DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico I

Programación Funcional

12 de abril de 2016 — Paradigmas de Lenguajes de Programación

Grupo Trinidad y estos vagos

Integrante	LU	Correo electrónico
Palladino, Julián Alberto	336/13	julianpalladino@hotmail.com
Roulet, Nicolas	587/13	nicoroulet@gmail.com
Schmit, Matias	714/11	matias.schmit@gmail.com

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

1. Tp.hs

```haskell
module Tp where

import Data.List
import Data.Maybe
import Data.Char -- ord

type Texto = String
type Feature = Float
type Instancia = [Feature]
type Extractor = (Texto -> Feature)

type Datos = [Instancia]
type Etiqueta = String
type Modelo = (Instancia -> Etiqueta)
type Medida = (Instancia -> Instancia -> Float)

tryClassifier :: [Texto] -> [Etiqueta] -> Float
tryClassifier x y = let xs = extraerFeatures ([longitudPromedioPalabras, repeticionesPromedio,
      palabrasClaves,frecuenciaParesConsecutivos] ++ frecuenciaTokens ++ frecuenciaClaves) x in
    nFoldCrossValidation 5 xs y
-- ++ frecuenciaClaves

mean :: [Float] -> Float
mean xs = realToFrac (sum xs) / genericLength xs

split :: Eq a => a -> [a] -> [[a]]
split d xs = filter(not . null)(foldr (\x r -> if (x == d) then []:r else (x:(head r)):(tail r)) [[]] xs)

longitudPromedioPalabras :: Extractor
longitudPromedioPalabras t = mean ( map (genericLength) (split ' ' t) )

cuentas :: Eq a => [a] -> [(Int, a)]
cuentas xs = [(elemCount x xs, x)| x <- nub(xs)]

elemCount :: Eq a => a -> [a] -> Int
elemCount a = foldr (\x r -> if (x==a) then r+1 else r ) 0

repeticionesPromedio :: Extractor
repeticionesPromedio t = mean (map (fromIntegral . fst) (cuentas (split ' ' t) ) )

tokens :: [Char]
tokens = "_,)(*;-=>/.{}\"&:+#[]<|%!\'@?~^$` abcdefghijklmnopqrstuvwxyz0123456789"

frecuenciaTokens :: [Extractor]
frecuenciaTokens = [(\text -> freqRel text t) | t <- tokens]

freqRel :: Texto -> Char -> Feature
freqRel t c = fromIntegral(elemCount c t ) / (genericLength t)

normalizarExtractor :: [Texto] -> Extractor -> Extractor
normalizarExtractor ts e = let maximo = (foldr (max . abs . e) 0 ts ) in (\t -> (e t) / maximo)

extraerFeatures :: [Extractor] -> [Texto] -> Datos
extraerFeatures fs txts = let es = [normalizarExtractor txts e | e <- fs] in
                                                    [[e t | e <- es ] | t <- txts ]
```

```haskell
dameValores :: [Extractor] -> [Texto] -> Texto -> Instancia
dameValores fs txts txt = [ (normalizarExtractor txts elem) txt | elem <- fs]

distEuclideana :: Medida
distEuclideana p q = sqrt (sum (zipWith (\a b -> (a-b)**2) p q ) )

distCoseno :: Medida
distCoseno p q = ( sum (zipWith (\a b -> (a*b)) p q ))  / ( (normaVector p) * (normaVector q) )

normaVector :: [Feature] -> Float
normaVector = sqrt . (foldr (\x r -> x*x + r) 0 ) -- otra version = (sqrt . sum . map (**2))

knn :: Int -> Datos -> [Etiqueta] -> Medida -> Modelo
knn k dss es m x = palabraMasRepetida (map snd (take k (ordenarVecinos dss es m x)))

-- devuelve una lista de tuplas (distancia, etiqueta)
ordenarVecinos :: Datos -> [Etiqueta] -> Medida -> Instancia -> [(Float, Etiqueta)]
 ordenadas por distancia
ordenarVecinos dss es m x = sort (zip (map (m x) dss) es)

palabraMasRepetida :: (Ord a) => [a] -> a
palabraMasRepetida = snd . (foldr1 max) . cuentas

accuracy :: [Etiqueta] -> [Etiqueta] -> Float
accuracy xs ys = mean ( zipWith (\ a b -> if a==b then 1 else 0) xs ys )

separarDatos :: Datos -> [Etiqueta] -> Int -> Int -> (Datos, Datos, [Etiqueta], [Etiqueta])
separarDatos ds es n p = (tomarTrain ds train1 train2 val, tomarVal ds val train1,
              tomarTrain es train1 train2 val, tomarVal es val train1)
