Trabajo Práctico 1
Programación Funcional

Reconociendo el paradigma reconociendo el paradigma.

Paradigmas de Lenguajes de Programación
1er cuatrimestre, 2016

Fecha de entrega: martes 12 de abril

1. Introducción

Aprendizaje automático es una rama de la computación especializada en lograr hacer programas que "aprendan" de datos a partir de reconocer patrones y lograr generalizar el conocimiento. El objetivo básico consiste en la construcción de modelos a partir de ejemplos que permitan aprender y luego hacer predicciones sobre nuevos datos.

Hay muchas categorías dentro del área: *aprendizaje supervisado*, *aprendizaje no supervisado*, *aprendizaje por refuerzos*, etc. Dentro de aprendizaje supervisado se encuentran las tareas de *regresión* (regresiones lineales por ejemplo) y *clasificación*. En nuestro trabajo, nos concentraremos en implementar una versión simplificada del proceso completo para la construcción de un clasificador (desde el tratamiento de datos hasta la evaluación de predicciones).

La tarea de clasificación consiste en construir un modelo que aprende de datos etiquetados y luego predice etiquetas sobre datos sin etiquetar.

A continuación, **un ejemplo** con el que **no** vamos a trabajar: Clasificar letras de canciones para determinar a qué género pertenece. En este caso, una forma de implementar el proceso sería:

- Conseguir muchas canciones (sus letras) acompañadas de etiquetas ('cumbia', 'merengue', 'pachanga', 'cha-cha-cha', etc).
- Extraer características (*features*) de este texto, como ser: longitud promedio de las palabras, cantidad de veces que se dice alguna palabra característica ('agite' por ejemplo), frecuencia de repetición de palabras/frases, etc.

- Construir un clasificador y hacer que "aprenda" a partir de una porción de estos datos junto con sus etiquetas
- Analizar predicciones hechas sobre el resto de los datos no utilizados para aprender
- En caso de no obtener buenos resultados, iterar el proceso cambiando los features utilizados o los parámetros del clasificador.

En este trabajo implementaremos un clasificador de texto (en este caso programas) para determinar si pertenecen a programas escritos en el paradigma **funcional** o **imperativo**. Es decir, contaremos con datos (programas) etiquetados en las clases: 'funcional' o 'imperativo' y ustedes deberán construir un clasificador que aprenda de estos datos y sirva para clasificar **nuevos** programas.

2. Implementación

A continuación se detalla el modelado del problema junto a los ejercicios.

2.1. Datos

Los datos con que contarán son programas provenientes de diversos problemas y lenguajes junto con etiquetas que indiquen a que paradigma pertenecen.

La forma de modelar estos programas será mediante el tipo `Texto` y sus etiquetas mediante el tipo `Etiqueta`

```
type Texto = String
type Etiqueta = String
```

Por ejemplo: `"a = 2; a++" :: Texto` y además `"Imperativo" :: Etiqueta`

2.2. Parte 1: Extracción de Atributos

La extracción de atributos (features) es un paso fundamental en Aprendizaje Automático que convierte instancias de datos crudos (raw data) en instancias de vectores numéricos.

En nuestro caso, los extractores, serán los encargados de convertir el código de los programas en números que representen algún atributo. Por ejemplo, dado el programa `let a = area (Rect 3 2) == 4` uno puede extraer diversos atributos como cantidad de números: 3, cantidad de palabras en mayúscula: 1, cantidad de aparición del símbolo '=': 3, etc y obtener de esta manera el vector $x = [3, 1, 3]$ (cada vector de atributos suele notarse con la letra x minúscula)

La forma de representar extractores será mediante un renombre de tipos:

```
type Extractor = (Texto -> Feature)
type Feature = Float
```

Esta sección tendrá como objetivo implementar algunos extractores de features[1]

Ejercicio 1

Implementar la función auxiliar `split :: Eq a => a -> [a] -> [[a]]` Dado un elemento separador y una lista, se deberá partir la lista en sublistas de acuerdo a la aparición del separador (sin incluirlo). Por ejemplo:

```
*Tp> split ',' "hola PLP, bienvenidos!"
["hola PLP"," bienvenidos!"]
```

[1] Cabe aclarar que el clasificador debería funcionar para otros lenguajes que no aparezcan en estos datos (es decir, extraer features como cantidadDeWhiles puede ser contraproducente).

