Bayesian Probabilistic Modelling in Haskell

and the Conditional Probability Category

T. Mark Ellison

April 17, 2019

Australian National University

Table of contents

Naive Probability Theory

Natural Probability

Finite set A

$$f : A \to [0, 1]$$

$$\sum_{a \in A} f(a) = 1$$

A Pair of Dice

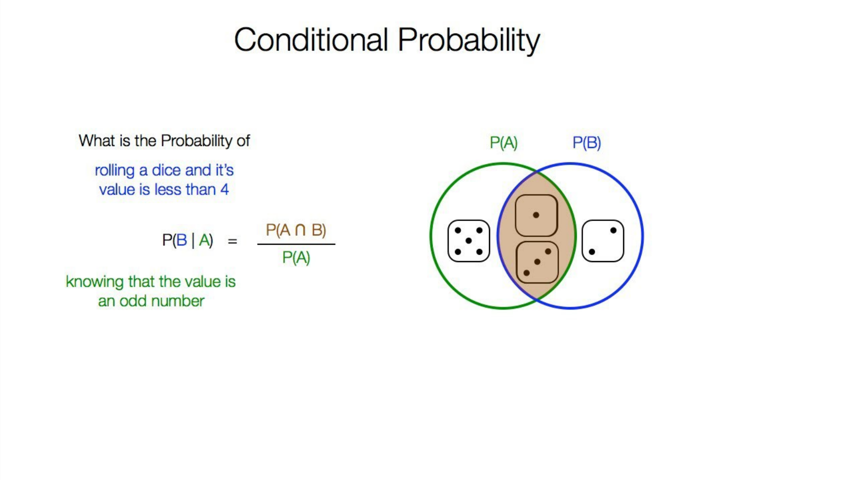
$P(2 \times H) = 1/4$
$P(H\&T) = 1/2$
$P(2 \times T) = 1/4$

Conditional Probability Reminder

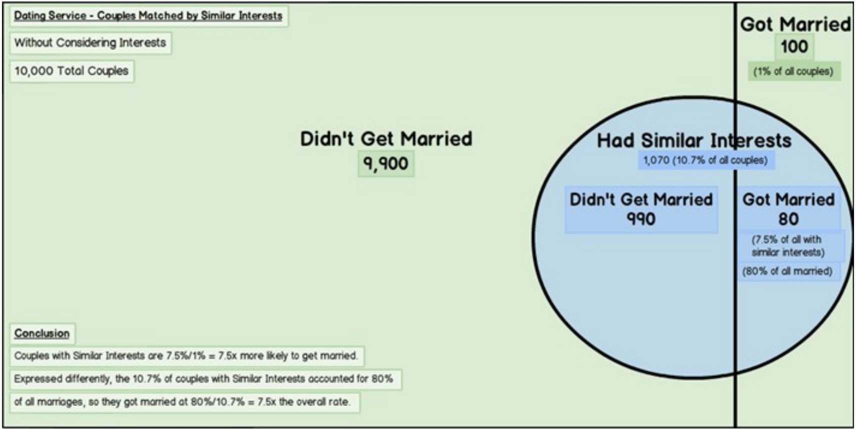
$$P(A|B) = \frac{P(A \cap B)}{P(B)}$$

$$P(B|A) = \frac{P(A \cap B)}{P(A)}$$

Conditional Probability Reminder



Bayes Theorem Reminder



Implementing Distributions as Arrays

Distribution as Array

Erwig and Kollmansberger [2006]

```
newtype Probability = P Float
newtype Dist a = D {unD :: [(a,Probability)]}

instance Monad Dist where
    return x     = D [(x,1)]
    (>>=) (D d) f = D [(y,q*p) | (x,p) <- d, (y,q) <- unD (f x)]
    fail         = D []
```

Distribution as Array

Erwig and Kollmansberger [2006]

Does it obey the monad laws?

Left Identity

```
return a >>= f == f a
D [(a,1)]  >>= f
== D [(y,q*p) | (x,p) <- [(a,1)], (y,q) <- unD (f x)]
== D [(y,q) | (y,q) <- unD (f a)]
== D (unD (f a))
== f a
```

Distribution as Array

Erwig and Kollmansberger [2006]

Does it obey the monad laws?

Right Identity

```
m >>= return == m
(D d) >>= return
== D [(y,q*p) | (x,p) <- d, (y,q) <- unD (return x)]
== D [(y,q*p) | (x,p) <- d, (x,1) <- unD (return x)]
== D [(x,p) | (x,p) <- d]
== D d
```

Distribution as Array

Erwig and Kollmansberger [2006]

Does it obey the monad laws?

Associative Law

```
(m >>= f) >>= g == m >>= (\x -> f x >>= g)
((D d) >>= f) >>= g
== D [(c,pq*r) | (b,pq) <- D [(b,p*q) | (a,p) <- d, (b,q) <- unD (f a)],
                 (c,r) <- unD (g b)]
== D [(c,p*q*r) | (a,p) <- d, (b,q) <- unD (f a), (c,r) <- unD (g b)]
== D [(c,p*qr) | (a,p) <- d,
                 (c,qr) <- D [(c,q*p) | (b,q) <- unD(f a), (c,q) <- unD (g b)]]
== (D d) >>= (\x -> f x >>= g)
```

Explore this Solution

Load the library: **Numeric**.Probability

Try out the examples

But is this the best way to represent probabilities?

