Distribuciones discretas con mónadas

Mario Román

<2018-05-25 Fri 18:34>

El modelo

Como parte de una serie de ejemplos sobre uso de mónadas, he escrito un poco de código para modelar distribuciones discretas usando mónadas. Por un lado, usa un generador congruencial lineal para generar números aleatorios; y por otro, usa la mónada State para pasar una semilla aleatoria de una función a otra que me permita seguir generado números aleatorios. Por último, aporta un método que deriva Show para probar las distribuciones y dibujar un histograma de cualquiera de ella.

Componiendo distribuciones

Lo más útil de esta idea es el poder generar unas distribuciones a partir de otras. La primera que intentamos es una uniforme discreta (un dado de `n` caras) usando una semilla inicial. En el siguiente código se implementa el generador congruencial.

```
dice :: Int -> Distribution Int
dice n = state (\s -> (s `mod` n + 1, 16807*s `mod` 2147483647))
```

Vemos que funciona como una distribución uniforme.

```
>>> dice 6

1:          ################
2:          ################
3:          ################
4:          ################
```

```
5:          ################
6:          ################
```

Y desde ella generar fácilmente otras usando funciones que compongan distribuciones. Un ejemplo es usar `() = liftM2 (+)` para sumar dados.

```
>>> dice 6  dice 6

2:          #####
3:          ##########
4:          ###############
5:          ####################
6:          ##########################
7:          ##############################
8:          ########################
9:          ####################
10:          ###############
11:          ##########
12:          #####
```

Otras distribuciones

Si seguimos componiendo usando la estructura de mónada, podemos crear otras distribuciones simples como la distribución de **Bernoulli** y la distribución **binomial**.

```
bernoulli :: Double -> Distribution Int
bernoulli p = do
  sample <- dice 1000000
  if (fromIntegral sample / 1000000.0 < p)
    then return 1
    else return 0

binomial :: Int -> Double -> Distribution Int
binomial k p = sum <$> replicateM k (bernoulli p)
```

Lo interesante de este código es que dejamos a la estructura de mónada encargarse internamente de el paso de la semilla de aleatoriedad y la construcción de distribuciones complejas puede hacerse composicionalmente.

El código

El siguiente código es una primera implementación de este post en Haskell.

```haskell
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE TypeSynonymInstances #-}
{-
En este archivo vamos a usar mónadas para definir distribuciones
discretas de probabilidad y aplicar operaciones algebraicas sobre ellas.
-}
import Control.Monad.State

-- Generación aleatoria
-- Para generar números pseudoaleatorios usaremos LCGs. La idea es tener
-- un dado que nos dé una distribución de probabilidad uniforme dada una
-- semilla y nos devuelva el resultado de la tirada y una nueva semilla
-- aleatoria. Buscamos que un dado de seis caras sea, por ejemplo:
--
--   dice 6 :: Seed -> (Int, Seed)
--
-- Si quisiéramos tirar dos dados, tendríamos que tomar la semilla resultante
-- del primer lanzamiento y pasarla al segundo; algo así:
--
--   let (a,newseed) = dice 6 seed
--   let (b,_)       = dice 6 newseed
--   print [a,b]
--
-- Pero esto se hace demasiado complejo. La semilla, en el fondo, es un
-- estado, así que podemos modelarla con la mónada State. Cada lanzamiento
-- será de la forma:
--
--   State Seed a   ===   Seed -> (a, Seed)
--
-- Luego podemos llamar a la distribución: Distribution a = State Seed a, y
-- trabajar con ella usando las funciones normales de mónadas.
type Seed = Int
type Distribution = State Seed
```

```haskell
-- Nuestra primera distribución es un dado de "n" lados que usa internamente un
-- generador de números aleatorios.
dice :: Int -> Distribution Int
dice n = state (\s -> (s `mod` n + 1, 16807*s `mod` 2147483647))

-- Una moneda es un dado de dos caras
coin :: Distribution Int
coin = dice 2

-- Estas funciones pueden ser llamadas con la mónada estado, dada una
-- semilla inicial, devuelven el resultado y la nueva semilla:
--
-- > runState (dice 6) 1
-- (2,16807)
-- > runState (dice 6) 16807
-- (2,282475249)
--
-- El usar composición con mónadas nos ahorraba controlar los errores
-- en el primer caso, aquí nos ahorra controlar el cambio de semilla,
-- por ejemplo: para lanzar dos dados y hacer que la semilla se pase
-- internamente.
twodices' :: Distribution Int
twodices' = do
  a <- dice 6
  b <- dice 6
  return (a+b)

(|) :: Distribution Int -> Distribution Int -> Distribution Int
(|) = liftM2 (+)
(|) :: Distribution Int -> Distribution Int -> Distribution Int
(|) = liftM2 (*)

twodices :: Distribution Int
twodices = dice 6 | dice 6

-- Igual que hago esto, podría hacer:
--
--   foldr () (return 0) [dice 6,dice 6,dice 6]
--   foldr () (return 0) (replicate 10 (dice 6))
```

```haskell
--
-- Que da un resultado que se aproxima a una distribución normal.

-- Ahora, desde ella, podemos crear otras distribuciones. La distribución de
-- bernoulli sería la de una moneda trucada donde una cara tiene probabilidad
-- p y la otra tiene probabilidad (1-p).
bernoulli :: Double -> Distribution Int
bernoulli p = do
  sample <- dice 1000000
  if (fromIntegral sample / 1000000.0 < p)
    then return 1
    else return 0

-- La distribución binomial es la suma de k distribuciones de Bernoulli
binomial :: Int -> Double -> Distribution Int
binomial k p = sum <$> replicateM k (bernoulli p)

-- La distribución constante y otra forma de escribir la distribución
-- binomial, de manera algebraica.
constant :: Int -> Distribution Int
constant n = return n

binomial' :: Int -> Double -> Distribution Int
binomial' k p = foldr (+) (constant 0) (replicate k (bernoulli p))



-- Muestra la distribución. Los detalles de implementación no son interesantes.
-- Hemos usado  TypeSynonymInstances para simplificar el proceso de sobrecargar
-- la instancia de Show y poder dibujar directamente por la pantalla las
-- demostraciones.
instance Show (State Seed Int) where
  show = showdist

showdist :: Distribution Int -> String
showdist dist = unlines $ map counter [minimum samples..maximum samples]
  where samples = fst $ runState (replicateM 50000 dist) 1
        counter n = show n ++ ":\t " ++ replicate ((count n samples) `div` (3000 `div`
        range = maximum samples - minimum samples + 1
```

```haskell
count :: Eq a => a -> [a] -> Int
count x = length . filter (x==)


main :: IO ()
main = return ()
```