**MEMORIA PRÁCTICA 1**

Victoria Pelayo e Ignacio Rabuñal.

-**Ejercicio1**

**1.1** Para este ejercicio, a parte de la función pedida por el enunciado, hemos creado dos auxiliares. Una que realiza el producto escalar entre dos vectores y otra que aplica la formula, una vez que ya se tienen los productos escalares.

**Pseudocódigo:**

**Scalar-product**

Entrada: x (vector), y (vector)

Salida: res (producto escalar de x e y)

Proceso:

Para cada i <= dimensión(x)

Res[i] = x[i] + y[i]

**Formula coseno distancia:**

Entrada: xx(norma vector x), yy(norma vector y), xy (producto escalar de x e y)

Salida: res (coseno-distancia entre x e y)

Proceso:

Res = 1 – (xy / (xx \* yy) )

**Cosine-distance-rec**

Entrada: x(vector), y(vector)

Salida: res (coseno distancia entre x e y)

Proceso:

Si (norma(x) = 0 o norma(y) = 0):

Res = NIL

Si no:

Res = Formula-coseno-distancia (procuto-escalar(x,x), producto-escalar(y,y) , producto-escalar(x,y))

**Scalar-product-mapcar**

Entrada: x(vector), y(vector)

Salida: res (producto escalar de x e y)

Proceso:

Lista [i] = x[i] \* y[i]

Para cada i <= dimension(lista)

Res = res + lista[i]

**Cosine-distance-mapcar**

Entrada: x(vector), y(vector)

Salida: res(coseno-distancia entre x e y)

Proceso:

Si x = 0 o y = 0:

Res = NIL

Si no:

Res = formula-coseno-distancia (producto-escalar-mapcar(x,x) , producto-escalar-mapcar(y,y), producto-escalar-mapcar(x,y))

**Código**

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; scalar-product (x y)

;;; Calcula el producto escalar de dos vectores.

;;; Se asume que los dos vectores de entrada tienen la misma longitud.

;;;

;;; INPUT: x: vector, representado como una lista

;;; y: vector, representado como una lista

;;; OUTPUT: producto escalar entre x e y

;;;

(defun scalar-product (x y)

(if (or(null x)(null y)) 0

(+(\* (car x) (car y))

(scalar-product (cdr x) (cdr y)))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; formula-cos-dis

;;; Aplica la formula de la distancia coseno

;;;

;;; INPUT: xx: norma del vector x

;;; yy: norma del vector y

;;; xy: producto escalar de x e y

;;; OUTPUT: coseno distancua entre x e y

;;;

(defun formula-cos-dis (xx yy xy)

(- 1 (/ xy

(\* (sqrt xx)

(sqrt yy)))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; cosine-distance-rec (x y)

;;; Calcula la distancia coseno de un vector de forma recursiva

;;; Se asume que los dos vectores de entrada tienen la misma longitud.

;;; Si uno de los dos es el vector 0, se devuelve NIL

;;;

;;; INPUT: x: vector, representado como una lista

;;; y: vector, representado como una lista

;;; OUTPUT: distancia coseno entre x e y

;;;

(defun cosine-distance-rec (x y)

(let ((xy (scalar-product x y)) (xx (scalar-product x x)) (yy (scalar-product y y)))

(if (or (= 0 xx) (= 0 yy))

nil

(formula-cos-dis xx yy xy))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; scalar-product-mapcar (x y)

;;; Calcula el producto escalar de dos vectores.

;;; En vez de recursivamente, utilizando "mapcar"

;;; Se asume que los dos vectores de entrada tienen la misma longitud.

;;;

;;; INPUT: x: vector, representado como una lista

;;; y: vector, representado como una lista

;;; OUTPUT: producto escalar entre x e y

;;;

(defun scalar-product-mapcar (x y)

(if (or (null x) (null y)) 0

(apply #'+ (mapcar #'\* x y))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; cosine-distance-mapcar

;;; Calcula la distancia coseno de un vector usando mapcar

;;; Se asume que los dos vectores de entrada tienen la misma longitud.

;;;

;;; INPUT: x: vector, representado como una lista

;;; y: vector, representado como una lista

;;; OUTPUT: distancia coseno entre x e y

;;;

(defun cosine-distance-mapcar (x y)

(let((xy (scalar-product-mapcar x y)) (xx (scalar-product-mapcar x x)) (yy (scalar-product-mapcar y y)))

(if (or (= 0 xx) (= 0 yy))

nil

(formula-cos-dis xx yy xy))))

**Ejemplos de ejecución**, con la función recursiva (hemos comprobado con una calculadora que el resultado es el correcto):

* (cosine-distance-rec ‘(1 2) ‘(1 2 3)) 🡪 0,40238577
* (cosine-distance-rec nil ‘(1 2 3)) 🡪 nil
* (cosine-distance-rec ’() ’()) 🡪 nil
* (cosine-distance-rec ‘(0 0) ‘(0 0)) 🡪 nil

Ejemplos de ejecución, con la función que utiliza mapcar:

* (cosine-distance-mapcar ‘(1 2) ‘(1 2 3)) 🡪 0,40238577
* (cosine-distance-mapcar nil ‘(1 2 3)) 🡪 nil
* (cosine-distance-mapcar ’() ’()) 🡪 nil
* (cosine-distance-mapcar ‘(0 0) ‘(0 0)) 🡪 nil

**1.2**  Para este apartado hemos creado una función auxiliar que devuelve la lista de tuplas de la confianza y el vector correspondiente.

