive numbers of parameters which are difficult to estimate, even when extraordinarily large corpora are used.

The art and science of estimating parameters is an area well represented in the speech processing community, and [1] provides an excellent introduction. Language model compression and storage is an entire subgenre in itself. Many approaches to compressing language models have been proposed, both lossless and lossy [2, 3, 4, 5, 6], and most of these techniques are complementary to what we present here.

A typical format that is used for static finite state acceptors in OpenFst [7] is illustrated in Figure 1. This format makes clear the set of arcs and weights associated with a particular state, but it does not make clear which state corresponds to a particular context, and it uses nearly half of its space to store the indices to the next state numbers and their offsets.

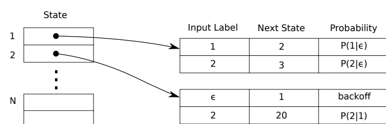


Figure 1: A typical language model storage format

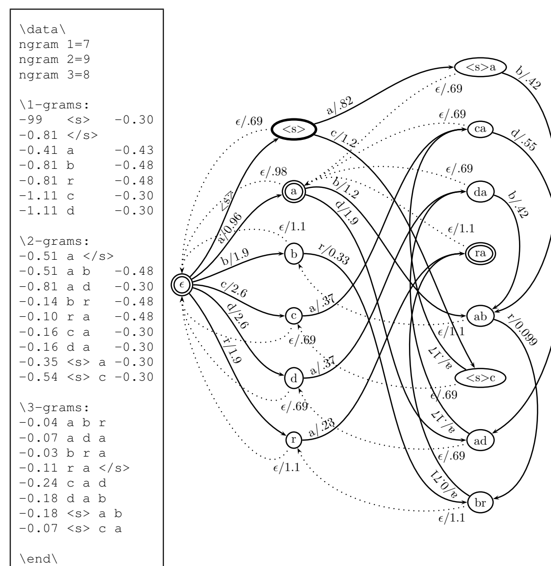


Figure 2: A trivial, but complete, trigram language model

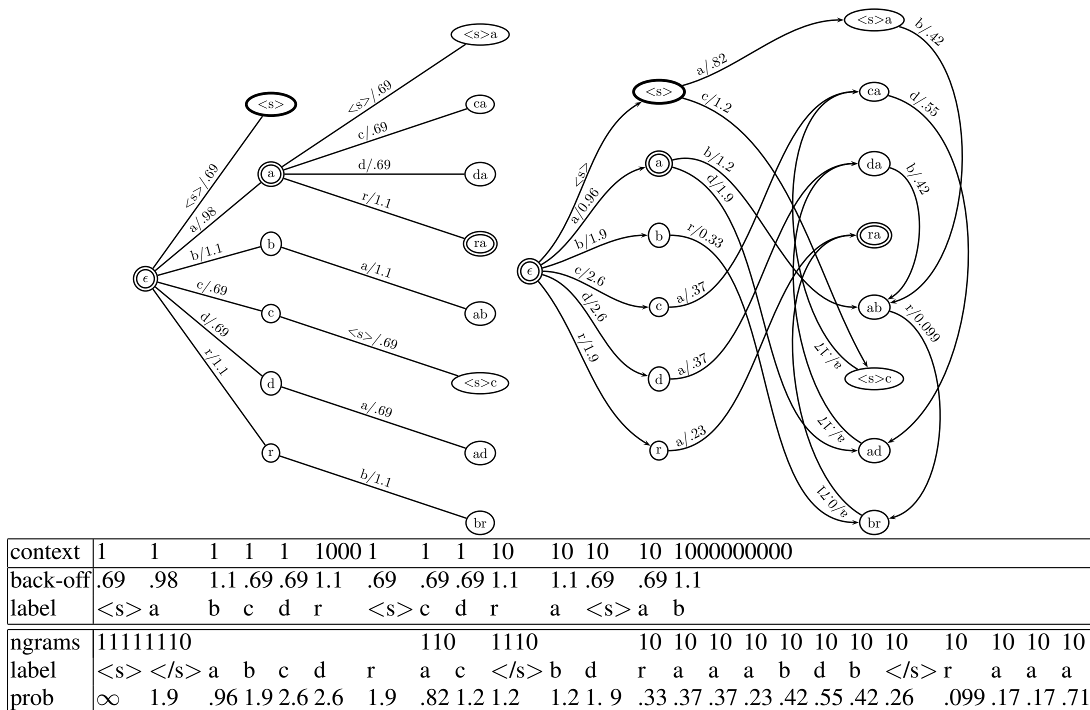
Language models are cyclic and non-deterministic, with both properties serving to complicate attempts to compress their representations. In order to illustrate the ideas and principles put forth in the paper, we introduce a complete, but trivial, language model based upon two "sentences" and a vocabulary of just five "words": `a b r a` and `c a d a b r a`.

