Implementation of the sequence memoizer in a probabilistic programming language

Stefan Webb

1 The Model

The sequence memoizer (SM) is an advanced probabilistic model for discrete sequences ([13], [14]). Normally the sparsity of the training data results in overconfident estimates of observed sequences and underestimates of those deviating from but similar to the training data. Ad-hoc methods, such as Kneser–Ney smoothing, have been developed to overcome these limitations—however, by using a hierarchical Bayesian model we can do so in a principled probabilistic way, sharing statistical strength amongst the distributions. The SM is a nonparametric model in the sense that the random variables in the hierarchy are infinite-dimensional objects. Each is a distribution over discrete distributions with a possibly infinite number of atoms. It can be understood as an extension of the hierarchical Pitman–Yor process (HPYP) language model ([11], [10], [12]) in which the Markov order tends to infinity. Special properties of the Pitman–Yor process (PYP) permit tractable inference despite the infinite depth of the hierarchy.

Two applications of the SM are language models and probabilistic compression ([2]). A language model forms a distribution over a sequence of words as a product of conditional distributions of each word given some of the previous words. The idea is that the immediate prior context of a word is usually very informative about what comes next. Compression and probabilistic modelling are deeply connected. An implication of the source coding theorem is that the more compact the probabilistic model of the data (that is, the greater its entropy), the higher the lossless compression ratio that can be attained. Entropy encoding ([8]) is a practical algorithm for lossless compression and decompression using a probabilistic model of the data.

Probabilistic programming (PP) is a nascent discipline that investigates how to write generative statistical models as computer programs along with general methods for performing inference in those programs ([5], [3]). By using Turing complete languages, augmented with special constructions to define the likelihood and prior of the model, any traditional generative model can be constructed. Indeed, we can define a wider class of generative models than those of the probabilistic graphical model framework, since the form of the likelihood may change depending on prior random choices in the program. In this project, we have used the Anglican language ([15]). It is integrated with Clojure, which is a dialect of Lisp using the Java Virtual Machine, and builds on the pioneering work of Church ([4]).

One advantage of PP, aside from permitting the expression of a wider class of models, is that the general inference methods obviate the need to derive inference methods by hand. This separation of representation and inference means that different researchers can specialize in each area and hopefully this greater specialization will lead to advances in each. Moreover, it increases productivity since the most time consuming part of developing new methods has been eliminated. The rapid prototyping and testing of different models makes data science a lot easier to perform, and will spread the methods to those with less expertise.

Hierarchical nonparametric Bayesian models for clustering ([7], [9]) have been implemented before in probabilistic programming languages (PPLs). However, hierarchical nonparametric Bayesian models for general sequences pose additional challenges, due to their deeper hierarchies.
and the fact that draws from a draw from the hierarchy are directly observed, rather than being latent as they are in the clustering models (where they represent the parameters of the clusters). As far as the author is aware, this is the first implementation of a hierarchical nonparametric Bayesian model for sequences in a PPL. In fact, it may be the most complex model so far implemented in a PPL.

The questions we are interested in are:

- Is the Anglican language expressive enough to represent this model? If not, what modifications need to be made? Experience implementing models provides good feedback for further work in designing the language.
- Does Anglican permit efficient inference in this model? How could inference be improved?
- Does the flexibility of a PPL representation afford inference in novel extensions to the model?

1.1 Pitman–Yor process

Before we discuss the exact form of the model, we give a brief overview of the PYP. The PYP is a distribution over discrete distributions, indexed by three parameters, that is a generalization of the Dirichlet process [7]. We denote \( G \sim \text{PYP}(\alpha, d, H) \) for a \( G \) distributed according to a PYP with concentration parameter \( \alpha \), discount parameter \( d \), and base measure \( H \). Since a draw from the PYP is almost surely discrete, we may write it as

\[
G = \sum_{k=1}^{\infty} w_k \delta_{\phi_k}
\]

where the \( \{\phi_k\} \) are drawn i.i.d. from \( H \). It turns out that the distribution of the weights \( \{w_k\} \) can be given by,

\[
\beta_k \overset{\text{ind}}{\sim} \text{Beta}(1 - d, \alpha + dk), \quad \text{for } k = 1, 2, \ldots
\]

\[
w_k = \beta_k \prod_{j=1}^{k-1} (1 - \beta_j), \quad \text{for } k = 1, 2, \ldots.
\]

This representation is described as “stick breaking,” because it is as if we were breaking off an amount of the probability in forming each subsequent \( w_k \) that is a fraction \( \beta_k \) of the remaining probability.

A hierarchical model can be formed by making the base measure of the PYP a draw from another PYP [12]. The atoms of a draw from the child measure conditioned on a draw from the parent measure must clearly be a subset of those from the parent measure, and this sharing of atoms can be exploited, for example, for hierarchical clustering. We call this model the hierarchical Pitman–Yor process (HPYP).

The PYP can capture a power law distribution in the weights of the atoms, a phenomenon that is present in many natural processes such as the distribution of words (see Zipf’s law) and population ranks of cities in various countries.

2
1.2 The sequence memoizer

The SM models a sequence \( x_{1:T} = (x_1, x_2, \ldots, x_T) \in \Sigma^* \) as a product of conditional probabilities,

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

where the \( G_u(s) \) terms are random measures modelling the probability of observing a symbol \( s \in \Sigma \) conditioned on the previous symbols being \( u \in \Sigma^* \), and are given a HPYP prior,

\[
G_\epsilon \sim PYP(\alpha_\epsilon, d_\epsilon, H)
\]

\[
G_u \mid G_{\sigma(u)} \sim PYP(\alpha_u, d_u, G_{\sigma(u)}) \quad \text{for all } u \in \Sigma^* \setminus \{\epsilon\},
\]

where \( \epsilon \) is the empty sequence and \( \sigma(u) \) is the sequence obtained by dropping the first symbol in \( u \).

Inference is made tractable by using a Chinese restaurant franchise (CRF) representation and using properties of the PYP to reduce the hierarchy from an infinite to a finite number of random measures. Consider the hierarchy required for the query of inferring the distribution of the next symbol having observed the string “oacac” (which is proportional to the joint distribution of “oacac” and the symbol). Draws are observed from the parameters \( G_\alpha, G_oa, G_oac, G_oaca, \) and \( G_oacac \), which depend on a number of additional parameters as depicted in Figure 1(a). This naive representation of the model scales quadratically in the length of the sequence.

All non-leaf distributions that do not have observations can be marginalized out using the coagulation operator to produce the reduced tree in Figure 1(b). Our model only specifies the
distributions of $G_u | G_{\sigma(u)}$—from this the coagulator operator will give us the form of $G_u | G_v$ where $v$ is an arbitrary subset of $u$. When constructing this data structure, a parameter that was previously marginalized out may need to be reintroduced, and this can be done using the fragmentation operator. We refer to this structure as the context tree, and it contains all information to perform inference in the model. A more detailed discussion of the coagulation and fragmentation operators is given in Section 2, and construction of the context tree in Section 3.

