CI4251 - Programación Funcional Avanzada
Tarea 1

Erick Marrero
09-10981
<09-10981@usb.ve>

Mayo 01, 2015

1. Machine Learning

1.1. Definiciones Generales

Para el desarrollo de la solución, serán necesarios los módulos

```
import Data.List
import Data.Functor
import Data.Monoid
import Data.Foldable (foldMap)
import Data.Tree
import Data.Maybe (fromJust)
```

Las técnicas de *Machine Learning* operan sobre conjuntos de datos o muestras. En este caso, existe un conjunto de muestras que serán usadas para "aprender", y así poder hacer proyecciones sobre muestras fuera de ese conjunto. Para user el método de regresión lineal multivariable, las muestras son *vectores* $(x_1, x_2, \ldots, x_n)$ acompañados del valor asociado y correspondiente.

En este ejercicio, nos limitaremos a usar vectores de dos variables, pero usaremos un tipo polimórfico basado en listas, para poder utilizar `Float` o `Double` según nos convenga, y modelar vectores de longitud arbitraria. Su programa no puede hacer ninguna suposición sobre la longitud de los vectores, más allá de que todos son del mismo tamaño.

Así, definiremos el tipo polimórfico

```
data Sample a = Sample { x :: [a], y :: a }
     deriving (Show)
```

Teniendo una colección de muestras como la anterior, el algoritmo calcula una *hipótesis*, que no es más que un *vector* de coeficientes $(\theta_0, \theta_1, \ldots, \theta_n)$ tal que minimiza el error de predicción $(\theta_0 + \theta_1 \times x_1 + \ldots + \theta_n x_n - y)$ para toda la colección de muestras.

```
data Hypothesis a = Hypothesis { c :: [a] }
     deriving (Show)
```

En el caso general, asegurar la convergencia del algoritmo en un tiempo razonable es hasta cierto punto "artístico". Sin entrar en detalles, es necesario un coeficiente α que regule cuán rápido se desciende por el gradiente

```
alpha :: Double
alpha = 0.03
```

También hace falta determinar si el algoritmo dejó de progresar, para lo cual definiremos un márgen de convergencia ϵ muy pequeño

```
epsilon :: Double
epsilon = 0.0000001
```

Finalmente, el algoritmo necesita una hipótesis inicial, a partir de la cual comenzar a calcular gradientes y descender hasta encontrar el mínimo, con la esperanza que sea un mínimo global. Para nuestro ejercicio, utilizaremos

```
guess :: Hypothesis Double
guess = Hypothesis { c = [0.0, 0.0, 0.0] }
```

1.2. Muestras de Entrenamiento

En este archivo se incluye la definición

```
training :: [Sample Double]
```

que cuenta con 47 muestras de entrenamiento listas para usar.

1.3. Funciones a desarrollar

1.3.1. Comparar en punto flotante

En esta función se restan los valores para chequear si es despreciable o no

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

1.3.2. Congruencia dimensional

Para agregar un coeficiente, se hace un map para recorrer cada muestra y se coloca un 1 adicional en x

```
addOnes :: [Sample Double] -> [Sample Double]
addOnes = map agregar
 where agregar m = Sample {x = add (x m), y = y m}
       add lis =  1:lis
```

1.3.3. Evaluando Hipótesis

Si tanto una hipótesis θ como una muestra X son vectores de $n+1$ dimensiones, entonces se puede evaluar la hipótesis en $h_\theta(X) = \theta^T X$ calculando el producto punto de ambos vectores.

Para ello se multiplica el vector 'h' con el vector 'x' componente a componente usando zipWith y luego se usa un foldl' para hacer las sumas de la lista resultante

```
theta :: Hypothesis Double -> Sample Double -> Double
theta h s = foldl' (+) 0 ( zipWith (*) (c h) (x s))
```

Una vez que pueda evaluar hipótesis, es posible determinar cuán buena es la hipótesis sobre el conjunto de entrenamiento. La calidad de la hipótesis se mide según su **costo** $J(\theta)$ que no es otra cosa sino determinar la suma de los cuadrados de los errores. Para cada muestra $x^{(i)}$ en el conjunto de entrenamiento, se evalúa la hipótesis en ese vector y se compara con el $y(i)$. La fórmula concreta para m muestras es

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2$$

Para el cálculo del costo, se usa un foldr para recorrer las muestras y aplicar la fórmula correspondiente, y finalmente se divide entre la cantidad de muestras

```
cost :: Hypothesis Double -> [Sample Double] -> Double
cost h ss =
       (foldr ((+).eval h) 0 ss) / (fromIntegral (length ss)*2)
        where eval hy sam = (theta hy sam - y sam)^2
```

1.4. Bajando por el gradiente

El algoritmo de descenso de gradiente por lotes es sorprendentemente sencillo. Se trata de un algoritmo iterativo que parte de una hipótesis θ que tiene un costo c, determina la dirección en el espacio vectorial que maximiza el descenso, y produce una nueva hipótesis θ' con un nuevo costo c' tal que $c' \leq c$. La "velocidad" con la cual se desciende por el gradiente viene dada por el coeficiente "de aprendizaje" α.