We have added a link from the ϵ unigram state to the $<$s$>$ start state in order to make our presentation clearer. This is not normally an allowed transition in typical applications, but without loss of generality we can assign a transition weight of ∞.

2. Succinct Tree Structures

Storing trees, asymptotically optimally, in a boolean array was first proposed in [8], who termed these data structures level-order unary degree sequences, or LOUDS. This was later generalized for application to labeled, ordinal trees by [9] and others. Succinct data structures differ from more traditional data structures, in part, because they avoid the use of indices and pointers, and [10] presents a good overview of the engineering issues. Using these data structures to store a language model was proposed in [11], although their structure does not use the decomposition we propose, nor does it provide a finite state acceptor structure.

Copyright © 2011 ISCA



context	1	1	1	1	1	1000	1	1	1	10	10	10	10	1000000000
back-off	.69	.98	1.1	.69	.69	1.1	.69	.69	.69	1.1	1.1	.69	.69	1.1
label	<s>	a	b	c	d	r	<s>	c	d	r	a	<s>	a	b

ngrams	11111110							110		1110			10	10	10	10	10	10	10	10	10	10	10	10
label	<s>	</s>	a	b	c	d	r	a	c	</s>	b	d	r	a	a	a	b	d	b	</s>	r	a	a	a
prob	∞	1.9	.96	1.9	2.6	2.6	1.9	.82	1.2	1.2	1.2	1. 9	.33	.37	.37	.23	.42	.55	.42	.26	.099	.17	.17	.71

Figure 3: Decomposed Language Model and its LOUDS representation

3. Decomposed Language Model

The language model in Figure 2 has states that have been labeled according to the context they represent. This property, while not an explicit part of standard language model notation, can easily be computed using the single-source shortest-path algorithm from [12] over the string semiring. The state labeled ϵ represents the *unigram* context or the probability of seeing each symbol without any context conditioning. The start state, in bold, represents the probability of seeing symbols in the initial context. Here the symbols a and c both have arcs leading to the appropriate context.

As language models must also admit strings that were not observed in the training corpus, we also have an failure arc to the unigram state. This arc is labeled with a back-off penalty that is dependent upon the smoothing algorithm chosen to build this model (this model was built using Witten-Bell [13] smoothing).

The language model is a cyclic graph with failure arcs, which we treat as ϵ arcs, on all states except the unigram state. The techniques of [8] cannot be applied to non-planar graphs, and typical language models are not planar (no graph with three times as many arcs as states is planar [14]). However, language models can be decomposed.

If you consider only the back-off arcs, the language model in Figure 2 can be represented by a tree of contexts. Figure 3 shows the subtree created by extracting all of the back-off arcs. Here we have relabeled the arcs (and removed the directionality) using the prefix of the state labels as determined from the shortest path above. And, we sorted the arcs lexicographically to support searching for a particular arc. This yields a tree of contexts, where each context is connected by an arc from a less specific to more specific context.

3.1. LOUDS Tree of Contexts

The LOUDS tree represents a tree by performing a breadth-first, or level-order, walk creating a bit vector that encodes the graph structure. Starting from the root node, we create a bit vector describing the tree by writing ones corresponding to the number of children at a node, followed by a zero. The resulting bit vector contains a one bit and a zero bit for each node, thus the storage required is two times the number of states in the language model.

In addition to the geometry of the tree, we must store the back-off weights, and the labels for the context arc transitions. These are stored, densely, in separate arrays, with values for each arc. In order to navigate the context tree, we make use of rank and select operations on the bit vector.

rank$_q(b, i)$ Returns the count of q valued bits in bitstring b for all $j < i$

select$_q(b, c)$ Returns the index i of the bit in b such that $c = \text{rank}_q(b, i)$

Using these two operators navigation from context to context can be done bidirectionally.

