An Exponential Family Basis for Probabilistic Programming

Chad Scherrer
Galois Inc.
chad.scherrer@galois.com

1. Introduction

Many common distributional families take the form of *exponential families*. The functional form of the conditional densities of these families gives them convenient compositional properties, yielding opportunities for powerful reasoning and optimization.

We present a Haskell-based approach for expressing probabilistic models in terms of *free arrows*, over a basis of exponential families. Arrows are more restrictive than the more common monadic approach, but this sacrifice in expressiveness is balanced with broader opportunities for inference, for example in terms of the dependency graph of random variables. Moreover, any monadic inference method is easily applied to arrow-based models.

2. Exponential Families

An *exponential family* (Barndorff-Nielsen 1978) is a family of distributions represented by a conditional density $f(x|\theta)$ satisfying

$$\log f(x|\theta) = \eta(\theta) \cdot t(x) - a(\theta) + b(x)$$

for some functions η, t, a, and b.[1] Note that the first term is a dot product; η and t are both vector-valued.

Exponential families encompass a wide variety of distributional families, including normal, Poisson, beta, gamma, multinomial, and Dirichlet. Combining these arithmetically or hierarchically (*e.g.*, as a mixture distribution) increases this even further.

Exponential families have a number of convenient properties:

- t is a *sufficient statistic*, which is useful for simplifying inference problems.
- Exponential families are "closed under replication"; the joint distribution of independent and identically distributed replicates of a given exponential family, itself constitutes an exponential family.
- The restriction of conditional distributions to members of exponential families leads to a striaghtforward implementation of mean-field variational inference (Blei 2011).

To specify an exponential family requires specifying the functional paramters η, t, a, and b above. For efficiency, we can also specify a function for sampling a value. Using the popular `mwc-random` package (O'Sullivan 2009), we arrive at

```
newtype Rand a = Rand (MWC.GenIO -> IO a)

data ExpFam θ x = ExpFam
  (θ -> [Double]) —— η
  (x -> [Double]) —— t
  (θ -> Double)   —— a
  (x -> Double)   —— b
  (θ -> Rand x)   —— sample
```

[1] In common notation, all of these but η are upper-case; we have changed this to be consistent with Haskell name requirements.

For example, the family of normal distributions with unknown mean μ and standard deviation σ can be represented as

```
normal :: ExpFam (Double, Double) Double
normal = ExpFam η t a b sample
  where
  η (μ,σ)      = [μ/sq σ, -0.5/sq σ]
  t x          = [x, sq x]
  a (μ,σ)      = 0.5 * sq (μ/σ) + log σ
  b x          = -0.5 * log (2*π)
  sample (μ,σ) = MWC.normal μ σ
```

The functional form of exponential families leads to convenient optimizations. For example, a vector of independent and identically distributed (iid) replicates from an exponential family constitutes another exponential family, which be be expressed via the following combinator.

```
iid :: Int -> ExpFam θ x -> ExpFam θ [x]
iid k (ExpFam η t a b sample) = ExpFam
  (\θ -> η θ)
  (\xs -> map sum [t x | x <- xs])
  (\θ  -> k * a θ)
  (\xs -> sum [b x | x <- xs])
  (replicateM k . sample)
```

For models with iid observations, this allows inference to be expressed in a natural way as a result of the sufficient statistic intrinsic to exponential families, without the need for complex computer algebra transformations.

3. Free Arrows

In functional programming languages, probabilistic programming is often expressed in terms of a free monad. This allows complex distributions and models to be built from a given set of components, and allows evaluation to be deferred and customized to each inference method of interest.

Though a free monad approach could conceivably be used in the current approach, we find it more instructive to explore the design space. We therefore carry the two-parameter kind of `ExpFam` forward to models, and describe a design based on *free arrows* (Visscher 2012). Figure 1 defines free arrows and operations on them.

4. Models

In the current approach, a probabilistic `Model` is a free arrow over the `ModelF` functor. Figure 2 gives details of the implementation.

The basis functor `ModelF` includes the ability to `Sample`, and also to increment the `LogWeight` dependent on the parameter value. The `LogWeight` constructor can be used to encode observed data or to express distributions that cannot easily be sampled from. Note that the "output" parameter of `LogWeight` has type unit; this encodes that it is effectful, corresponding to scaling the distribution by some function of its parameters.

