Práctica 1 Inteligencia Artificial

Lucía Asencio y Juan Riera

Febrero 2018

1 Ejercicio 1: Similitud Coseno

1.1

PSEUDOCÓDIGO

Entrada: x, y: vectores, representados como listas
Salida: similitud coseno entre los vectores
Procesamiento:
Se calculan los productos escalares de los operandos de la siguiente manera:
– Caso recursivo:
1) Si cualquiera de los dos vectores es NIL se devuelve 0
2) Si no, se devuelve el resultado de multiplicar los primeros elementos de los dos vectores y sumarlo con el siguiente nivel de recursión.
– Caso de mapcar:
1) Se aplica mapcar con la multiplicación entre los vectores 2) Se aplica reduce a todo el vector con la operacion dela suma Por último calculamos el coseno del ángulo con la fórmula

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; prodEscRec (x y)
;;; Calcula el producto escalar de dos vectores de forma recursiva.
;;;
;;; INPUT: x: vector, representado como una lista
;;; y: vector, representado como una lista
;;;
;;; OUTPUT: producto escalar
;;;
(defun prodEscRec (x y)
    (if (or (null x) (null y))
        0
      (+ (* (first x) (first y)) (prodEscRec (cdr x) (cdr y)))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; sc-rec (x y)
;;; Calcula la similitud coseno de un vector de forma recursiva
;;;
;;; Se asume que los dos vectores de entrada tienen la misma longitud.
;;; La semejanza coseno entre dos vectores que son listas vacias o que son
;;; (0 0...0) es NIL.
;;; INPUT: x: vector, representado como una lista
;;; y: vector, representado como una lista
;;;
;;; OUTPUT: similitud coseno entre x e y
;;;
(defun sc-rec (x y)
```

```
   ;;Falta comprobar la division por 0
    (/ (prodEscRec x y) (sqrt (* (prodEscRec x x) (prodEscRec y y)))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; prodEscMapcar (x y)
;;; Calcula el producto escalar de dos vectores usando mapcar
;;;
;;; INPUT: x: vector, representado como una lista
;;; y: vector, representado como una lista
;;;
;;; OUTPUT: producto escalar
;;;

(defun prodEscMapcar (x y)
    (reduce #'+ (mapcar #'* x y)))



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; sc-mapcar (x y)
;;; Calcula la similitud coseno de un vector usando mapcar
;;;
;;; Se asume que los dos vectores de entrada tienen la misma longitud.
;;; La semejanza coseno entre dos vectores que son listas vacias o que son
;;; (0 0...0) es NIL.
;;; INPUT: x: vector, representado como una lista
;;; y: vector, representado como una lista
;;;
;;; OUTPUT: similitud coseno entre x e y
;;;


(defun sc-mapcar (x y)
   ;; Falta comprobar dvision por 0
    (/ (prodEscMapcar x y) (sqrt (* (prodEscMapcar x x) (prodEscMapcar y y)))))
```

COMENTARIOS

- Comentario

1.2

PSEUDOCÓDIGO

Entrada: Nivel de confianza, vector de categorías y vector de vectores
Salida: Vector de vectores cuyasimilitud con la categoria supera el nivel de confianza ordenados
Procesamiento: 1) Se ordenan los vectores con sort 2) Se seleccionan los que superan el umbral

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; select (conf cat vs)
;;; Selecciona, de entre unos vectores ya ordenados (vs)
;;; los que superan el umbral de similaridad
;;;
;;; INPUT: conf: Nivel de confianza
;;; cat: vector que representa a una categoria, representado como una lista
;;; vs: vector de vectores
;;; OUTPUT: Vectores cuya similitud con respecto a la categoria es superior al
;;; nivel de confianza, ordenados

(defun select (conf cat vs)
   (if (>= (sc-rec (car vs) cat) conf)
      vs
      (select conf cat (cdr vs))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; sc-conf (cat vs conf)
;;; Devuelve aquellos vectores similares a una categoria
;;; INPUT: cat: vector que representa a una categoria, representado como una lista
;;; vs: vector de vectores
;;; conf: Nivel de confianza
;;; OUTPUT: Vectores cuya similitud con respecto a la categoria es superior al
;;; nivel de confianza, ordenados



(defun sc-conf (cat vs conf)
   (select conf cat (sort (copy-list vs)
      #'(lambda (x y) (< (sc-rec x cat) (sc-rec y cat))))))
```

COMENTARIOS

- Comentario

1.3

PSEUDOCÓDIGO

Entrada: Vector de categorías, vector de textos y funcion con la que calcular la similaridad.