Ejercicio 2

Implementar `longitudPromedioPalabras :: Extractor`, que dado un texto, calcula la longitud promedio de sus palabras. Consideraremos palabra a cualquier secuencia de caracteres separadas por espacios. por ejemplo,

```
*Tp> longitudPromedioPalabras "Este test tiene palabras $$++$$"
5.4
```

Para esta función recomendamos utilizar la función genericLength del módulo List de Haskell y la función `mean :: [Float] -> Float` provista por la cátedra.

Ejercicio 3

Implementar la función auxiliar: `cuentas :: Eq a => [a] -> [(Int, a)]` que dada una lista, deberá devolver la cantidad de veces que aparece cada elemento en la lista. Por ejemplo:

```
*Tp> cuentas ["x", "x", "y", "x", "z"]
[(3,"x"),(1,"y"),(1,"z")]
```

Ejercicio 4

Implementar `repeticionPromedio :: Extractor` que calcula la cantidad promedio de repeticiones por cada palabra

```
*Tp> repeticionesPromedio "lalala $$++$$ lalala lalala $$++$$"
2.5
```

Para esta función recomendamos utilizar la función `fromIntegral` que puede ser utilizada para convertir de `Int` a `Float`.

Ejercicio 5

Implementar `frecuenciasTokens :: [Extractor]` que devuelve un extractor por cada uno de los símbolos definidos en la constante `tokens :: [Char]`[2] con la frecuencia relativa de estos con respecto a todos los caracteres presentes en el programa.

```
*Tp> head tokens
'_'
*Tp> (head frecuenciaTokens) "use_snake_case !"
0.125
```

Ejercicio 6

Muchas veces, es necesario normalizar los datos obtenidos por los extractores para evitar problemas al clasificar. En este punto, implementaremos la función
`normalizarExtractor :: [Texto] -> Extractor -> Extractor` que dado un extractor, lo "modifica" de manera que el valor de los features se encuentre entre -1 y 1 para todos los datos con los que se dispone. Por ejemplo, suponiendo los textos t_1, t_2 y t_3 y un extractor que devuelve los valores -20.3, 1.0 y 10.5 respectivamente, debería ser modificado para que al aplicarlo nuevamente a esos textos, los valores sean -1.0, 0.04 y 0.51. [3]

[2] La expresión tokens está definida en el esqueleto del trabajo práctico

[3] No es necesario separar el caso en que los extractores ya se encuentren en el rango, asumir que no lo están

Ejercicio 7

Los tipos `Instancia` y `Datos` definido como:

```
type Instancia = [Feature]
type Datos = [Instancia]
```

Representan el tipo de los vectores de atributos y matriz de vectores respectivamente. Es decir, cada fila de la matriz de tipo `Datos` representa un vector de atributos (`Instancia`) y a su vez estos vectores son los obtenidos a partir de extraer distintos features de un programa.

Implementar la función `extraerFeatures :: [Extractor] -> [Texto] -> Datos` que permita aplicar varios extractores a todos los programas que se reciban como parámetro y de esta manera lograr obtener una matriz de atributos. Para ello, primero deberán **normalizar** utilizando los mismos programas pasados como parámetros, todos los extractores antes de ser aplicados. Por ejemplo,

```
*Tp> let ts = ["b=a", "a = 2; a = 4", "C:/DOS C:/DOS/RUN RUN/DOS/RUN"]
*Tp> extraerFeatures [longitudPromedioPalabras, repeticionesPromedio] ts
[[0.33333334,0.6666667],[0.12962963,1.0],[1.0,0.6666667]]
```

2.3. Parte 2: Construcción de un clasificador

Una vez obtenidos los vectores de atributos (o features), se procede a construir el clasificador. Un clasificador es un modelo encargado de predecir el valor de etiquetas de nuevas instancias del problema. Por ejemplo, dado un nuevo vector de features de un programa, predice a qué paradigma pertenece. Para que este modelo funcione, debe ser "entrenado" utilizando datos de entrenamiento (vectores de features junto a sus etiquetas).

