# CI4251 - Programación Funcional Avanzada
# Tarea 1

Stefani Castellanos
11-11394

Abril 27, 2016

# 1. Machine Learning

En este ejercicio, investigaremos la técnica de Regresión Lineal Multivariable con el método de *Gradient Descent*, aprovechando las ecuaciones normales. Implantaremos el algoritmo paso a paso en Haskell, aprovechando *folds* y *unfolds*.

No es necesaria experiencia previa con el Algoritmo, pues todos los detalles serán presentados a lo largo del problema. Sugiero que construya las funciones en el mismo orden en que se van proponiendo, pues van aumentando de complejidad.

## 1.1. Definiciones Generales

Para el desarrollo de la solución, serán necesarios los módulos [1]

```
import Data.List
import Data.Functor
import Data.Monoid
import Data.Foldable (foldMap)
import Data.Tree
import Data.Maybe (fromJust)
--import Graphics.Rendering.Chart.Easy
--import Graphics.Rendering.Chart.Backend.Cairo
```

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)
```

[1] Los dos últimos opcionales si quiere tener el gráfico que muestra la convergencia.

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 $\alpha$ 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 $\epsilon$ 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. Tampoco debe preocuparle mucho qué representan – al algoritmo no le importan y Ud. tampoco tiene que preocuparse por eso. [2]

## 1.3. Ahora le toca a Ud.

### 1.3.1. Comparar en punto flotante

No se puede y Ud. lo sabe. Pero necesitamos una manera de determinar si la diferencia entre dos números en punto flotante es $\epsilon-$despreciable

[2] En la práctica, las muestras suelen estar en diferentes escalas (precio en miles, unidades de Frobs, porcentajes, etc.) y uno de los trabajos previos es normalizar las medidas. Ya eso fue hecho por Ud. porque no es importante para esta materia.

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

Por favor **no** use un `if`...

La función veryClose se encarga de esta comparación, su implementación es directa y autoexplicativa.

### 1.3.2. Congruencia dimensional

Seguramente notó que los vectores con muestras tienen dimensión $n$, pero la hipótesis tiene dimensión $n + 1$. Eso es porque la hipótesis necesariamente debe agregar un coeficiente constante para la interpolación lineal. En consecuencia *todas* las muestras necesitan incorporar $x_0 = 1$.

```
addOnes :: [Sample Double] -> [Sample Double]
addOnes = map (\s -> Sample { x = 1.0 : (x s), y = y s })
```

Para añadir el 1.0 a cada muestra se utiliza map que aplica a cada elemento de la lista la función lambda que agrega al principio de la lista del campo s y conserva el resto.

Escriba la función `addOnes` usando exclusivamente funciones de orden superior y en estilo *point-free* (con argumento implícito, como puede ver).

### 1.3.3. Evaluando Hipótesis

Si tanto una hipótesis $\theta$ 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

```
theta :: Hypothesis Double -> Sample Double -> Double
theta h s = sum (zipWith (*) (c h) (x s))
```

Escriba la función `theta` usando exclusivamente funciones de orden superior. Puede suponer que ambos vectores tienen las dimensiones correctas.

El producto punto se calcula como la sumatoria de la multiplicación de cada elemento del vector hipótesis por el vector x en la muestra. Se implementa esta theta utilizando la función del preludio zipWith que, dada una operación binaria y dos listas, aplica dicha operación entre los elementos de las listas. Finalmente se realiza la sumatoria con la función sum.

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

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

```
cost :: Hypothesis Double -> [Sample Double] -> Double
cost h ss = su / (2 * n)
   where auxSum (n, su) s = (n + 1, ((theta h s) - (y s)) ** 2 + su)
         (n, su) = foldl' auxSum (0, 0) ss
```

Su función debe ser escrita como un *fold* que realice todos los cómputos en *una sola pasada* sobre el conjunto de muestras. Debe escribirla de manera general, suponiendo que `ss` podría tener una cantidad arbitraria de muestras disponibles.

Para implementar el costo en una sola pasada se realiza un foldl', dado que la lista debe ser finita que la sumatoria tenga sentido, y como acumulador una tupla cuya primera posición representa el tamaño de la lista y la segunda posición representa la sumatoria descrita en el párrafo anterior en cada paso. Finalmente se divide el resultado entre dos veces el tamaño de la muestra.

## 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 $\theta$ que tiene un costo $c$, determina la dirección en el espacio vectorial que maximiza el descenso, y produce una nueva hipótesis $\theta'$ 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" $\alpha$.

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.

La primera parte de este algoritmo, y sin duda la función más complicada de este ejercicio, es aquella que dada una hipótesis y un conjunto de entrenamiento, debe producir una nueva hipótesis mejorada según el coeficiente de aprendizaje.