Existing methods for inference in HPYP, such as those based on Gibbs samplings and particle filters, can be used, as the model has been reduced to one with a finite number of parameters. Importantly, the number of parameters scales linearly in the length of the sequence.

1.3 Chinese restaurant processes

An understanding of Chinese restaurant processes (CRF) and the CRF is necessary for implementation of the model, and we give a brief overview here. The Chinese restaurant process is an exchangeable distribution over partitions of the integers. According to the metaphor, samples from the process represent customers in a Chinese restaurant and the partitions represent tables in the restaurant. Let $\pi_{[n]}$ denote a partition based on the first $n$ customers and consider the following exchangeable distribution,

$$P(\text{customer } n + 1 \text{ joins table } c \mid \pi_{[n]}) = \begin{cases} \frac{|c| - d}{\alpha + |c| - d K_n} & \text{if } c \in \pi_{[n]}, \\ \frac{\alpha}{\alpha + n} & \text{otherwise.} \end{cases} \tag{1}$$

where $K_n$ is the number of tables in $\pi_{[n]}$. This equation describes an infinite partition of the integers. If we associate a sequence of i.i.d. values from the base measure with each table, and associate a variable with each customers that is equal to its table’s value, we produce an exchangeable sequence of draws from the base measure. It turns out that the PYP is the de Finetti measure associated with this Blackwell-MacQueen (BM) urn model. That is, draws from the BM urn model correspond to draws from the expectation of the PYP (that is, where the PYP has been marginalized out).

Consider seating arrangements for a finite number of customers. Let $A_c$ be the set of seating arrangements of $c$ customers, and $A_{ct}$ those with $t$ tables. Multiplying the conditional probabilities in (1),

$$P(A) = \frac{[\alpha + d]_{|A|}^{-1}}{[\alpha + 1]_{|A| - 1}} \prod_{a \in A} [1 - d]_1^{[a] - 1} \quad \text{for each } A \in A_{ct},$$

where

$$[y]_d^n \equiv \prod_{i=1}^{n-1} (y + id).$$

Fixing the number of tables to be $t \leq c$, the distribution is

$$P(A \mid |A| = t) \propto P(A) \propto \prod_{a \in A} [1 - d]_1^{[a] - 1}, \tag{2}$$

since $P(|A| = t \mid A) = 1$. The normalization constant is a generalized Stirling number of type $(-1, -d, 0)$ denoted

$$S_{\sigma}(c, t) \equiv \sum_{A \in A_{ct}} \prod_{a \in A} [1 - d]_1^{[a] - 1}.$$
Consider the corresponding Blackwell-MacQueen urn, and define by \( z_{1:c} = (z_1, \ldots, z_c) \) a draw from this model.

In order to work with the minimal sufficient statistics, the seating arrangement will be broken up in sections serving the same dish. Let \( s \in \Sigma \) be an arbitrary dish, \( c_s \) the number of observations served the dish \( s \), \( t_s \) the number of tables served dish \( s \), and \( A_s \in A_{c_s t_s} \) the seating arrangement of customers around the tables serving dish \( s \). The joint distribution is,

\[
P(\{c_s, t_s, A_s\}, z_{1:c}) = \left( \prod_{s \in \Sigma} G_0(s)^{t_s} \right) \left( \frac{[\alpha]^{d_1^{t_s} - 1}}{[\alpha + 1]^{d_1^{t_s} - 1}} \prod_{s \in \Sigma} \prod_{a \in A_s} [1 - d_a^{[\alpha] - 1}] \right)
\]

Summing over all seating arrangements,

\[
P(\{c_s, t_s\}, z_{1:c}) = \left( \prod_{s \in \Sigma} G_0(s)^{t_s} \right) \left( \frac{[\alpha + d^{t_s}_{\sigma}]^{\sum_{1}^{d^{t_s}_{\sigma}} - 1}}{[\alpha + 1]^{\sum_{1}^{d^{t_s}_{\sigma}} - 1}} \prod_{s \in \Sigma} S_d(c_s, t_s) \right)
\]

Hence, we can perform inference only considering the number of tables for each symbol given \( z_{1:c} \) since the \( c_s \) are fixed (see (3) below).

### 1.4 Chinese restaurant franchises

The Chinese restaurant franchise is an extension of the Chinese restaurant process that is the distribution whose de Finetti measure is the HPYP. Consider the hierarchy of PY measures formed in the sequence memoizer. For each measure we form the equivalent Chinese restaurant, and a draw from a leaf node follows (1) where when we are required to draw from the base formed in the sequence memoizer. For each measure we form the equivalent Chinese restaurant, and a draw from a leaf node follows (1) where when we are required to draw from the base.

The joint distribution of the Chinese restaurant franchise is obtained by multiplying together the probabilities of all the seating arrangements, 

\[
P(\{c_{us}, t_{us}, A_{us}\}, x_{1:T}) = \left( \prod_{s \in \Sigma} H(s)^{t_{us}} \right) \prod_{u \in \Sigma^*} \left( \frac{[\alpha_u - d_u]^{t_{us}_{u}}_{d_u} - 1}{[\alpha_u + 1]^{c_u_{u} - 1}} \right) \prod_{1 \in \Sigma} \prod_{a \in A_{us}} [1 - d_u^{[\alpha] - 1}]
\]

Note one important difference from the Chinese restaurant process; we must condition on the joint seating arrangement because the parent measures are random. That is, to get a draw from the expectation of the HPYP we must also marginalize out the seating arrangements.

The joint distribution of the Chinese restaurant franchise is obtained by multiplying together the probabilities of all the seating arrangements,
As for the single restaurant case, we can marginalize out the seating arrangements,

\[
P(\{c_{us}, t_{us}\}, x_{1:T}) = \left(\prod_{s \in \Sigma} H(s)^{t_{us}}\right) \prod_{u \in \Sigma^*} \left(\frac{[\alpha_u - \sigma_u |T_u - 1]}{[\alpha + 1]}\right) \prod_{s \in \Sigma} S_{du}(c_{us}, t_{us})
\]

Thus it suffices to work with the sufficient statistics \{c_{us}, t_{us}\}.

2 Fragmentation/coagulation

The fragmentation and coagulation operators describe the relationship between the distributions in a HPYP. That is, when

\[
G_1 \mid G_0 \sim \text{PYP}(\alpha, d_1)
\]
\[
G_2 \mid G_1 \sim \text{PYP}(\alpha d_2, d_2),
\]

we have

\[
G_2 \mid G_0 \sim \text{PYP}(\alpha d_2, d_1 d_2).
\]

They will be of use to us in marginalizing and reintroducing PYPs in the hierarchy as we construct the context tree. The results are traditionally given in terms of the stick breaking representation of the draws from the PYPs. Nonetheless, it is not clear how to implement fragmentation in terms of a stick breaking representation since it involves an infinite limit (see Eq (6) in [13]).

