A Stochastic Memoizer for Sequence Data
Wood, Archambeau, Gasthaus, James, and Teh

Presented by: Will Allen

October 27, 2011
Quick Note

- This paper builds on the Teh’s 2006 ACL article on PYP for language models presented on Tuesday.
- And relies on details about the coagulation and fragmentation operators in Gasthaus and Teh’s 2010 NIPS article presented next.
- So I’ll go over those topics relatively quickly and incompletely.
Problem: Want to model sequences of symbols $x_{1:T} = (x_1 x_2 ... x_T) \in \Sigma^*$, without making Markov assumptions. (Preferably maintaining power-law symbol occurrence statistics.)

E.g.: Given some new symbol, $x_{T+1}$, we’d like to find $p(x_{T+1} = s | x_{1:T}) \forall s \in \Sigma$.

This requires a large (infinite) number of latent variables.

Solution: Use a tree data structure, and clever use of marginalization, to efficiently represent a hierarchical Pitman-Yor process prior over the predictive distribution.
Markov Models

- Normally, when modeling language, we make a *Markov assumption*:

  - Given sequence $x_{1:T} = (x_1x_2...x_T)$, for $x_i \in \Sigma$, where $\Sigma$ is a set of symbols, assume each $x_i$ depends on the previous $n$ variables in the sequence:

    $$p(x_{1:T}) = \prod_{i=1}^{T} p(x_i|x_{(i-n+1):i-1})$$

- As $n$ gets larger, computational complexity grows and probability of each $n$-gram occurring goes down, and smoothing is required.

- In the paper we covered last class, Teh used a Pitman-Yor process prior on the previous $n$ words for Bayesian smoothing.
What if we let \( n \) grow with the length of the data?

Get a non-Markov model:

\[
p(x_{1:T}) = \prod_{i=1}^{T} p(x_i | x_{1:i-1})
\]

Each symbol is conditioned on every previous symbol.

How do we actually compute this?
The Sequence Memoizer Model

- For each symbol $s \in \Sigma$, and some context $\mathbf{u}$, create a latent variable $G_\mathbf{u} = [G_\mathbf{u}(s)]_{s \in \Sigma}$ (a probability vector). (I.e. $G_\mathbf{u}(s) = p(u_{T+1} = s | u_{1:T})$).

- Let $G = \{ G_s \}_{s \in \Sigma^*}$ be the (infinite) set of all such probability vectors for every possible sequence made from elements of $\Sigma$.

- So $p(\mathbf{x}_{1:T}, G) = p(G) \prod_{i=1}^{T} G_{x_{1:i-1}}(x_i)$ for the particular sequences we observe. Notice that this is recursive.

- But what is $p(G)$?
The Sequence Memoizer Model

- Take $\mathcal{G}$ to be a Pitman-Yor process prior.
- They set $c$, the concentration parameters, to always be 0. Zach’s paper covers the more general case.
- With this choice, we can model the power-law properties of language.
- Will also use some nice marginalization properties later.
The Sequence Memoizer Model

In particular, the Sequence Memoizer model gives a distribution over $G = \{ G_u \}_{u \in \Sigma^*}$ using hierarchical Pitman-Yor process:

$$
G[] | d_0, H \sim \mathcal{PY}(d_0, 0, H)
\quad
G[u] | G[\sigma(u)], d[u] \sim \mathcal{PY}(d[s], 0, G[\sigma(u)]) \quad \forall u \in \Sigma^+
\quad
x_i | x_{1:i-1} = u \sim G[u]
$$

where $\sigma(u)$ means the suffix of context $u$ (e.g. if $u = abcd$, $\sigma(u) = bcd$).