Dejando el álgebra vectorial de lado por un momento, porque no importa para esta materia, es natural pensar que nuestro algoritmo iterativo tendrá que detenerse cuando la diferencia entre c y c' sea $\epsilon-$despreciable.

Para calcular la nueva hipótesis, se hace un foldl' para ir recorriendo las componentes de la hipótesis anterior, y el valor semilla es una tupla que contiene una lista que guarda los valores calculados de la nueva hipótesis y el otro valor de la tupla es la posición de la componente que se está modificando. La función `calculo` es la que permite crear un nuevo valor a la lista y llama a `sumatoria` que hace un foldr para sumar todos los valores que retorna la función `funaux` que aplica la fórmula correspondiente. Y finalmente como es una tupla lo que se lleva, se toma el primero con "fst" y "reverse" para colocarlo en el orden correcto.

```
descend :: Double -> Hypothesis Double -> [Sample Double]
        -> Hypothesis Double
descend alpha h ss =
 Hypothesis {c = reverse ( fst (
   foldl' (calculo (length ss) alpha h ss ) ([],0) (c h)))}
        where calculo tam alpha hs ss (lis, num) h =
                (sumatoria tam alpha ss hs num : lis, num+1)
       sumatoria tam alpha ss hy num = c hy !! num -
                ((foldr ((+).funaux hy num) 0 ss)
                * alpha/ fromIntegral tam)
       funaux hyp num s = ((theta hyp s) - y s)* (x s !! num)
```

Sea θ_j el $j-$ésimo componente del vector θ correspondiente a la hipótesis actual que pretendemos mejorar. La función debe calcular, para todo j

$$\theta'_j \leftarrow \theta_j - \frac{\alpha}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}$$

donde m es el número de muestras de entrenamiento.

La segunda parte de este algoritmo debe partir de una hipótesis inicial y el conjunto de entrenamiento, para producir una lista de elementos tales que permitan determinar, para cada iteración, cuál es la hipótesis mejorada y el costo de la misma.

En la función `gd`, se uso "unfoldr" para ir construyendo la lista con nuevas hipótesis y nuevos costos en cada iteración, éstas usan la función `calculo` para calcular los nuevos valores. Esto se hace hasta que los costos seas despreciables usando la función `veryClose` que es la condición de parada.

```
gd :: Double -> Hypothesis Double -> [Sample Double]
   -> [(Integer,Hypothesis Double,Double)]
gd alpha h ss =
   unfoldr (\(i,hypo,c) ->
   if veryClose c
       (cost (descend alpha hypo (addOnes ss)) (addOnes ss))
        then Nothing
       else
         Just ((i,hypo,c),calculo i alpha hypo (addOnes ss)))
               (0,h,cost h (addOnes ss))
 where calculo i alpha hy ss =
       (i+1,descend alpha hy ss, cost (descend alpha hy ss) ss)
```

1.5. Resultados Obtenidos

Probar las funciones `veryClose`, `addOnes` y `theta` es trivial por inspección. Para probar la función `cost` tendrá que hacer algunos cálculos a mano con muestras pequeñas y comprobar los resultados que arroja la función. Preste atención que estas funciones *asumen* que las muestras ya incluyen el coeficiente constante 1.

Probar la función `descend` es algo más complicado, pero la sugerencia general es probar paso a paso si se produce una nueva hipótesis cuyo costo es, en efecto, menor.

```
 *Main>  take 3 (gd alpha guess training)
[(0,Hypothesis {c = [0.0,0.0,0.0]},6.559154810645744e10),
(1,Hypothesis {c = [10212.379787234042,3138.9880129854737,
                    1623.7105153222735]},
                    6.1759853666230095e10),
(2,Hypothesis {c = [20118.388180851063,6159.113611965673,
                    3148.136171427171]},
                    5.8164574323311745e10)]
```

y si se deja correr hasta terminar converge (el *unfold* **termina**) y los resultados numéricos en la última tripleta.

```
 *Main> last (gd alpha guess training)
       (1213,Hypothesis {c = [340412.65957446716,
               110631.04200819538,-6649.466000169089]},
               2.0432800506028578e9)
```

Una vez que el algoritmo converge, obtenga la última hipótesis y úsela para predecir el valor y asociado al vector $(-0{,}44127, -0{,}22368)$.

```
ghci> let (_,h,_) = last (gd alpha guess training)
ghci> let s = Sample ( x = [1.0, -0.44127,-0.22368], y = undefined }
ghci> theta h s
293081.8522224286
```

