

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 hipotesis 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 calculo 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

```
descend :: Double -> Hypothesis Double -> [Sample Double]
        -> Hypothesis Double
descend alpha h ss = undefined
```

Sea  $\theta_j$  el  $j$ -ésimo componente del vector  $\theta$  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.

Su función debe ser escrita exclusivamente empleando funciones de orden superior. En particular, se trata de dos *fold* anidados, cada uno de ellos realizando sus cálculos en *una sola pasada*. Debe escribirla de manera general, suponiendo que **ss** podría tener una cantidad arbitraria de muestras disponibles y que  $j$  es arbitrario.

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.

```
gd :: Double -> Hypothesis Double -> [Sample Double]
    -> [(Integer, Hypothesis Double, Double)]
gd alpha h ss = undefined
```

Su función debe ser escrita como un *unfold*. Note que esta será la función “tope”, por lo tanto debe asegurarse de agregar los coeficientes 1 antes de comenzar a iterar, y mantener la iteración hasta que la diferencia entre los costos de dos hipótesis consecutivas sean  $\epsilon$ -despreciables.

### 1.3. ¿Cómo sé si lo estoy haciendo bien?

Probar las funciones `veryClose`, `add0nes` 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.

Con las definiciones en este archivo, si su algoritmo está implantado correctamente, hay convergencia. Para que tenga una idea

```
ghci> take 3 (gd alpha guess training)
[(0,Hypothesis {c = [0.0,0.0,0.0]} ,6.559154810645744e10),
 (1,Hypothesis {c = [10212.379787234042,3138.9880129854737,...
 (2,Hypothesis {c = [20118.388180851063,6159.113611965675,...]
```

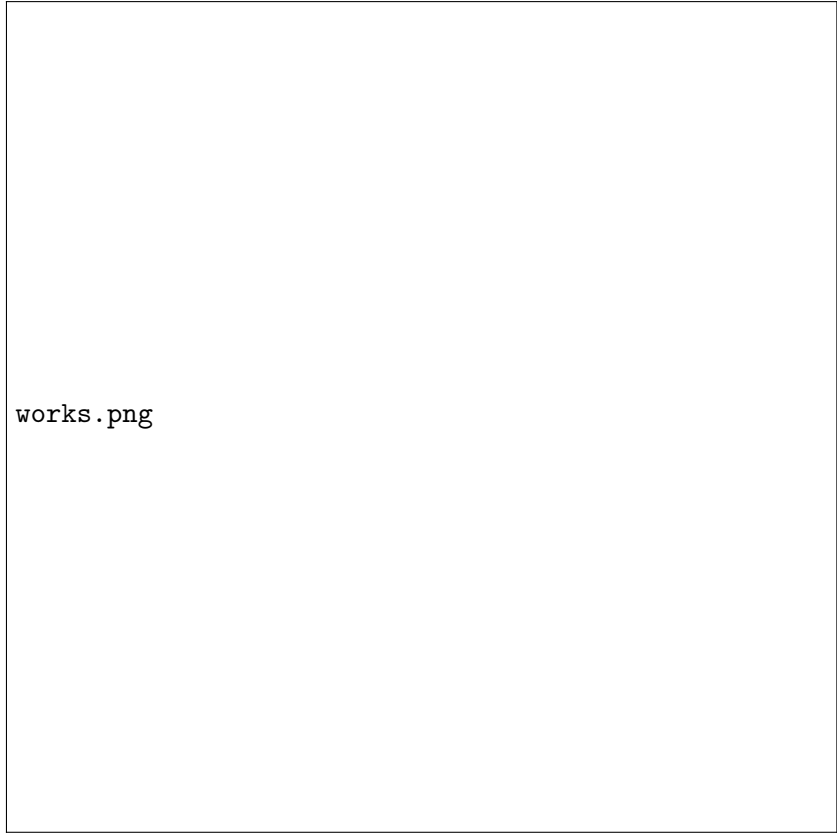
y si se deja correr hasta terminar converge (el *unfold termina*) y los resultados numéricos en la última tripleta deberían ser muy parecidos a (indentación mía)

```
(1072,
Hypothesis {c = [340412.65957446716,
                  110631.04133702737,
                  -6649.4653290010865]} ,
2.043280050602863e9)
```

Para su comodidad, he incluido la función `graph` de la magnífica librería `chart` que permite hacer gráficos sencillos (línea, torta, barra, etc.). Puede usarla para verificar que su función está haciendo el trabajo

```
ghci> graph "works" (gd alpha guess training)
```

y en el archivo `works.png` debe obtener una imagen similar a



works.png

#### 1.4. ¿Aprendió?

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
293081.85236
```

## 2. Monoids

Durante la discusión en clase acerca de **Monoid** se dejó claro que para algunos tipos de datos existe más de una instancia posible. En concreto, para los números puede construirse una instancia **Sum** usando  $(+)$  como operación y  $0$  como elemento neutro, pero también puede construirse una

instancia `Product` usando `(*)` como operación y `1` como elemento neutro. La solución al problema resultó ser el uso de tipos equivalentes pero incompatibles aprovechando `newtype`.

Siguiendo esa idea, 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
```

Note que en este caso `a` es el tipo comparable, y la primera función debe levantar el valor libre al `Monoid` calculador de máximos. Piense que el `Foldable t` *podría* estar vacío (lista, árbol, ...) así que el `Monoid` debe operar con “seguridad”

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]
```

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   ::
```