Encodes prior knowledge that contexts sharing suffixes will be similar to each other, so later symbols in a context will be more important in prediction.
The Sequence Memoizer Model: Infinite

This hierarchy can be viewed as an infinite tree (a context tree), beginning at the empty root node, where each node has a branch for each $s \in \Sigma$. E.g. for $\Sigma = \{0, 1\}$:

![Context Tree Diagram]


The parent of node $u$ is $\sigma(u)$, the longest proper suffix of that node. (E.g. 110 is the longest proper suffix of 0110).
Prefix Trie: $O(T^2)$

- When given a particular sequence (e.g. $x = 0110$), we can integrate all of the nodes in the context tree not associated with data in $x$. The resulting tree looks like a suffix trie.
- Every prefix is a path in the tree.
- Requires $O(T^2)$ time and space to build this tree for a sequence of length $T$.
- Intuition: One-to-one correspondence between nodes of suffix trie and distinct substrings.
Prefix Tree: $O(T)$

- Obtained by compacting non branching, non leaf nodes.
- If we need those internal nodes, can recreate them.
- Better algorithm requires only $O(T)$ time and space to build! (At most $2T$ nodes.)

Coagulation and Fragmentation

- Key concept: Compacting internal nodes of prefix trie ↔ marginalizing PYP.
- For certain parameter settings, chains of conditional PYP are closed under marginalization.
- Theorem: If $G_2|G_1 \sim \mathcal{PY}(d_1, 0, G_1)$ and $G_3|G_2 \sim \mathcal{PY}(d_2, 0, G_2)$, then $G_3|G_1 \sim \mathcal{PY}(d_1d_2, 0, G_1)$ with $G_2$ marginalized out.
- Just multiply discount parameters along collapsed edge!
- Can also go backwards, to recreate $G_2$. Zach may go over this in more detail, shortly.
Inference Algorithm: Posterior

Intractable to do exact inference in this model, so they use a Gibbs sampler. Zach will probably cover this in his talk.

- Building the suffix tree for $x$ gives the structure of a graphical model.
- Traverse that tree, collecting parameters for a hierarchical Pitman-Yor process.
- Instantiate a Chinese Restaurant Franchise representation of the HPYP.
- Use Gibbs sampling to simulate the posterior distribution conditioned on the observed sequence (as with any other CRF).
Inference Algorithm: Prediction

- Given some context $s$ not in the training set, and some next symbol $v$, want to compute $p(v|s, x)$.
- $p(v|s, x) = \mathbb{E}[G[s](v)] = \mathbb{E}[G[s'](v)]$, where $s'$ is the longest suffix of $s$ in the prefix trie.
- If $s'$ doesn’t appear in the prefix tree, can use fragmentation to reinstate the corresponding restaurants into the model.
- $\mathbb{E}[G[s](v)] = \mathbb{E} \left[ \frac{N(sv) - d_s | M(sv) + \sum_{v' \in \Sigma} d_s | M(sv') G_{\sigma(s)}(v)}{\sum_{v' \in \Sigma} N(sv')} \right]$, where $\{N(s'v'), M(s'v')\}$ are random counts given some context $s'$ and symbol $v'$.
- Use samples from posterior distribution to approximate this expectation.
Results

- On New York Times corpus and AP corpus
- Used CRF sampler with special Metropolis-Hastings updates for discount parameters (because collapsed nodes have products of discount parameters).
- Did really short burn-in (10 iterations) and collected 5 samples.
Results: Number of nodes in tree and number which require sampling

Shown as function of number of New York Times observations. Grows linearly with corpus size. Leaf nodes don’t require sampling.
Results: Nodes in prefix trees vs \( n \)-gram trie

Figure 4. Nodes in prefix tries and trees as a function of \( n \) (of \( n \)-gram) and for different NYT corpora sizes. Horizontal lines are prefix tree node counts. Curved lines are prefix trie node counts. Sizes of corpora are given in the legend.

Figure 5. NYT test perplexity for \( n \)-gram and \( \infty \)-gram HPYP language models given a 4 million word subset of the NYT training corpus. The dotted line indicates the first \( n \)-gram model that has more nodes than the \( \infty \)-gram model.

Significantly beyond \( n = 6 \) we observe that this linear growth continues for a long time. This transition between quadratic and linear growth can be explained by observing that virtually all branch points of the trie occur above a certain depth, and below this depth only linear paths remain. Also, at \( n = 5 \) the number of nodes in the \( n \)-gram trie is roughly the same (greater in all but one case) as the number of nodes in the \( \infty \)-gram.

