Scaling High-Order Character Language Models to Gigabytes

Bob Carpenter
Alias-i, Inc.
181 North 11th St., #401, Brooklyn, NY 11211
carp@colloquial.com

Abstract

We describe the implementation steps required to scale high-order character language models to gigabytes of training data without pruning. Our online models build character-level PAT trie structures on the fly using heavily data-unfolded implementations of an mutable daughter maps with a long integer count interface. Terminal nodes are shared. Character 8-gram training runs at 200,000 characters per second and allows online tuning of hyperparameters. Our compiled models precompute all probability estimates for observed n-grams and all interpolation parameters, along with suffix pointers to speedup context computations from proportional to n-gram length to a constant. The result is compiled models that are larger than the training models, but execute at 2 million characters per second on a desktop PC. Cross-entropy on held-out data shows these models to be state of the art in terms of performance.

1 Introduction

Character *n*-gram language models have been applied to just about every problem amenable to statistical language modeling. The implementation we describe here has been integrated as the source model in a general noisy-channel decoder (with applications to spelling correction, tokenization and case normalization) and the class models for statistical classification (with applications including spam filtering, topic categorization, sentiment analysis and word-sense disambiguation). In addition to these human language tasks, *n*-grams are also popular as estimators for entropy-based compression and source models for cryptography. (Teahan, 2000) and (Peng, 2003) contain excellent overviews of character-level models and their application from a compression and HMM perspective, respectively.

Our hypothesis was that language-model smoothing would behave very much like the classifiers explored in (Banko and Brill, 2001), in that more data trumps better estimation technique. We managed to show that the better of the interpolation models used in (Chen and Goodman, 1996), namely Dirichlet smoothing with or without update exclusion, Witten-Bell smoothing with or without update exclusion, and absolute discounting with update exclusion converged for 8-grams after 1 billion characters to cross entropies of 1.43+/-0.01. The absolute discounting with update exclusion is what Chen and Goodman refer to as the Kneser-Ney method, and it was the clear winner in their evaluation. They only tested non-parametric Witten-Bell with a suboptimal hyperparameter setting (1.0, just as in Witten and Bell's original implementation). After a billion characters, roughly 95 percent of the characters were being estimated from their highest-order (7) context. The two best models, parametric Witten-Bell and absolute discounting with update exclusion (aka Kneser-Ney), were even closer in cross-entropy, and depending on the precise sample (we kept rolling samples as described below), and after a

million or so characters, the differences even at the higher variance 12-grams were typically in the +/- 0.01 range. With a roughly 2.0 bit/character deviation, a 10,000 character sample, which is the size we used, leads to a 2σ (95.45%) confidence interval of +/-0.02, and the conclusion that the differences between these systems was insignificant.

Unlike in the token-based setting, we are not optimistic about the possibility of improving these results dramatically by clustering character contexts. The lower-order models are very well trained with existing quantities of data and do a good job of this kind of smoothing. We do believe that training hyperparameters for different model orders independently might improve cross-entropy fractionally; we found that training them hierarchically, as in (Samuelsson, 1996), actually increased cross-entropy. We believe this is a direct correlate of the effectiveness of update exclusion; the lower-order models do not need to be the best possible models of those orders, but need to provide good estimates when heavily weighted, as in smoothing. The global optimization allows a single setting to balance these attributes, but optimizing each dimension individually should do even better. But with the number of estimates taking place at the highest possible orders, we do not believe the amount of smoothing will have that large an impact overall.

These experiments had a practical goal — we needed to choose a language modeling implementation for LingPipe and we didn't want to take the standard Swiss Army Knife approach because most of our users are not interested in running experiments on language modeling, but rather using language models in applications such as information retrieval, classification, or clustering. These applications have actually been shown to perform better on the basis of character language models than token models ((Peng, 2003)). In addition, character-level models require no decisions about tokenization, token normalization and subtoken modeling (as in (Klein et al., 2003)).

We chose to include the Witten-Bell method in our language modeling API because it is derived from full corpus counts, which we also use for collocation and relative frequency statistics within and across corpora, and thus the overall implementation effort was simpler. For just language modeling, an update exclusion implementation of Kneser-Ney is no more complicated than Witten-Bell.

In this paper, we describe the implementation details behind storing the model counts, how we sample the training character stream to provide low-cost, online leave-one-out style hyperparameter estimation, and how we compile the models and evaluate them over text inputs to achieve linear performance that is nearly independent of *n*-gram length. We also describe some of the design patterns used at the interface level for training and execution. As far as we know, the online leave-one-out analysis is novel, though there are epoch-based precursors in the compression literature.

As far as we know, no one has built a character language model implementation that will come close to the one presented here in terms of scalability. This is largely because they have not been designed for the task rather than any fundamental limitation. In fact, we take the main contribution of this paper to be a presentation of simple data sharing and data unfolding techniques that would also apply to token-level language models. Before starting our presentation, we'll review some of the limitations of existing systems. For a start, none of the systems of which we are aware can scale to 64-bit values for counts, which is necessary for the size models we are considering without pruning or count scaling. It's simply easier to find 4 billion instances of a character than of a token. In fact, the compression models typically use 16 bits for storing counts and then just scale downward when necessary, thus not even trying to store a full set of counts for even modest corpora. The standard implementations of character models in the compression literature represent ordinary trie nodes as arrays, which is hugely wasteful for large sparse implementations; they represent PAT-trie nodes as pointers into the original text plus counts, which works well for long *n*-gram lengths (32) over small data sets (1 MB) but does not scale well for reasonable *n*-gram lengths (8-12) over larger data sets (100MB-1GB). The standard token-level language models used to restrict attention to 64K tokens and thus require 16-bit token representatives per node just as our character-based approach; with the advent of large vocabulary speech recognition, they now typically use 32-bits per node just to represent the token. Arrays of

daughter nodes and lack of sharing of low-count terminal nodes were the biggest space hogs in our experiments, and as far as we know, none of the standard approaches take the immutable data unfolding approach we adopt to eliminate this overhead. Thus we would like to stress again that existing character-level compression and token-level language modeling systems were simply not designed for handling large character-level models.

We would also like to point out that the standard finite state machine implementations of language models do not save any space over the trie-based implementations, typically only approximate smoothing using backoff rather than interpolation, and further suffer from a huge space explosion when determinized. The main advantage of finite state approaches is at the interface level in that they work well with hand-written constraints and can interface on either side of a given modeling problem. For instance, typical language models implemented as trivial finite state transducers interface neatly with triphone acoustic models on the one side and with syntactic grammars on the other. When placed in that context, the constraints from the grammar can often create an overall win in space after composition.