Utilizaremos el tipo:

```
type Modelo = (Instancia -> Etiqueta)
```

para representar clasificadores. Un clasificador entonces, será de tipo `Modelo` y podrá ser utilizado como una función de predicción de etiquetas.

Ejercicio 8

Para el clasificador que construiremos en este trabajo, necesitaremos definir alguna medida de distancia entre vectores (o puntos) en la dimensión correspondiente:

Dados $P = (p_1, p_2, \ldots, p_n)$ *y* $Q = (q_1, q_2, \ldots, q_n)$ dos vectores, implementar las siguientes medidas de distancia:

1. `distEuclideana :: Medida` que calcula la *distancia Euclideana*:

$$d_E(P, Q) = \sqrt{\sum_{i=1}^{n} (p_i - q_i)^2}.$$

   ```
   *Tp> distEuclideana [1.0,0.75,0.8125] [0.75,1.0,0.5]
   0.47186464
   ```

2. `distCoseno :: Medida` que calcula la *distancia Coseno*:

$$d_C(P, Q) = \frac{P \cdot Q}{||P||.||Q||}$$

   donde

$$P \cdot Q = \sum_{i=1}^{n} P_i Q_i \qquad ||P|| := \sqrt{P \cdot P}$$

   La particularidad de esta distancia es que sólo toma en cuenta el ángulo entre los dos vectores (y no su módulo).

```
*Tp> distCoseno [0,3,4] [0,-3,-4]
-1.0
```

Ejercicio 9

El algoritmo de clasificación que implementaremos es un modelo simple: *K-Vecinos Más Cercanos* (https://es.wikipedia.org/wiki/K-vecinos_m%C3%A1s_cercanos). La versión que utilizaremos del modelo funciona como se explica a continuación:

- Ante una instancia que debe etiquetar, se calcula la distancia a todas las instancias de entrenamiento utilizando alguna medida de distancia definida previamente.
- Una vez computadas las distancias, se seleccionan las K instancias más cercanos y se obtiene su etiqueta.
- Ante esta lista de etiquetas, se calcula la moda estadística.
- Luego, se etiqueta a la nueva instancia con esa moda [4] [5]

Implementar un clasificador de K-Vecinos más cercanos. `knn :: Int -> Datos -> [Etiqueta] -> Medida -> Modelo`. El primer parámetro corresponde al número de vecinos K, luego recibe datos de entrenamiento junto a sus etiquetas y una medida de distancia, devuelve un modelo de predicción. Por ejemplo:

```
*Tp> (knn 2 [[0,1],[0,2],[2,1],[1,1],[2,3]] ["i","i","f","f","i"] distEuclideana) [1,1]
"f"
```

2.4. Parte 3: Evaluación de Resultados y Cross validation

Cross validation se refiere a la técnica de probar nuestro clasificador separando de distintas maneras nuestro conjunto de datos.

Ejercicio 10

Como primer paso, necesitamos separar nuestros datos en datos de entrenamiento y datos de validación, para ello utilizaremos la siguiente función:

```
separarDatos :: Datos -> [Etiqueta] -> Int -> Int -> (Datos, Datos, [Etiqueta], [Etiqueta])
```

Esta función, toma una matriz de datos xs y sus respectivas etiquetas y, luego dos números n y p, y devuelve el resultado de partir la matriz xs en n particiones dejando la partición número p para validación y el resto para entrenamiento. Es precondición que el número de partición estará entre 1 y n. Para este ejercicio, se debe **mantener el orden original** en estas particiones.