Questions of model size automatically lead to questions of the expressive power of the models. Figure 5 compares the expressive power of the \( n \)-gram HPYP language model against the \( \infty \)-gram model, using the test set perplexity as a proxy. We see that the predictive performance of the \( n \)-gram HPYP asymptotically approaching that of the \( \infty \)-gram. While the performance gain over the \( n \)-gram model is modest, and certainly goes to zero as \( n \) increases, remember that the computational cost associated with the \( n \)-gram surpasses the \( \infty \)-gram after \( n = 5 \). This indicates that there is no reason, computational or statistical, for preferring \( n \)-gram models over the \( \infty \)-gram when \( n \geq 5 \).

In the next set of experiments we switch to using the AP corpus instead. Figure 6 shows the test perplexities of both the 5-gram HPYP and the \( \infty \)-gram fit to AP corpora of increasing size. For small training corpora its performance is indistinguishable from that of the \( \infty \)-gram. Furthermore, as the corpus size grows, enough evidence about meaningful long contexts begins to accrue to give the \( \infty \)-gram a slight relative advantage. It bears repeating here that the AP corpus is preprocessed in a way that will minimize this advantage.

Finally, despite the AP corpus being semi-adversarially preprocessed, the \( \infty \)-gram achieves the best test perplexity of any language model we know of that has been applied to the AP corpus. Comparative results are given in Table 1. This is somewhat surprising and worth further study.

Remember the trade-off between using the \( \infty \)-gram vs. the \( n \)-gram HPYP: the \( n \)-gram HPYP allows for non-zero concentrations at all levels, whereas the \( \infty \)-gram requires all concentrations to be set to zero. Conversely, the advantage of the \( \infty \)-gram is that it can utilize arbitrarily long contexts whereas the

\[ \frac{3.8 \times 10^5}{3.9 \times 10^6} \]

Horizontal lines are tree counts, curved lines are trie counts. Note: as amount of data grows, tree has about same number of nodes as a 5-gram.
Results: Sequence Memoizer vs \( n \)-gram performance

The Sequence Memoizer always below HPYP. Af \( n = 5 \), HPYP begins to have more nodes (by previous figure).
Results

Figure 4. Nodes in prefix tries and trees as a function of \( n \) (of \( n \)-gram) and for different NYT corpora sizes. Horizontal lines are prefix tree node counts. Curved lines are prefix trie node counts. Sizes of corpora are given in the legend.

Figure 5. NYT test perplexity for \( n \)-gram and \( \infty \)-gram HPYP language models given a 4 million word subset of the NYT training corpus. The dotted line indicates the first \( n \)-gram model that has more nodes than the \( \infty \)-gram model.

Using a 5-gram HPYP model. \( \infty \)-gram becomes better than 5-gram as dataset size increases.

Questions of model size automatically lead to questions of the expressive power of the models. Figure 5 compares the expressive power of the \( n \)-gram HPYP language model against the \( \infty \)-gram model, using the test set perplexity as a proxy. We see that the predictive performance of the \( n \)-gram HPYP asymptotically approaches that of the \( \infty \)-gram. While the performance gain over the \( n \)-gram model is modest, and certainly goes to zero as \( n \) increases, remember that the computational cost associated with the \( n \)-gram surpasses the \( \infty \)-gram after \( n = 5 \). This indicates that there is no reason, computational or statistical, for preferring \( n \)-gram models over the \( \infty \)-gram when \( n \geq 5 \).

In the next set of experiments we switch to using the AP corpus instead. Figure 6 shows the test perplexities of both the 5-gram HPYP and the \( \infty \)-gram fit to AP corpora of increasing size. For small training corpora its performance is indistinguishable from that of the \( \infty \)-gram. Furthermore, as the corpus size grows, enough evidence about meaningful long contexts begins to accrue to give the \( \infty \)-gram a slight relative advantage. It bears repeating here that the AP corpus is preprocessed in a way that will minimize this advantage.