2 Online Character Language Models

For generality, we use the 16-bit subset of unicode as provided by Java 1.4.2 to represent characters. This presents an additional scaling problem compared to ASCII or Latin1, which fit in 7 and 8 bits.

Formally, if Char is a set of characters, a *language model* is defined to be a mapping P from the set Char^* of character sequences into non-negative real numbers. A *process* language model is normalized over sequences of length n: $\sum_{X \in \mathsf{Char}^*, |X|=n} P(X) = 1.0$. We also implement bounded language models which normalize over all sequences, but their implementation is close enough to the process models that we do not discuss them further here. The basic interfaces are provided in Figure 1 (with names shortened to preserve space). Note that the process and sequence distribution is represented through marker interfaces, whereas the cross-cutting dynamic language models support training and compilation, as well as the estimation inherited from the language model interface.

```
interface LM {
    double log2Prob(char[] cs,
        int start, int end);
}
interface ProcessLM extends LM {
}
interface SequenceLM extends LM {
}
interface DynamicLM extends LM {
    double train(char[] cs,
        int start, int end);
    void compile(ObjectOutput out)
        throws IOException;
}
```

Figure 1: Language Model Interface

We now turn to the statistics behind character-level langauge models. The chain rule factors $P(x_0,\ldots,x_{k-1}) = \prod_{i<k} P(x_i|x_0,\ldots,x_{i-1})$. An n-gram language model estimates a character using only the last $n-1$ symbols, $\hat{P}(x_k|x_0,\ldots,x_{k-1}) = \hat{P}(x_k|x_{k-n+1},\ldots,x_{k-1})$; we follow convention in denoting generic estimators by $\hat{P}$.

The maximum likelihood estimator for n-grams is derived from frequency counts for sequence X and symbol c, $P_{\text{ML}}(c|X) = \mathsf{count}(Xc)/\mathsf{extCount}(X)$, where $\mathsf{count}(X)$ is the number of times the sequence X was observed in the training data and $\mathsf{extCount}(X)$ is the number of single-symbol extensions of X observed: $\mathsf{extCount}(X) = \sum_{c \in \mathsf{Char}} \mathsf{count}(Xc)$. When training over one or more short samples, the disparity between $\mathsf{count}(X)$ and $\mathsf{extCount}(X)$ can be large: for *abracadabra*, $\mathsf{count}(a) = 5$, $\mathsf{count}(bra) = 2$, $\mathsf{extCount}(a) = 4$, and $\mathsf{extCount}(bra) = 1$.

We actually provide two implementations of language models as part of LingPipe. For language models as random processes, there is no padding. They correspond to normalizing over sequences of a given length in that the sum of probabilities for character sequences of length k will sum to 1.0. With a model that inserts begin-of-sequence and end-of-sequence characters and estimates only the end-of-sequence character, normalization is over all strings. Statistically, these are very different models. In practice, they are only going to be distinguishable if the boundaries are very significant and the total string length is relatively small. For instance, they are not going to make much difference in estimating probabilities of abstracts of 1000 characters,

even though the start and ends are significant (e.g. capitals versus punctuation being preferred at beginning and end of abstracts) because cross-entropy will be dominated by the other 1000 characters. On the other hand, for modeling words, for instance as a smoothing step for token-level part-of-speech, named-entity or language models, the begin/end of a word will be significant, representing capitalization, prefixes and suffixes in a language. In fact, this latter motivation is why we provide padded models. It is straightforward to implement the padded models on top of the process models, which is why we discuss the process models here. But note that we do not pad all the way to maximum *n*-gram length, as that would bias the begin/end statistics for short words.

We use linear interpolation to form a mixture model of all orders of maximum likelihood estimates down to the uniform estimate $P_U(c) = 1/|\mathsf{Char}|$. The interpolation ratio $\lambda(dX)$ ranges between 0 and 1 depending on the context dX.

$$\begin{aligned}\hat{P}(c|dX) &= \lambda(dX)P_{\mathrm{ML}}(c|dX) \\ &+ (1-\lambda(dX))\hat{P}(c|X) \\ \hat{P}(c) &= \lambda()P_{\mathrm{ML}}(c) \\ &+ (1-\lambda())(1/|\mathsf{Char}|)\end{aligned}$$

The Witten-Bell estimator computed the interpolation parameter $\lambda(X)$ using only overall training counts. The best performing model that we evaluated is parameterized Witten-Bell interpolation with hyperparameter K, for which the interpolation ratio is defined to be:

$$\lambda(X) = \frac{\mathsf{extCount}(X)}{\mathsf{extCount}(X) + K \cdot \mathsf{numExts}(X)}$$

We take $\mathsf{numExts}(X) = |\{c|\mathsf{count}(Xc) > 0\}|$ to be the number of different symbols observed following the sequence X in the training data. The original Witten-Bell estimator set $K = 1$. We optimize the hyperparameter K online (see the next section).

3 Online Models and Hyperparameter Estimation

A language model is *online* if it can be estimated from symbols as they arrive. An advantage of online models is that they are easy to use for adaptation to documents or styles, hence their inclusion in commercial dictation packages such as DragonDictate and ViaVoice. Another advantage is that they are easy to integrate into tag-a-little/learn-a-little systems such as MITRE's Alembic Workbench.

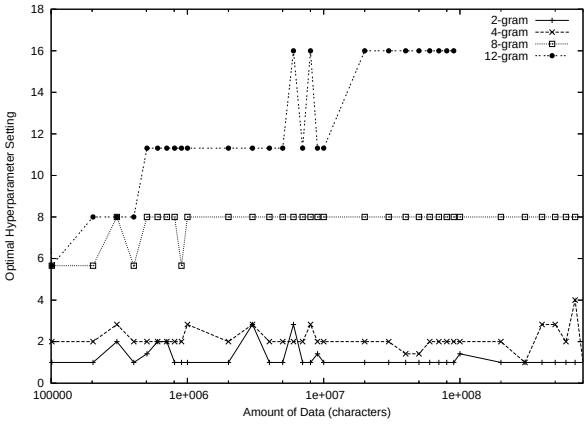

Figure 2: Optimal Hyperparameter Settings for Witten-Bell

With online models, we are able to estimate hyperparameters using an online form of leave-one-out analysis (Ney et al., 1995). This can be performed in a number of ways as long as the model efficiently estimates likelihoods given a set of hyperparameter settings. We opted for the simplest technique we could muster to find the right settings. This was made easier because we only have a single hyperparameter whose behavior is fairly flat around the optimal setting and because the optimal setting didn't change quickly with increasing data. The optimal settings are shown in Figure 2. Also note that the optimal value is rarely at 1 except for very low-order *n*-grams. To save the complexity of maintaining an interval around the best estimate do do true hill climbing, we simply kept rolling averages of values logarithmically spaced from 1/4 to 32. We also implemented a training method that kept track of the last 10,000 character estimates (made before the characters were used for training, of course). We used a circular queue for this data structure because its size is fixed and it allowed a constant time insert of the last recorded value. We used one circular queue for each hyperparameter setting, thus storing around 5MB or so worth of samples. These samples can be used to provide an estimate of the best hyperparameter

