Universidad Autónoma

Inteligencia Artificial

Práctica 1

Memoria

Autores:
Adrián Fernández
Santiago González-Carvajal

11 de febrero de 2018

Índice

1. Similitud Coseno

1.1. Función similitud coseno

Batería de pruebas

```
» (sc-rec '(1 0) '(1 0)) --> 1.0
» (sc-rec '(1 0) '(0 1)) --> 0.0
» (sc-rec '(1 0) '(1 1)) --> 0.70710677
» (sc-mapcar '(1 0) '(1 0)) --> 1.0
» (sc-mapcar '(1 0) '(0 1)) --> 0.0
» (sc-mapcar '(1 0) '(1 1)) --> 0.70710677

» (sc-rec '(1 2 3) '(3 2 1)) --> 0.7142857
» (sc-mapcar '(1 2 3) '(3 2 1)) --> 0.7142857

» (sc-rec '(3 4) '(1 0)) --> 0.6
» (sc-rec '(3 4) '(3 0)) --> 0.6
» (sc-mapcar '(3 4) '(1 0)) --> 0.6
» (sc-mapcar '(3 4) '(3 0)) --> 0.6
```

Pseudocódigo

```
Entrada: x (vector de números reales), y (vector de números reales)
Salida: sc (valor del coseno del ángulo que forman los vectores x, y)

Procesamiento:
sim-cos (x y):
    denom <- sqrt(pesc(x x))*sqrt(pesc(y y))
    si denom es 0.0
        devuelve NIL
    en caso contrario
        devuelve pesc(x y)/denom
fin sim-cos
```

Comentarios sobre implementación

Para realizar ambas implementaciones, hemos creado 2 funciones que implementan el producto escalar. Una de forma recursiva y otra utilizando

mapcar. El procesamiento de la función similitud coseno sería el mismo, sustituyendo pesc por una de las implementaciones para cada caso.

1.2. Función de categorías

Batería de pruebas

```
» (sc-conf '(1 0) '((1 0.2) (1 0.4) (1 0.6) (1 0.8) (1 1))  1)
--> NIL
» (sc-conf '(1 0) '((1 0) (1 0.2) (1 0.4) (1 0.6) (1 0.8) (1 1))  1)
-- > ((1 0))
» (sc-conf '(1 0) '((1 0) (1 0.2) (1 0.4) (1 0.6) (1 0.8) (1 1))  0.95)
-- > ((1 0) (1 0.2))
» (sc-conf '(1 0) '((1 0) (1 0.2) (1 0.4) (1 0.6) (1 0.8) (1 1))  0.9)
-- > ((1 0) (1 0.2) (1 0.4))
» (sc-conf '(1 0) '((1 0) (1 0.2) (1 0.4) (1 0.6) (1 0.8) (1 1))  0.8)
--> ((1 0) (1 0.2) (1 0.4) (1 0.6))
» (sc-conf '(1 0) '((1 0) (1 0.2) (1 0.4) (1 0.6) (1 0.8) (1 1))  0.75)
--> ((1 0) (1 0.2) (1 0.4) (1 0.6) (1 0.8))
» (sc-conf '(1 0) '((1 0) (1 0.2) (1 0.4) (1 0.6) (1 0.8) (1 1))  0.7)
--> ((1 0) (1 0.2) (1 0.4) (1 0.6) (1 0.8) (1 1))
```

Pseudocódigo

```
Entrada: x (vector de números reales), vs (lista de vectores reales),
conf (número real)
Salida: vconf (vectores cuya similitud con respecto a la categoría es
superior al nivel de confianza, ordenados)

Procedimiento:
sc-conf (x vs conf):
    para cada v en vs:
        si sim(v) > conf
        add(lst similitud(x v))
    devolver ordenar(lst)
fin sc-conf
```

Comentarios sobre implementación

Utilizar un lenguaje funcional para implementar esta función es especialmente útil, ya que requiere aplicar la función similitud coseno a una lista de vectores y filtrar los vectores con el valor de esa función. Esto se logra fácilmente mediante mapcar y expresiones lambda.