Como no se especifica nada en el enunciado, para calcular la distancia coseno utilizamos la función implementada con mapcar.

Ejemplos de ejecución:

* (order-vectors-cosine-distance '(1 2 3) '((32 454 123) (133 12 1) (4 2 2)) 0.99) NIL
* (order-vectors-cosine-distance '(1 2 3) '((32 454 123) (133 12 1) (4 2 2)) 0.3) ((4 2 2) (32 454 123) (133 12 1))
* (order-vectors-cosine-distance '(1 2 3) '((32 454 123) (133 12 1) (4 2 2)) 0.5) ((4 2 2) (32 454 123)) NIL
* (order-vectors-cosine-distance '(1 2 3) '()) NIL
* (order-vectors-cosine-distance '() '((4 3 2) (1 2 3))) NIL

**1.3 y 1.4.**

Para esta última función hemos creado dos funciones auxiliares. Una para calcular la distancia coseno entre cada una de las categorías y un vector; y otra que elige la mejor categoría para un vector.

Pruebas de ejecución:

* (get-vectors-category '(()) '(()) #'cosine-distance-rec) NIL
* (get-vectors-category '(()) '(()) #'cosine-distance-mapcar) NIL
* (get-vectors-category '((1 4 2) (2 1 2)) '((1 1 2 3)) #'cosine-distance-mapcar) ((2 0.40238577))
* (get-vectors-category '((1 4 2) (2 1 2)) '((1 1 2 3)) #'cosine-distance-rec) ((2 0.40238577))
* (get-vectors-category '(()) '((1 1 2 3) (2 4 5 6)) #'cosine-distance-rec) ((NIL NIL) (NIL NIL))
* (get-vectors-category '(()) '((1 1 2 3) (2 4 5 6)) #'cosine-distance-mapcar) ((NIL NIL) (NIL NIL))
* (setf categories '((1 43 23 12) (2 33 54 24))) ((1 43 23 12) (2 33 54 24))

(setf texts '((1 3 22 134) (2 43 26 58))) ((1 3 22 134) (2 43 26 58))

(get-vectors-category categories texts #'cosine-distance-rec) ((2 0.5101813) (1 0.18444914))

(get-vectors-category categories texts #'cosine-distance-mapcar) ((2 0.5101813) (1 0.18444914))

**EJERCICIO3.**

**3.1**

Para este apartado no hemos necesitado crear ninguna función auxiliar.

Ejemplos de ejecución:

* (combine-elt-lst 'a '(1 2 3)) 🡪 ((A 1) (A 2) (A 3))
* (combine-elt-lst nil nil) 🡪 NIL
* (combine-elt-lst nil ’(a b)) 🡪 ((NIL A) (NIL B))

**3.2**

En este ejercicio teníamos que crear una función que hiciese el producto cartesiano de dos listas.

Ejemplo de ejecución:

* (combine-lst-lst nil nil) 🡪 NIL
* (combine-lst-lst ’(a b c) nil) 🡪 NIL
* (combine-lst-lst nil ’(a b c)) 🡪 NIL

Si una de las dos listas es NIL el producto cartesiano es NIL.

**3.3**

En este apartado hay que crear una función que combine todos los elementos de varias listas dadas.

Para ello hemos creado una función auxiliar que es una modificación de la del ejercicio1, esta modificación evita los problemas de paréntesis.

Y otra función auxiliar que combina todos los elementos de dos listas.

Ejemplos de ejecución:

* (combine-list-of-lsts ’(() (+ -) (1 2 3 4))) 🡪 ((+ 1) (+ 2) (+ 3) (+ 4) (- 1) (- 2) (- 3) (- 4))

(Si una lista es nil, contamos como si no estuviese)

* (combine-list-of-lsts ’((a b c) () (1 2 3 4))) 🡪 ((A 1) (A 2) (A 3) (A 4) (B 1) (B 2) (B 3) (B 4) (C 1) (C 2) (C 3) (C 4))
* (combine-list-of-lsts ’((a b c) (1 2 3 4) ())) 🡪 ((A . 1) (A . 2) (A . 3) (A . 4) (B . 1) (B . 2) (B . 3) (B . 4) (C . 1) (C . 2) (C . 3) (C . 4))
* (combine-list-of-lsts ’((1 2 3 4))) 🡪 ((1) (2) (3) (4))
* (combine-list-of-lsts ’(nil)) 🡪 NIL