at any given point in the algorithm's execution. We used this explicit method rather than the much less costly rolling average method so that results would be easier to report. We actually believe just keeping a rolling average of measured cross-entropies on online held-out samples is sufficient.

We also sampled the character stream rather than estimating each character before training. With a gigabyte of characters, we only needed to sample 1 in 100,000 characters to find enough data for estimates. At this rate, online hyperparameter estimate did not measurably affect training time, which was dominated by simply constructing the trie.

We only estimated a single hyperparameter rather than one for each order to avoid having to solve a multivariate estimation problem; although we can collect the data online, we would either have to implement an EM-like solution or spend a lot time per estimate iterating to find optimal parameters. This may be worthwhile for cases where less data is available. As the training data increased, the sensitivity to training parameters decreased. Counterintuitively, we found that recursively estimating each order from low to high, as implemented in (Samuelsson, 1996), actually increased entropy considerably. Clearly the estimator is using the fact that lower-order estimates should not necessarily be optimal for use on their own. This is a running theme of the discounting methods of smoothing such as absolute discounting or Kneser-Ney.

Rather than computing each estimate for hyperparameter and *n*-gram length separately, we first gather the counts for each suffix and each context and the number of outcomes for that context. This is the expensive step, as it require looking up counts in the trie structure. Extension counts require a loop over all the daughters of a context node in the trie because we did not have enough space to store them on nodes. With all of these counts, the *n*-gram etimates for each *n* and each hyperparameter setting can be computed from shortest to longest, with the lower order estimates contributing the smoothed estimate for the next higher order.

4 Substring Counters

Our *n*-gram language models derive estimates from counts of substrings of length *n* or less in the training corpus. Our counter implementation was the trickiest component to scale as it essentially holds the statistics derived from the training data. It contains statistics sufficient to implement all of the estimators defined above. The only non-trivial case is Kneser-Ney, which is typically implemented using the technique known in the compression literature as "update exclusion" (Moffat, 1990). Under update exclusion, if a count "abc" is updated and the context "ab" was known, then counts for "a" and "ab" are excluded from the update process. We actually compute these counts from the total counts by noting that the update exclusion count is equal to the number of unique characters found following a shorter context. That is, the count for "ab" for smoothing is equal to the number of characters "x" such that "xab" has a non-zero count, because these are the situations in which the count of "ab" is not excluded. This is not an efficient way to implement update exclusion, but merely an expedient so we could share implementations for experimental purposes. Straight update exclusion is actually more efficient to implement than full counts, but we wanted the full set of character substring counts for other purposes, as well as language modeling.

```
interface Node {
    Node increment(char[] cs,
                   int start, int end);
    long count(char[] cs,
               int start, int end);
    long extCount(char[] cs,
                  int start, int end);
    int numExts(char[] cs,
                int start, int end);
    Node prune(long minCount);
}
```

Figure 3: Trie Node Interface

Our implementation relies heavily on a data unfolded object-oriented implementation of Patricia tries. Unlike the standard suffix tree algorithms for constructing this trie for all substrings as in (Cleary and Teahan, 1997), we limit the length and make copies of characters rather than pointing back into the original source. This is more space efficient than the suffix-tree approach for our data set sizes and *n*-gram lengths.

The basic node interface is as shown in Figure 3. Note that the interface is in terms of long integer val-

ues. This was necessary to avoid integer overflow in our root count when data size exceeded 2 GB and our 1-gram counts when data sizes exceeded 5 or 6GB. A widely used alternative used for compression is to just scale all the counts by dividing by two (and typically pruning those that go to zero); this allows PPM to use 8-bit counters at the cost of arithmetic precision ((Moffat, 1990)). We eschew pruning because we also use the counts to find significant collocations. Although most collocation and significance statistics are not affected by global scaling, cross-entropy suffers tremendously if scaling is done globally rather than only on the nodes that need it.

Next note that the interface is defined in terms of indexed character slices. This obviates a huge amount of otherwise unnecessary object creation and garbage collection. It is simply not efficient enough, even with the newer generational garbage collectors, to create strings or even lighter character sequences where needed on the heap; slice indices can be maintained in local variables.

The `increment` method increments the count for each prefix of the specified character slice. The `count` method returns the count of a given character sequence, `extensionCount` the count of all one-character extensions, `numExtensions` the number of extensions. The `extensions` method returns all the observed extensions of a character sequence, which is useful for enumerating over all the nodes in the trie.

Global pruning is implemented, but was not necessary for our scalability experiments. It *is* necessary for compilation; we could not compile models nearly as large as those kept online. Just the size of the floating point numbers (two per node for estimate and interpolation) lead to 8 bytes per node. In just about every study every undertaken, including our informal ones, unpruned models have outperformed pruned ones. Unfortunately, applications will typically not have a gigabyte of memory available for models. The best performing models for a given size are those trained on as much data available and pruned to the specified size. Our pruning is simply a minimum count approach, because the other methods have not been shown to improve much on this baseline.