Problems

how to handle analytic distributions? e.g.

$$P(x|\sigma) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{x^2}{2\sigma^2}}$$

. In performing actions like taking marginal probabilities, or conditionalisation, there may/may not be efficient analytical solutions

even for finite models questions arise about how to ameliorate computation in places where there are predictably low probabilities

not satisfying to a category theorist - the construction is very ad-hoc.

Category Theory of Probabilities

Category-Theoretic Accounts of Probabilities

Cencov [2000] (originally 1982) gives a basis for non-Bayesian statistical inference in Category Theory,

Lawvere 1962 unpublished gave first construction of a probabilistic category, extending this as the basis for Bayesian statistics,

Giry [1982] showed that the endofunctor on the category of measurable spaces G: M M associated to the probability adjunction given by Lawvere forms a monad, and that Lawveres category of probabilistic mappings is the Kleisli category of that monad.

But tonight's award in the category of Categorical Underpinnings of Probability goes to Culbertson and Sturtz [2014] for *the most general category in which Bayesian probability can be realized.*

Culbertson & Sturtz 2014

Appl Categor Struct (2014) 22:647–662
DOI 10.1007/s10485-013-9324-9

A Categorical Foundation for Bayesian Probability

Jared Culbertson · Kirk Sturtz

Received: 18 June 2012 / Accepted: 11 July 2013 / Published online: 21 August 2013

Abstract Building on the work of Lawvere and others, we develop a categorical framework for Bayesian probability. This foundation will then allow for Bayesian representations of uncertainty to be integrated into other categorical modeling applications. The main result uses an existence theorem for regular conditional probabilities by Faden, which holds in more generality than the standard setting of Polish spaces. This more general setting is advantageous, as it allows for non-trivial decision rules (Eilenberg–Moore algebras) on finite (as well as non finite) spaces. In this way, we obtain a common framework for decision theory and Bayesian probability.

Keywords Bayesian probability · Decision rules · Giry monad · Regular conditional probability

Conditional Probabilities as Fundamental

$$\begin{tikzcd}
 & 1 \arrow[dl, "P_X"'] \arrow[d, dashed, "J_h"] & \\
X \arrow[rr, bend right, "h"'] & X \times Y \arrow[l, dashed, "\pi_X"] \arrow[r, dashed, "\pi_Y"'] & Y
\end{tikzcd}$$

Conditional Probabilities as Fundamental

$$
\begin{tikzcd}
 & 1 \arrow[ddl, "P_X"'] \arrow[ddr, "P_Y"] \arrow[d, dashed, shift right, "J_k"'] \arrow[d, dashed, shift left, "J_h"] & \\
 & X \times Y \arrow[dl, dashed, "\pi_X"] \arrow[dr, dashed, "\pi_Y"'] & \\
X \arrow[rr, shift left, "h"] & & Y \arrow[ll, shift left, "k"]
\end{tikzcd}
$$

GADTs of Probabilities

What are GADTs?

Generalised Abstract Data Types

Not native in Haskell, but can be adopted.

```haskell
{-# Language GADTs #-}
```

Read more about them at: e.g. the Haskell wikibook

What are GADTs?

Constructors do not make data-types, they just corral their sets of arguments.

```haskell
data Expr a where
      I   :: Int  -> Expr Int
      B   :: Bool -> Expr Bool
      Add :: Expr Int -> Expr Int -> Expr Int
      Mul :: Expr Int -> Expr Int -> Expr Int
      Eq  :: Eq a => Expr a -> Expr a -> Expr Bool
```

```haskell
eval  :: Expr a -> a
eval (I n) = n
eval (B b) = b
eval (Add e1 e2) = eval e1 + eval e2
eval (Mul e1 e2) = eval e1 * eval e2
eval (Eq e1 e2) = eval e1 == eval e2
```

GADTs for Probabilities

Ścibior *et al.* [2015, p169]

```haskell
data Dist a where
  Return      :: a -> Dist a
  Bind        :: Dist b -> (b -> Dist a) -> Dist a
  Primitive   :: Sampleable d => d a -> Dist a
  Conditional :: (a -> Prob) -> Dist a -> Dist a
condition = Conditional

instance Functor Dist where
  fmap = liftM

instance Monad Dist where
  return = Return
  (>>=)  = Bind
```

Sampleable

Ścibior *et al.* [2015, p169]

```
instance Sampleable Dist where
  sample g (Return x)          = x
  sample g (Bind d f)          = sample g1 y  where
    y       = f (sample g2 d)
    (g1,g2) = split g
  sample g (Primitive d)       = sample g d
  sample g (Conditional c d) = undefined
```