Finally, despite the AP corpus being semi-adversarially preprocessed, the \( \infty \)-gram achieves the best test perplexity of any language model we know of that has been applied to the AP corpus. Comparative results are given in Table 1. This is somewhat surprising and worth further study.

Comparative results are given in Table 1. This is somewhat surprising and worth further study.

<table>
<thead>
<tr>
<th>Source</th>
<th>Perplexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Mnih &amp; Hinton, 2009)</td>
<td>112.1</td>
</tr>
<tr>
<td>(Bengio et al., 2003)</td>
<td>109.0</td>
</tr>
<tr>
<td>4-gram Modified Kneser-Ney (Teh, 2006)</td>
<td>102.4</td>
</tr>
<tr>
<td>4-gram HPYP (Teh, 2006)</td>
<td>101.9</td>
</tr>
<tr>
<td>( \infty )-gram (Sequence Memoizer)</td>
<td>96.9</td>
</tr>
</tbody>
</table>

Using a 5-gram HPYP model. \( \infty \)-gram becomes better than 5-gram as dataset size increases.
Results

Figure 4. Nodes in prefix tries and trees as a function of \( n \) (of \( n \)-gram) and for different NYT corpora sizes. Horizontal lines are prefix tree node counts. Curved lines are prefix trie node counts. Sizes of corpora are given in the legend.

Figure 5. NYT test perplexity for \( n \)-gram and \( \infty \)-gram HPYP language models given a 4 million word subset of the NYT training corpus. The dotted line indicates the first \( n \)-gram model that has more nodes than the \( \infty \)-gram model.

Questions of model size automatically lead to questions of the expressive power of the models. Figure 5 compares the expressive power of the \( n \)-gram HPYP language model against the \( \infty \)-gram model, using the test set perplexity as a proxy. We see that the predictive performance of the \( n \)-gram HPYP asymptotically approaching that of the \( \infty \)-gram. While the performance gain over the \( n \)-gram model is modest, and certainly goes to zero as \( n \) increases, remember that the computational cost associated with the \( n \)-gram surpasses the \( \infty \)-gram after \( n = 5 \). This indicates that there is no reason, computational or statistical, for preferring \( n \)-gram models over the \( \infty \)-gram when \( n \geq 5 \).

In the next set of experiments we switch to using the AP corpus instead. Figure 6 shows the test perplexities of both the 5-gram HPYP and the \( \infty \)-gram fit to AP corpora of increasing size. For small training corpora its performance is indistinguishable from that of the \( \infty \)-gram. Furthermore, as the corpus size grows, enough evidence about meaningful long contexts begins to accrue to give the \( \infty \)-gram a slight relative advantage. It bears repeating here that the AP corpus is preprocessed in a way that will minimize this advantage.

Finally, despite the AP corpus being semi-adversarially preprocessed, the \( \infty \)-gram achieves the best test perplexity of any language model we know of that has been applied to the AP corpus. Comparative results are given in Table 1. This is somewhat surprising and worth further study.

Recall the trade-off between using the \( \infty \)-gram vs. the \( n \)-gram HPYP: the \( n \)-gram HPYP allows for non-zero concentrations at all levels, whereas the \( \infty \)-gram requires all concentrations to be set to zero. Conversely, the advantage of the \( \infty \)-gram is that it can utilize arbitrarily long contexts whereas the...

<table>
<thead>
<tr>
<th>Source</th>
<th>Perplexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Mnih &amp; Hinton, 2009)</td>
<td>112.1</td>
</tr>
<tr>
<td>(Bengio et al., 2003)</td>
<td>109.0</td>
</tr>
<tr>
<td>4-gram Modified Kneser-Ney (Teh, 2006)</td>
<td>102.4</td>
</tr>
<tr>
<td>4-gram HPYP (Teh, 2006)</td>
<td>101.9</td>
</tr>
<tr>
<td>( \infty )-gram (Sequence Memoizer)</td>
<td>96.9</td>
</tr>
</tbody>
</table>