2.1 Chinese restaurant franchise representation

We can always work in the Chinese restaurant franchise representation instead of the stick-breaking one. It turns out that the fragmentation operation is tractable in this representation—let us describe it now.

Let \(A_2 \in \mathcal{A}_c\) and \(A_1 \in \mathcal{A}_{|A_2|}\) be the seating arrangements of two adjacent Chinese restaurants in a hierarchical Pitman-Yor process. \(A_1\) is the parent of \(A_2\) so that the number of customers in the former is the same as the number of tables in the later. Let \(C \in \mathcal{A}_c\) be the seating arrangement formed by merging all tables in \(A_2\) being served dishes from the same table in \(A_1\), that is, the coagulation of \(A_1\) with \(A_2\). Also, let \(\{F_a\}_{a \in C}\) represent a fragmentation of the tables in \(C\), so that each \(F_a \in \mathcal{A}_{|a|}\) contains the customers at \(a \in C\) split into (possibly) multiple tables. (See Figure 2) Note that \(A_1\) and \(C\) have the same number of tables, \(|A_1| = |C|\), and the number of tables in a section of \(A_2\) is equal to the number of customers at the corresponding table in \(A_1\), \(|F_a| = |a|\).

Translating the hierarchical models of (5) and (6) into their equivalent Chinese restaurant franchises gives the following analogous theorem due to [1].

**Theorem 1.** Suppose \(A_2 \in \mathcal{A}_c, A_1 \in \mathcal{A}_{|A_2|}, C \in \mathcal{A}_c\) and \(F_a \in \mathcal{A}_{|a|}\) for each \(a \in C\) are related as above. Then the following describe equivalent distributions,

(a) \(A_2 \sim \text{CRP}_c(\alpha d_2, d_2)\) and \(A_1 \mid A_2 \sim \text{CRP}_{|A_2|}(\alpha, d_1)\)

(b) \(C \sim \text{CRP}_c(\alpha d_2, d_1 d_2)\) and \(F_a \mid C \sim \text{CRP}_{|a|}(d_1 d_2, d_2)\) for each \(a \in C\).

An elementary proof is possible—we equate the two joint distributions.
Proof. The joint distribution of \((a)\) is,

\[
P(A_1, A_2) = \left(\frac{\alpha + d_1}{\alpha + 1}\right)^{|A_1| - 1} \left(\frac{\alpha + d_2}{\alpha + 1}\right)^{|A_2| - 1} \left(\prod_{a \in A_1} [1 - d_1]^{|a| - 1}\right) \left(\prod_{b \in A_2} [1 - d_2]^{|b| - 1}\right).
\]

Using the fact that

\[
[\alpha d_2 + d_2]^{|A_2| - 1} = d_2^{|A_2| - 1} [\alpha + 1]^{|A_2| - 1},
\]

the above expression simplifies,

\[
\frac{[\alpha d_2 + d_1 d_2]^{|A_1| - 1}}{[\alpha d_2 + 1]^{|A_1| - 1}} \cdot \frac{d_2^{|A_2| - |A_1|}}{d_2} \left(\prod_{a \in A_1} [1 - d_1]^{|a| - 1}\right) \left(\prod_{b \in A_2} [1 - d_2]^{|b| - 1}\right).
\]

The quantity \(|A_2| - |A_1|\) is the excess of customers over tables in \(A_1\), that is, the number of customers who joined an existing table. Thus, \(d_2\) can be distributed over the first product to give,

\[
\frac{[\alpha d_2 + d_1 d_2]^{|A_1| - 1}}{[\alpha d_2 + 1]^{|A_1| - 1}} \left(\prod_{a \in A_1} [d_2 - d_1 d_2]^{|a| - 1}\right) \left(\prod_{b \in A_2} [1 - d_2]^{|b| - 1}\right).
\]

Since our aim is to equate this expression to \(P(C, \{F_a\}_{a \in C})\), we regroup in terms of the coagulated representation,

\[
\frac{[\alpha d_2 + d_1 d_2]^{|C|}}{[\alpha d_2 + 1]^{|C| - 1}} \left(\prod_{a \in C} [d_2 - d_1 d_2]^{|F_a| - 1}\right) \left(\prod_{b \in F_a} [1 - d_2]^{|b| - 1}\right)
\]

\[
= \frac{[\alpha d_2 + d_1 d_2]^{|C|}}{[\alpha d_2 + 1]^{|C| - 1}} \left(\prod_{a \in C} [1 - d_2]^{|a| - 1}\right) \left(\prod_{b \in F_a} [1 - d_2]^{|b| - 1}\right)
\]

\[
= P(C) \prod_{a \in C} P(F_a | C),
\]

where

\[
P(C) = \left(\frac{\alpha d_2 + d_1 d_2}{\alpha + 1}\right)^{|C| - 1} \left(\prod_{a \in C} [1 - d_1 d_2]^{|a| - 1}\right)
\]

\[
P(F_a | C) = \left(\frac{d_2 - d_1 d_2}{1 - d_1 d_2}\right)^{|a| - 1} \left(\prod_{b \in F_a} [1 - d_2]^{|b| - 1}\right).
\]
are the distributions of the Chinese restaurant processes given in \((b)\). Reversing the argument proves the other direction of the equivalence.

Theorem 1 tells us how to perform coagulation and fragmentation of the Chinese restaurants. Given \(A_2 \mid A_1\), we form the coagulation \(C\) by merging the tables in \(A_2\) drawn from customers at the same table in \(A_1\). Given \(C\), we form the fragmentation \(\{F_a\}_{a \in C}\) by breaking up each table in \(C\) according to the Chinese restaurant process given in \((b)\). \(A_1\) and \(A_2\) can be reconstructed from \(C\) and \(\{F_a\}\). It might seem counterintuitive that the direction of conditioning in the Chinese restaurant franchise has been reversed relative to the HPYP.

2.2 Form of the parameters

The most general form of the parameters that is compatible with Theorem 1 is,

\[
\alpha = \alpha > 0 \\
\alpha_u = \alpha_{\sigma(u)} d_u.
\]

where the discounts and the concentration of the root measure are allowed to vary freely. For simplicity, we restrict the discount parameters to be a function of the length of the context only in our implementation, and for the moment set them all equal to a constant.

Now, we wish to understand the form of the parameters after measures have been marginalized. Plugging these forms of the parameters into a simple hierarchy,

\[
G_u \mid G_{\sigma(u)} \sim \text{PYP}(\alpha_{\sigma(u)} d_u, d_u, G_{\sigma(u)}) \\
G_{\sigma(u)} \mid G_{\sigma(\sigma(u))} \sim \text{PYP}(\alpha_{\sigma(\sigma(u))} d_{\sigma(u)}, d_{\sigma(u)}, G_{\sigma(\sigma(u))})
\]

implies

\[
G_u \mid G_{\sigma(\sigma(u))} \sim \text{PYP}(\alpha_{\sigma(u)} d_u, d_u d_{\sigma(u)}, G_{\sigma(\sigma(u))}).
\]

In general,

\[
G_u \mid G_{\sigma^n(u)} \sim \text{PYP}(\alpha \prod_{i=0}^{\lfloor u \rfloor-1} d_{\sigma^i(u)}, \prod_{i=0}^{n-1} d_{\sigma^i(u)}, G_{\sigma^n(u)}).
\]

where \(\sigma^i = \sigma \circ \cdots \circ \sigma\).