En caso de particionar los datos y que la división no sea exacta, se descartarán los casos sobrantes, por ejemplo:

```
*Tp> let xs = [[1,1],[2,2],[3,3],[4,4],[5,5],[6,6],[7,7]] :: Datos
*Tp> let y = ["1","2","3","4","5","6","7"]
*Tp> let (x_train, x_val, y_train, y_val) = separarDatos xs y 3 2

*Tp> (x_train, y_train)
([[1.0,1.0],[2.0,2.0],[5.0,5.0],[6.0,6.0]],["1","2","5","6"])
*Tp> (x_val, y_val)
([[3.0,3.0],[4.0,4.0]],["3","4"])
```

[4] En caso de utilizar un K par y encontrar más de una moda, cualquiera de las etiquetas mayoritarias será válida.

[5] En caso de haber puntos que se encuentren a la misma distancia, será equivalente tomar cualquiera de ellos. Lo importante es que no exista un punto más cercano a los K seleccionados.

Ejercicio 11

Además, necesitaremos alguna medida para evaluar nuestros resultados. Implementar la función `accuracy :: [Etiqueta] -> [Etiqueta] -> Float` que devuelve la proporción de aciertos de nuestras predicciones sobre las verdaderas etiquetas, por ejemplo:

```
*Tp> accuracy ["f", "f", "i", "i", "f"] ["i", "f", "i", "f", "f"]
0.6
```

Ejercicio 12

Implementaremos ahora la función
`nFoldCrossValidation :: Int -> Datos -> [Etiqueta] -> Float` que calculará el accuracy promedio de nuestro modelo en datos no etiquetados. Para ello, utilizaremos una versión de n-fold cross-validation[6] en donde la matriz de datos (y sus respectivas etiquetas) son particionadas en N particiones (siendo N el primer parámetro de nuestra función).

Una vez particionados los datos, seleccionaremos $N - 1$ particiones para entrenamiento y la restante para validar nuestro modelo. Para este ejercicio el modelo será: vecinos más cercanos con K = 15 y la distancia Euclideana como medida.

Repetimos este proceso variando cuál es la partición seleccionada para validación entre las N posibles y de esta manera obtenemos N resultados intermedios (accuracy para cada conjunto de validación). El resultado será el de promediar estos N resultados intermedios.

2.5. Parte 4: Cómo probar

Una vez implementados los puntos anteriores, pueden compilar el archivo run.hs utilizando ghc: `ghc run.hs` y luego correr el programa `./run`, allí verán los resultados de su clasificador. Este clasificador tomará instancias de las subcarpetas "./imperativo/" y "./funcional/" que ustedes pueden descargar de http://www.dc.uba.ar/materias/plp/cursos/2016/cuat1/descargas/tps/tp-funcional-archivos/tp-funcional-programas y extraer en la misma carpeta que los archivos `.hs`.

Es posible que se produzca un error si no tienen instalado el paquete `Random`, para instalarlo, recomendamos utilizar el instalador de paquetes cabal: `> cabal install random`

La salida esperada de este programa es:

```
> ghc run & ./run
[1 of 2] Compiling Tp         ( Tp.hs, Tp.o )
"Funcional: 2000 instancias"
"Imperativo: 2000 instancias"
"Accuracy promedio: 0.8255"
"random value: 0.5"
```

2.6. Parte 5: ¿No es suficiente?

Ejercicio 13 (opcional)

Implementar alguna de las siguientes ideas:

- Más extractores de features que piensen que pueden ayudar a la clasificación.
- Implementar algún otro clasificador (regresión logística, decision tree, redes neuronales, etc)
- Probar los clasificadores para otro dominio (por ejemplo, letras de canciones)

Si deciden implementar más extractores o más clasificadores, haremos pruebas finales con datos que ustedes no poseen pero de las características de los provistos.