Finally, note that both the increment and prune

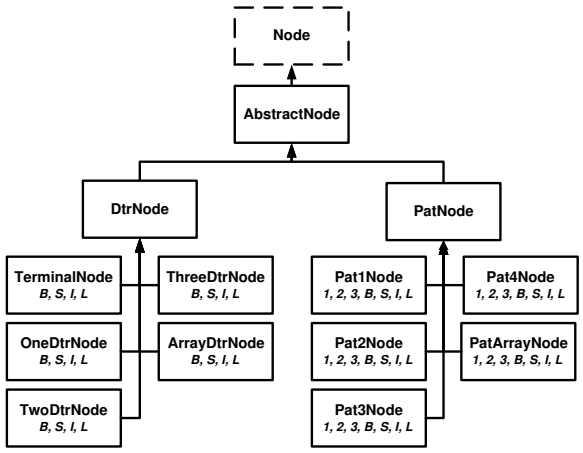

Figure 4: Unfolded Trie Classes

methods return nodes themselves. This is to support the key implementation technique for scalability – replacing immutable objects during increments. Rather than having a fixed mutable node representation, nodes can return results that are essentially replacements for themselves. For instance, there is an implementation of `Node` that provides a count as a byte (8 bits) and a single daughter. If that class gets incremented above the byte range, it returns a node with a short-based counter (16 bits) and a daughter that's the result of incrementing the daughter. If the class gets incremented for a different daughter path, then it returns a two-daughter implementation. Of course, both of these can happen, with a new daughter that pushes counts beyond the byte range. This strategy may be familiar to readers with experience in Prolog (O'Keefe, 1990) or Lisp (Norvig, 1991), where many standard algorithms are implemented this way.

A diagram of the implementations of `Node` is provided in Figure 4. At the top of the diagram is the `Node` interface itself. The other boxes all represent abstract classes, with the top class, `AbstractNode`, forming an abstract adapter for most of the utility methods in `Node` (which were not listed in the interface).

The abstract subclass `DtrNode` is used for nodes with zero or more daughters. It requires its extensions to return parallel arrays of daughters and characters and counts from which it implements all the update methods at a generic level.

```
abstract class TwoDtrNode
    extends DtrNode {

    final char mC1; final Node mDtr1
    final char mC2; final node mDtr2;

    TwoDtrNode(char c1, Node dtr1,
                      char c2, Node dtr2,
        mC1 = c1; mDtr1 = dtr1;
        mC2 = c2; mDtr2 = dtr2;
    }

    Node getDtr(char c) {
        return c == mC1
            ? mDaughter1
            : ( c == mC2
                ? mDaughter2
                : null );
    }

    [] chars() {
        return new char[] { mC1, mC2 };
    }

    Node[] dtrs() {
        return new Node[] { mDaughter1,
                            mDaughter2 };
    }

    int numDtrs() { return 2; }
}
```

Figure 5: Two Daughter Node Implementation

The subclass `TerminalNode` is used for nodes with no daughters. Its implementation is particularly simple because the extension count, the number of extensions and the count for any non-empty sequence starting at this node are zero. The nodes with non-empty daughters are not much more complex. For instance, the two-daughter node abstract class is shown in Figure 5.

All daughter nodes come with four concrete implementations, based on the size of storage allocated for counts: `byte` (8 bits), `short` (16 bits), `int` (32 bits), or `long` (64 bits). The space savings from only allocating bytes or shorts is huge. These concrete implementations do nothing more than return their own counts as long values. For instance, the `short` implementation of three-daughter nodes is shown in Figure 6. Note that because these nodes are not public, the factory can be guaranteed to only call the constructor with a count that can be cast to a short value and stored.

Increments are performed by the superclass

```
final class ThreeDtrNodeShort
    extends ThreeDtrNode {

    final short mCount;

    ThreeDtrNodeShort(char c1, Node dtr1,
                      char c2, Node dtr2,
                      char c3, Node dtr3,
                      long count) {
        super(c1,dtr1,c2,dtr2,c3,dtr3);
        mCount = (short) count;
    }

    long count() { return mCount; }
}
```

Figure 6: Three Daughter Short Node

and will call constructors of the appropriate size. The `increment` method as defined in `AbstractDtrNode` is given in Figure 7. This method increments all the suffixes of a string.

The first line just increments the local node if the array slice is empty; this involves taking its characters, its daughters and calling the factory with one plus its count to generate a new node. This generates a new immutable node. If the first character in the slice is an existing daughter, then the daughter is incremented and the result is used to increment the entire node. Note the assignment to `dtrs[k]` after the increment; this is to deal with the immutability. The majority of the code is just dealing with the case where a new daughter needs to be inserted. Of special note here is the factory instance called on the remaining slice; this will create a PAT node. This appears prohibitively expensive, but we refactored to this approach from a binary-tree based method with almost no noticeable hit in speed; most of the arrays stabilize after very few characters and the resizings of big arrays later on is quite rare. We even replaced the root node implementation which was formerly a map because it was not providing a measurable speed boost.

Once the daughter characters and daughters are marshalled, the factory calls the appropriate constructor based on the number of the character and daughters. The factory then just calls the appropriately sized constructor as shown in Figure 8.

Unlike other nodes, low count terminal nodes are stored in an array and reused. Thus if the result of an increment is within the cache bound, the stored

```
Node increment(char[] cs,
               int start, int end) {
  // empty slice; incr this node
  if (start == end)
    return NodeFactory
           .createNode(chars(),dtrs(),
                       count()+1l);
  char[] dtrCs = chars();
  // search for dtr
  int k = Arrays.binarySearch(dtrCs,
                              cs[start]);
  Node[] dtrs = dtrs();
  if (k >= 0) {  // found dtr
    dtrs[k] = dtrs[k]
              .increment(cs,start+1,end);
    return NodeFactory
           .createNode(dtrCs,dtrs,
                       count()+1l);
  }
  // insert new dtr
  char[] newCs = new char[dtrs.length+1];
  Node[] newDtrs = new Node[dtrs.length+1];
  int i = 0;
  for (; i < dtrs.length
         && dtrCs[i] < cs[start];
       ++i) {
    newCs[i] = dtrCs[i];
    newDtrs[i] = dtrs[i];
  }
  newCs[i] = cs[start];
  newDtrs[i] = NodeFactory
               .createNode(cs,start+1,
                           end,1);
  for (; i < dtrCs.length; ++i) {
       newCs[i+1] = dtrCs[i];
       newDtrs[i+1] = dtrs[i];
  }
  return NodeFactory
         .createNode(newCs,newDtrs,
                     count()+1l);
}
```

Figure 7: Increment in `AbstractDtrNode`

version is returned. Because terminal nodes are immutable, this does not cause problems with consistency. In practice, terminal nodes are far and away the most common type of node, and the greatest saving in space came from carefully coding terminal nodes.