Salida: pares compuestos por el identificador de la categoría y el resultado de similitud coseno

Procesamiento:

Para cada texto se busca la categoría de la siguiente manera: 1) Se forma la tupla con el identificador de la categoria y el resultado de la funcion coseno. 2) Se continúa la recursión, iterando sobre todas las categorías. 3) Si se llega al final (NIL) se devuelve una tupla con valor coseno inferior al mínimo dela función, de forma que siempre será superada a menos que sea la última, en cuyo caso esta tupla será un indicativo de error. 4) Tras la recursión la funcion devuelve la tupla con "second" máximo, el suyo propio, o el devuelto por la recursión. De esta manera el devuelto al final es el máximo de todos.

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; maxcadr (x y)
;;; Compara los segundos elementos de los vectores, y devuelve el
;;; que tenga el maximo.
;;;
;;; INPUT: x, y: los vectores a comparar
;;; OUTPUT: vector con el mayor "second"
;;;

(defun maxcadr (x y)
   (if(> (second x) (second y)) x y))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; classify (cats text func)
;;; Halla la categoria que se ajusta mas al texto text.
;;;
;;; INPUT: cats: vector de categorias
;;; text: texto para el que se quiere averiguar categoria
;;; func: funcion con la que calcular la similaridad
;;; OUTPUT: par formado por la categoria que mas se ajusta
;;; al texto y por el nivel de similaridad categoria-texto
;;;

(defun classify (cats text func)
   (if (null cats) '(-2 0)
      (maxcadr (list (caar cats) (funcall func (cdr text) (cdar cats)))
           (classify (cdr cats) text func))))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; sc-classifier (cats texts func)
;; Clasifica a los textos en categorias.
;;;
;;; INPUT: cats: vector de vectores, representado como una lista de listas
;;; texts: vector de vectores, representado como una lista de listas
;;; func: funcion para evaluar la similitud coseno
;;; OUTPUT: Pares identificador de categoria con resultado de similitud coseno
;;;

(defun sc-classifier (cats texts func)
   (if (null texts) nil
      (cons (classify cats (car texts) func)
         (sc-classifier cats (cdr texts) func))))
```

COMENTARIOS

- Tras haber medido tiempos de ejecución hemos obtenido los siguientes resultados:
  - Ambos algoritmos llegan al mismo resultado correcto.
  - La versión recursiva del algoritmo es del orden de 6 veces mas rápida que la versión de mapcar.

2 Ejercicio 2: Raíces de una función

2.1

PSEUDOCÓDIGO

```
Entrada:  f, función y [a, b] intervalo en que buscar raíces
Salida:  raíz encontrada
Procesamiento:
Si f(a)f(b) > 0, no se cumplen hipótesis
Si f(a)=0, a es raiz
Si f(b)=0, b es raiz
Si (b-a) < tol, punto-medio es raiz
Si f(a)f(puntomedio) < 0, busca raiz en [a, puntomedio]
En otro caso, busca raiz en [ puntomedio, b ]
```

CÓDIGO

```
;; Finds a root of f between the points a and b using bisection.
;;
;; If f(a)f(b)>=0 there is no guarantee that there will be a root in the
;; interval, and the function will return NIL.
;; INPUT:
;; f: function of a single real parameter with real values whose root we want to find
;; a: lower extremum of the interval in which we search for the root
```

