

Memoria práctica 1

Rafael Sánchez, Miguel Baquedano

7 de marzo de 2019

Revisión del 7 de marzo de 2019 a las 21:48.

Índice general

I	Ejercicios	5
1.	Distancia coseno	7
1.1.	Implementación distancia coseno	7
1.1.1.	Pseudocódigo	7
1.1.2.	Comentarios	7
1.1.3.	Pruebas	7
1.2.	Ordenación según distancia coseno	7
1.2.1.	Pseudocódigo	7
1.2.2.	Comentarios	8
1.2.3.	Pruebas	8
1.3.	Clasificador por distancia coseno	8
1.3.1.	Pseudocódigo	8
1.3.2.	Comentarios	8
1.3.3.	Pruebas	8
2.	Raíces de una función	9
2.1.	Implementación Newton-Raphson	9
2.1.1.	Pseudocódigo	9
2.1.2.	Comentarios	9
2.1.3.	Pruebas	9
2.2.	Una raíz para una lista de semillas	9
2.2.1.	Pseudocódigo	9
2.2.2.	Comentarios	10
2.2.3.	Pruebas	10
2.3.	Todas las raíces para una lista de semillas	10
2.3.1.	Pseudocódigo	10
2.3.2.	Comentarios	10
2.3.3.	Pruebas	10
3.	Combinación de listas	11
3.1.	Combinación elemento - lista	11
3.1.1.	Pseudocódigo	11
3.1.2.	Comentarios	11
3.1.3.	Pruebas	11
3.2.	Combinación lista-lista	11
3.2.1.	Pseudocódigo	11
3.2.2.	Comentarios	11
3.2.3.	Pruebas	12
3.3.	Producto cartesiano n-ario	12
3.3.1.	Pseudocódigo	12
3.3.2.	Comentarios	12
3.3.3.	Pruebas	12
4.	Árboles de verdad	13
4.1.	Funciones de derivación	13
4.1.1.	Pseudocódigo	13

4.1.2. Comentarios	13
4.1.3. Pruebas	13
4.2. Construcción del árbol de verdad	13
4.2.1. Pseudocódigo	13
4.2.2. Comentarios	13
4.2.3. Pruebas	13
5. Búsqueda en anchura	15
5.1. Ilustración del algoritmo	15
5.1.1. Grafo especial	15
5.1.2. Caso típico	15
5.1.3. Caso típico	16
5.2. Pseudocódigo	16
5.3. BFS de ANSI Common Lisp	16
5.4. Comentarios al código de BFS en ANSI Common Lisp	16
5.5. BFS camino más corto	16
5.6. Evaluación de <code>shortest-path</code>	17
5.7. Bucle infinito en grafo con ciclos	17
5.8. Corrección del código	17
II Código	19
6. Distancia coseno	21
6.1. Implementación distancia coseno	21
6.2. ordenación según distancia coseno	22
6.3. Clasificador por distancia coseno	23
7. Raíces de una función	25
7.1. Implementación Newton-Raphson	25
7.2. Una raíz para una lista de semillas	25
7.3. Todas las raíces para una lista de semillas	26
8. Combinación de listas	27
8.1. Combinación elemento - lista	27
8.2. Combinación lista-lista	28
8.3. Producto cartesiano n-ario	28
9. Árboles de verdad	29
9.1. Funciones de derivación	29
9.2. construcción del árbol de verdad	30
10. Búsqueda en anchura	33
10.1. Comentarios al código de BFS en ANSI Common Lisp	33
10.2. Corrección del código	33

Parte I

Ejercicios

Capítulo 1

Distancia coseno

1.1. Implementación distancia coseno

1.1.1. Pseudocódigo

Dados dos vectores $u, v \in \mathbb{R}^n$, se define la distancia coseno entre ellos como:

$$1 - \frac{\langle u, v \rangle}{\|u\|_2 \|v\|_2}$$

Es claro que nuestro código tendrá la estructura

```
dist-cos (u, v) :  
    return ( 1 - scalar-prod(x, y) ) / (norm(x) * norm(y)) )
```

Por lo que dividiremos nuestra función en diversas subrutinas.

1.1.2. Comentarios

Tanto la implementación recursiva como la iterativa de la distancia coseno considera que la distancia coseno entre $u \in \mathbb{R}^m$ y $v \in \mathbb{R}^n$ con $m < n$ es la de la extensión del vector u a \mathbb{R}^n con sus últimas coordenadas nulas.

1.1.3. Pruebas

1. (cosine-distance '(1 2) '(1 2 3)) --> 0.40238565
2. (cosine-distance nil '(1 2 3)) --> NIL
3. (cosine-distance '() '()) --> NIL
4. (cosine-distance-rec '(0 0) '(0 0)) --> NIL

1.2. Ordenación según distancia coseno

1.2.1. Pseudocódigo

El pseudocódigo de nuestra rutina es el siguiente:

```
sort-cos-dist(vec, list, threshold) :  
    si semejanza(list[0], vec) > threshold:  
        inserta-en-orden(list[0], sort-cos-dist(vec, list++, threshold))  
    si no:  
        sort-cos-dist(vec, list++, threshold)
```

Donde list++ hace referencia al resto de la lista.

1.2.2. Comentarios

En la implementación hemos usado distintas funciones auxiliares para cumplir con la descomposición funcional que sigue el pseudocódigo.

Además, la función `inserta-en-orden` la implementamos de forma general permitiendo pasar por parámetro la función de comparación.