2.3 Reinstantiating a restaurant

During construction of the context tree, we may need to reintroduce a restaurant that was previously marginalized out. How do we do this having only the sufficient statistics for a restaurant \(A_u, \{c_u, t_u\}_{s \in S}\)?

The restaurant \(A_u\) can be broken up into a number of sub-restaurants, one for each symbol. Let \(A_{us}\) be the part of the restaurant serving dish \(s\). We can sample a table configuration for \(A_{us}\) given \(\{c_{us}, t_{us}\}\), fragment \(A_{us}\), and count the resulting tables to generate the new sufficient statistics (discarding the table configurations).

In particular, if we use the notation of Theorem 1 \((C = A_{us}, \text{etc.})\), we fragment each table \(a \in C\) into the subtables of the \(\{F_a\}\) according to a CRP with parameters \((-d_1 d_2, d_2)\), and
calculate the new sufficient statistics for the parent and child fragmented restaurants as,
\[
c_{us}^{A_1} = \sum_{a \in C} t_{us}^{A_a},
\]
\[
t_{us}^{A_1} = t_{us}^{C},
\]
\[
c_{us}^{A_2} = c_{us}^{C},
\]
\[
t_{us}^{A_2} = c_{us}^{A_1}.
\]

2.4 Sampling from \( A_{ct} \)

It is not immediately clear how to sample a seating arrangement from \( A_{ct} \) using equation (2).

One solution, following [1], is to convert the seating arrangement to an alternative representation. Let \( z_i \) denote the number of tables occupied by the first \( i \) customers, and \( y_i \) the label of the table at which customer \( i \) sits, where the tables are labeled by the index of the first customer at the table. To enforce the constraints \( z_1 = 1 \), \( z_c = t \) we define a Markov network over \( z_1: c \) (see Figure 3).

Clearly, we desire to define the factors so that
\[
P(z_1: c, y_1: c) \propto \prod_{i: z_i = z_{i-1}} (i - 1 - z_id).
\]

Thus let the factors be,
\[
f(z_i, z_{i-1}) = \begin{cases} 
  i - 1 - z_id & \text{if } z_i = z_{i-1}, \\
  1 & \text{if } z_i = z_{i-1} + 1, \\
  0 & \text{otherwise.}
\end{cases} \quad (8)
\]

The normalization constant is \( S_d(c, t) \). Conditioned on \( z_1: c \), the distribution of a table label is,
\[
P(y_i \mid z_1: c, y_{1:i-1}) = \begin{cases} 
  1 & \text{if } y_i = i \text{ and } z_i = z_{i-1} + 1, \\
  \sum_{j=1}^{i-1} 1(y_j = y_i) - d & \text{if } z_i = z_{i-1} \text{ and } y_i \in [i - 1],
\end{cases} \quad (9)
\]
or in other words, when \( y_i \) starts a new table its label is \( i \). Otherwise, we have to choose amongst the existing tables, using the CRP equation to determine the probability of selecting a table.

This alternative representation defines a distribution equivalent to (2),
\[
P(z_1: c, y_1: c) = \sum_{i: z_i = z_{i-1}} (i - 1 - z_id) \prod_{i: z_i = z_{i-1}} \sum_{j=1}^{i-1} 1(y_j = y_i) - d \\
= \sum_{i: z_i = z_{i-1}} \left( \sum_{j=1}^{i-1} 1(y_j = y_i) - d \right) \\
= \prod_{a \in A} [1 - d]|a|^{-1} \\
= \frac{S_d(c, t)}{S_d(c, t)}
\]
so the method is well-defined. To obtain the last equality we use the fact that there are \(|a| - 1\) terms for which \( z_i = z_{i-1} \) for each table, since the equation \( z_i = z_{i-1} \) holds when a customer joins an existing table.
2.5 Implementation in Anglican

Before we consider how to implement sampling of the $z_{1:i}$, let us determine their range. This will suggest an algorithm that uses the least memory possible. From the left boundary condition, and the fact that each $z_i$ may start a table,

$$1 \leq z_2 \leq 2$$
$$1 \leq z_3 \leq 3$$
$$\vdots$$

Thus, $z_i \leq \text{min}\{t, i\}$. Similarly, a lower bound is obtained from the right boundary condition. The range of $z_i$ is then,

$$\text{max}\{1, i + t - c\} \leq z_i \leq \text{min}\{t, i\}. \quad (10)$$

To sample from the Markov network for $z_{1:i}$, we calculate the probability of each variable $z_i$ for $i = 2, \ldots, c-1$ using the sum-product algorithm in the clique tree of Figure 4, then sample from the resulting discrete distributions. The messages are passed from right-to-left for numerical stability, and when we have reached the first clique, we sample the variables from left-to-right. Following the basic properties of clique trees, the distribution of $z_i$ is given by fixing the value of $z_{i-1}$ (it having been sampled previously), and using,

$$p(z_i) = \frac{1}{Z} f(z_{i-1}, z_i) \phi_{i\rightarrow i-1}(z_i)$$

where $Z$ is the normalizing constant.

The message from clique $i$ to clique $i-1$ is formed by multiplying the incoming message from clique $i+1$ (except at the boundary clique $c-1$) with the factors at that clique, and summing out the variables not in the sepset. That is,

$$\phi_{i\rightarrow i-1}(z_i = k) = \sum_{j=1}^{t} \phi_{i+1\rightarrow i}(z_{i+1} = j)f(z_i = k, z_{i+1} = j)$$
$$= (i - kd)\phi_{i+1\rightarrow i}(z_{i+1} = k) + \phi_{i+1\rightarrow i}(z_{i+1} = k + 1). \quad (12)$$

The range of $k$ is given by equation (10), which also says that the message $\phi_{i\rightarrow i-1}$ can be stored in a vector of length $\text{min}\{t, i\} - \text{max}\{1, i + t - c\} + 1$. Using the reduced representations of $\phi_{i\rightarrow i-1}$ and $\phi_{i+1\rightarrow i}$, the only complication in implementing (11) is determining how the entries of the later line up with the former. When $i > c - t$, or rather the lower limit of $Z_{i+1}$ is greater than 1, then the first entry of $\phi_{i+1\rightarrow i}$ corresponds to a value of $Z_{i+1}$ that is one greater than the lower limit of $Z_i$. Otherwise, the first entry of $\phi_{i+1\rightarrow i}$ corresponds to a value of $Z_{i+1}$ that is equal to the lower limit of $Z_i$. 