```
;; b: b>a upper extremum of the interval in which we search for the root
;; tol: tolerance for the stopping criterion: if b-a < tol the function returns
;; (a+b)/2 as a solution.
;; OUTPUT: Root of the function

(defun bisect (f a b tol)
   (let ((medio (/ (+ a b) 2)))
       (cond ((< 0 (* (funcall f a) (funcall f b))) NIL )
             ((= 0 (funcall f a)) a)
             ((= 0 (funcall f b)) b)
             ((> tol (- b a)) medio)
             ((>= 0 (* (funcall f a) (funcall f medio))) (bisect f a medio tol))
             ((>= 0 (* (funcall f b) (funcall f medio))) (bisect f medio b tol)))))
;; Ejemplos
;; (bisect #'(lambda(x) (sin (* 6.26 x))) 0.1 0.7 0.001) --> 0.5016602
;; (bisect #'(lambda(x) (sin (* 6.26 x))) 0.0 0.7 0.001) --> 0.0
```

COMENTARIOS

- En un ejemplo del enunciado, en caso de que la raíz fuera uno de los extremos del intervalo, no se detectaba como raíz. Nos pareció más lógico encontrar las raíces en [a, b] y no es (a, b) y así lo hemos implementado.

2.2

PSEUDOCÓDIGO

Entrada: f, función; l, lista ordenada, tol para la condición de parada del algoritmo
Salida: lista con raíces encontradas
Procesamiento:
Si la lista no esta vacía,
1) Busca raíz entre los 2 1ºs eltos de la lista
2) Evalúa en el resto de la lista
3) Concatena resultados de 1) y 2)

CÓDIGO

```
;; Funcion auxiliar
;; Igual que la funcion allroot, pero devuelve una lista que contiene las raices
;; intercaladas con NILs
;;
;;
;; INPUT:
;; f: function of a single real parameter with real values whose root
;; we want to find
;; lst: ordered list of real values (lst[i] < lst[i+1])
;; tol: tolerance for the stopping criterion: if b-a < tol the function
;; returns (a+b)/2 as a solution.
```

```
;;
;; Whenever sgn(f(lst[i])) != sgn(f(lst[i+1])) this function looks for a
;; root in the corresponding interval.
;;
;; OUTPUT:
;; A list o real values containing the roots of the function in the
;; given sub-intervals

(defun nils-n-root (f lst tol)
   (unless (null (rest lst))
      (cons (bisect f (first lst) (first (rest lst)) tol) (nils-n-root f (rest lst) tol ))))

;; Finds all the roots that are located between consecutive values of a list
;; of values
;;
;; INPUT:
;; f: function of a single real parameter with real values whose root
;; we want to find
;; lst: ordered list of real values (lst[i] < lst[i+1])
;; tol: tolerance for the stopping criterion: if b-a < tol the function
;; returns (a+b)/2 as a solution.
;;
;; Whenever sgn(f(lst[i])) != sgn(f(lst[i+1])) this function looks for a
;; root in the corresponding interval.
;;
;; OUTPUT:
;; A list o real values containing the roots of the function in the
;; given sub-intervals

(defun allroot (f lst tol)
   (remove nil (nils-n-root f lst tol)))

;; Ejemplos
;; (allroot #'(lambda(x) (sin (* 6.28 x))) '(0.25 0.75 1.25 1.75 2.25) 0.0001)
;; --> (0.50027466 1.0005188 1.5007629 2.001007)
;; (allroot #'(lambda(x) (sin (* 6.28 x))) '(0.25 0.9 0.75 1.25 1.75 2.25) 0.0001)
;; --> (0.5002166 1.0005188 1.5007629 2.001007)
;; (allroot #'(lambda(x) (sin (* 6.28 x))) '() 0.0001) --> NIL
;;  (allroot #'(lambda(x) (sin (* 6.28 x))) '(0.1 0.2 0.3) 0.0001) --> NIL
```

COMENTARIOS

- La lista que genera la recursión pura contiene NILs en las posiciones correspondientes a intervalos donde no se encuentran raíces. Por esta razón hacemos uso de dos funciones: la función principal hace de envoltorio y elimina estos NILs, y la funcion auxiliar se encarga de la recursión.

2.3

PSEUDOCÓDIGO

Entrada: f función, [a, b] intervalo donde buscar raíces, N 2^N es el tamaño def la partición, tol define la condición de parada del algoritmo
Salida: lista con las raíces encontradas
Procesamiento:
Actual = a,
Mientras actual <= b,
1) Busca raices en [actual, actual + tamintervalo]
2) Evalua en actual + tamintervalo
3) Concatena resultado de 1) y 2)

CÓDIGO