1.2.3. Pruebas

1. `(order-vectors-cosine-distance '(1 2 3) '()) --> NIL`
2. `(order-vectors-cosine-distance '() '((4 3 2) (1 2 3))) --> NIL`
3. `(order-vectors-cosine-distance '(1 2 3) '((32 454 123) (133 12 1) (4 2 2)) 0.5) --> ((4 2 2) (32 454 123))`

1.3. Clasificador por distancia coseno

1.3.1. Pseudocódigo

```
clasifica(categorias, textos, fdistancia) :
    return concatena(mejor-cat(categorias, textos[0], fdistancia),
                     clasifica(categorias, textos++, fdistancia))
```

Donde `mejor-cat` devuelve una tupla (categoría óptima, distancia a ella) para cada texto, y `concatena` añade un objeto (tupla en este caso), a una lista de objetos.

1.3.2. Comentarios

La rutina se desarrolló con una función auxiliar que construye el par (categoría, distancia). La complejidad temporal del algoritmo es $O(m \cdot n)$ con $m = \#categorias$, $n = \#textos$. Se podría mejorar con memoria auxiliar.

1.3.3. Pruebas

1. `(time (get-vectors-category categories texts #'cosine-distance-rec))`

```
Real time: 0.0010296 sec.
Run time: 0.0 sec.
Space: 4624 Bytes
((2 0.5101813) (1 0.18444908))
```

2. `(time (get-vectors-category categories texts #'cosine-distance-mapcar))`

```
Real time: 9.973E-4 sec.
Run time: 0.0 sec.
Space: 11344 Bytes
((2 0.5101813) (1 0.18444908))
```

3. `(get-vectors-category '() '() #'cosine-distance-rec) --> NIL`
4. `(get-vectors-category '((1 4 2) (2 1 2)) '((1 1 2 3)) #'cosine-distance-rec) --> ((2 0.40238565))`
5. `(get-vectors-category '() '((1 1 2 3) (2 4 5 6)) #'cosine-distance-rec) --> NIL`

Como era de esperar, la implementación iterativa tarda menos en la ejecución puesto que no tiene que deshacer la recursión. Sin embargo, puede verse como `mapcar` hace mayor uso de memoria que la pila de llamadas recursivas.

Capítulo 2

Raíces de una función

2.1. Implementación Newton-Raphson

2.1.1. Pseudocódigo

```
Newton-Raphson (f, df, max-iter, x0, tol) :  
  hasta que f(x0) o df(x0) = 0:  
    si |h| < tol :  
      devolver x0  
    si no:  
      Newton-Raphson(f, df, --max-iter, x0-h, tol)
```

Por lo que dividiremos nuestra función en diversas subrutinas. En este caso, $h : \mathbb{R} \rightarrow \mathbb{R}$ tal que $h(x) = \frac{f(x)}{df(x)}$.

2.1.2. Comentarios

Podría mejorarse considerando tolerancia en altura.

2.1.3. Pruebas

1. (newton #'(lambda(x) (* (- x 4) (- x 1) (+ x 3))) #'(lambda (x) (- (* x (- (* x 3) 4)) 11)) 20 3.0) --> 4.000084
2. (newton #'(lambda(x) (* (- x 4) (- x 1) (+ x 3))) #'(lambda (x) (- (* x (- (* x 3) 4)) 11)) 10 100.0) --> NIL

2.2. Una raíz para una lista de semillas

2.2.1. Pseudocódigo

```
one-root(f, df, max-iter, semillas, tol):  
  si (solucion <- Newton-Raphson(f, df, max-iter, semilla[0], tol):  
    devolver solución  
  si no:  
    one-root(f, df, max-iter, semillas++, tol)
```

Donde `semillas++` hace referencia al resto de la lista.

2.2.2. Comentarios

Simplemente encapsula el hacer varias pruebas con **Newton-Raphson** hasta que se encuentra la primera solución.

2.2.3. Pruebas

1. `(one-root-newton #'(lambda(x) (* (- x 4) (- x 1) (+ x 3))) #'(lambda (x) (- (* x (- (* x 3) 4)) 11)) 20 '(0.6 3.0 -2.5)) --> 0.99999946`
2. `(one-root-newton #'(lambda(x) (* (- x 4) (- x 1) (+ x 3))) #'(lambda (x) (- (* x (- (* x 3) 4)) 11)) 1 '(3.0 -2.5)) --> NIL`

2.3. Todas las raíces para una lista de semillas

2.3.1. Pseudocódigo

```
all-root(f, df, max-iter, semillas, tol):
  para s en semillas:
    l.append(Newton-Raphson(f, df, semilla, tol))
  devuelve l
```

2.3.2. Comentarios

La rutina se implementa fácilmente con el uso de una expresión lambda y `mapcar`. Para eliminar los `NIL` del resultado puede implementarse usando `mapcan`.