```
descend :: Double -> Hypothesis Double -> [Sample Double]
        -> Hypothesis Double
descend alpha h ss =  Hypothesis $ (reverse . fst) (foldl' iterJ ([], 0) (c h))
    where iterJ (th', j) th = (th - (alpha/m)* s : th', j + 1)
                where (s, m) = iterI j ss
          iterI j = foldl' sumI (0, 0)
                where sumI (su, l) s = (su + ((theta h s) - (y s)) * ((x s)!! j) , l +
```

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 cómputos 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.

Se necesitan dos fold's, uno para recorrer $\theta_j$ y otro para recorrer los $x_i$. Para lograr esta función se pueden definir dos funciones auxiliares: la primera de ellas iterJ se encarga de las operaciones inherentes a iterar sobre las j, es decir, trabaja con el resultado de la sumatoria y realiza las operaciones pertinentes. Mientras que iterI se encarga de realizar la sumatoria con el $x_j$ apropiado y calcular el tamaño de la muestra para el cálculo sobre las j, evitando así el uso de length.

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 = unfoldr newH (h, 0)
   where newH (h1, i) = if veryClose (cost h1 ss') (cost h2 ss')
                        then Nothing
                        else Just ((i, h1, cost h1 ss'),(h2, i + 1))
                        where h2 = descend alpha h1 ss'
         ss' = addOnes ss
```

Para usar unfoldr se necesita una función que retorne Nothing cuando el costo sea despreciable y Just a mientras no lo sea. El tipo de esta a debe ser una tupla que coincida con el tipo de los elementos de la lista que se desea

retornar, en este caso (Integer, Hypothesis Double, Double). En la segunda posición el tipo del acumulador, un tupla cuya primera es la nueva hipotesis y la segunda el número de iteración en la que se generó.

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.

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

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.

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

### 1.6. ¿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 [] [])
```

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

instance Ord a => Monoid (Max a) where
    mempty = Max Nothing
    mappend (Max (Just a)) (Max (Just b))
        |a > b = Max (Just a)
        |otherwise = Max (Just b)
    mappend (Max (Just a)) mempty = (Max (Just a))
    mappend mempty (Max (Just b)) = (Max (Just b))
```

Se define un newtype que almacena un Maybe a para operar con seguridad sobre el tipo de dato polimórfico. Para construir la instancia de Monoid se debe proporcionar la función mempty que no es más que Nothing ya que este es el elemento neutro.

También se tiene que implementar el operador binario mappend que compara dos elementos que están en el monoide. Evidentemente el tipo de dato a debe ser comparable, lo que explica la restricción Ord a

## 3. Zippers

Considere el tipo de datos

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

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.

```
data FsCrumb a = Crumb a [Filesystem a] [Filesystem a]
               deriving (Show)
data Breadcrumbs a = C [FsCrumb a] deriving (Show)

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

Para determinar el tipo de dato del zipper se escribe el tipo de dato como una ecuación de tipos,en este caso Filesystem a = a + a x [Filesystem a] x [Filesystem a] y luego se debe derivar para obtener Filesystem a = a x [Filesystem a] x [Filesystem a] que representa el tipo de dato de FsCrumb, luego Breadcrumbs es una lista de este tipo de dato.

Se hace uso de Maybe para realizar operaciones de manera segura.

```
goDown :: Zipper a -> Maybe (Zipper a)
goDown (Directory a (f : hs), C bs) = Just (f, C (Crumb a [] hs : bs))
```

```
goDown _                                = Nothing
```

Bajar en el Filesystem solo tiene sentido para Directory y significa poner el foco en el primer elemento de la lista en él.

```
goLeft :: Zipper a -> Maybe (Zipper a)
goLeft (f, C ((Crumb a (fs : fss) hs) : bs)) = Just (fs, C (Crumb a fss (f : hs) : bs
goLeft _                                     = Nothing
```

También ir a la izquierda y a la derecha tiene sentido solo en Directory. Se interpreta como avanzar o retroceder entre los archivos del directorio actual. En la lista de la izquierda se tiene los archivos que han sido "vistosz en la derecha los que quedan por "ver". Según sea el caso se agrega en una lista u otra.

```
goRight :: Zipper a -> Maybe (Zipper a)
goRight (f, C ((Crumb a fss (fs : hs)) : bs)) = Just (fs, C (Crumb a (f : fss) hs : b
goRight _                                      = Nothing
```

Para regresar en un Directory es necesario reconstruir la lista con los archivos contenidos en él, para ello concatenamos las listas poniendo en medio el archivo en foco y ubicando el nuevo foco en el Directory que lo contenía. Note que la lista de la izquierda está al revés, he allí la razón para usar reverse.

```
goBack :: Zipper a -> Maybe (Zipper a)
goBack (f, C ((Crumb a fss hs) : bs)) = Just (Directory a ((reverse fss) ++ [f] ++ hs
goBack _                              = Nothing
```

tothetop es simplemente usar goBack de manera recursiva hasta llegar a la ráiz del Filesystem

```
tothetop :: Zipper a -> Zipper a
tothetop (f, C []) = (f, C [])
tothetop z = tothetop . fromJust $ goBack z
```

modify recibe una función que cambia el contenido del archivo en foco.

```
modify :: (a -> a) -> Zipper a -> Zipper a
modify g (Directory a fs, bs) = (Directory (g a) fs, bs)
modify g (File a, bs)         = (File (g a), bs)
```

Focus y defocus para entrar o salir del zipper

```
focus :: Filesystem a -> Zipper a
focus fs = (fs, C [])

defocus  :: Zipper a -> Filesystem a
defocus (fs, _) = fs
```