Class Bayes

Bayes Class

```
class (Functor m) => Bayes m where
      bpure             :: a -> m a
      cond              :: m (a,b) -> a -> m b
      decond            :: m a -> (a -> m b) -> m (a,b)
```

Bayes Class

```
class (Functor m) => Bayes m where
    bpure                                   :: a -> m a
    cond                                    :: m (a,b) -> a -> m b
    decond                                  :: m a -> (a -> m b) -> m (a,b)
```

Laws:

```
cond2                                   :: m (a,b) -> (m a, a -> m b)
cond2 mab                                = (fmap fst mab, cond mab)
cond2 $\cdot$ decond == id
decond $\cdot$ cond2 == id
```

and more to work out

Bayes Class - Applicative, Monad

```
instance (Bayes m) => Applicative m where
      pure                              = bpure
      (<*>) mf ma                       = fmap ($) $ decond mf $ const ma
```

```
instance (Bayes m) => Monad m where
      (>>=) ma mbKa                     = fmap snd $ decond ma mbKa
```

Monads in Bayes Class

Maybe is a Bayes Class

```
instance Bayes Maybe where
    -- bpure                            :: a -> m a
    bpure                             = pure
    -- cond                             :: m (a,b) -> a -> m b
    cond | Nothing _                  = Nothing
         | (Just (aa,bb))             = \ a -> if aa = a then Just bb else N
    -- decond                           :: m a -> (a -> m b) -> m (a,b)
    decond Nothing _                  = Nothing
    decond (Just a) mbKa              = let mk (Just b) = Just (a,b)
                                            mk Nothing = Nothing
                                        in  mk $ mbKa a
```

Set is a Bayes Class

```
instance (Eq a) => Bayes Set where
    -- bpure                          :: a -> m a
    bpure a                           = Set [ a ]
    -- cond                           :: m (a,b) -> a -> m b
    cond mab aa                       = map snd $ filter (\ (a,b) -> (a == a
    -- decond                         :: m a -> (a -> m b) -> m (a,b)
    decond ma mbKa                    = ...
```

Slicings are Bayes Class

A *slicing* over X is a category with objects a −> X for each type a.

Functions are (a −> X) −> (b −> X) with the contravariant functor defined by fmap fab b2x = b2x . fab

Currying helps construct the methods for slice type families provide the Bayes Class functions.

```
instance (Eq a) => Bayes Set where
    -- bpure                          :: a -> m a
    bpure a                          = Set [ a ]
    -- cond                           :: m (a,b) -> a -> m b
    cond mab aa                      = curry mab aa
    -- decond                         :: m a -> (a -> m b) -> m (a,b)
    decond ma mbKa                   = uncurry mbKa -- ignores first argume
```

Slices with Normalisation

If the slice target is a group (using $*$ and $/$), then we can incorporate normalisation.

```
instance (Eq a) => Bayes Set where
    -- cond                    :: m (a,b) -> a -> m b
    cond mab aa                = let scale  = fmap fst mab a
                                  in  \ b -> (curry mab a b) / scale
    -- decond                  :: m a -> (a -> m b) -> m (a,b)
    decond ma mbKa             = \ (a,b) -> let scale = ma a
                                            in  mbKa a b * scale
```

Lists are Bayes Class - instance of Slice plus Normalisation

I'm just going to state this without proof.

Back to Probabilities with GADTs

Probabilities and GADTs - Applicative

```haskell
data D where
      Pure    ::  a -> D a
      Cond    ::  D (a,b) -> a -> D b
      DeCond :: D a -> (a -> D b) -> D (a,b)
      --
      FromList  ::  [(a,Double)] -> D a
      FromFunction :: (a -> Double) -> D a
eval  ::  D a -> a -> Double

Formally  nicer  than  \'{S}cibior's,  but  lots  of work needed to handle continuous
```

Membership of Bayes class follows immediately.

Construction from lists or functions assigning probabilities.

Conclusion

- probabilities can be represented by List or Map type associating items with probabilities
- this is not satisfying / efficient for all cases
- GADTs allow these implementation decisions to be postponed
- Category Theory allows neat generalisation of symmetric conditionalisation: **class** Bayes, which implies a monad
- many standard monads are in this class
- can inspire a simple GADT characterisation of probabilities and distributions

Thank You for your Attention

As this work comes together, I'll add to my github repository https://github.com/tyrannomark/HaskellBayes.

References

N. N. Cencov. *Statistical Decision Rules and Optimal Inference*. American Mathematical Soc., April 2000. Google-Books-ID: 63CPCwAAQBAJ.

Jared Culbertson and Kirk Sturtz. A Categorical Foundation for Bayesian Probability. *Applied Categorical Structures*, 22(4):647–662, August 2014.

Martin Erwig and Steve Kollmansberger. Functional pearls: Probabilistic functional programming in Haskell. *Journal of Functional Programming*, 16(01):21–34, 2006.

Michele Giry. A categorical approach to probability theory. In *Categorical aspects of topology and analysis*, pages 68–85. Springer, 1982.

Adam Ścibior, Zoubin Ghahramani, and Andrew D Gordon. Practical