2.3.3. Pruebas

1. `(all-roots-newton #'(lambda(x) (* (- x 4) (- x 1) (+ x 3))) #'(lambda (x) (- (* x (- (* x 3) 4)) 11)) 20 '(0.6 3.0 -2.5)) --> (0.99999946 4.000084 -3.0000203)`
2. `(all-roots-newton #'(lambda(x) (* (- x 4) (- x 1) (+ x 3))) #'(lambda (x) (- (* x (- (* x 3) 4)) 11)) 20 '(0.6 3.0 10000.0)) --> (0.99999946 4.000084 NIL)`
3. `(list-not-nil-roots-newton #'(lambda(x) (* (- x 4) (- x 1) (+ x 3))) #'(lambda (x) (- (* x (- (* x 3) 4)) 11)) 20 '(0.6 3.0 10000.0)) --> (0.99999946 4.000084)`

Capítulo 3

Combinación de listas

3.1. Combinación elemento - lista

3.1.1. Pseudocódigo

```
lista = []
comb-elt-lst(elt lst):
    para cada l en lst:
        lista.append( par(elt l) )
    devuelve lista
```

3.1.2. Comentarios

La implementación es algo más delicada. Hemos orientado esta función a las siguientes, de tal forma que distinguimos si `elt` es un elemento o un conjunto (podría serlo). Además, tenemos que considerar un caso especial cuando la lista es `(NIL)`, es decir, $\{\emptyset\}$.

3.1.3. Pruebas

1. `(combine-elt-lst 'a '(1 2 3)) --> ((A 1) (A 2) (A 3))`
2. `(combine-elt-lst 'a nil) --> NIL`
3. `(combine-elt-lst nil nil) --> NIL`
4. `(combine-elt-lst nil '(a b)) --> ((A) (B))`

3.2. Combinación lista-lista

3.2.1. Pseudocódigo

La función a implementar es exactamente el producto cartesiano binario.

```
cart-binary(a, b):
    lista = []
    para cada ai en a:
        l.append(comb-elt-lst(ai, b))
    devuelve l
```

3.2.2. Comentarios

Hemos implementado la función haciendo uso de `mapcan` para obtener el resultado del producto escalar. Además, como operación bien definida, `NIL` es el 0 y `(NIL)` es el 1 de la operación.

3.2.3. Pruebas

1. `(combine-lst-lst '(a b c) '(1 2)) --> ((A 1) (A 2) (B 1) (B 2) (C 1) (C 2))`
2. `(combine-lst-lst nil nil) --> NIL`
3. `(combine-lst-lst '(a b c) nil) --> NIL`
4. `(combine-lst-lst nil '(a b c)) --> NIL`
5. `(combine-lst-lst '(1 2 3) '(nil)) --> (1 2 3)`

3.3. Producto cartesiano n-ario

3.3.1. Pseudocódigo

Siguiendo el estándar de implementación de operaciones n-arias a partir de binarias, el pseudocódigo se resume usando una reducción.

```
cart-nary( lists ):
    devuelve (reduce (cart-binary , lists))
```

3.3.2. Comentarios

Gracias a esta implementación, hemos conseguido una forma elegante de implementar el producto cartesiano n-ario.

3.3.3. Pruebas

1. `(combine-list-of-lsts '((a b c) (+ -) (1 2 3 4))) --> ((A + 1) (A + 2) (A + 3) (A + 4) (A - 1) (A - 2) (A - 3) (A - 4) (B + 1) (B + 2) (B + 3) (B + 4) (B - 1) (B - 2) (B - 3) (B - 4) (C + 1) (C + 2) (C + 3) (C + 4) (C - 1) (C - 2) (C - 3) (C - 4))`
2. `(combine-list-of-lsts '(() (+ -) (1 2 3 4))) --> NIL`
3. `(combine-list-of-lsts '((a b c) () (1 2 3 4))) --> NIL`
4. `(combine-list-of-lsts '((a b c) (1 2 3 4) ())) --> NIL`
5. `(combine-list-of-lsts '((1 2 3 4))) --> (1 2 3 4)`
6. `(combine-list-of-lsts '(nil)) --> NIL`
7. `(combine-list-of-lsts nil) --> NIL`

Capítulo 4

Árboles de verdad

4.1. Funciones de derivación

4.1.1. Pseudocódigo

Son diversas funciones y el pseudocódigo se omite por que no aporta mucho ya que el código es simple. Se basan en el álgebra de Boole y las leyes de De Morgan.

4.1.2. Comentarios

Hemos implementado el desarrollo a forma normal negativa de la negación de: la negación, la conjunción, la disyunción, el condicional y el bicondicional, además de el desarrollo a forma normal negativa de estos dos últimos.

4.1.3. Pruebas

1. (double-neg '(! (! A))) --> A
2. (neg-bicond '(! (<=> A B))) --> (V (∧ A (! B)) (∧ B (! A)))

4.2. Construcción del árbol de verdad

4.2.1. Pseudocódigo

Nuestra función ha de comprobar si alguna rama del árbol es satisfacible.

```
truth-tree(fbf):  
  tree <- construye-arbol(fbf)  
  para cada rama en tree:  
    si rama es satisfacible:  
      devuelve SAT  
  devuelve UNSAT
```

4.2.2. Comentarios

En el pseudocódigo, la lista **tree** se corresponde con una lista de listas, cada una representando una rama del árbol de verdad. La función **construye-arbol** se encarga de ir generando la lista, convirtiendo a forma normal negativa con las funciones de derivación y separando en ramas cada \vee que encuentra. Después, la función encargada de comprobar si una rama es satisfacible simplemente busca por un literal que haya visto conforme va deshaciendo recursivamente la rama.