Though a `Model` is expressed in terms of exponential families, combining elements from these families allows for very flexible

```
type f :~> g = forall a b. f a b -> g a b

newtype FreeA f a b = FreeA {
  runFreeA :: forall arr. Arrow arr =>
    (f :~> arr) -> arr a b }

unit :: x :~> FreeA x
unit a = FreeA $ \k -> k a

leftAdjunct :: (FreeA x :~> y)->(x :~> y)
leftAdjunct f = f . unit

rightAdjunct :: Arrow y => (x :~> y)->(FreeA x :~> y)
rightAdjunct f a = runFreeA a f
```

Figure 1. Operations on free arrows, adpated from Visscher (2012)

```
data ModelF θ x where
  Sample :: ExpFam θ x -> ModelF θ x
  LogWeight :: (θ -> Double) -> ModelF θ ()

type Model θ x = FreeA ModelF θ x

sample :: ExpFam θ x -> Model θ x
sample dist = unit $ Sample dist

observe :: x -> ExpFam θ x -> Model θ ()
observe x d = logWeight $ \θ -> logPdf d θ x

logWeight :: (θ -> Double) -> Model θ ()
logWeight f = unit $ LogWeight f

logPdf :: ExpFam θ x -> θ -> x -> Double
logPdf dist θ x = η θ · t x - a θ + b x
  where
  dist = ExpFam η t a b
  xs · ys = sum $ zipWith (*) xs ys
```

Figure 2. `Model`s, and some helpful combinators.

modeling. For example, Student's T distribution cannot be expressed within a single exponential family, but it can be built using and inverse Gamma and normal distributions. If

$$\sigma^2 \sim \texttt{InverseGamma}(\nu/2, \nu/2)$$
$$x \sim \mathcal{N}(0, \sqrt{\sigma^2}) \,,$$

then $x \sim \texttt{StudentT}(\nu)$. Thus, using the `proc` syntax for arrows (Paterson 2001), we can write

```
studentT :: Model Double Double
studentT = proc ν -> do
  s2 <- sample inverseGamma -< (ν/2, ν/2)
  sample normal -< (0, sqrt s2)
```

For a model with observed data, consider the simple example

$$\mu \sim \mathcal{N}(0, 1)$$
$$x \sim \mathcal{N}(\mu, 1) \,,$$

where data x is observed, and we'd like to perform inference on μ. We can express this as follows:

```
model :: Double -> Model () Double
model x = proc () -> do
  μ <- sample normal -< (0,1)
  observe x normal -< (μ,1)
  returnA -< μ
```

Note that the observed data `x` cannot be part of the arrow in this case, because an expression of the form `proc x -> f x -< k` would desugar to `arr (\x -> k) >>> f x`, which is nonsense.

5. Inference

The `rightAdjunct` function specializes to

```
runModel:: Arrow a=> (ModelF :~> a)->(Model :~> a)
```

This makes it easy to map a `Model` into any arrow, through specification of the targets of the basis elements. Note that this preserves transparency of the `ExpFam` constructor, so inference methods have access to the functional decomposition of the pdf as well as the sampling function. In addition, mapping to a *Kleisli arrow* like `(θ -> Rand x)` gives access to any inference method that can be written monadically.

Beyond monadic inference, this arrow-based approach preserves information about the dependency graph, leading to opportunties for graph-based methods like belief propagation.

6. Conclusion

We have introduced an approach for probabilistic programming using free arrows over a basis comprised of exponential families and log-density weighting.

Prior to this work, Toronto et al. (2015) described an approach for using arrows to restrict the functional form of distributions, yielding a way to reason about inverse images. This relied heavily on inverse transform sampling, and hid the arrow-based logic, rather than exposing it as the programming model.

This work is in its very early stages, but we see great potential in the conciseness and composability of this approach. We are hopeful that future explorations in this area will lead to further improvements in expressive and composable models with access to a wide variety of inference methods.

Acknowledgement

The author is grateful for discussions on this topic with Frank Wood, Nathan Collins, and Rob Dockins. This research was supported by the DARPA PPAML program, contract number FA8750-14-C-0003.

References

O. Barndorff-Nielsen. *Information and Exponential Families in Statistical Theory*. John Wiley & Sons, 1978.

D. Blei. *Variational Inference*, 2011. URL `https://www.cs.princeton.edu/courses/archive/fall11/cos597C/lectures/variational-inference-i.pdf`.

B. O'Sullivan. *mwc-random*, 2009. URL `https://hackage.haskell.org/package/mwc-random`.

R. Paterson. A new notation for arrows. *ACM SIGPLAN Notices*, 36(10): 229–240, 2001.

N. Toronto, J. McCarthy, and D. Van Horn. Running probabilistic programs backwards. In *European Symposium on Programming Languages and Systems*, pages 53–79. Springer, 2015.

S. Visscher. *Useful operations on free arrows*, 2012. URL `http://stackoverflow.com/questions/12001350`.