Because context arc labels, for each node, are stored contiguously, it makes cache-sensitive binary searches practical. Although there are several methods, for simplicity we use 1 based indexing, where each node in the tree is referred to by specifying the offset to its corresponding 1 in the bitstring for the tree. To illustrate, let us chose the a context which is identified by the second 1 bit, or offset $i = 2$.

$$\text{is-leaf}(b, i) = b[\text{select}_0(\text{rank}_1(b, i)) + 1] \qquad (2)$$

To find the context ba, starting from the index, we can check if the node is a leaf using in this case, $\text{rank}_1(b, 2) = 1$, and $\text{select}_0(b, 1)$ = 8. $b[9]$ is a 1 bit, and it happens to be the bit

```
function NextState(context, future)
  node = findChild(root, future)
  if numChildren(node) == 0 return node
  from = context
  i = 0
  while (from != root)
    history[i] = label[rank1(from)]
    from = rank1(select0(from) − 1)
    i = i + 1
  while true
    nextNode = findChild(node, history[i])
    if (nextNode == null) break
    i = i − 1
    node = nextNode
  return node
```

Figure 4: Algorithm to find next context

corresponding to the first child of context a. Thus, for non-leaves

$$\text{first-child}(b, i) = \text{select}_0 \left[\text{rank}_1(b, i)\right] + 1 \quad (3)$$

and

$$\text{last-child}(b, i) = \text{select}_0 \left[\text{rank}_1(b, i) + 1\right] - 1 \quad (4)$$

which tells us that the children of a span from 9 to 12 in the label array. The value b does not appear in this span, which is not surprising because the sequence ba does not occur in our training corpus.

Unlike techniques based on hashing, bidirectional navigation to arbitrary contexts is also supported, via

$$\text{parent}(b, i) = \text{select}_1 \left[\text{rank}_0(b, i)\right] \quad (5)$$

and, finally, the arc labels and weights are simply done by accessing the $\text{rank}_1(b, i)$th element of the back-off and label arrays. We discuss how to implement rank and select efficiently in Section 3.3.

The LOUDS tree of context provides an invertible one-to-one mapping between state numbers and the sequence of inputs required to reach that state. This is important for decoding and allows us to remove entirely the next state value from the language model arcs.

3.2. Future Words

In addition to the tree of contexts, the non-epsilon arcs, which form a partition of the entire language model, cannot be represented as a tree. For our example language model, due to the limit of n previous terms, there are many states that are reachable from many different contexts.

The list of future words does not need to store the identity of the next state for any of the arcs, because the tree of contexts allows these to be determined on-the-fly using the state transition algorithm in Figure 4. Thus, the only information that needs to be stored for each arc is the sorted list of outbound arc labels (for fast searching) and the probability associated with each label. We use a simple bit vector indicating the number of outbound arcs for each state with a sequence of ones, followed by a zero. The word ids and the probabilities for each of these arcs are stored in dense auxiliary arrays, which are indexed using $\text{rank}_1 \left[i + \text{select}_0(s)\right]$ for arc i of state s.

3.3. Fast Algorithms for Tree Access

While we have presented the data structures used to store the language model, there are many engineering details that also need to be addressed to make the proposed data structures competitive with more conventional approaches based on sorted lists, hash tables, or other index based approaches.

The bitstrings used to store the tree geometry in our language model can be billions of bits long. A naive implementations of rank, even using contemporary special purpose instructions like popcnt, would require $O(|b|)$ time to compute. To make these operations practical, we use the highly tuned implementation by [15] which adds to the bitstring both a primary and secondary index that provide constant time access. These techniques are based upon the idea of breaking the bitstring into blocks of 64 bits (as is already done by the processor) and then maintaining a running count of the number of one bits per block. This *second level count*, itself of fixed size, is accumulated into a 64 bit *first level count*.

This index, of course, requires an additional storage, using $0.25|b|$ bits. Implementation of select can be done, naively, using a binary search with rank operations. However, [15] presents a constant time algorithm that requires at most $0.375|b|$ bits of index data.