4.2.3. Pruebas

1. (truth-tree '(! (! A))) --> T

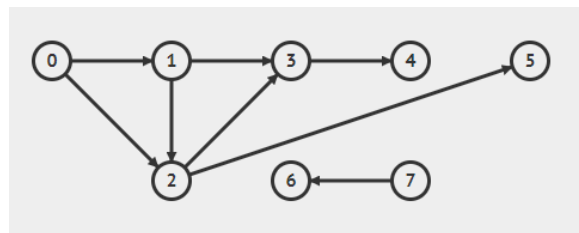
2. `(truth-tree '(\ A (! A))) --> NIL`
3. `(truth-tree '(V A (! A))) --> T`
4. `(truth-tree '(\ (V A B))) --> T`
5. `(truth-tree '(\ (! B) B)) --> NIL`
6. `(truth-tree '(<=> (=> (\ P Q) R) (=> P (v (! Q) R)))) --> T`

Capítulo 5

Búsqueda en anchura

5.1. Ilustración del algoritmo

5.1.1. Grafo especial



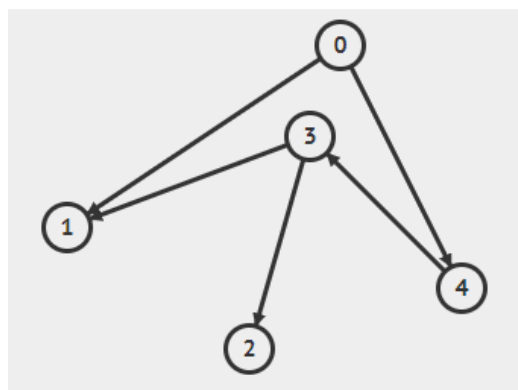
Empezando en el nodo 1 obtendríamos:

1→3→4

1→2→5

Y no llegaríamos ni a 0, ni a 6 ni a 7.

5.1.2. Caso típico

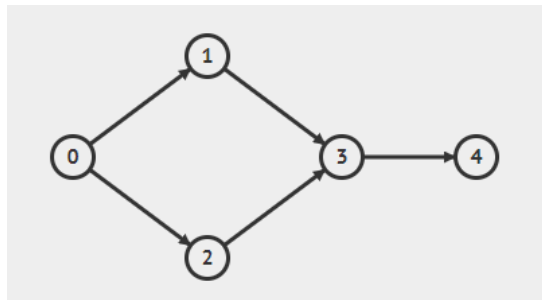


Empezando en el nodo 0 obtendríamos:

0→1

0→4→3→2

5.1.3. Caso típico



Empezando en el nodo 0 obtendríamos:

0->1->3->4

0->2

5.2. Pseudocódigo

```

BFS(grafo G, nodo_inicial s){
  para cada nodo u de G{
    estado[u]=NO_VISITADO
    distancia[u]= INFINITO
    padre[u]=null
  }
  estado[s]=VISITADO
  distancia[s]=0
  crear_cola(Q);
  Encolar(Q, s)
  while (! vacia(Q)){
    u = extraer(Q)
    para cada v de adyacencia[u]{
      if (estado[v]==NO_VISITADO){
        estado[v]= VISITADO
        distancia[v] = distancia[u]+1
        padre[v]=u
        Encolar(Q,v)
      }
    }
  }
}

```

5.3. BFS de ANSI Common Lisp

5.4. Comentarios al código de BFS en ANSI Common Lisp

Tanto 5.3 como 5.4 se resuelven en el código entregado. Se pueden ver en 10.1.

5.5. BFS camino más corto

La función hace uso de BFS para encontrar un camino entre dos nodos. Una forma sencilla de ver que BFS encuentra el camino más corto entre dos nodos es ver que BFS es un caso especial de Dijkstra con costes 1. Como Dijkstra es completo y óptimo, BFS lo es.

5.6. Evaluación de shortest-path

El *trace* de la función esclarece por completo el funcionamiento de la evaluación.

1. Trace: (SHORTEST-PATH 'A 'F '((A D) (B D F) (C E) (D F) (E B F) (F)))
 2. Trace: (BFS 'F '((A)) '((A D) (B D F) (C E) (D F) (E B F) (F)))
 3. Trace: (BFS 'F '((D A)) '((A D) (B D F) (C E) (D F) (E B F) (F)))
 4. Trace: (BFS 'F '((F D A)) '((A D) (B D F) (C E) (D F) (E B F) (F)))
 4. Trace: BFS \Rightarrow (A D F)
 3. Trace: BFS \Rightarrow (A D F)
 2. Trace: BFS \Rightarrow (A D F)
 1. Trace: SHORTEST-PATH \Rightarrow (A D F)
- (A D F)

BFS va añadiendo nodos a la lista que inicialmente solo contiene a A, que es el nodo inicial. Añade D que es su único vecino, y finalmente F.

5.7. Bucle infinito en grafo con ciclos

En este caso, entramos en un bucle infinito. Se puede comprobar si evaluamos la siguiente expresión:

```
(shortest-path 'b 'g '((a b c d e) (b a d e f) (c a g) (d a b e g h)
(e a b d g h) (f b h) (g c d e h) (h d e f g)))
```

5.8. Corrección del código

Se puede ver en el código entregado. Se basa en usar una lista auxiliar para llevar una cuenta de qué nodos hemos visitado ya.