```
;; Funcion auxiliar
;; Igual que la funcion allind, pero devuelve una lista que contiene las raices
;; intercaladas con NILs
;;
;; INPUT:
;; f: function of a single real parameter with real values whose root we want to find
;; a: lower extremum of the interval in which we search for the root
;; b: b>a upper extremum of the interval in which we search for the root
;; N: Exponent of the number of intervals in which [a,b] is to be divided:
;; [a,b] is divided into 2^N intervals
;; tol: tolerance for the stopping criterion: if b-a < tol the function
;; returns (a+b)/2 as a solution.
;;
;; The interval (a,b) is divided in intervals (x[i], x[i+i]) with
;; x[i]= a + i*dlt; a root is sought in each interval, and all the roots
;; thus found are assembled into a list that is returned.
;;
;; OUTPUT: List with all the found roots.

(defun introot (f incr fin actual tol)
  (unless (<= fin actual)
    (cons (bisect f actual (+ actual incr) tol)
          (introot f incr fin (+ actual incr) tol))))


;; Divides an interval up to a specified length and find all the roots of
;; the function f in the intervals thus obtained.
;; INPUT:
;; f: function of a single real parameter with real values whose root we want to find
;; a: lower extremum of the interval in which we search for the root
;; b: b>a upper extremum of the interval in which we search for the root
;; N: Exponent of the number of intervals in which [a,b] is to be divided:
;; [a,b] is divided into 2^N intervals
;; tol: tolerance for the stopping criterion: if b-a < tol the function
```

```
;; returns (a+b)/2 as a solution.
;;
;; The interval (a,b) is divided in intervals (x[i], x[i+i]) with
;; x[i]= a + i*dlt; a root is sought in each interval, and all the roots
;; thus found are assembled into a list that is returned.
;;
;; OUTPUT: List with all the found roots.

(defun allind (f a b N tol)
   (remove nil (introot f (/ (- b a) (expt 2 N)) b a tol)))

;; Ejemplos
;; (allind #'(lambda(x) (sin (* 6.28 x))) 0.1 2.25 1 0.0001) --> NIL
;; (allind #'(lambda(x) (sin (* 6.28 x))) 0.1 2.25 2 0.0001)
;; --> (0.50027096 1.000503 1.5007349 2.0010324)
;; (allind #'(lambda(x) (sin x)) 0 16 4 0.0001)
;; --> (0 102943/32768 205887/32768 308831/32768 411775/32768 514719/32768)
```

COMENTARIOS

- Hemos encontrado un error en la función que no nos da tiempo a resolver. Como la función encuentra raíces en intervalos de la forma $[a, a+i], [a+i, a+2i], ... [a+ki, a+(k+1)i]$... en caso de encontrarse una raíz en un extremo de intervalo, ésta aparece repetida en el resultado. Podemos resolver esto eliminando repetidos o, de manera más eficiente y menos correcta (perderíamos una raíz en caso de ser f(b) = 0) modificar la función bisect para que encontrara raíces en $[a+ki, a+(k+1)i)$

3 Ejercicio 3: Combinación de listas

3.1

PSEUDOCÓDIGO

```
Entrada:  elt elemento que combinar con lst lista
Salida:  lista con la combinacion
Procesamiento:
Si elt != null and lst != null,
 1) Combina elt con first(lst)
 2) Evalua en resto de lst
 3) Concatena resultados de 1) y 2)
```

CÓDIGO