The abstract class `PatNode` implements a so-called "Patricia" trie node, which has a single chain of descendants each of which has the same count. There are four fixed-length implementations for the one, two, three and four daughter case. For these implementations, the daughter characters are stored in member variables. For the array implementation, `PatArrayNode`, the daughter chain is stored

```
static Node createNode(char c, Node dtr,
                       long n) {
  if (n <= Byte.MAX_VALUE)
    return new OneDtrNodeByte(c,dtr,n);
  if (n <= Short.MAX_VALUE)
    return new OneDtrNodeShort(c,dtr,n);
  if (n <= Integer.MAX_VALUE)
    return new OneDtrNodeInt(c,dtr,n);
  return new OneDtrNodeLong(c,dtr,n);
}
```

Figure 8: One Daughter Factory Method

as an array. Like the generic daughter nodes, PAT nodes contain implementations for byte, short, int and long counters. They also contain constant implementations for one, two and three counts. We found in profiling that the majority of PAT nodes had counts below four. By providing constant implementations, no memory at all is used for the counts (other than a single static component per class). Pat nodes themselves are actually more common that regular daughter nodes in high-order character tries, because most long contexts are deterministic. As *n*-gram order increases, so does the proportion of PAT nodes. Implementing increments for PAT nodes is only done once in the abstract class `PatNode`. Each PAT node implementation supplied an array in a standardized interface to the implementations in `PatNode`. That array is created as needed and only lives long enough to carry out the required increment or lookup. Java's new generational garbage collector is fairly efficient at dealing with garbage collection for short-lived objects such as the trie nodes.

5 Compilation

Our online models are tuned primarily for scalability, and secondarily for speed of substring counts. Even the simplest model, Witten-Bell, requires for each context length that exists, summing over extension counts and doing arithmetic including several divisions and multiplications per order a logarithm at the end. Thus straightforward estimation from models is unsuitable for static, high throughput applications. Instead, models may be compiled to a less compact but more efficient static representation.

We number trie nodes breadth-first in unicode order beginning from the root and use this indexing for four parallel arrays following (Whittaker and Raj, 2001). The main difference is that we have not

		char	int	float	float	int
Idx	Ctx	C	Suf	$\log P$	$log(1\text{-}\lambda)$	Dtr
0	n/a	n/a	n/a	n/a	-0.63	1
1		a	0	-2.60	-0.41	6
2		b	0	-3.89	-0.58	9
3		c	0	-4.84	-0.32	10
4		d	0	-4.84	-0.32	11
5		r	0	-3.89	-0.58	12
6	a	b	2	-2.51	-0.58	13
7	a	c	3	-3.49	-0.32	14
8	a	d	4	-3.49	-0.32	15
9	b	r	5	-1.40	-0.58	16
10	c	a	1	-1.59	-0.32	17
11	d	a	1	-1.59	-0.32	18
12	r	a	1	-1.17	-0.32	19
13	ab	r	9	-0.77	n/a	n/a
14	ac	a	10	-1.10	n/a	n/a
15	ad	a	11	-1.10	n/a	n/a
16	br	a	12	-0.67	n/a	n/a
17	ca	d	8	-1.88	n/a	n/a
18	da	b	6	-1.55	n/a	n/a
19	ra	c	7	-1.88	n/a	n/a

Figure 9: Compiled Representation of 3-grams for "abracadabra"

coded to a fixed *n*-gram length, costing us a bit of space in general, and also that we included context suffix pointers, costing us more space but saving lookups for all suffixes during smoothing.

The arrays are (1) the character leading to the node, (2) the log estimate of the last character in the path of characters leading to this node given the previous characters in the path, (3) the log of one minus the interpolation parameter for the context represented by the full path of characters leading to this node, (4) the index of the first daughter of the node, and (5) index of the suffix of this node. Note that the daughters of a given node will be contiguous and in unicode order given the breadth-first nature of the indexing, ranging from the daughter index of the node to the daughter index of the next node.

We show the full set of parallel arrays for trigram counts for the string "abracadabra" in Figure 9. The first column is for the array index, and is not explicitly represented. The second column, labeled "Ctx", is the context, and this is also not explicitly represented. The remaining columns are explicitly represented. The third column is for the character. The fourth column is an integer backoff suffix pointer; for instance, in the row with index 13, the context is "ab", and the character is "r", meaning it represents "abr" in the trie. The suffix index is 9, which is for "br", the suffix of "abr". The fifth and sixth columns are 32-bit floating point estimates, the fifth of $\log_2 P(r|ab)$, and the sixth is empty because there is no context for "abr", just an outcome. The value of $\log_2(1-\lambda(ab))$ is found in the row indexed 6, and equal to -0.58. The seventh and final column is the integer index of the first daughter of a given node. The value of the daughter pointer for the following node provides an upper bound. For instance, in the row index 1 for the string "a", the daughter index is 6, and the next row's daughter index is 9, thus the daughters of "a" fall between 6 and 8 inclusively — these are "ab", "ac" and "ad" respectively. Note that the daughter characters are always in alphabetical order, allowing for a binary search for daughters.

For *n*-gram estimators, we need to compute $\log P(c_n|c_0\cdots c_{n-1})$. We start with the longest sequence $c_k,\ldots,c_{n-1}$ that exists in the trie. If binary search finds the outcome c_n among the daughters of this node, we return its log probability estimate; this happens in over 90 percent of estimates with reasonably sized training sets. If the outcome character is not found, we continue with shorter and shorter contexts, adding log interpolation values from the context nodes until we find the result or reach the uniform estimate at the root, at which point we add its estimate and return it. For instance, the estimate of $\log_2 P(r|ab) = -0.77$ can be read directly off the row indexed 13 in Figure 9. But $\log_2 P(a|ab) = -0.58 + \log_2 P(a|b) = -0.58 + -0.58 + \log_2 P(a) = -0.58 + -0.58 + -2.60$, requiring two interpolation steps.

For implementation purposes, it is significant that we keep track of where we backed off from. The row for "a", where the final estimate was made, will be the starting point for lookup next time. This is the main property of the fast string algorithms — we know that the context "ba" does not exist, so we do not need to go back to the root and start our search all over again at the next character. The result is a linear bound on lookup time because each back-

off of *n* characters guarantees at least *n* steps to get back to the same context length, thus there can't be more backoff steps than characters input. The main bottleneck in run time is memory bandwidth due to cache misses.

The log estimates can be compressed using as much precision as needed (Whittaker and Raj, 2001), or even reduced to integral values and integer arithmetic used for computing log estimates. We use floats and full characters for simplicity and speed.