Parte II

Código

Capítulo 6

Distancia coseno

6.1. Implementación distancia coseno

```
1  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
2  ;; scalar-product-rec (x y)
3  ;; Calcula el producto escalar de dos vectores x e y de forma recursiva
4  ;; Se asume que los dos vectores de entrada tienen la misma longitud.
5  ;;
6  ;; INPUT: x: vector, representado como una lista
7  ;;        y: vector, representado como una lista
8  ;; OUTPUT: producto escalar de x e y
9  ;;
10 (defun scalar-product-rec (x y)
11   (if (or (null x) (null y))
12       0
13       (+
14         (* (car x) (car y))
15         (scalar-product-rec (cdr x) (cdr y)))))

1  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
2  ;; squared-norm-rec (x)
3  ;; Calcula la norma al cuadrado de un vector x
4  ;; INPUT: x: vector, representado como una lista
5  ;; OUTPUT: norma al cuadrado de x
6  ;;
7  (defun squared-norm-rec (x)
8   (scalar-product-rec x x))

1  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
2  ;; cosine-distance-rec (x y)
3  ;; Calcula la distancia coseno de un vector de forma recursiva
4  ;; Se asume que los dos vectores de entrada tienen la misma longitud.
5  ;;
6  ;; INPUT: x: vector, representado como una lista
7  ;;        y: vector, representado como una lista
8  ;; OUTPUT: distancia coseno entre x e y
9  ;;
10 (defun cosine-distance-rec (x y)
11   (unless (or (= 0 (squared-norm-rec x)) (= 0 (squared-norm-rec y)))
12     (-
13       1
14       (/
15         (scalar-product-rec x y)
16         (sqrt
17          (* (squared-norm-rec x) (squared-norm-rec y)))))))

1  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
2  ;; scalar-product-mapcar (x y)
3  ;; Calcula el producto escalar de dos vectores x e y usando mapcar
```

```

4  ;; Se asume que los dos vectores de entrada tienen la misma longitud.
5  ;;
6  ;; INPUT:  x: vector, representado como una lista
7  ;;        y: vector, representado como una lista
8  ;; OUTPUT: producto escalar de x e y
9  ;;
10 (defun scalar-product-mapcar (x y)
11   (apply #'+
12    (mapcar #'* x y)))

1  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
2  ;; squared-norm-mapcar (x)
3  ;; Calcula la norma al cuadrado de un vector x
4  ;; INPUT: x: vector, representado como una lista
5  ;; OUTPUT: norma al cuadrado de x
6  ;;
7  (defun squared-norm-mapcar (x)
8   (scalar-product-mapcar x x))

1  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
2  ;; cosine-distance-mapcar
3  ;; Calcula la distancia coseno de un vector usando mapcar
4  ;; Se asume que los dos vectores de entrada tienen la misma longitud.
5  ;;
6  ;; INPUT:  x: vector, representado como una lista
7  ;;        y: vector, representado como una lista
8  ;; OUTPUT: distancia coseno entre x e y
9  ;;
10 (defun cosine-distance-mapcar (x y)
11   (unless (or (= 0 (squared-norm-mapcar x)) (= 0 (squared-norm-mapcar y)))
12     (-
13      1
14      (/
15       (scalar-product-mapcar x y)
16       (sqrt
17        (* (squared-norm-mapcar x) (squared-norm-mapcar y)))))))

```

6.2. ordenación según distancia coseno

```

1  (defun likelihood (x y)
2   (- 1 (cosine-distance-mapcar x y)))

1  (defun less-likelihood (x y vector)
2   (< (likelihood x vector) (likelihood y vector)))

1  (defun insert-in-descending-order (vector element lst less-function)
2   (if (null lst)
3       (cons element lst)
4       (if (funcall less-function element (car lst) vector)
5           (cons (car lst) (insert-in-descending-order vector element (cdr lst) less-function))
6           (cons element lst))))

1  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
2  ;; order-vectors-cosine-distance
3  ;; Devuelve aquellos vectores similares a una categoria
4  ;; INPUT:  vector: vector que representa a una categoria,
5  ;;        representado como una lista
6  ;;        lst-of-vectors vector de vectores
7  ;;        confidence-level: Nivel de confianza (parametro opcional)
8  ;; OUTPUT: Vectores cuya semejanza con respecto a la
9  ;;        categoria es superior al nivel de confianza ,
10 ;;        ordenados
11 ;;

```

```

12  ;;
13  (defun order-vectors-cosine-distance (vector lst-of-vectors &optional (confidence-level 0))
14    (unless (or (null lst-of-vectors) (null vector))
15      (if (> (likelihood vector (car lst-of-vectors)) confidence-level)
16          (insert-in-descending-order vector (car lst-of-vectors) (
              order-vectors-cosine-distance vector (cdr lst-of-vectors) confidence-level) '
              less-likelihood)
17          (order-vectors-cosine-distance vector (cdr lst-of-vectors) confidence-level))))

```

6.3. Clasificador por distancia coseno

```

1  (defun get-min-category (categories text distance-measure)
2    (if (null categories)
3        (NIL 3)
4        (let ((current-distance (funcall distance-measure (cdr text) (cdr (car categories)))))
5          (last-pair (get-min-category (cdr categories) text distance-measure)))
6          (if (< current-distance (NTH 1 last-pair))
7              (cons (car (car categories)) (list current-distance))
8              last-pair))))