4. Experimental Results

We performed two seperate experiments to compare the proposed LOUDS based language model in typical decoding applications. The first experiment was a simple composition of 4K voice search utterances consisting of 17K total words with a 5-gram, 1M word vocabulary, 11M n-gram language model.

In order to evaluate our proposed LOUDS data format in a decoder, we compared the composition speed of a variety of FST representations of a language model consisting of arcs and states. We compare 4 representations for the LM, the proposed LOUDS format, and the const, compact_acceptor, and vector formats of OpenFst with the decoding time (composition and shortest path) normalized to the speed of the vector fst format.

Model Format	Storage	Time
vector	≈500 MB	1.0
const	292 MB	0.97
compact acceptor	206 MB	1.02
LOUDS fst	148 MB	0.87

Table 1: Storage requirement and average recognition time per utterance.

To test the LOUDS language model in a more realistic speech recognition application, a 4-gram language model was constructed using Katz backoff trained from a variety of sources [16]. The model consists of 14.3 million n-grams: 1 million unigrams, 7.5 million bigrams, 8 million trigrams and 0.8 million 4-grams.

The acoustic model is a tied-state triphone GMM-based HMM whose input features are 13 PLP-cepstral coefficients, frame-stacked and projected onto 39 dimensions using LDA/STC, trained using ML, MMI, and boosted-MMI objective functions as described in [17].

Model Format	Storage	Time
const fst	307 MB	4.03s
compact acceptor fst	204 MB	4.15s
LOUDS fst	132 MB	3.77s

Table 2: Storage requirement and average recognition time per utterance.

At recognition time, for each utterance, a new instance of the on-demand composition of the context-dependent lexicon transducer and the language model is created and searched using a open-pass Viterbi decoder. The composition is performed using the algorithm from [18] using the label-lookeahead filter with weight pushing. The results are presented in Table 2.

5. Asymptotic Storage Requirements

If we use a simple percentage n-gram frequency threshold to build a language model from a corpus of N words, the empirically observed distribution of n-grams [19] suggests

$$|V| \propto \log N \quad n \propto \log N \tag{6}$$

with n growing substantially slower than $|V|$ as the size of the corpus grows without bound. We consider the asymptotic storage requirements for three different alternatives. Data structures that use a format similar to OpenFst's const format, or any data structure that indexes the n-grams via pointers or tables, require $\mathrm{O}\big(N \log N\big)$ storage. This can be shown by noting that each arc contains the state id of the next state, and the number of states grows with N, and state id's require $\log N$ storage.

A potentially effective technique, storing the n-grams in hash tables, which supports fast queries but not fast arc iteration, requires storing the n-gram keys for each arc which require $\mathrm{O}\big(\log |V|^n\big)$ space, making the total storage $\mathrm{O}\big(N \log N \log\log N\big)$.

The storage of arcs for our proposed model does not depend upon the n-gram depth. But, the vocabulary items stored in the auxiliary arrays need $\mathrm{O}\big(\log |V|\big)$ storage, making the total storage requirements $\mathrm{O}\big(N \log\log N\big)$.

This is not to say that the format, as presented, would be competitive with the state-of-the-art compression schemes presented in [2, 6], both of which use variable length block encoding schemes. Certainly, nothing precludes the use of similar compression techniques with our format. However, variable length encoding comes at great additional cost in terms of access time.

6. Conclusion and Future Work

The proposed LOUDS format provides considerable savings over the existing OpenFst language model formats. And, unlike previous proposed systems for compressing language models, it does so while simultaneously improving access time. The use of quantization or block compression should be easy to incorporate into our proposed data structure. More easily, in fact, considering that these data values are stored in densely packed auxiliary arrays.

For a vocabulary of size V, the potential number of n-grams of order n is V^n. The set of bigrams is denser than all other n-gram orders. Even so, for a typical language model, the number of bigrams is much less than V^2. And, under reasonable pruning constraints, the set of higher order n-gram arcs is a subset of the $n-1$ state's arcs.

