A stochastic programming perspective on nonparametric Bayes

Daniel Roy, Vikash Mansinghka, Noah Goodman, Josh Tenenbaum DROY,VKM,NDG,JBT@MIT.EDU
Massachusetts Institute of Technology

Abstract: We use Church, a Turing-universal language for stochastic generative processes and the probability distributions they induce, to study and extend several objects in nonparametric Bayesian statistics. We connect exchangeability and de Finetti measures with notions of purity and closures from functional programming. We exploit delayed evaluation to provide finite, machine-executable representations for various nonparametric Bayesian objects. We relate common uses of the Dirichlet process to a stochastic generalization of memoization, and use this abstraction to compactly describe and extend several nonparametric models. Finally, we briefly discuss issues of computability and inference.

The Church language is a stochastic generalization of the lambda calculus [3] based on a lexically-scoped, applicative order Lisp [7]. Due to space constraints, we refer to [4] for the detailed syntax and semantics of Church. Here, we focus on the novel perspective Church offers on several nonparametric Bayesian objects, and the connections between foundational issues in functional programming and mathematical issues in the theory of stochastic processes.

Church expressions formally specify stochastic generative processes: programs that, when executed, generate random values. If a Church expression describes a process that halts with probability 1, it induces a well-defined (marginal) distribution on values. Church procedures are first-class, i.e. they can be returned by and passed as arguments to procedures. Therefore, procedures that return procedures represent computable random measures. This makes Church particularly well suited to the study of Bayesian nonparametrics.

Our starting point is the notion of *pure* expressions in functional programming. An expression is pure if it causes no side effects during evaluation (such as writing to disk). An essential property that purity guarantees is independence from the order of evaluation. In Church, the random procedure `flip` has detectable side effects: multiple coin `(flip)`s will not in general yield the same value, because different random bits were used to generate each one. However, despite these side effects, a set of `(flip)`s is still *exchangeable*: the order of evaluation does not matter. In fact, a set of `(flip)`s is simply i.i.d.. However, consider the Beta-Bernoulli model:

```
(define (sample-coin)
  (let ((coin-weight (random-beta 1 1)))  ;; sample a coin weight Beta(1,1)
    (lambda () (flip coin-weight))))      ;; return a procedure that flips it

(define my-coin (sample-coin))            ;; construct a coin
(my-coin)                                 ;; flip it. flips are exchangeable,
(my-coin)                                 ;;   but i.i.d. conditioned on the
(my-coin)                                 ;;   (marginalized) coin weight.
(define your-coin (sample-coin))          ;; construct a second coin, with
(your-coin)                               ;;   (in general) different weight
```

This Church program makes explicit the de Finetti representation of the exchangeable sequence of coin flips. A functional programmer would say the constructed coins are *closures* containing their weights: a caller cannot reach inside the `my-coin` procedure to discover (or depend on) the weight of the coin from which i.i.d. flips are being generated. A statistician would say that a caller can only reason about any sequence of flips via their marginal, in which the individual `flips` are not i.i.d., but are exchangeable. Any sequence of calls to a pure thunk is clearly exchangeable. This makes the construction of Daniell-Kolmogorov consistent distributions – often desirable in data modeling – straightforward. Furthermore, this criterion yields a sound (but not complete) program analysis for verifying exchangeability, which could be automatically exploited during inference.

Functions of no arguments are traditionally called *thunks*. Our example suggests that procedures which return pure thunks correspond to computable de Finetti representations. The infinitely exchangeable sequence of random variables being represented is the sequence of repeated calls to the thunk being returned. The central challenge in extending this to nonparametric objects is to deal with countably infinite de Finetti representations using only finite computation time and memory. To address this challenge, we employ *delayed evaluation*, a central idea in functional programming.

The value of a delayed expression is computed only if some other evaluation depends on it. We support delays via the use of the primitive `mem`, which implements *memoization*. `mem` accepts any procedure as its sole argument, and behaves as though it evaluates (i.e. samples from) the procedure (distribution) for every possible collection of argument values, returning a procedure that produces the corresponding sampled value for each argument when called.