**Figure 4:** Clique tree for the Markov network above. The factors associated with a clique are denoted below each clique, and the messages are passed from right to left.
3 Context tree

3.1 Data structure

In a declarative programming language, we could represent the context tree as a vector of nodes that reference their children, with a stack structure to keep track of the path followed. Alternatively, if the nodes also contained a reference to their parent it would not be necessary to use an additional structure to keep track of the path followed. In an object-orientated programming language, an iterator class could be used to encapsulate a pointer to the current node and the navigational operations such as up, down, dereference.

These straightforward and efficient design patterns are not possible, however in a functional programming language like Clojure because it is not possible to create back-references with the built-in immutable data structures. The fundamental data structure in Lisp is the singly linked. Clojure also has a few other data structures built in, such as maps, sets, and vectors. So our challenge is, how do we represent a tree structure that can be efficiently navigated and modified using only these primatives?

One elegant solution is the zipper structure of [6]. The basic idea is that we transform a tree into a structure that can be unpeeled as we navigate further into the depth of the tree. A copy of the subtree of all descendants of the current node is kept, along with enough information to zip up the structure and navigate back up as required. Clojure has an implementation of the zipper. However, it does not support all of the operations we need, such as non-leaf node values and testing whether there is a given child of a node in constant time.

Here are the details of our implementation. Our current location in the tree data structure is represented as a vector,

\[
\begin{array}{c}
\text{[Position Subcontext]}
\end{array}
\]

The context is broken up into two parts: there is the smaller subcontext, which is formed by following symbols along each node in the tree, and the complete context, which is represented at each node as indices into the training data. It is much more efficient store the contexts as indices rather than the complete string because of the amount of repetition between the contexts. Storing the subcontext at our current location separately from the position means we do not have to store the substrings of the subcontext, we can simply remove the first character to form the new subcontext when navigating one level up the tree.

The Position element is a list of the following form,

\[
\begin{array}{c}
\text{(Current-Restaurant Current-Children Parent-Position)}
\end{array}
\]

It contains the Chinese restaurant at the current node, the children of the current node (which is a map from symbols to subtrees), and the unzipped structure used to navigate back up the tree.

3.2 Constructing the context tree

It is easiest to explain the details of constructing the context tree with a simple example. We will consider the context tree required for calculating the probability of the string “oacacs.” See Figure 5 for an illustration of the steps for building the context tree. The following steps are needed to calculate the probabilities of each symbol given the previous symbols, in an increasing order.

For,

1. \( o \mid c \): create a restaurant for \( H_c \) and insert the symbol \( o \).
Figure 5: Building the context tree for the string “oacac.” The figure is to be read top-to-bottom, left-to-right. The counts after the symbols represent the numbers of customers and tables respectively. In some cases, multiple counts are possible depending on how far a customer percolates up the hierarchy, that is, whether a sample is drawn from the parent measure or an existing table.
2. $a | o$: the context $o$ is prefixed by $\epsilon$, so we insert $G_o$ as a child of $G_\epsilon$. The symbol $a$ is inserted into $G_o$, and since there were no tables with $o$ we must pass this customer to the parent, $G_\epsilon$.

3. $c | oa$: likewise, insert $G_{oa}$ as a child of $G_\epsilon$ with symbol $c$, and pass this customer upwards. Note that in this case we have implicitly marginalized out $G_o$.

4. $a | oac$: insert $G_{oac}$ as a child of $G_\epsilon$ with symbol $a$, and pass this customer upwards. When $G_\epsilon$ receives the count it can either attribute it to an existing table, or draw it from the base measure $H$ to start a new table. Hence there are two possible counts for the symbol $a$ at this step.

5. $c | oaca$: in deciding where to insert $G_{oaca}$, we find the measure corresponding to the context that has the largest prefix in common with $oaca$, which in this case is $oa$. Since $oa$ is not a complete prefix of $oaca$, however, we need to fragment $oa$ and reintroduce the restaurant for $G_o$. This becomes the parent of both $G_{oaca}$ and $G_{oa}$. When the customer is passed upwards, there is already a table in $G_a$ for $a$, so it may or may not be passed further up the hierarchy. If it is, a similar choice occurs at $G_\epsilon$.

6. $s | oacac$: similarly for this step.

From this example, the general method should be clear. For each successive context, we find the restaurant that has the longest prefix in common. If the two contexts are equal, we insert the customer into the existing restaurant. If the existing context is a prefix of the new context, we insert the new restaurant as a child. Otherwise we fragment the restaurant with the longest prefix in common to produce a new restaurant whose context is the shared prefix. It can never be the case that the new context is shorter than the existing context since we add contexts of increasing length.

4 Further work

To finish the implementation of the model, we have to calculate the likelihood of the model as the context tree is being created. The probability of observing a symbol conditioned on the context is given by (4). The probabilities are calculated recursively, so I intend to store the probabilities in the restaurant data structure and perform a downwards update after the upwards update of the counts.

When the model is working I will run it on a standard text data set and compare the probabilities given to the results of the Java implementation with standard sampling methods. This will give an indication as to how well inference is performing, and the speed of the implementation.

It may be possible to extend the family of distributions that the model can express. I investigated $\alpha$-stable distributions and processes and their properties, writing up a series of notes, but did not reach that stage of the project. I believe either the class of sub-Gaussian or symmetric $\alpha$-stable distributions will provide a suitable generalization.

References


Appendix

The following is the Anglican source code for the sequence memoizer model with commentary.

A.1 Parameters

Here are the functions to calculate the parameters, following (7),

```clojure
(defn make-discount
  ([context-length] 0.5)
  ([context-length parent-context-length] (reduce * (map make-discount (range
    (inc parent-context-length)
    (inc context-length)))))
)
```

```clojure
(defn make-concentration
  [context-length]
  (if (= 0 context-length)
    alpha-base
    (* alpha-base (reduce * (map make-discount (range 1 (inc context-length))))))
)
```