```
;; Combina un elto dado con todos los eltos de una lista
;;
;; INPUT:
;; elt: elemento a combinar con la lista
;; lst: llista quue sera combinada con el elemnto
```

```
;;
;; OUTPUT: lista que contenga la combinacion

(defun combine-elt-lst (elt lst)
   (unless (or (null lst) (null elt))
      (cons (list elt (first lst))
           (combine-elt-lst elt (rest lst)) )))

;; Ejemplos
;; (combine-elt-lst nil nil) --> NIL
;; (combine-elt-lst 'a nil) --> NIL
;; (combine-elt-lst 'a '(1 2 3) ) --> ((A 1) (A 2) (A 3))
;; (combine-elt-lst 'a '(1) ) --> ((A 1))
```

COMENTARIOS

- (Ningún comentario hasta ahora)

3.2

PSEUDOCÓDIGO

Entrada: lst1, lst2 listas a combinar
Salida: lista con producto cartesiano
Procesamiento:
Si lst1 != null and lst2 != null,
 1) Combina first(lst1) con lst2
 2) Evalua con resto de lst1 y lst2
 3) Concatena resultados de 1) y 2)

CÓDIGO

```
;; Funcion que devuelve el producto cartesiano de dos listas
;;
;; INPUT:
;; lst1, lst2 : listas a combinar
;;
;; OUTPUT: lista que contenga la combinacion

(defun combine-lst-lst (lst1 lst2)
   (unless (or (null lst1) (null lst2))
      (append (combine-elt-lst (first lst1) lst2)
           (combine-lst-lst (rest lst1) lst2))))

;; (combine-lst-lst nil nil ) --> NIL
;; (combine-lst-lst '(a b c) nil ) --> NIL
;; (combine-lst-lst '(a b c) '(1 2) )
;; ((A 1) (A 2) (B 1) (B 2) (C 1) (C 2))
```

```
;; (combine-lst-lst nil '(1 2) ) --> NIL
```

COMENTARIOS

- (Ningún comentario hasta ahora)

3.3

PSEUDOCÓDIGO

Entrada: lstolsts
Salida: lista que contenga todas las combinaciones
Procesamiento: 1) Si la lista contiene una única lista devuelve una lista de listas con los elementos separados. 2) Se avanza recursivamente si hay mas listas hasta la penultima lista, se haceel producto cartesiano de las dos ultimas listas y se devuelve. 3) Se combina el car de la lista actual en otro caso, con la lista de listas devuetas por la recursión.

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;; EJERCICIO 3.3 ;;;;;;;;;;;;;;;;;;

;; Funcion que calcula todas las posibles disposiciones de elementos
;; pertenecientes a N listas de forma que en cada disposicion aparezca unicamente
;; un elemento de cada lista
;;
;; INPUT:
;; lstolsts : listas a combinar
;;
;; OUTPUT: lista que contenga todas las combinaciones


(defun cortar (lst)
   (unless (null lst)
      (append (list (list(car lst))) (cortar (cdr lst)))))


(defun append-elt-lst (elt lst)
   (if (null lst) (list (list elt))
      (if (null (cdr lst))
         (list (append (list elt) (first lst)))
         (cons (append (list elt) (first lst))
           (append-elt-lst elt (rest lst)) ))))


(defun combine-list-lsts (lst lsts)
   (unless (or (null lst) (null lsts))
      (append (append-elt-lst (car lst) lsts) (combine-list-lsts (cdr lst) lsts))))
```

```
(defun combine-list-of-lsts (lstolsts)
   (print lstolsts)
   (if (null (cdr lstolsts))
      (cortar (car lstolsts))
      (if (null (cddr lstolsts))
         ;; Las combina
         (combine-lst-lst (car lstolsts) (cadr lstolsts))
         ;; Si no, la combina con el resultado de la recursion
         (combine-list-lsts (car lstolsts) (combine-list-of-lsts (cdr lstolsts))))))
```

COMENTARIOS

- Comentario