# Memoria Práctica 1

## Ejercicio 1:

Apartado 1:

Para este ejercicio hemos creado una función auxiliar que calcula el producto escalar de dos vectores:

*(defun producto-escalar-rec (x y)*

*(if (or (null x) (null y))*

*0*

*(+ (\* (first x) (first y)) (producto-escalar-rec (rest x) (rest y))))*

*)*

Y luego una función que haciendo uso de esa función calcule la distancia coseno entre dos vectores de manera recursiva:

*(defun cosine-distance-rec (x y)*

*(if (= 0 (\* (producto-escalar-rec x x) (producto-escalar-rec y y)))*

*nil*

*(cosine-distance (producto-escalar-rec x y) (\* (sqrt (producto-escalar-rec x x)) (sqrt (producto-escalar-rec y y)))))*

*)*

Haciendo uso de una formula que aplica la fórmula dada llamada cosine distance:

*(defun cosine-distance (x y)*

*(- 1 (/ x y))*

*)*

Luego haciendo uso de las funciones mapcar hemos reducido la función de la distancia coseno:

*(defun cosine-distance-mapcar (x y)*

*(if (= 0 (\* (apply #'+ (mapcar #'\* x x)) (apply #'+ (mapcar #'\* y y))))*

*nil*

*(cosine-distance (apply #'+ (mapcar #'\* x y)) (\* (sqrt (apply #'+ (mapcar #'\* x x))) (sqrt (apply #'+ (mapcar #'\* y y))))))*

*)*

Los resultados a las pruebas pedidas en el ejercicio 1.1 son:

1. (cosine-distance ’(1 2) ’(1 2 3)) → 0.40238577

2. (cosine-distance nil ’(1 2 3)) → 0

3. (cosine-distance ’() ’()) → 0

4. (cosine-distance ’(0 0) ’(0 0)) → 0

### Apartado 2:

En este apartado se pide crear una función que ordene los vectores de la lista según el valor de confianza con el vector de referencia, siempre y cuando superen el valor de confianza dado como argumento de entrada:

Para ello hemos creado dos funciones auxiliares: la primera para insertar en orden en una lista:

*(defun insert-ordered-cosine-distance (reference vector lst-of-vectors)*

*(cond ((null vector)*

*lst-of-vectors)*

*((null lst-of-vectors)*

*(cons vector lst-of-vectors))*

*((< (cosine-distance-mapcar reference vector) (cosine-distance-mapcar reference (first lst-of-vectors)))*

*(cons vector lst-of-vectors))*

*(t*

*(cons (first lst-of-vectors) (insert-ordered-cosine-distance reference vector (rest lst-of-vectors)))))*

*)*

Esta función recibe una referencia (el vector respecto del cual se ordena la lista) un vector y la lista para insertarlo, si el vector es nulo devuelve la lista como tal, si lo nulo es la lista devuelve el vector como único elemento de la lista, en caso de que el vector sea más cercano que la lista se inserta el primero y en caso contrario se pasa al siguiente por recursión.

Y la siguiente como parte recursiva de la función principal:

*(defun order-vectors-cosine-distance-rec (vector lst-of-vectors &optional (confidence-level 0))*

*(cond ((null lst-of-vectors)*

*nil)*

*(t*

*(insert-ordered-cosine-distance vector (first lst-of-vectors) (order-vectors-cosine-distance-rec vector (rest lst-of-vectors) confidence-level))))*

*)*

Esta función envía a la anterior de manera recursiva todos los elementos de la lista para que los vaya ordenando.

Por lo que al final la función principal se dedicaría solamente a llamar a la recursiva y ésta a la de insertar en orden:

*(defun order-vectors-cosine-distance (vector lst-of-vectors &optional (confidence-level 0))*

*(order-vectors-cosine-distance-rec vector*

*(remove-if (lambda (v) (< (- 1 confidence-level) (cosine-distance-mapcar vector v))) lst-of-vectors)*

*confidence-level)*

*)*

Esta función una vez recibe la lista ordenada por distancia elimina los que no cumplen el nivel de confianza.

La salida a los ejercicios puestos es:

(order-vectors-cosine-distance ’(1 2 3) ’()) → NIL

(order-vectors-cosine-distance ’() ’((4 3 2) (1 2 3))) → ((1 2 3) (4 3 2))

### Apartado 3:

En este ejercicio se pide que dada una lista de textos y de categorías se le asigne a cada texto la categoría más cercana según la función de distancia asignada, para ello hemos creado dos funciones, la primera que dado un texto y una lista de categorías saque la que le corresponde al texto y la otra que dada una lista de textos y otra de categorías llame repetidamente a la primera para que saque las categorías.