6 Corpora and Parsers

Our first corpus is 7 billion characters from the *New York Times* section of the Linguistic Data Consortium's Gigaword corpus. Only the body of documents of type story were used. Paragraphs indicated by XML markup were begun with a single tab character. All newlines were converted to single whitepspaces, and all other data was left unmodified. The data is problematic in at least two ways. First, the document set includes repeats of earlier documents. Language models provide a good way of filtering these repeated documents out, but we did not do so for our measurements because there were few enough of them that it made little difference and we wanted to simplify other comparative evaluations. Second, the document set includes numerical list data with formatting such as stock market reports. The *Times* data uses 87 ASCII characters.

Our second corpus is the 5 billion characters drawn from abstracts in the United States' National Library of Medicine's 2004 MEDLINE baseline citation set. Abstract truncation markers were removed. MEDLINE uses a larger character set of 161 characters, primarily extending ASCII with diacritics on names and Greek letters.

By comparison, (Banko and Brill, 2001) used one billion tokens for a disambiguation task, (Brown et al., 1991) used 583 million tokens for a language model task, and (Chen and Goodman, 1996) cleverly sampled from 250 million tokens to evaluate higher-order models by only training on sequences used in the held-out and test sets.

Our implementation is based a generic text parser and text handler interface, much like a simplified version of XML's SAX content handler and XML parser. A text parser is implemented for the various data sets, including decompressing their zipped and gzipped forms and parsing their XML, SGML and tokenized form. A handler is then implemented that adds data to the online models and polls the model for results intermittently for generating graphs.

7 Results

We used a 1.4GB Java heap (unfortunately, the maximum allowable with Java on 32-bit Intel hardware without taking drastic measures), which allowed us to train 6-grams on up to 7 billion characters with room to spare. Roughly, 8-grams ran out of memory at 1 billion characters, 12 grams at 100 million characters, and 32 grams at 10 million characters. We did not experiment with pruning for this paper, though our API supports both thresholded and pdivisive scaling pruning. Training the counters depends heavily on the length of *n*-gram, with 5-grams training at 431,000 characters per second, 8-grams at 204,000 char/s, 12-grams at 88,000 char/s and 32-grams at 46,000 char/s, including online hyperparameter estimation (using a $2000 PC running Windows XP and Sun's 1.4.2 JDK, with a 3.0GHz Pentium 4, 2GB of ECC memory at 800MHz, and two 10K SATA drives in RAID 0).

Our primary results are displayed in Figure 11 and Figure 10, which plot sample cross-entropy rates against amount of text used to build the models for various *n*-gram lengths. Sample cross entropy is simply the average log (base 2) probability estimate per character. All entropies are reported for the best hyperparameter settings through online leave-one-out estimation for parameterized Witten-Bell smoothing. Each data point in the plot uses the average entropy rate over a sample size of up to 10,000 for MEDLINE and 100,000 for the *Times*, with the samples being drawn evenly over the data arriving since the last plot point. For instance, the point plotted at 200,000 characters for MEDLINE uses a sample of every 10th character between character 100,000 and 200,000 whereas the sample at 2,000,000,000 characters uses every 100,000th character between characters 1,000,000,000 and 2,000,000,000.

Like the Tipster data used by (Chen and Goodman, 1996), the immediately noticeable feature of the plots is the jaggedness early on, including some