A.2 Tree data structure

Here are the functions to initialize the zipper structure and extract basic properties of the current location,

```clojure
(defn crf-zipper
  "Takes a tree representation of a Chinese restaurant franchise and starts its zipper data structure"
  [crf-tree]
  (vector (list (first crf-tree) (second crf-tree) nil) "")
)
```

```clojure
(defn crf-location
  "Given a point in the unzipped tree, returns the location part"
  [crf]
  (get crf 0)
)
```

```clojure
(defn crf-path
  "Given a point in the unzipped tree, returns the path followed to get here"
  [crf]
  (get crf 1)
)
```

```clojure
(defn crf-parent-path
  "Remove the first character from the current path"
  [crf]
  (let [this-path (crf-path crf)]
    (subs this-path 1 (count this-path))
  )
)
```

```clojure
(defn crf-children
  "Returns the children of the current location"
  [crf]
  (second (crf-location crf))
)
It will also be useful to have functions that return higher-level properties of the data structure,

\begin{Verbatim}
(defn crf-leaf?
"Determines whether the current position in the Chinese restaurant franchise is a leaf"
[crf]
(empty? (crf-children crf))
)
\end{Verbatim}

\begin{Verbatim}
(defn crf-root?
"Determines whether the current position in the Chinese restaurant franchise is the root"
[crf]
(nil? (crf-parent-loc crf))
)
\end{Verbatim}

\begin{Verbatim}
(defn crf-child
"Returns the Chinese restaurant franchise (subtree) for a given symbol, or nil"
[crf symbol]
(get (crf-children crf) symbol)
)
\end{Verbatim}

\begin{Verbatim}
(defn crf-has-child?
"Determines whether there is a child node along a given symbol"
[crf symbol]
(contains? (crf-children crf) symbol)
)
\end{Verbatim}

The data structure for a restaurant is a map with (currently) four entries: context, counts, customers and tables. The context key contains a vector of indices into the training string that index the substring that is the context of the restaurant. The counts key contains a map between the symbols and the customer/table counts associated with each symbol. The customer/table counts are in a vector. The customers and tables keys contain the total number, respectively, of the customers and tables in this restaurant. They could, of course, be calculated by summing the entries from the counts map, but this would be an unnecessary calculation.

Here are some functions for creating a restaurant, and adding the counts for each symbol.

\begin{Verbatim}
(defn make-restaurant
"Build a restaurant data structure"
[context-idx counts customers tables]
{:context context-idx :counts counts :customers customers :tables tables}
)
(defn add-customers
  "Sum the number of customers in the counts map"
  [counts]
  (let [counter (fn [result [key val]] (+ result (first val)))]
    (reduce counter 0 counts))
)

(defn add-tables
  "Sum the number of tables in the counts map"
  [counts]
  (let [counter (fn [result [key val]] (+ result (second val)))]
    (reduce counter 0 counts))
)

Here are functions that extract the context indices from the restaurant and related properties,

(defn crf-context-idx
  "Return a vector with the indices of the substring associated with the context of a node"
  [crf]
  (get (crf-restaurant crf) :context))

(defn crf-context-len
  "Return the length of the substring associated with the context of a node"
  [crf]
  (let [[[l u]] (crf-context-idx crf)]
    (- u l)))

(defn crf-parent-context-idx
  "Return a vector with the indices of the substring associated with the context of the parent node"
  [crf]
  (if (crf-root? crf)
    nil
    (get (first (crf-parent-loc crf)) :context)))

(defn crf-parent-context-len
  "Return the length of the substring associated with the context of a node"
  [crf]
  (if (crf-root? crf)
    0
    (let [[[l u]] (crf-parent-context-idx crf)]
      (- u l))))

(defn crf-context
  "Return the full context at leaf nodes that is given as a substring of the training string"
  [crf string]
  (let [indices crf-context-idx]
    (subs string (get indices 0) (get indices 1)))
Here are functions that extract the remainder of the restaurant properties. The parameters are implicitly defined by the context length of this node and the parent.

(defn crf-counts
  "Return the counts map for the current node"
  [crf]
  (:counts (crf-restaurant crf)))

(defn crf-tables
  "Return the counts map for the current node"
  [crf]
  (:tables (crf-restaurant crf)))

(defn crf-customers
  "Return the counts map for the current node"
  [crf]
  (:customers (crf-restaurant crf)))

(defn crf-parameters
  "Return the parameters of the restaurant at the current node"
  [crf]
  (if (crf-root? crf)
    (list (make-concentration 0) (make-discount 0))
    (list (make-concentration (crf-context-len crf))
          (make-discount (crf-context-len crf) (crf-parent-context-len crf))))
)

(defn crf-discount
  [crf]
  (second (crf-parameters crf)))

(defn crf-concentration
  [crf]
  (first (crf-parameters crf)))

To navigate to a child node, we have to turn the current position into the parent position by removing the link to the child node, update the subcontext by prefixing the new character, and form the new position from these updated variables,

(defn crf-make-zip
  "Makes a zipper structure given a location and a subcontext"
  ([restaurant children parent-loc this-path]
   (vector (list restaurant children parent-loc this-path))
   ([location this-path]
    (vector location this-path))))

(defn crf-remove
  ...)
"Remove a subtree along a given symbol from the current node, returning a location"
[crf symbol]
(list (crf-restaurant crf) (dissoc (crf-children crf) symbol)
 (crf-parent-loc crf))
)

(defn crf-down
 "Navigate to a child node from the current position"
[crf symbol]
(let [parent-loc (crf-remove crf symbol)
 child-node (crf-child crf symbol)]
 (if (nil? child-node)
 crf
 (crf-make-zip (first child-node) (second child-node) parent-loc
 (str symbol (crf-path crf))))
)

Navigating to the parent node is straightforward—it is just reversing the effects of downward navigation. We add the current node as a child in the parent position, remove the leftmost character of the subcontext, and return the modified parent position,

(defn crf-add-to-loc
 "Add a subtree along a given symbol from a location"
[loc symbol subtree]
(list (first loc) (assoc (second loc) symbol subtree) (last loc))
)

(defn crf-add
 "Add a subtree along a given symbol from a location"
[crf symbol subtree]
(let [loc (crf-location crf)
 new-loc (crf-add-to-loc loc symbol subtree)
 this-path (crf-path crf)]
 (crf-make-zip new-loc this-path)
)

(defn crf-up
 "Navigate to a parent node from the current position"
[crf]
(let [parent-loc (crf-parent-loc crf)
 this-path (crf-path crf)]
 (if (nil? parent-loc)
 crf
 (crf-make-zip (crf-add-to-loc parent-loc (first this-path)
 (crf-subtree crf)) (crf-parent-path crf)))
)
)

Navigating to the top is just navigating up the tree until we reach the root. This is a useful operation when more complicated transformations are performed so that we can return the tree in a known position.

(defn crf-top
 "Navigate to the top of a zipped CRF structure"
[crf]
(if (crf-root? crf)
 crf
)

19
Replacing the restaurant at the current node is required for when we update the seating arrangement.

(defn crf-replace-restaurant
  "Replace the restaurant at the current node"
  [crf new-restaurant]
  (crf-make-zip new-restaurant (crf-children crf) (crf-parent-loc crf)
   (crf-path crf))
)

A.3 Fragmentation

First, we define the factors of (8) and the clamped versions,

(defn factor
  "The factor $f(z_i, z_{i-1})$ for the Markov network used to sample from a table configuration given the number of customers and tables"
  [i discount]
  (fn [y x]
   (cond (= x y) (- (dec i) (* y discount))
     (= x (dec y)) 1.
     :else 0)
  )
)

(defn right-factor
  "The right-clamped factor $f(z_i = y, z_{i-1})$ for the Markov network used to sample from a table configuration given the number of customers and tables"
  [i discount y]
  (partial (factor i discount) y)
)

(defn left-factor
  "The left-clamped factor $f(z_i, z_{i-1} = x)$ for the Markov network used to sample from a table configuration given the number of customers and tables"
  [i discount x]
  (fn [y] (((factor i discount) y) x))
)

It will be useful to have functions that calculate the bounds in Anglican, both as a pair of numbers and as a range,

(defn bounds-z
  "Returns bounds on the auxiliary variables $z_i$ used to sample from a table configuration given the number of customers and tables"
  [i customers tables]
  (vector (max 1 (+ i (- tables customers))) (min tables i))
)

(defn range-z
  "Returns the valid range of the auxiliary variables $z_i$ used to sample from a table configuration given the number of customers and tables"
  [i customers tables]
  (let [bounds (bounds-z i customers tables)]
   (range (first bounds) (inc (second bounds))))
)
Here is the code to calculate a single message,