We can implement `mem` using finite computation by having it immediately return a stateful procedure closed over an initially empty mapping from argument values to samples. When the procedure is called, it

checks to see if the mapping for its given arguments already exists, and if so, returns that cached value. Otherwise, it applies the underlying procedure, stores the result in the map, and returns it. This statefulness allows us to delay countably many computations while preserving exchangeability. Using `mem`, we can implement the Dirichlet process, following [12]:

```
(define (DP concentration base-measure)
  (let ((sticks (mem (lambda j (random-beta 1.0 concentration))))
        (atoms  (mem (lambda j (base-measure)))))
    (lambda ()
      (let loop ((j 1))
        (if (flip (sticks j))    ;; with probability (stick j)
            (atoms j)            ;;   return j'th sample from base measure
            (loop (+ j 1)))))))  ;; otherwise move to (j+1)'th stick
```

In fact, `DP` lets us generalize memoization to a form more useful in the stochastic setting. On repeated calls with the same arguments, we want a *stochastically memoized* procedure that sometimes returns old values, but sometimes samples new values.

```
(define (DPmem alpha proc)
  (let ((restaurants (mem (lambda args (DP alpha
                                            (lambda () (apply proc args)))))))
    (lambda args ((apply restaurants args)) )))
```

`DPmem` with `alpha` set to 0 recovers `mem`, and with `alpha` set to ∞ recovers no memoization (wasting space). This idiom lets us compactly describe a wide range of Dirichlet process based models in the literature, in particular recursively structured models not easily describable in graphical terms (see Figure 1). Many other higher order procedures play important roles in functional programming, and may suggest new stochastic processes.

We also provide a Church program that uses the stick breaking representation of the Indian Buffet Process [16, 5] introduced in [15]:

```
(define (ibp-stick-breaking-process concentration base-measure)
  (let ((sticks (mem (lambda j (random-beta 1.0 concentration))))
        (atoms  (mem (lambda j (base-measure)))))
    (lambda ()
      (let loop ((j 1) (dualstick (sticks 1))}
        (append (if (flip dualstick)           ;; with prob. dualstick
                    (atoms j)                  ;;   add feature j
                    '())                       ;; otherwise, next stick
                (loop (+ j 1) (* dualstick (sticks (+ j 1)))) ))))))
```

This procedure does not halt, and therefore does not induce a well-defined distribution on values, although the original IBP does. This raises the question of whether the IBP has a computable de Finetti representation and may have implications for sampler design.

Church also introduces `(query <expr> <pred>)`, which samples a value `v` from the marginal distribution on values of `<expr>` given that `(<pred> v)` returns true. This provides a Turing-universal target for exact and approximate inference, and exposes further connections between probability and computing. For example, a Church representation of a distribution also has well-defined time, space and entropy complexity, which interacts with the complexity of inference

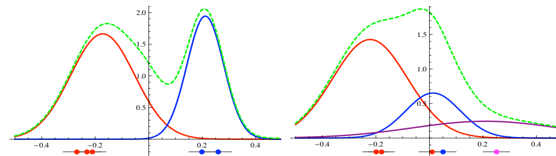

Figure 1. Exact posterior samples from a DP mixture of Gaussians (with Gaussian mean and inverse gamma variance), using the collapsed rejection algorithm for `query`.

schemes that use forward simulation (including our MH algorithm). Different Church representations of a given nonparametric object may thus be more or less suitable for different inference algorithms. Furthermore, although our generic inference algorithms are currently less efficient than special-purpose alternatives, techniques for functional program analysis and transformation between marginally equivalent representations could yield significant improvements. For example, we think exploiting the dynamic programming ideas from [9] in the general context of Church (or programmatically identifying the subset of programs to which those techniques apply) will implicate flow analysis techniques [13].

Figure 1 Examples of stochastic transition models.

This deterministic higher-order function defines the basic structure of stochastic transition models:

```
(define (unfold expander symbol)
  (if (terminal? symbol)
      symbol
      (map (lambda (x) (unfold expander x))
           (expander symbol) )))
```

A Church model for a PCFG transitions via a fixed multinomial over expansions for each symbol:

```
(define (PCFG-productions symbol)
  (cond ((eq? symbol 'S) (multinomial '((S a) (T a)) (0.2 0.8)))
        ((eq? symbol 'T) (multinomial '((T b) (a b)) (0.3 0.7))) ))

(define (sample-pcfg) (unfold PCFG-productions 'S))
```

