

CI4251 - Programación Funcional Avanzada  
Respuesta de la Tarea 1

Gustavo Gutiérrez  
11-10428  
[<11-10428@usb.ve>](mailto:11-10428@usb.ve)

Abril 27, 2016

# 1. Machine Learning

## 1.1. Implementación

### 1.1.1. Comparar en punto flotante

Dos flotantes se consideraran *cercanos* en el caso en el que la diferencia entre ambos sea menor a  $\epsilon$ .

```
veryClose :: Double -> Double -> Bool
veryClose v0 v1 = abs (v1 - v0) < epsilon
```

### 1.1.2. Congruencia dimensional

Para solucionar la congruencia dimensional se le hace un map a la lista de Samples con una función que toma un Sample y le agrega un uno a su lista de coeficientes.

```
addOnes :: [Sample Double] -> [Sample Double]
addOnes = map addOne
      where addOne sam = Sample { x = 1:(x sam), y = y sam }
```

### 1.1.3. Evaluando Hipótesis

La evaluación de la hipótesis sobre una muestra implica realizar el producto punto de el vector de coeficientes de la hipótesis con el vector  $x$  de la muestra. Partiendo de la suposición de que ambos vectores tienen la misma dimensión se puede definir el producto punto como un `foldl` sobre el `zip` de ambos vectores. La función pasada al fold toma el par  $(h_i, x_i)$ , multiplica ambos términos y los suma al acumulador.

```
theta :: Hypothesis Double -> Sample Double -> Double
theta h s = foldl acum 0 $ zip (c h) (x s)
      where acum sum (h,s) = sum + (h*s)
```

El cálculo del costo de la hipótesis se realizó con un `foldl` y una función que saque las cuentas finales. Para poder realizar el cálculo en una sola pasada el acumulador del fold debe ser un par ordenado donde se vayan acumulando la suma de las evaluaciones y la cantidad de muestras observadas. Una vez terminado el fold se aplica la función `result` que extrae la información del par generado y calcula el costo final.

```

cost :: Hypothesis Double -> [Sample Double] -> Double
cost h ss = result $ foldl acum (0,0) ss
  where acum (sum, n) s = (sum + (theta h s - y s)^2 , n+1)
        result (finalSum, finaln) = finalSum / (2 * finaln)

```

## 1.2. Bajando por el gradiente

La función `descend` fue escrita con dos folds como fue indicado. El fold externo recorre la lista de variables de la hipótesis a mejorar, llevando un control de cual es la posición que se está recorriendo, pues es necesario para el fold interno. Luego el fold interno se encarga de realizar el cálculo de la variable mejorada. Para hacer esto recorre todo el conjunto de muestras acumulando la sumatoria de error y contando cuantas muestras se han observado. Los resultados de ambos fold deben ser pasados a sendas funciones auxiliares que extraigan el valor deseado del acumulador resultante.

```

descend :: Double -> Hypothesis Double -> [Sample Double]
        -> Hypothesis Double
descend alpha h ss = Hypothesis $
  extract $ foldl improve ([],0) (c h)
  where extract (xs,_) = reverse xs
        improve (h',n) hj = ((hj -
                               (result $ foldl (err n) (0,0) ss)
                              ):h', n+1)
        where result (ac, m) = ac * alpha / m
              err n (ac, m) s = (ac +
                                  (theta h s - (y s))*
                                  ((x s) !! n)
                                 , m+1)

```

Finalmente queda definir la función `gd`. Fue escrita usando un `unfoldr`. Las semillas son de tipo `(Hypothesis Double, Integer)` pues se debe llevar también el número de iteraciones para agregarlo a la tupla de resultado. La iteración se termina cuando el costo de la nueva hipótesis y el costo de la anterior son  $e$ -cercanas.

```

gd :: Double -> Hypothesis Double -> [Sample Double]
    -> [(Integer, Hypothesis Double, Double)]
gd alpha h ss = unfoldr go (h,0)
  where go (h,n) = if veryClose (cost h ssw1) (cost h' ssw1)
                      then Nothing
                      else Just ((n, h, cost h ssw1), (h', n+1))

```

```

where ssw1 = addOnes ss
      h'   = descend alpha h ssw1

```

## 2. Monoids

Para tomar en cuenta los casos en los que el `Foldable` esté vacío se define el tipo `Max a` como una instancia de `Maybe a`.

```

newtype Max a = Max { getMax :: Maybe a } deriving (Eq, Show)

```

Para que nuestro tipo `Max a` pueda instanciar la clase `Monoid` debemos pedir como mínimo que el tipo `a` sea instancia de `Ord`.

```

instance Ord a => Monoid (Max a) where

```

Solo falta definir las funciones `mempty` y `mappend`. `Mempty` lo usaremos para representar el caso base de nuestro *Monoide*, que en el caso de la clase `Maybe` es `Nothing`.

```

mempty = Max Nothing

```

`Mappend` de dos elementos se define como el máximo de sus dos valores, en caso de que ninguno sea `Nothing`.

```

mappend x y = case getMax x of
    Nothing -> y
    Just n -> case getMax y of
        Just m -> (Max . Just) $ max m n
        _      -> x

```

## 3. Zippers

Considere el tipo de datos

```

data Filesystem a = File a | Directory a [Filesystem a]

```

Diseñe un zipper seguro para el tipo `Filesystem` proveyendo todas las funciones de soporte que permitan trasladar el foco dentro de la estructura de datos, así como la modificación de cualquier posición dentro de la estructura.

```
ata Breadcrumbs a = undefined

type Zipper a = (Filesystem a, Breadcrumbs a)

oDown    ::
oRight   ::
oLeft    ::
oBack    ::
othetop  ::
odify    ::
ocus     ::
efocus   ::
```