```
(defn max-normalize
  [x]
  (let [max-x (double (reduce max x))]
    (into [] (map #( / % max-x) x)))
)

(defn calculate-msg
  "Calculate a message in the Markov network used to sample from a table configuration given the number of customers and tables"
  [i msg-in discount customers tables]
  (let [[Lk Uk] (bounds-z i customers tables)]
    ; An upper bound on the index of msg-in Ul (count msg-in)]
    (loop [; The starting value for z_i
      k Lk
      ; The index of the first value msg-in to use
      l (if (> i (- customers tables)) -1 0)
      ; The message that we build up for every k
      msg []]
      (let [first-term (if (>= l 0) (* (- i (* k discount)) (get msg-in l)) 0)
            second-term (if (< (inc l) Ul) (get msg-in (inc l)) 0)]
        (if (< k Uk)
          ; While k is in range, calculate the next element of the message
          (recur (inc k) (inc l) (conj msg (+ first-term second-term))))
          ; When finished return the message vector
          (max-normalize (conj msg (+ first-term second-term)))))
    )
)
```

The messages are then passed from right to left,

```
(defn range-msgs
  "Calculate the order of the clique indices to pass messages from"
  [customers]
  (reverse (range 2 (dec customers)))
)

(defn pass-msg-factory
  "Return a function that passes a message from clique_i to clique_i-1, given a list of the current messages etc."
  [discount customers tables]
  (fn [messages clique-idx]
    (let [msg-in (last messages)]
      (conj messages (calculate-msg clique-idx msg-in
discount customers tables))))
)
```

```
(defn calculate-msgs
  "Calculate all of the right-to-left messages in the Markov network used to sample from a table configuration given the number of customers and tables"
  [discount customers tables]
  (let [pass-msg (pass-msg-factory discount customers tables)])
```

21
clique−idxs (range−msgs customers)  
first−msg (into []) (map (right−factor customers discount  
tables)  
(range−z (dec customers) customers  
tables))]

(reduce pass−msg (vector first−msg) clique−idxs)
)

And the \( z_{1:c} \) sampled from left to right,

(defn normalize
[x]
(let [sum−x (double (reduce + x))]
  (into [] (map #/(% sum−x) x))
)
)

(defn sample−z−factory
"Returns a function that samples a given auxiliary variable \( z_{i+2} = z_{j} \)"
[msgs discount customers tables]
(fn [prev−z i]
  (let [msg−in (get msgs i)  
z−j−1 (last prev−z)  
j (+ i 2)  
range−z−j (range−z j customers tables)  
  factor−term (into [] (map (left−factor j discount z−j−1)  
        range−z−j))  
  z−dist (normalize (into [] (map * factor−term msg−in)))]  
    (conj prev−z z−j)  
)
)

(defn sample−zs
"Samples the auxiliary variables \( z_{i} \) used to construct a table configuration given the number of customers and tables"  
[discount customers tables]
(cond (and (= customers 2) (= tables 2)) [1 2]  
  (and (= customers 2) (= tables 1)) [1 1]  
  (= customers 1) [1]  
  :else (let [msgs (calculate−msgs discount customers tables)  
         sample−z (sample−z−factory msgs discount customers tables)]  
    (conj (reduce sample−z (vector 1) (range 0 (count msgs))) tables)
)
)

The \( y_{1:c} \) are sampled according to (9), although we only need to calculate the table counts.

(defn apply−element
[v f i]
  (into [] (concat (take i v) (conj (take−last (− (count v) (inc i))) v)  
(f (get v i))))
)

(defn inc−element
[v i]
  (apply−element v inc i)  
)
(defn add-all
[v x]
(into [] (map #(+ x) v)))

(defn sample-table-counts
[discount customers tables]
(let [zs (sample-zs discount customers tables)]
(loop [i 1
  ys (vector 1)
  table-counts (inc-element (into [] (take tables (repeat (* -1. discount)))) 0)]
(let [zi (get zs i)
  zi-1 (get zs (dec i))]
  (if (< i customers)
    (if (= zi zi-1)
      ; Observation does not start a new table
      (let [table-dist (into [] (take zi table-counts))
            table-idx (inc (sample (discrete table-dist)))
            (recur (inc i) (conj ys table-idx) (inc-element table-counts
                                           (dec table-idx)))
      )
      ; In this case, observation starts a new table
      (let [table-idx zi]
        (recur (inc i) (conj ys table-idx) (inc-element table-counts
                                           (dec table-idx)))
      )
    )
    )
  )
)
)

Having sampled a seating arrangement for a fixed number of tables and customers, it is simply a matter of running the CRP on each table, and summing the number of tables to get the sufficient statistics for the new parent and child fragmented tables.

(defn fragment-table
"Fragments a table with given number of customers by running a CRP"
[customers concentration discount]
(loop [: customer index i 2
  table-dist [(- 1. discount) (+ concentration discount)]
  table-idx (sample (discrete table-dist))
  table-count (dec (count table-dist))]
  (if (<= i customers)
    (if (= table-idx table-count)
      ; In this case we start new table
      (recur (inc i) (conj (conj (into [] (butlast table-dist))
                             (- 1. discount))
               (+ (last table-dist) discount)))
    ; Otherwise increment existing table count
      (recur (inc i) (inc-element table-dist table-idx)))
  )
)

; Return number of tables
defn fragment-symbol
"Produce the sufficient statistics for fragmenting part of a table for a given symbol"
[customers tables old-discount child-discount parent-discount]
(let [] ; Sample the table configuration
table-counts (sample-table-counts old-discount customers tables)
 ; Calculate the parameters of the CRP
concentration (* -1. (* parent-discount child-discount))
discount child-discount
 ; Run a CRP on each table
fragmented-table-counts (into [] (map #(fragment-table
% concentration discount) table-counts))
(fragmented-table-count (reduce + fragmented-table-counts))
{:child-customers customers
 :child-tables fragmented-table-count
 :parent-customers fragmented-table-count
 :parent-tables tables}
)

This procedure is run on the statistics for all the symbols in the restaurant to produce the sufficient statistics for the two new restaurants.

(defn fragment-counts-helper
"Given an element of the counts map, fragment symbol"
[old-discount child-discount parent-discount]
(fn [[symb [customers tables]]]]
(let [new-count (fragment-symbol customers tables old-discount
child-discount parent-discount)]
(vector symb (vector (:child-customers new-count)
(:child-tables new-count)
(:parent-customers new-count)
(:parent-tables new-count))
)
)
)

(defn fragment-counts
"Fragment a table"
[counts old-discount child-discount parent-discount]
(let [fragment-mapper (fragment-counts-helper old-discount child-discount
parent-discount)
fragments-bundled (map fragment-mapper counts)
get-child-counts (fn [[key val]] (vector key (into [] (take 2 val)))))
get-parent-counts (fn [[key val]] (vector key (into [] (drop 2 val)))))
child-counts (into {} (map get-child-counts fragments-bundled))
parent-counts (into {} (map get-parent-counts fragments-bundled))
(list child-counts parent-counts)
)

Here is the function to fragment the current node.
(defn crf-fragment
A.4 Sampling

As we sample, we have to updates the counts, passing the customers up the hierarchy as the random choices dictate. Here then, are two functions to increment the counts for the current restaurant.

\begin{verbatim}
(defn crf-inc-customer
  "Returns a modified CRF, incrementing the customer count for a given symbol" 
  [crf sym]
  (let [old-counts (crf-counts crf) 
    sym-counts (get old-counts sym) 
    new-sym-counts (if (nil? sym-counts) (vector 1 0) 
                      (vector (inc (first sym-counts)) (second sym-counts))) 
    new-counts (assoc old-counts sym new-sym-counts) 
    old-customers (crf-customers crf) 
    new-restaurant (assoc old-restaurant :counts new-counts :customers (inc old-customers)) 
  ]
  (crf-replace-restaurant crf new-restaurant))
)

(defn crf-inc-customer-and-table
  "Returns a modified CRF, incrementing both the customer and the table count for a given symbol" 
  [crf sym]
  (let [old-counts (crf-counts crf) 
    sym-counts (get old-counts sym) 
    new-sym-counts (if (nil? sym-counts) (vector 1 0) 
                      (vector (inc (first sym-counts)) (second sym-counts))) 
    new-counts (assoc old-counts sym new-sym-counts) 
    old-customers (crf-customers crf) 
    old-tables (crf-tables crf) 
    new-restaurant (assoc old-restaurant :counts new-counts :customers (inc old-customers)) 
    new-restaurant (assoc old-restaurant :counts new-counts :tables (inc old-tables)) 
  ]
  (crf-replace-restaurant crf new-restaurant))
)
\end{verbatim}
new-sym-counts (if (nil? sym-counts) (vector 1 1)
  (vector (inc (first sym-counts))
    (inc (second sym-counts))))

new-counts (assoc old-counts sym new-sym-counts)
old-restaurant (crf-restaurant crf)
old-customers (crf-customers crf)
old-tables (crf-tables crf)
new-restaurant (assoc old-restaurant :counts new-counts
  :customers (inc old-customers)
  :tables (inc old-tables))

(crf-replace-restaurant crf new-restaurant)

Now it simply a matter of percolating the counts up the hierarchy.

(defn crf-prob-sym
  "A quantity proportional to the probability of sampling a symbol from existing tables at the current restaurant"
  [crf sym]
  (let [counts (get (crf-counts crf) sym)
        [customers tables] (if (nil? counts) [0 0] counts)
        discount (crf-discount crf)]
    (- customers (* tables discount)))
)

(defn crf-prob-parent
  "A quantity proportional to the probability of sample from the parent restaurant"
  [crf]
  (let [tables (crf-tables crf)
        discount (crf-discount crf)
        concentration (crf-concentration crf)]
    (+ concentration (* tables discount)))
)

(defn crf-sample
  "Sample a symbol from CRF by updating sufficient statistics, possibly going up hierarchy."
  [crf sym]
  (let [prob-sample-this (crf-prob-sym crf sym)
        prob-sample-parent (crf-prob-parent crf)
        sample-this? (sample (flip (/ prob-sample-this
          (+ prob-sample-this prob-sample-parent)))]
    (if (= sample-this? true)
      ; In this case, we sample from the current restaurant and increment
      ; just the customer count
      (crf-top (crf-inc-customer crf sym))

    (if (crf-root? crf)
      ; Sample from H
      (crf-inc-customer-and-table crf sym)

      ; Sample from the parent PYP
      (crf-sample (crf-up (crf-inc-customer-and-table crf sym)) sym)
    )))
)
A.5 Constructing context tree

The model begins with just a PYP for the empty context.