The HDP-HMM [2, 14] uses memoized symbols for states and memoizes transitions. Fresh symbols are generated by the exchangeable (but stateful) primitive `gensym`, which returns distinct symbols on each call:

```
(define get-symbol (DPmem 1.0 gensym))
(define get-observation-model (mem (lambda (symbol) (make-100-sided-die))))
(define ihmm-transition (DPmem 1.0 (lambda (state)
                                     (if (flip) 'stop (get-symbol)) )))
(define (ihmm-expander symbol)
  (list ((get-observation-model symbol)) (ihmm-transition symbol)) )
(define (sample-ihmm) (unfold ihmm-expander 'S))
```

The HDP-PCFG [8] is also straightforward:

```
(define terms      '( a  b  c  d))
(define term-probs '(.1 .2 .2 .5))
(define rule-type (mem (lambda symbol)
                        (if (flip) 'terminal 'binary-production))
(define ipcfg-expander (DPmem 1.0 (lambda (symbol)
                                    (if (eq? (rule-type symbol) 'terminal)
                                        (multinomial terms term-probs)
                                        (list (get-symbol) (get-symbol)) ))))
(define (sample-ipcfg) (unfold ipcfg-expander 'S))
```

Making adapted versions of any of these models [6] only requires stochastically memoizing `unfold`:

```
(define adapted-unfold
  (DPmem 1.0 (lambda (expander symbol)
               (if (terminal? symbol)
                   symbol
                   (map (lambda (x) (adapted-unfold expander x))
                        (expander symbol)) ))))
```

References

[1] H. Abelson and G. J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA, USA, 1996.

[2] M. Beal, Z. Ghahramani, and C. Rasmussen. The infinite hidden Markov model. *NIPS 14*, 2002.

[3] A. Church. A Set of Postulates for the Foundation of Logic. *The Annals of Mathematics*, 33(2):346–366, 1932.

[4] N. Goodman, V. Mansinghka, D. M. Roy, K. Bonawitz, and J. Tenenbaum. Church: a language for generative models with non-parametric memoization and approximate inference. In *Uncertainty in Artificial Intelligence*, 2008.

[5] T. L. Griffiths and Z. Ghahramani. Infinite latent feature models and the indian buffet process. In *Advances in Neural Information Processing Systems 18*, 2006.

[6] M. Johnson, T. Griffiths, and S. Goldwater. Adaptor grammars: A framework for specifying compositional nonparametric Bayesian models. *NIPS 19*, 2007.

[7] R. Kelsey, W. Clinger, and J. R. (eds.). Revised^5 Report on the Algorithmic Language Scheme. *Higher-Order and Symbolic Computation*, 11(1): 7–105, 1998.

[8] P. Liang, S. Petrov, M. Jordan, and D. Klein. The Infinite PCFG using Hierarchical Dirichlet Processes. *Proc. EMNLP-CoNLL*, 2007.

[9] I. P. Max Welling and E. Bart. Infinite State Bayesian Networks For Structured Domains. In *Neural Information Processing Systems*, 2007.

[10] J. L. McCarthy. Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. *Communications of the ACM*, 3(4):184–195, 1960.

[11] J. C. Reynolds. Definitional interpreters for higher-order programming languages. In *ACM '72: Proceedings of the ACM annual conference*, pages 717–740, New York, NY, USA, 1972. ACM. doi: http://doi.acm.org/10.1145/800194.805852.

[12] J. Sethuraman. A Constructive definition of Dirichlet priors. *Statistica Sinica*, 4, 1994.

[13] O. G. Shivers. *Control-flow analysis of higher-order languages or taming lambda*. PhD thesis, Carnegie Mellon University, 1991.

[14] Y. W. Teh, M. I. Jordan, M. J. Beal, and D. M. Blei. Hierarchical Dirichlet processes. *Journal of the American Statistical Association*, 101(476): 1566–1581, 2006.

[15] Y. W. Teh, D. Görür, and Z. Ghahramani. Stick-breaking construction for the Indian buffet process. In *Proceedings of the International Conference on Artificial Intelligence and Statistics*, volume 11, 2007.

[16] R. Thibaux and M. I. Jordan. Hierarchical beta processess and the indian buffet process. In *Proceedings of the Eleventh International Conference on Artificial Intelligence and Statistics*, 2007.