```

```

1  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
2  ;; get-vectors-category (categories vectors distance-measure)
3  ;; Clasifica a los textos en categorias .
4  ;;
5  ;; INPUT : categories: vector de vectores, representado como
6  ;;          una lista de listas
7  ;;          texts:      vector de vectores, representado como
8  ;;          una lista de listas
9  ;;          distance-measure: funcion de distancia
10  ;; OUTPUT: Pares formados por el vector que identifica la categoria
11  ;;          de menor distancia , junto con el valor de dicha distancia
12  ;;
13  (defun get-vectors-category (categories texts distance-measure)
14    (unless (or (null (car categories)) (null (car texts))) ;
15      (unless (or (null texts))
16        (cons
17          (get-min-category categories (car texts) distance-measure)
18          (get-vectors-category categories (cdr texts) distance-measure)))))

```


Capítulo 7

Raíces de una función

7.1. Implementación Newton-Raphson

```
1  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
2  ;; h-value
3  ;; Calcula el incremento en el eje X para la siguiente iteración del
4  ;; método de Newton.
5  ;; INPUT:  current-point: punto actual del eje X.
6  ;;         f: función cuyo cero se desea encontrar
7  ;;         df: derivada de f
8  ;;
9  ;; OUTPUT: incremento para la siguiente iteración
10 (defun h-value (current-point f df)
11   (/
12    (funcall f current-point)
13    (funcall df current-point)))

1  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
2  ;; newton
3  ;; Estima el cero de una función mediante Newton-Raphson
4  ;;
5  ;; INPUT : f: función cuyo cero se desea encontrar
6  ;;         df: derivada de f
7  ;;         max-iter: máximo número de iteraciones
8  ;;         x0: estimación inicial del cero (semilla)
9  ;;         tol: tolerancia para convergencia (parámetro opcional)
10 ;; OUTPUT: estimación del cero de f o NIL si no converge
11 ;;
12 (defun newton (f df max-iter x0 &optional (tol 0.001))
13   (unless (or (= (funcall df x0) 0) (= max-iter 0))
14     (let ((h (h-value x0 f df)))
15       (if (< (ABS h) tol)
16         x0
17         (newton f df (- max-iter 1) (- x0 h) tol))))))
```

7.2. Una raíz para una lista de semillas

```
1  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
2  ;; one-root-newton
3  ;; Prueba con distintas semillas iniciales hasta que Newton
4  ;; converge
5  ;;
6  ;; INPUT: f : función de la que se desea encontrar un cero
7  ;;         df : derivada de f
8  ;;         max-iter : máximo número de iteraciones
9  ;;         semillas : semillas con las que invocar a Newton
10 ;;         tol : tolerancia para convergencia ( parámetro opcional )
```

```

11 ;;
12 ;;; OUTPUT: el primer cero de f que se encuentre , o NIL si se diverge
13 ;;;           para todas las semillas
14 ;;;
15 (defun one-root-newton (f df max-iter semillas &optional (tol 0.001))
16   (unless (null semillas)
17     (let ((solution (newton f df max-iter (car semillas) tol)))
18       (if (null solution)
19           (one-root-newton f df max-iter (cdr semillas) tol)
20           solution))))

```

7.3. Todas las raíces para una lista de semillas

```

1  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
2  ;;; all-roots-newton
3  ;;; Prueba con distintas semillas iniciales y devuelve las raices
4  ;;; encontradas por Newton para dichas semillas
5  ;;;
6  ;;; INPUT: f: funcion de la que se desea encontrar un cero
7  ;;;         df: derivada de f
8  ;;;         max-iter: maximo numero de iteraciones
9  ;;;         semillas: semillas con las que invocar a Newton
10 ;;;         tol : tolerancia para convergencia ( parametro opcional )
11 ;;;
12 ;;; OUTPUT: las raices que se encuentren para cada semilla o nil
13 ;;;          si para esa semilla el metodo no converge
14 ;;;
15 (defun all-roots-newton (f df max-iter semillas &optional ( tol 0.001))
16   (mapcar #'(lambda(seed) (newton f df max-iter seed tol)) semillas))

1 (defun list-not-nil-roots-newton (f df max-iter semillas &optional ( tol 0.001))
2   (mapcan (lambda (x) (unless (null x) (list x))) (all-roots-newton f df max-iter semillas tol
3   )))

```

Capítulo 8

Combinación de listas

```
1  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
2  ;; to-list
3  ;; Si x no es una lista, devuelve la lista que contiene a x.
4  ;;
5  ;; INPUT: x: objeto a enlistar
6  ;;
7  ;; OUTPUT: Si x no es una lista, la lista que contiene a x.
8  ;;         Si x es una lista, devuelve la propia x.
9  (defun to-list (x)
10    (if (listp x)
11        x
12        (list x)))
13
14
15  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
16  ;; merge-two-lst
17  ;; Concatena dos listas sin modificar las originales
18  ;;
19  ;; INPUT: lst1: lista a unir
20  ;;        lst2: lista a unir
21  ;;
22  ;; OUTPUT: listas unidas
23  (defun merge-two-lst (lst1 lst2)
24    (if (null lst1)
25        lst2
26        (cons (car lst1) (merge-two-lst (cdr lst1) lst2))))
```

8.1. Combinación elemento - lista

```
1
2  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
3  ;; combine-elt-lst
4  ;; Combina un elemento dado con todos los elementos de una lista
5  ;;
6  ;; INPUT: elem: elemento a combinar
7  ;;        lst: lista con la que se quiere combinar el elemento
8  ;;
9  ;; OUTPUT: lista con las combinaciones del elemento con cada uno de los
10 ;;         de la lista
11 (defun combine-elt-lst (elt lst)
12   (if (equal lst '(NIL))
13       (to-list elt)
14       (mapcar
15        (lambda (x)
16          (if (listp elt) ; Si elt es un conjunto
17              (merge-two-lst elt (list x))
18              (list elt x)))
19        lst)))
```