 where l = tamanioParticion ds n         -- l es el tamanio de las particiones
       train1 = l * (p-1)
       val = l                -- val es el tamanio de la particion de validacion
       train2 = (n - p) * l
-- train1 es la cantidad de elementos que tiene en total el primer conjunto de particiones de entrenamien
-- train2 es la cantidad de elementos que tiene en total el segundo conjunto de particiones de entrenamie


-- tomarTrain, dado una lista de t1+d+t2 elementos, remueve los "d" del medio
-- La usamos para remover la particion de validacion, dejando solamente las de entrenamiento
tomarTrain :: [a] -> Int -> Int -> Int -> [a]
tomarTrain xs t1 t2 d = (take t1 xs) ++ (take t2 (drop (t1 + d) xs))

tomarVal :: [a] -> Int -> Int -> [a]
tomarVal xs t d = take t (drop d xs)

tamanioParticion :: Datos -> Int -> Int
tamanioParticion ds n = div (length ds) n

nFoldCrossValidation :: Int -> Datos -> [Etiqueta] -> Float
nFoldCrossValidation n ds es = mean [validateAccuracy (separarDatos ds es n p ) | p <- [1..n]]

-- toma una particion y calcula el accuracy de los datos de validacion contra los de training
validateAccuracy :: (Datos, Datos, [Etiqueta], [Etiqueta]) -> Float
validateAccuracy (td, vd, te, ve) = accuracy ve (validate td te vd)

-- valida cada elemento de la particion de validación contra las de training con knn
validate :: Datos -> [Etiqueta] -> Datos -> [Etiqueta]
validate ts es vs = map (knn 15 ts es distEuclideana) vs
```

```haskell
--Ejercicio opcional: Otros extractores

diccionarioClaves :: [(Texto,Int)]          --DiccClaves = clavesImperativo ++ clavesFuncional
diccionarioClaves = [(";",30), ("++",10), ("--",40), ("namespace",100), ("public",50), ("module",40),
("void",40), ("while",40), ("self",30),("class",80), ("for",42), ("cout",80), ("endl",80), ("static",100),
(">>>",60), ("//",40), ("argv",50), ("catch",10),
("try",30) ] ++
[("-->",-100), ("::",-80), ("!!",-80), ("type",-50), ("Eq",-80), ("Show",-100), ("Data.",-100),
("where",-70),("fold",-200), ("let",-60),
("define",-80),("defn",-100),("Nothing",-70),
("Just",-70), ("Maybe",-70)]


--Extractor que calcula un puntaje por programa dependiendo las palabras "claves" que aparecen
palabrasClaves :: Extractor
palabrasClaves = (\text -> fromIntegral (puntajeCadena text diccionarioClaves) )

-- Puntaje total del texto comparado con todas las tuplas
puntajeCadena :: Texto -> [(Texto,Int)] -> Int
puntajeCadena t = foldr f 0
        where f = (\(x,y) r -> if (isInfixOf x t ) then r +
    ((aparicionesEnIntersec x (intersect t x ) ) * y) else r )

-- Cantidad de apariciones de un texto en la intersección entre el y una clave
aparicionesEnIntersec :: Texto -> Texto -> Int
aparicionesEnIntersec t qs = length [ True | c <- tails qs, isPrefixOf t c ]

--Otro, similar al de frecuenciasTokens
frecuenciaClaves :: [Extractor]
frecuenciaClaves = [(\text -> freqCl text t) | t <- diccionarioClaves]

freqCl :: Texto -> (Texto,Int) -> Feature
freqCl t c = fromIntegral(puntajeCadena t [c] ) / (genericLength t)

-- frecuencia de pares de palabras consecutivos

frecuenciaParesConsecutivos :: Extractor
frecuenciaParesConsecutivos t = let l = (split ' ' t) in
            mean (map (fromIntegral . fst) (cuentas (zip l (tail l) ) ) )

autosplit :: [Char] -> [[Char]]
autosplit s = foldr (\ x rec -> (breakAlphaNum x) ++ rec) [[]] (split ' ' s)

breakAlphaNum :: [Char] -> [[Char]]
breakAlphaNum [] = [[]]
breakAlphaNum [c] = [[c]]
breakAlphaNum (c1:c2:cs) = let rec = (breakAlphaNum (c2:cs)) in
        if (isAlphaNum c1) == (isAlphaNum c2) then
                (c1:(head rec)):(tail rec)
        else
                ([c1]:rec)
```