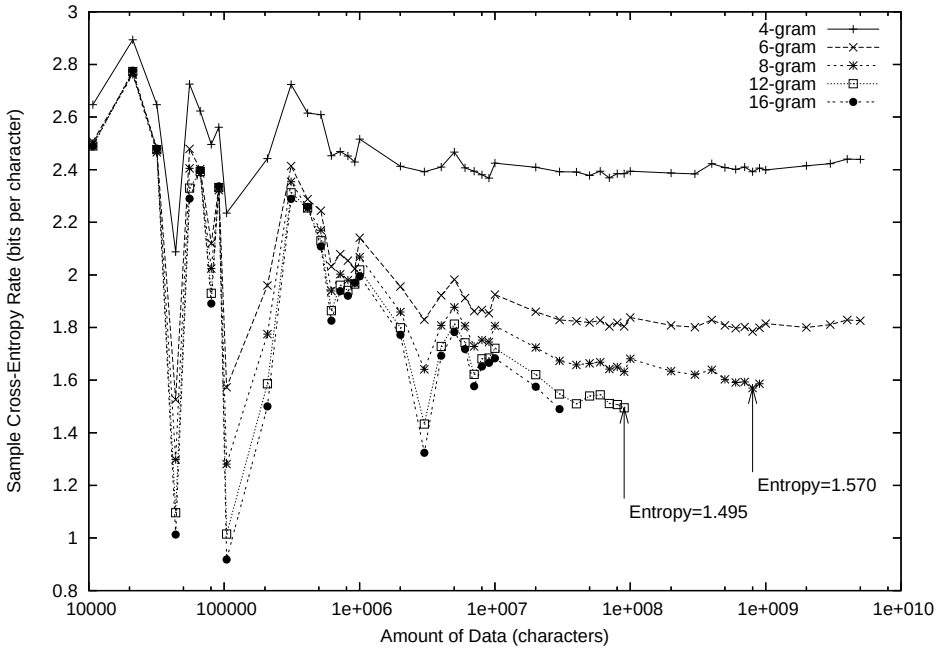

Figure 10: *NY Times* Sample Cross-Entropy Rates

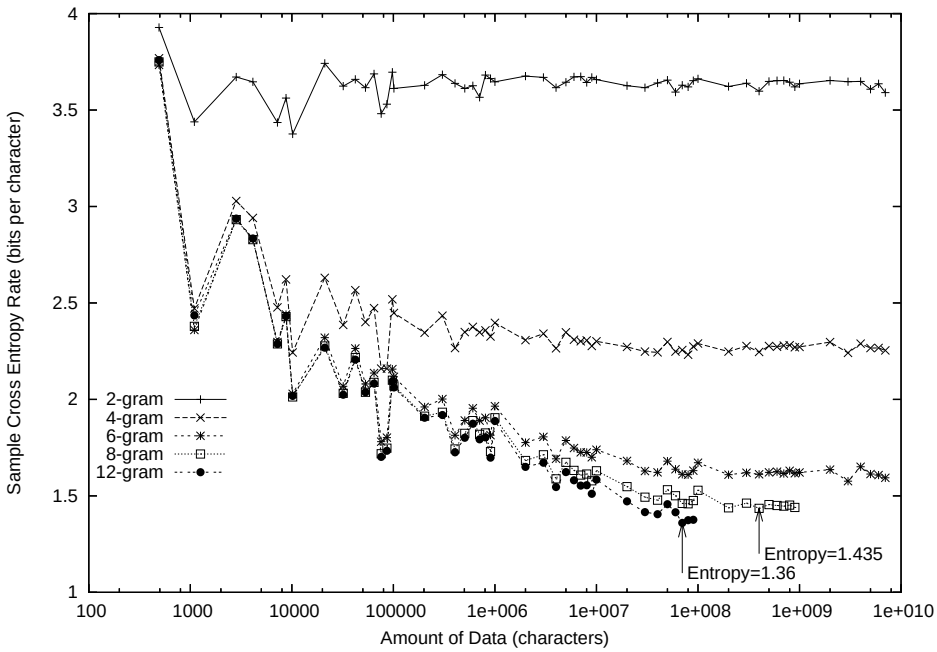

Figure 11: MEDLINE Sample Cross-Entropy Rates

ridiculously low cross-entropy rates reported for the *Times* data. This is largely due to low training data count, high *n*-gram models being very good at matching repeated passages coupled with the fact that a 2000 word article repeated out of 10,000 sample characters provides quite a cross-entropy reduction. For later data points, samples are sparser and thus less subject to variance.

For applications other than cross-entropy bake-offs, 5-grams to 8-grams seem to provide the right

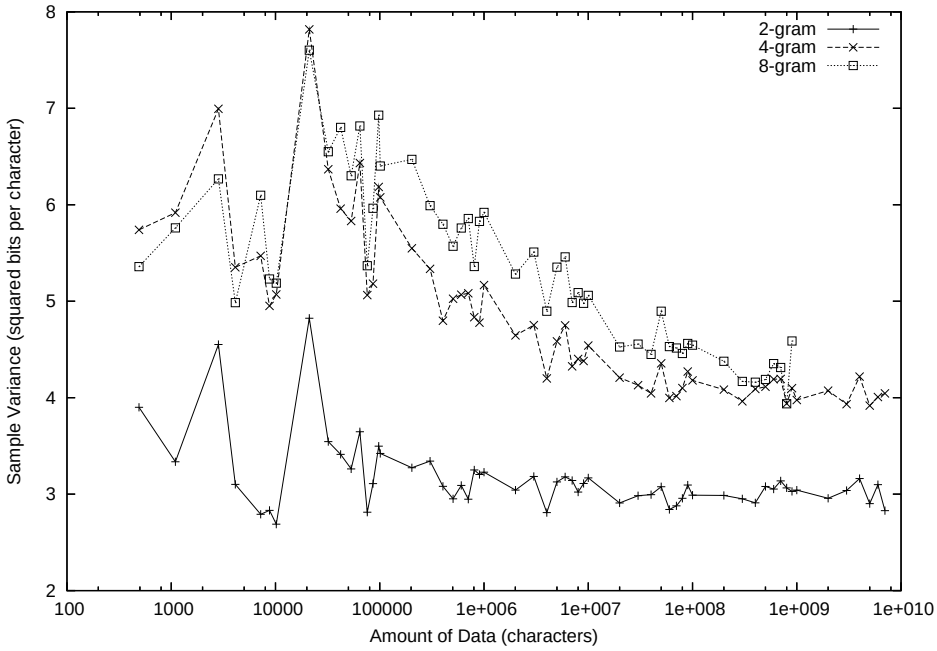

Figure 12: MEDLINE Sample Variances

compromise between accuracy and efficiency.

We were surprised that MEDLINE had lower *n*-gram entropy bounds than the *Times*, especially given the occurrence of duplication within the *Times* data (MEDLINE does not contain duplicates in the baseline). The best MEDLINE operating point is indicated in the figure, with a sample cross-entropy rate of 1.36 for 12-grams trained on 100 million characters of data; 8-gram entropy is 1.435 at nearly 1 billion characters. The best performance for the *Times* corpus was also for 12-grams at 100 million characters, but the sample cross-entropy was 1.49; with 8-gram sample cross-entropy as low as 1.570 at 1 billion characters. Although MEDLINE may be full of jargon and mixed-case alphanumeric acronyms, the way in which they are used is highly predictable given enough training data. Data in the *Times* such as five and six digit stock reports, sports scores, etc., seem to provide a challenge.

The per-character sample variances for 2-grams, 4-grams and 8-grams for MEDLINE are given in Figure 12. We did not plot results for higher-order *n*-grams, as their variance was almost identical to that of 8-grams. Standard error is the square root of variance, or about 2.0 in the range of interest. With 10,000 samples, variance should be 4/10,000, with standard error the square root of this, or 0.02. This is in line with measurement variances found at the tail end of the plots, but not at the beginnings.

Most interestingly, it turned out that smoothing method did not matter once *n*-grams were large, thus bringing the results of (Banko and Brill, 2001) to bear on those of (Chen and Goodman, 1996). The comparison for 12-grams and then for the tail of more data for 8-grams in Figures 13 and 14. Figure 14 shows the smoothing methods for 8-grams on an order of magnitude more data.

Conclusions

We have shown that it is possible to use object oriented techniques to scale language model counts to very high levels without pruning on relatively modest hardware. Even more space could be saved by unfolding characters to bytes (especially for token models). Different smoothing models tend to converge to each other after gigabytes of data, making smoothing much less critical.

Full source with unit tests, javadoc, and applications is available from the LingPipe web site:

`http://www.alias-i.com/lingpipe`

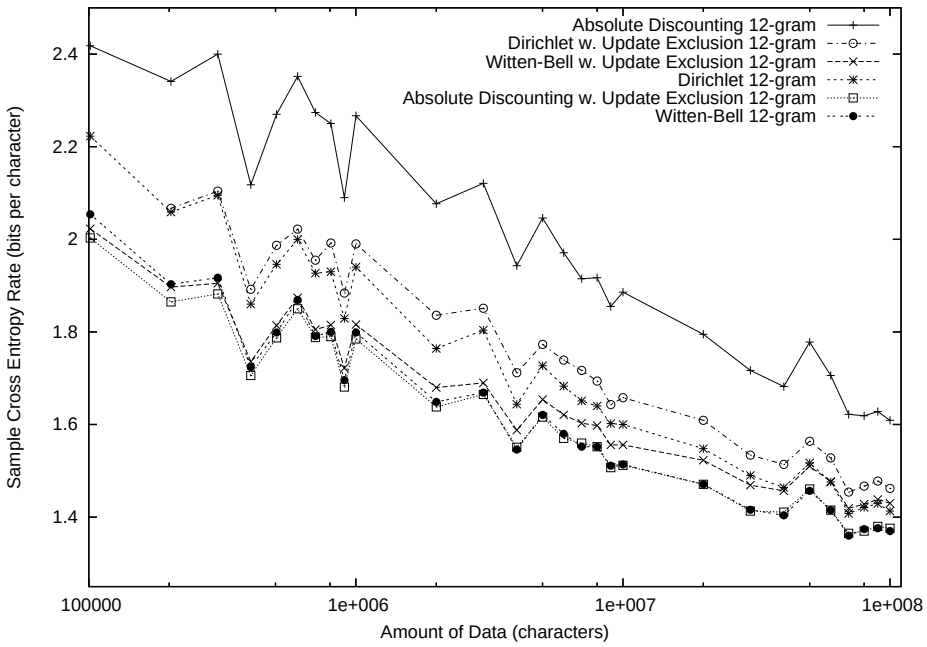

Figure 13: Comparison of Smoothing for 12-grams

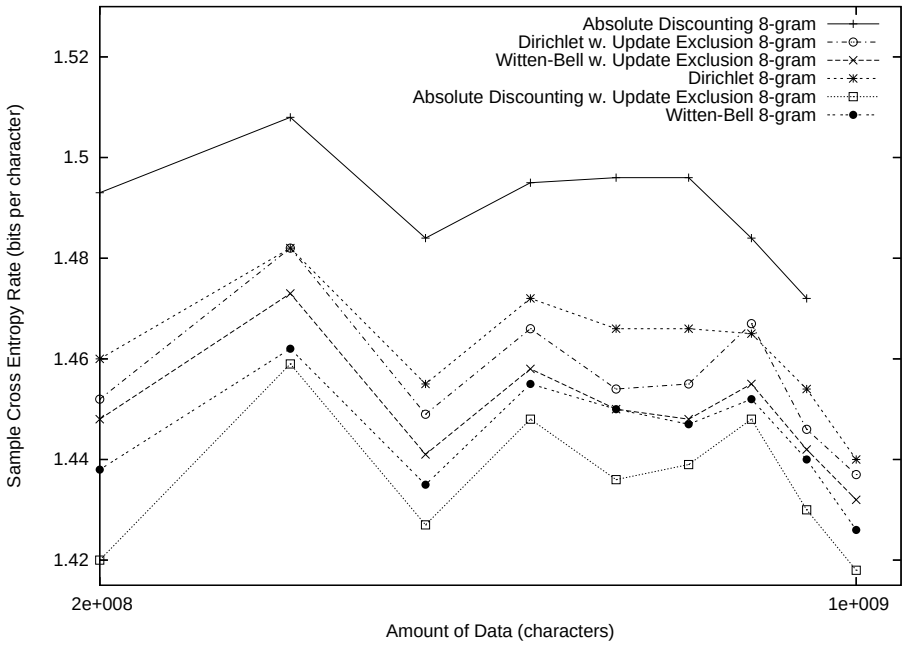

Figure 14: Comparison of Smoothing for 8-grams

References

Michele Banko and Eric Brill. 2001. Scaling to very very large corpora for natural language disambiguation. In *Proceedings of the 39th Meeting of the ACL*.

Eric Brill and Robert C. Moore. 2000. An improved error model for noisy channel spelling correction. In *Proceedings of the 38th Annual Meeting of the ACL*.

Peter F. Brown, Stephen Della Pietra, Vincent J. Della Pietra, and Robert L. Mercer. 1991. Word-sense disambiguation using statistical methods. pages 264–270.

Stanley F. Chen and Joshua Goodman. 1996. An empirical study of smoothing techniques for language modeling. In *Proceedings of the 34th Annual Meeting of the ACL*, pages 310–318.

John G. Cleary and William J. Teahan. 1997. Unbounded length contexts for PPM. *The Computer Journal*, 40(2/3):67–??

Thomas M. Cover and Joy A. Thomas. 1991. *Elements of Information Theory*. John Wiley.

Frederick Jelinek and Robert L. Mercer. 1980. Interpolated estimation of Markov source parameters from sparse data. In *Proceedings of the Workshop on Pattern Recognition in Practice*. North-Holland.

Dan Klein, Joseph Smarr, Huy Nguyen, and Christopher D. Manning. 2003. Named entity recognition with character-level models. In *Proceedings the 7th ConNLL*, pages 180–183.

Reinhard Kneser and Hermann Ney. 1995. Improved backing off for n-gram language modeling. In *Proceedings of ICASSP*, pages 181–184.

Kevin Knight and Vasileios Hatzivassiloglou. 1995. Two-level, many-paths generation. In *Proceedings of the 33rd Annual Meeting of the ACL*.

Lucian Vlad Lita, Abe Ittycheriah, Salim Roukos, and Nanda Kambhatla. 2003. tRuEcasIng. In *Proceedings of the 41st Annual Meeting of the ACL*, pages 152–159.

David J. C. MacKay and Linda C. Peto. 1995. A hierarchical Dirichlet language model. *Natural Language Engineering*, 1(3):1–19.

Alistair Moffat. 1990. Implementing the PPM data compression scheme. *IEEE Transactions on Communications*, 38:1917–1921.

Hermann Ney, U. Essen, and Reinhard Kneser. 1995. On the estimation of 'small' probabilities by leaving-one-out. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 17:1202–1212.

Peter Norvig. 1991. *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp*. Morgan Kaufmann.

Richard O'Keefe. 1990. *The Craft of Prolog*. MIT Press.

Fuchun Peng. 2003. *Building Probabilistic Models for Language Independent Text Classification*. Ph.D. thesis.

Gerasimos Potamianos and Frederick Jelinek. 1998. A study of n-gram and decision tree letter language modeling methods. *Speech Communication*, 24(3):171–192.

Christer Samuelsson. 1996. Handling sparse data by successive abstraction. In *Proceedings of COLING-96*, Copenhagen.

William J. Teahan and John G. Cleary. 1996. The entropy of english using PPM-based models. In *Data Compression Conference*, pages 53–62.

William J. Teahan. 2000. Text classification and segmentation using minimum cross-entropy. In *Proceeding of RIAO 2000*.

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

ChengXiang Zhai and John Lafferty. 2004. A study of smoothing methods for language models applied to information retrieval. *ACM Transactions on Information Systems*, 2(2):179–214.