2. Monoids

Construya una instancia `Monoid` *polimórfica* para *cualquier* tipo comparable, tal que al aplicarla sobre cualquier `Foldable` conteniendo elementos de un tipo concreto comparable, se retorne el máximo valor almacenado, si existe. La aplicación se logra con la función

```
foldMap :: (Foldable t, Monoid m) => (a -> m) -> t a -> m
```

Para la construcción de este Monoid, se puede notar claramente que tienen que ser elementos comparables entre ellos ya que se va a calcular el máximo de los mismos. Como no se sabe con que estructura va a trabajar el Monoid, entonces puede ser que no se obtenga un resultado. Por ello se trabajará con el Maybe. Por lo tanto, el neutro para la función `mempty` es el tipo Nothing y para la función `mappend` retorna el máximo entre esos elementos colocando la marca Max.

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

instance (Ord a) => Monoid ( Max a ) where
       mempty                        = Max Nothing
       mappend (Max x) (Max y) = Max $ max x y
```

Oriéntese con los siguientes ejemplos

```
ghci> foldMap (Max . Just) []
Max {getMax = Nothing}
ghci> foldMap (Max . Just) ["foo","bar","baz"]
Max {getMax = Just "foo"}
ghci> foldMap (Max . Just) (Node 6 [Node 42 [], Node 7 [] ])
Max {getMax = Just 42}
ghci> foldMap (Max . Just) (Node [] [])
```

3. Zippers

Considere el tipo de datos

```
data Filesystem a = File a | Directory a [Filesystem a]
       deriving(Show,Eq)
```

Para moverse dentro de esta estructura, se propone un data Breadcrumbs que contenga el nombre del directorio donde se esta actualmente, seguido de una tripla que contienen listas. La primera lista indicará los directorios por lo que se ha bajado, es decir que son los padres del directorio actual. La segunda lista es para mostrar los `Filesystem` que tiene a la izquierda del foco, y la tercera lista representa los `Filesystem` que están a la derecha del foco.

De esta manera tenemos lo siguiente:

```
data Breadcrumbs a =
       Down a ([Filesystem a],[Filesystem a],[Filesystem a])
               deriving(Show,Eq)

type Zipper a = (Filesystem a, Breadcrumbs a)
```

Note que habrán movimientos que no se podrán hacer, y como no se sabe que movimientos se haga, se trabajará con el tipo `Maybe`.

Con esta estructura queda ahora definir las funciones:

```
goDown:: Zipper a -> Maybe (Zipper a)
goDown  (File a,_) = Nothing
goDown  (Directory a [],Down name ( lisDir ,_, _)) = Nothing
goDown (Directory a xs,Down name ( lisDir ,y,ys)) =
   Just (head xs ,
      Down a (Directory name
             (reverse y++ [Directory a xs]++ys ):lisDir,
             [],
             tail xs))

goRight:: Zipper a -> Maybe (Zipper a)
goRight  (filesys, Down name ( lis, x, xs)) =
         if null xs then Nothing
          else Just (head xs ,Down name (lis,filesys:x,tail xs))

goLeft:: Zipper a -> Maybe (Zipper a)
goLeft (filesys, Down name ( lis, x, xs)) =
       if null x then Nothing
       else Just (head x ,Down name (lis,tail x, filesys:xs))
```

```
goBack:: Zipper a -> Maybe (Zipper a)
goBack (_,Down name ( [] ,_,_)) = Nothing
goBack (_,Down name ( lisDir ,_,_)) =
       Just (prilista (head lisDir),
             Down (nameDir (head lisDir))
             (tail lisDir,[],reslista (head lisDir)))
         where prilista (Directory a b) = head b
               nameDir (Directory a b) = a
               reslista (Directory a b) = tail b


tothetop:: Zipper a -> Zipper a
tothetop (tope,Down name ( [] ,lis,lis1)) =
               (tope,Down name ( [] ,lis,lis1))
tothetop (tope,Down name ( lisdir,lis,lis1)) =
               tothetop $
               fromJust $
               goBack (tope,Down name ( lisdir,lis,lis1))

modify:: ( a -> a ) -> Zipper a -> Zipper a
modify f (File a ,Down name ( lisdir,lis,lis1)) =
               (File (f a) ,Down name ( lisdir,lis,lis1))
modify f (Directory a b,Down name ( lisdir,lis,lis1)) =
               (Directory (f a) b ,Down name ( lisdir,lis,lis1))

focus :: Filesystem a -> Zipper a
focus (File a) = (File a,Down a ([],[],[]))
focus (Directory a lis) = (Directory a lis ,Down a([] ,[],[]))

defocus :: Zipper a -> Filesystem a
defocus (File a, _ ) = File a
defocus (Directory a lis,_) = Directory a lis
```