8.2. Combinación lista-lista

```

1  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
2  ;; combine-lst-lst
3  ;; Calcula el producto cartesiano de dos listas
4  ;;
5  ;; INPUT: lst1: primera lista
6  ;;       lst2: segunda lista
7  ;;
8  ;; OUTPUT: producto cartesiano de las dos listas
9
10 (defun combine-lst-lst (a b)
11   (mapcan
12    (lambda (ai)
13      (combine-elt-lst ai b))
14    a))

```

8.3. Producto cartesiano n-ario

```

1  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
2  ;; combine-list-of-lsts
3  ;; Calcula todas las posibles disposiciones de elementos
4  ;; pertenecientes a N listas de forma que en cada disposicion
5  ;; aparezca unicamente un elemento de cada lista
6  ;;
7  ;; INPUT: lstolsts: lista de listas
8  ;;
9  ;; OUTPUT: lista con todas las posibles combinaciones de elementos
10
11 (defun combine-list-of-lsts (a)
12   (unless (null a)
13     (reduce #'combine-lst-lst a)))

```

Capítulo 9

Árboles de verdad

9.1. Funciones de derivación

```
1  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
2  ;; double-neg (x)
3  ;; Deshace la doble negacion de una fbf
4  ;; INPUT: formula bien formada x representada como lista
5  ;; OUTPUT: la fbf x sin la doble negacion
6  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
7  (defun double-neg (x)
8    (car (cdr (car (cdr x)))))

1  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
2  ;; neg-conj (x)
3  ;; Aplica las formula de De Morgan a la negacion de un conjuncion
4  ;; INPUT: formula bien formada x representada como lista
5  ;; OUTPUT: la fbf x tras aplicar De Morgan
6  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
7  (defun neg-conj (x)
8    (cons +or+ (combine-elt-lst +not+ (cdr (car (cdr x))))))

1  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
2  ;; neg-disj (x)
3  ;; Aplica las formula de De Morgan a la negacion de una disjuncion
4  ;; INPUT: formula bien formada x representada como lista
5  ;; OUTPUT: la fbf x tras aplicar De Morgan
6  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
7  (defun neg-disj (x)
8    (cons +and+ (combine-elt-lst +not+ (cdr (car (cdr x))))))

1  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
2  ;; implies (x)
3  ;; Transforma la implicacion x en un disjuncion aplicando las
4  ;; reglas de derivacion
5  ;; INPUT: formula bien formada x representada como lista
6  ;; OUTPUT: la fbf x tras la regla de derivacion
7  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
8  (defun implies (x)
9    (let ((literals (cdr x)))
10     (list +or+ (list +not+ (car literals)) (car (cdr literals)))))

1  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
2  ;; neg-implies (x)
3  ;; Transforma la negacion de una implicacion x en un conjuncion aplicando
4  ;; las reglas de derivacion y De Morgan
5  ;; INPUT: formula bien formada x representada como lista
6  ;; OUTPUT: la fbf x tras la regla de derivacion y De Morgan
7  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

9.2. construcción del árbol de verdad

```

1  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
2  ;;; expand-truth-tree-aux
3  ;;; Recibe una expresion y construye una lista de ramas a partir de
4  ;;; disyunciones o conjunciones, pasando a forma normal negativa.
5  ;;;
6  ;;; INPUT  : fbf - Formula bien formada (FBF) a analizar.
7  ;;; OUTPUT : lista de ramas
8  ;;;
9  (defun expand-truth-tree-aux (fbf)
10   (cond
11     ((literal-p fbf)
12      (list fbf))
13     ((unary-connector-p (car fbf))
14      (cond
15        ((unary-connector-p (car (car (cdr fbf))))
16         (expand-truth-tree-aux (double-neg fbf)))
17        ((cond-connector-p (car (car (cdr fbf))))
18         (expand-truth-tree-aux (neg-implies fbf)))
19        ((bicond-connector-p (car (car (cdr fbf))))
20         (expand-truth-tree-aux (neg-bicond fbf)))
21        ((n-ary-connector-p (car (car (cdr fbf))))
22         (if (eql (car (car (cdr fbf))) +or+)
23             (expand-truth-tree-aux (neg-disj fbf))
24             (expand-truth-tree-aux (neg-conj fbf))))))
25     ((cond-connector-p (car fbf))
26      (expand-truth-tree-aux (implies fbf)))
27     ((bicond-connector-p (car fbf))
28      (expand-truth-tree-aux (bicond fbf)))
29     ((n-ary-connector-p (car fbf))
30      (if (eql (car fbf) +and+)
31          (unless (null (cdr fbf))

```

```

38      (nconc (expand-truth-tree-aux (car (cdr fbf))) (expand-truth-tree-aux (cons +
39      and+ (cddr fbf)))))
40      (if (null (cddr fbf))
41      (expand-truth-tree-aux (car (cdr fbf)))
42      (mapcan (lambda (x) (list (expand-truth-tree-aux x))) (combine-list-of-lsts (
43      list (list +or+) (cdr fbf)))))))