1. **Primera función:**

*(defun get-category (categories text distance-measure minimum)*

*(cond ((null categories)*

*(list (first (first minimum)) (second minimum)))*

*(t*

*(if (< (funcall distance-measure text (first categories)) (second minimum))*

*(get-category (rest categories) text distance-measure (list (first categories) (funcall distance-measure text (first categories))))*

*(get-category (rest categories) text distance-measure minimum))))*

*)*

1. **Segunda función:**

*(defun get-vectors-category (categories texts distance-measure)*

*(cond ((null texts)*

*nil)*

*(t*

*(cons (get-category (rest categories) (first texts) distance-measure (list (first categories) (funcall distance-measure (first texts) (first categories)))) (get-vectors-category categories (rest texts) distance-measure))))*

*)*

### Apartado 4:

(get-vectors-category ’(()) ’(()) #’cosine-distance-mapcar) → **(NIL 0)**

(get-vectors-category ’((1 4 2) (2 1 2)) ’((1 1 2 3)) #’cosine-distance-mapcar) → (**2 0.39753592)**

(get-vectors-category ’(()) ’((1 1 2 3) (2 4 5 6)) #’cosine-distance-mapcar) **→ ((NIL 0) (NIL 0))**

# Ejercicio 2:

### Apartado 1:

*(defun newton (f df max-iter x0 &optional (tol 0.001))*

*(cond ((= max-iter 0)*

*nil)*

*((< (abs (funcall f x0)) tol)*

*x0)*

*(t*

*(newton f df (- max-iter 1) (- x0 (/ (funcall f x0) (funcall df x0))) tol)))*

*)*

### Apartado 2:

*(defun one-root-newton (f df max-iter semillas &optional (tol 0.001))*

*(cond ((null semillas)*

*nil)*

*((null (newton f df max-iter (first semillas) tol))*

*(one-root-newton f df max-iter (rest semillas) tol))*

*(t*

*(newton f df max-iter (first semillas) tol)))*

*)*

### Apartado 3:

1. **Función 1:**

*(defun all-roots-newton (f df max-iter semillas &optional (tol 0.001))*

*(cond ((null semillas)*

*nil)*

*(t*

*(cons (newton f df max-iter (first semillas) tol) (all-roots-newton f df max-iter (rest semillas) tol))))*

*)*

1. **Función 2:**

*(defun list-not-nil-roots-newton (f df max-iter semillas &optional ( tol 0.001))*

*(mapcan #'(lambda (x) (if (null x) nil (list x))) (all-roots-newton f df max-iter semillas tol))*

*)*

## Ejercicio 3:

### Apartado 1:

*(defun combine-elt-lst (elt lst)*

*(cond ((or (null lst) (null elt))*

*nil)*

*(t*

*(mapcar #'(lambda (x) (list elt x)) lst)))*

*)*

La salida para los casos propuestos es:

1. (combine-elt-lst ’a nil) **→** **NIL**

2. (combine-elt-lst nil nil) **→** **NIL**

3. (combine-elt-lst nil ’(a b)) **→ NIL**

### Apartado 2:

*(defun combine-lst-lst (lst1 lst2)*

*(mapcan #'(lambda (x) (combine-elt-lst x lst2)) lst1)*

*)*

La salida para los casos propuestos es:

1. (combine-lst-lst nil nil) **→** **NIL**

2. (combine-lst-lst ’(a b c) nil) **→** **NIL**

3. (combine-lst-lst nil ’(a b c)) **→** **NIL**

### Apartado 3:

1. **Función 1:**

*(defun combine-list-of-lsts (lstolsts)*

*(if (null lstolsts)*

*(list nil)*

*(combine-lst-lst-cons (first lstolsts) (combine-list-of-lsts (rest lstolsts))))*

*)*

1. **Función 2:**

*(defun combine-elt-lst-cons (elt lst)*

*(cond ((or (null lst) (null elt))*

*nil)*

*(t*

*(mapcar #'(lambda (x) (cons elt x)) lst)))*

*)*

1. **Función 3:**

*(defun combine-lst-lst-cons (lst1 lst2)*

*(mapcan #'(lambda (x) (combine-elt-lst-cons x lst2)) lst1)*

*)*

La salida para los casos propuestos es:

1. (combine-list-of-lsts ’(() (+ -) (1 2 3 4))) **→** **NIL**

2. (combine-list-of-lsts ’((a b c) () (1 2 3 4))) **→NIL**

3. (combine-list-of-lsts ’((a b c) (1 2 3 4) ())) **→** **NIL**

4. (combine-list-of-lsts ’((1 2 3 4))) **→** **((1) (2) (3) (4))**

5. (combine-list-of-lsts ’(nil)) **→** **NIL**

6. (combine-list-of-lsts nil) **→** (**NIL)**

Ejercicio 5:

5.2 Pseudocódigo del bfs:

*BFS (G, s) //Where G is the graph and s is the source node*

*let Q be queue*

*Q.enqueue( s ) //Inserting s in queue until all its neighbour vertices are marked*

*mark s as visited.*

*while ( Q is not empty)*

*//Removing that vertex from queue,whose neighbour will be visited now*

*v = Q.dequeue( )*

*//processing all the neighbours of v*

*for all neighbours w of v in Graph G*

*if w is not visited*

*Q.enqueue( w ) //Stores w in Q to further visit its neighbor*

*mark w as visited.*