Very good perplexity results!
Application: Compression

- Used the predictive ability of the SM to very efficiently compress text.
- Developed an approximate incremental inference algorithm for the SM.
**Table 1:** Compression performance in terms of average log-loss (average bits per character under optimal entropy encoding) for the Calgary corpus. Boldface type indicates best performance. Ties are resolved in favour of lowest computational complexity. The results for PPM* (PPM with unbounded-length contexts) are copied from [3] and are actual compression rates, while the results for PPMZ are average log-losses obtained using a modified version of PPMZ 9.1 under Linux [8] (which differ slightly from the published compression rates). The results for CTW were taken from [14].

<table>
<thead>
<tr>
<th>File</th>
<th>Size</th>
<th>DEPLUMP</th>
<th>PPM</th>
<th>PPMZ</th>
<th>CTW</th>
</tr>
</thead>
<tbody>
<tr>
<td>bib</td>
<td>111261</td>
<td>1.73</td>
<td><strong>1.72</strong></td>
<td>1.91</td>
<td>1.74</td>
</tr>
<tr>
<td>book1</td>
<td>768771</td>
<td>2.17</td>
<td>2.20</td>
<td>2.40</td>
<td>2.21</td>
</tr>
<tr>
<td>book2</td>
<td>610856</td>
<td>1.83</td>
<td>1.84</td>
<td>2.02</td>
<td>1.87</td>
</tr>
<tr>
<td>geo</td>
<td>102400</td>
<td><strong>4.40</strong></td>
<td>4.40</td>
<td>4.83</td>
<td>4.64</td>
</tr>
<tr>
<td>news</td>
<td>377109</td>
<td><strong>2.20</strong></td>
<td>2.20</td>
<td>2.42</td>
<td>2.24</td>
</tr>
<tr>
<td>obj1</td>
<td>21504</td>
<td><strong>3.64</strong></td>
<td>3.65</td>
<td>4.00</td>
<td>3.66</td>
</tr>
<tr>
<td>obj2</td>
<td>246814</td>
<td>2.21</td>
<td><strong>2.19</strong></td>
<td>2.43</td>
<td>2.23</td>
</tr>
<tr>
<td>paper1</td>
<td>53161</td>
<td>2.21</td>
<td><strong>2.20</strong></td>
<td>2.37</td>
<td>2.22</td>
</tr>
<tr>
<td>paper2</td>
<td>82199</td>
<td><strong>2.18</strong></td>
<td>2.18</td>
<td>2.36</td>
<td>2.21</td>
</tr>
<tr>
<td>pic</td>
<td>513216</td>
<td>0.77</td>
<td>0.82</td>
<td>0.85</td>
<td><strong>0.76</strong></td>
</tr>
<tr>
<td>progc</td>
<td>39611</td>
<td>2.23</td>
<td><strong>2.21</strong></td>
<td>2.40</td>
<td>2.25</td>
</tr>
<tr>
<td>progl</td>
<td>71646</td>
<td>1.44</td>
<td><strong>1.43</strong></td>
<td>1.67</td>
<td>1.46</td>
</tr>
<tr>
<td>progp</td>
<td>49379</td>
<td>1.44</td>
<td><strong>1.42</strong></td>
<td>1.62</td>
<td>1.47</td>
</tr>
<tr>
<td>trans</td>
<td>93695</td>
<td>1.21</td>
<td><strong>1.20</strong></td>
<td>1.45</td>
<td>1.23</td>
</tr>
<tr>
<td>avg.</td>
<td></td>
<td><strong>2.12</strong></td>
<td>2.12</td>
<td>2.34</td>
<td>2.16</td>
</tr>
<tr>
<td>w. avg.</td>
<td></td>
<td><strong>1.89</strong></td>
<td>1.91</td>
<td>2.09</td>
<td>1.93</td>
</tr>
</tbody>
</table>

In average bits/byte.
Final Note

- All of the code for the Sequence Memoizer is available online at www.sequencememoizer.com.
- There are C++ and Java implementations, and bindings to Python and R.