1  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
2  ;; evaluate (fbf)
3  ;; Comprueba si una fbf es satisfacible
4  ;;
5  ;; INPUT : auxlist: lista de literales encontrados
6  ;;       : fbf - Formula bien formada (FBF) a analizar.
7  ;; OUTPUT : T - FBF es SAT
8  ;;       : N - FBF es UNSAT
9  ;;
10 (defun evaluate (auxlist fbf)
11   (if (null fbf)
12       T
13       (if (listp (car fbf))
14           (unless (find (car (cdr (car fbf))) (car auxlist) :test #'equal)
15               (if (find (car (cdr (car fbf))) (car (cdr auxlist)) :test #'equal)
16                   (evaluate auxlist (cdr fbf))
17                   (evaluate (cons (car auxlist) (list (NCONC (list (car (cdr (car fbf)))) (car
18                   (cdr auxlist))))) (cdr fbf))))
19           (unless (find (car fbf) (car (cdr auxlist)) :test #'equal)
20               (if (find (car fbf) (car auxlist) :test #'equal)
21                   (evaluate auxlist (cdr fbf))
22                   (evaluate (list (NCONC (car auxlist) (list (car fbf))) (car (cdr auxlist)))
23                   (cdr fbf)))))))

1  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
2  ;; branch-is-sat (fbf)
3  ;; Comprueba si una rama es satisfacible
4  ;;
5  ;; INPUT : fbf - Formula bien formada (FBF) a analizar.
6  ;; OUTPUT : T - FBF es SAT
7  ;;       : N - FBF es UNSAT
8  ;;
9  ;;
10 (defun branch-is-sat (fbf)
11   (evaluate nil fbf))

1  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
2  ;; truth-tree
3  ;; Recibe una expresion y construye su arbol de verdad para
4  ;; determinar si es SAT o UNSAT
5  ;;
6  ;; INPUT : fbf - Formula bien formada (FBF) a analizar.
7  ;; OUTPUT : T - FBF es SAT
8  ;;       : N - FBF es UNSAT
9  ;;
10 (defun truth-tree (fbf)
11   (unless (null fbf)
12       (some #'branch-is-sat (list (expand-truth-tree-aux fbf)))))

```


Capítulo 10

Búsqueda en anchura

10.1. Comentarios al código de BFS en ANSI Common Lisp

```
1  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
2  ;;; Breadth-first-search in graphs
3  ;;;
4  ;;; INPUT:  end: nodo final
5  ;;;         queue: cola de nodos por explorar
6  ;;;         net: grafo
7  ;;; OUTPUT: arbol de busqueda en anchura
8  ;;;         nil si no lo encuentra
9  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
10 (defun bfs (end queue net)
11   (if (null queue) ;;; Si la cola esta vacia devuelve una lista vacia
12       '()
13       ;;; Inicializa path como el primer elemento de queue y node como el primero
14       ;;; de path
15       (let* ((path (first queue))
16              (node (first path)))
17         ;;; Si node es igual a el nodo buscado devolvemos el camino inverso de path
18         (if (eql node end)
19             (reverse path)
20             ;;; Si no, llama recursivamente a BFS
21             (bfs end
22                  (append (rest queue)
23                           (new-paths path node net))
24                  net )))))
25
26 (defun new-paths (path node net)
27   (mapcar #'(lambda (n)
28               (cons n path))
29           (rest (assoc node net))))
```

10.2. Corrección del código

```
1  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
2  ;;; bfs-improved
3  ;;; busqueda en anchura mejorada sin recursion infinita
4  ;;; INPUT:  end: nodo final
5  ;;;         queue: cola de nodos por explorar
6  ;;;         net: grafo
7  ;;;         expanded-nodes: nodos ya expandidos o vistos
8  ;;; OUTPUT: arbol de busqueda en anchura
9  ;;;         nil si no lo encuentra
10 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
11 (defun bfs-improved (end queue net expanded-nodes)
12   (if (null queue) ;;; Si la cola esta vacia devuelve una lista vacia
13       '()
14       ;;; Inicializa path como el primer elemento de queue y node como el primero
15       ;;; de path
16       (let* ((path (first queue))
17              (node (first path))
18              (new-paths (new-paths path node net expanded-nodes)))
19         ;;; Si node es igual a el nodo buscado devolvemos el camino inverso de path
20         (if (eql node end)
21             (reverse path)
22             ;;; Si no, llama recursivamente a BFS
23             (bfs-improved end
24                             (append (rest queue)
25                                     new-paths)
26                             net
27                             (cons node expanded-nodes))))))
```

```

14      ;; Inicializa path como el primer elemento de queue y node como el primero
15      ;; de path
16      (let* ((path (first queue))
17             (node (first path)))
18        ;; Si node es igual a el nodo buscado devolvemos el camino inverso de path
19        (if (EQL node end)
20            (reverse path)
21            (if (find node expanded-nodes)
22                (bfs-improved end (cdr queue) net expanded-nodes)
23                (bfs-improved ebd (APPEND (cdr queue) (new-paths path
24                                                (cons node expanded-nodes) net)) net expanded-nodes))))))

1  ;;
2  ;;; shortest-path-improved
3  ;;; Version de busqueda en anchura que no entra en recursion
4  ;;; infinita cuando el grafo tiene ciclos
5  ;;; INPUT:  end: nodo final
6  ;;;        queue: cola de nodos por explorar
7  ;;;        net: grafo
8  ;;; OUTPUT: camino mas corto entre dos nodos
9  ;;;        nil si no lo encuentra
10 (defun shortest-path-improved (start end net)
11   (bfs-improved end (list(list start)) net nil))

```