```clj
(defn empty-chinese-restaurant []
  {:context [0 0] :counts {} :customers 0 :tables 0})
```

```clj
(defn empty-crf []
  (list (empty-chinese-restaurant) {}))
```

```clj
(defn crf-init
  "Create an empty CRF with a single restaurant for \( \epsilon \)
  []
  (crf-zipper (empty-crf))
)
```

To know where to insert the restaurant for the new context, we need to calculate how much of the new context matches the context of a given node, where characters are matched from right to left.

```clj
(defn crf-count-match
  "Check how much of a given context matches part of the current context"
  [training-str l u l2 k2]
  (loop [i k j k2 in-common 0]
    (if (and (> i 1) (> j l2))
      (let [first-symbol (get training-str i)
            second-symbol (get training-str j)]
        (if (= first-symbol second-symbol)
          (recur (dec i) (dec j) (inc in-common))
          in-common)
      )
    )
    in-common)
)
```

Building the context tree will be broken into steps of inserting new contexts. The following function takes a CRF and inserts the new restaurant at the correct position, performing a single sampling operation afterwards.

```clj
(defn crf-insert-context
  "Navigate to the right place to insert a node. Make sure you start from the top!"
  [training-str l u sym crf-zipped]
  (loop [crf-navigated crf-zipped]
    (let [[[l2 u2] (crf-context-idx crf-navigated)] count-common (if (crf-root? crf) 0 (crf-count-common-subcontext crf-navigated))]
      k (dec (u count-common))
      k2 (dec (u2 count-common))
      count-match (crf-count-match training-str l l2 k2)
      count-remaining (inc (k2 l2))
      new-remaining (inc (k l1))
      k-new (k count-match)
      this-symbol (get training-str k-new)]
    )
  )
)
(if (= count-match count-remaining)
  (cond
    ; Check if context already exists
    (= count-remaining new-remaining) (crf-sample crf-navigated sym)
    ; See if can follow child
    (crf-has-child? crf-navigated this-symbol) (recur (crf-down crf-navigated this-symbol))
    ; Otherwise insert new restaurant as child
    :else (crf-sample (crf-down (crf-add crf-navigated this-symbol)
      (list (make-restaurant [1 u] {} 0 0) {})))
    ))
  ))
  ))
)

; Perform fragmentation (afterwards, should be able to insert as child)
(recur (crf-fragment crf-navigated (+ count-match count-common) training-str))

Then to build the context tree, we insert the contexts corresponding to increasing prefixes of
the training string, sampling the next symbol.

(defun build-context-tree
  "Build the context tree for a given string, and seat the customers"
  [training-str]
  (let [u (count training-str)]
    (loop [k 0 crf (crf-init)]
      (if (>= k u)
        crf
        (recur (inc k) (crf-insert-context training-str 0 k
          (get training-str k) crf))
        )
      )
    )
  )