This enables us to employ a similar tree based storage technique for storing these arcs. As the unigrams probabilities have already been addressed, we need to store the list of extensions, or next words, for each unigram context. For all higher order n-grams, the list of futures for any state is a subset of the arcs of the back-off context. For example, state ca contains only one of the two arcs from the a context. In order to avoid storing the arc labels again, we refer to the back-off context and store a boolean for each bigram indicating which bits survived to the next level.

This could be done at the expense of a slightly more complicated lookup. Searching bigrams first could turn out to be an effective strategy, instead of doing binary searches at each n-gram order, as binary searches have demonstrably bad cache behavior. We intend to explore this, and more traditional compression schemes in the future, and their effect on composition speed in decoders.

7. References

[1] S. F. Chen and J. Goodman, "An empirical study of smoothing techniques for language modeling," in *Proc. of ACL*, 1996, pp. 310–318.

[2] B. Harb, C. Chelba, J. Dean, and S. Ghemawat, "Back-off language model compression," in *Proc. of Interspeech*, 2009, pp. 325–355.

[3] D. Guthrie and M. Hepple, "Storing the web in memory: space efficient language models with constant time retrieval," in *Proc. of EMNLP*, 2010, pp. 262–272.

[4] K. Church, T. Hart, and J. Gao, "Compressing trigram language models with Golomb coding," in *Proc. of EMNLP-CoNLL*, pp. 199–207.

[5] U. Germann, E. Joanis, and S. Larkin, "Tightly packed tries: How to fit large models into memory, and make them load fast, too," in *Proc. of SETQA-NLP*, 2009, pp. 31–39.

[6] A. Pauls and D. Klein, "Faster and smaller n-gram language models," in *Proceedings of the 49th annual meeting of the Association for Computational Linguistics: Human Language Technologies*, 2011.

[7] C. Allauzen, M. Riley, J. Schalkwyk, W. Skut, and M. Mohri, "OpenFst: A general and efficient weighted finite-state transducer library." in *Proc. of CIAA*, 2007, pp. 11–23.

[8] G. Jacobson, "Space-efficient static trees and graphs," in *Proc. of FOCS*, 1989, pp. 549–554.

[9] R. F. Geary and R. Raman, "Succinct ordinal trees with level-ancestor queries," in *Proc. of SODA*, 2004, pp. 1–10.

[10] N. Rahman and R. Raman, "Engineering the louds succinct tree representation," in *Proc. of WEA*, 2006, p. 145.

[11] T. Watanabe, H. Tsukada, and H. Isozaki, "A succinct n-gram language model," in *Proc. of the ACL-IJCNLP 2009 Conference Short Papers*, 2009, pp. 341–344.

[12] M. Mohri, "Semiring frameworks and algorithms for shortest-distance problems," *Journal of Automata, Languages and Combinatorics*, vol. 7, pp. 321–350, 2002.

[13] I. Witten and T. Bell, "The zero-frequency problem: estimating the probabilities of novel events in adaptive text compression," *Information Theory, IEEE Transactions on*, vol. 37, no. 4, pp. 1085–1094, 1991.

[14] J. Hopcroft and R. Tarjan, "Efficient planarity testing," *J. ACM*, vol. 21, pp. 549–568, October 1974.

[15] S. Vigna, "Broadword implementation of rank/select queries," in *Proc. of WEA*, 2008, pp. 154–168.

[16] B. Ballinger, C. Allauzen, A. Gruenstein, and J. Schalkwyk, "On-demand language model interpolation for mobile speech input," in *Proc. of Interspeech*, 2010, pp. 1812–1815.

[17] J. Schalkwyk, D. Beeferman, F. Beaufays, B. Byrne, C. Chelba, M. Cohen, M. Kamvar, and B. Strope, "Google Search by Voice: A case study," in *Advances in Speech Recognition: Mobile Environments, Call Centers and Clinics*. Springer, 2010.

[18] C. Allauzen, M. Riley, and J. Schalkwyk, "A generalized composition algorithm for weighted finite-state transducers," in *Proc. of Interspeech*. ISCA, 2009, pp. 1203–1206.

[19] L. Q. Ha, P. Hanna, J. Ming, and F. J. Smith, "Extending Zipf's law to n-grams for large corpora," *Artif. Intell. Rev.*, vol. 32, pp. 101–113, December 2009.