[6] Normalmente este método se llama K-fold cross-validation pero optamos por utilizar N para no generar colisión de nombres con el clasificador

Pautas de Entrega

Se debe entregar el código impreso con la implementación de las funciones pedidas. Cada función debe contar con un comentario donde se explique su funcionamiento. Cada función asociada a los ejercicios debe contar con ejemplos que muestren que exhibe la funcionalidad solicitada. Además, se debe enviar un e-mail conteniendo el código fuente en Haskell a la dirección plp-docentes@dc.uba.ar. Dicho mail debe cumplir con el siguiente formato:

- El título debe ser `[PLP;TP-PF]` seguido inmediatamente del nombre del grupo.
- El código Haskell debe acompañar el e-mail y lo debe hacer en forma de archivo adjunto (puede adjuntarse un `.zip` o `.tar.gz`).
- El código entregado **debe** incluir tests que permitan probar las funciones definidas.

El código debe poder ser ejecutado en Haskell2010. No es necesario entregar un informe sobre el trabajo, alcanza con que el código esté **adecuadamente** comentado (son comentarios adecuados los que ayudan a entender lo que no es evidente o explican decisiones tomadas; no son adecuadas las traducciones al castellano del código). Los objetivos a evaluar son:

- Corrección.
- Declaratividad.
- Prolijidad: evitar repetir código innecesariamente y usar adecuadamente las funciones previamente definidas (tener en cuenta tanto las funciones definidas en el enunciado como las definidas por ustedes mismos).
- Uso adecuado de funciones de alto orden, currificación y esquemas de recursión: Es necesario para los ejercicios que usen las funciones que vimos en clase y aprovecharlas, por ejemplo, usar zip, map, filter, take, takeWhile, dropWhile, foldr, foldl, listas por comprensión, etc, cuando sea necesario y no volver a implementarlas.

Salvo donde se indique lo contrario, **no se permite utilizar recursión explícita**, dado que la idea del TP es aprender a aprovechar las características enumeradas en el ítem anterior. Se permite utilizar listas por comprensión y esquemas de recursión definidos en el preludio de Haskell y los módulos `Prelude`, `List`, `Maybe`, `Data.Char`, `Data.Function`, `Data.List`, `Data.Maybe`, `Data.Ord` y `Data.Tuple`. Las sugerencias de los ejercicios pueden ayudar, pero no es obligatorio seguirlas. Pueden escribirse todas las funciones auxiliares que se requieran, pero estas no pueden usar recursión explícita (ni mutua, ni simulada con `fix`).

Importante: se admitirá un único envío, sin excepción alguna. Por favor planifiquen el trabajo para llegar a tiempo con la entrega.

Tests: se recomienda la codificación de tests. Tanto HUnit `https://hackage.haskell.org/package/HUnit` como HSpec `https://hackage.haskell.org/package/hspec` permiten hacerlo con facilidad.

Para instalar HUnit usar: `> cabal install hunit`

Para instalar cabal ver: `https://wiki.haskell.org/Cabal-Install`

Referencias del lenguaje Haskell

Como principales referencias del lenguaje de programación Haskell, mencionaremos:

- **The Haskell 2010 Language Report:** el reporte oficial de la última versión del lenguaje Haskell a la fecha, disponible online en `http://www.haskell.org/onlinereport/haskell2010`.

- **Learn You a Haskell for Great Good!**: libro accesible, para todas las edades, cubriendo todos los aspectos del lenguaje, notoriamente ilustrado, disponible online en http://learnyouahaskell.com/chapters.
- **Real World Haskell**: libro apuntado a zanjar la brecha de aplicación de Haskell, enfocándose principalmente en la utilización de estructuras de datos funcionales en la "vida real", disponible online en http://book.realworldhaskell.org/read.
- **Hoogle**: buscador que acepta tanto nombres de funciones y módulos, como signaturas y tipos *parciales*, online en http://www.haskell.org/hoogle.
- **Hayoo!**: buscador de módulos no estándar (i.e. aquéllos no necesariamente incluídos con la plataforma Haskell, sino a través de **Hackage**), online en http://holumbus.fh-wedel.de/hayoo/hayoo.html.