

CI4251 - Programación Funcional Avanzada

Tarea 1

Ernesto Hernández-Novich
86-17791
[<emhn@usb.ve>](mailto:emhn@usb.ve)

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 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 usar el método de regresión lineal multivariable, las muestras son *vectores* (x_1, x_2, \dots, 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)
```

¹Los 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 $(\theta_0, \theta_1, \dots, \theta_n)$ tal que minimiza el error de predicción $(\theta_0 + \theta_1 \times x_1 + \dots + \theta_n x_n - y)$ 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 margen 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. Tampoco debe preocuparle mucho qué representan – al algoritmo no le importan y Ud. tampoco tiene que preocuparse por eso. ²

²En 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.

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 ϵ —despreciable

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

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

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 addOne
          where addOne sam = Sample { x = 1:(x sam), y = y sam }
```

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 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

```
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)
```

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

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 = undefined
```

Su función debe ser escrita como un *fold* que realice todos los cálculos 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.

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 ϵ —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 = undefined
```

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.

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 ϵ —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)

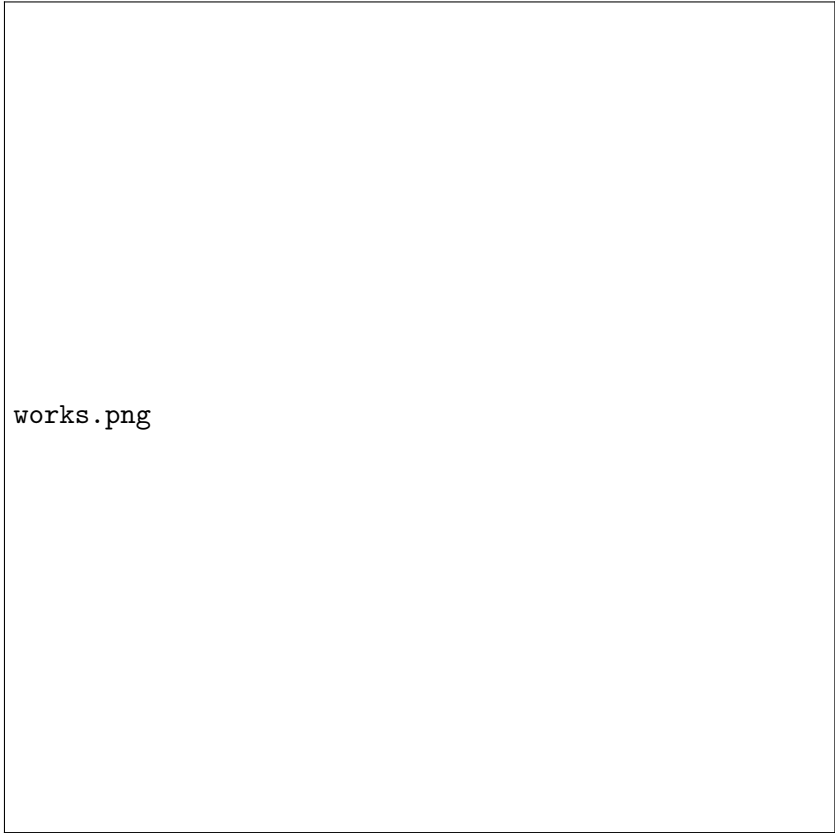
```
(1072,
Hypothesis {c = [340412.65957446716,
```

```
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



works.png

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 [] [])
```

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   ::
```