1.3. Clasificador por similitud coseno

Pseudocódigo

```
Entrada: cats (lista vectores cuyo primer elemento es un
identificador), texts(lista vectores cuyo primer elemento
es un identificador), f (función)
Salida: lsol (lista de pares cuyo  segundo elemento es
la mayor similitud coseno del conjunto de vectores cats con
el vector de texts que ocupa la posición del par y el primero
es el identificador de dicho vector)

Proceso:
sc-classifier (cats texts f):
    para cada t en texts:
        (id valor) <-- maximo(f(t cats))
        add(lst (id valor))
    devuelve lst
fin sc-classifier
```

Comentarios sobre implementación

La implementación es parecida a la de la función anterior, pero la función es más general.

1.4. Pruebas llamando a sc-classifier

Batería de pruebas

```
» (setf cats '((1 43 23 12) (2 33 54 24)))
» (setf texts '((1 3 22 134) (2 43 26 58)))
» (sc-classifier cats texts #'sc-rec) --> ((2 0.48981872) (1 0.81555086))
```

```
» (sc-classifier cats texts #'sc-mapcar) --> ((2 0.48981872) (1 0.81555086))

» (time
   (sc-classifier
    '((1 43 23 12) (2 33 54 24) (3 4 3 1) (4 2 1 2) (5 2 1 2)
    (6 2 1 2) (7 2 1 2))
    '((1 3 22 134) (2 43 26 58) (3 23 21 83) (4 12 32 1 4))
    #'sc-rec)) --> ((4 0.72641367) (4 0.99035937) (4 0.87609464)
    (2 0.9219237))
                  time: 0.007000 sec
» (time
   (sc-classifier
    '((1 4 3 1 93 1 23 9) (2 2 1 2 2321 21 21 3))
    '((1 3 22 134 122 41 48 4) (2 43 26 58 3 4 5 6))
    #'sc-rec)) --> ((1 0.6841774) (1 0.10203057))
                  time: 0.001000 sec
» (time
   (sc-classifier
    '((1 4 3 1 93 1 23 9 43 34) (2 2 1 2 2321 21 21 3 12 43))
    '((1 3 22 134 122 41 48 4 12 34) (2 43 26 58 3 4 5 6 95 23))
    #'sc-rec)) --> ((1 0.6613261) (1 0.4072973))
                  time: 0.001000 sec

» (time
   (sc-classifier
    '((1 43 23 12) (2 33 54 24) (3 4 3 1) (4 2 1 2) (5 2 1 2)
    (6 2 1 2) (7 2 1 2))
    '((1 3 22 134) (2 43 26 58) (3 23 21 83) (4 12 32 1 4))
    #'sc-mapcar)) --> ((4 0.72641367) (4 0.99035937) (4 0.87609464)
    (2 0.9219237))
                     time: 0.008000 sec
» (time
   (sc-classifier
    '((1 4 3 1 93 1 23 9) (2 2 1 2 2321 21 21 3))
    '((1 3 22 134 122 41 48 4) (2 43 26 58 3 4 5 6))
    #'sc-mapcar)) --> ((1 0.6841774) (1 0.10203057))
                     time: 0.000000 sec
» (time
```

```
(sc-classifier
 '((1 4 3 1 93 1 23 9 234 12) (2 2 1 2 2321 21 21 3 21 34))
 '((1 3 22 134 122 41 48 44 23 12) (2 43 26 58 3 4 5 6 12 34))
 #'sc-mapcar)) --> ((1 0.6613261) (1 0.4072973))
                   time: 0.001000 sec
```

Comentarios sobre las pruebas realizadas

Recursión es más eficiente para vectores pequeños, ya que mapcar paraleliza el proceso, y para procesos pequeños se pierde más tiempo paralelizando que ejecutando. Sin embargo mapcar es más rápido para vectores más grandes y mediante recursión se puede llenar la pila de recursión si los vectores son lo suficientemente grandes.

2. Raíces de una función

2.1. Función bisect

Batería de pruebas

```
» (bisect #'(lambda(x) (sin (* 6.26 x))) 0.1 0.7 0.001) ;;---> 0.5020995
» (bisect #'(lambda(x) (sin (* 6.26 x))) 0.0 0.7 0.001) ;;---> NIL
» (bisect #'(lambda(x) (sin (* 6.28 x))) 1.1 1.5 0.001) ;;---> NIL
» (bisect #'(lambda(x) (sin (* 6.28 x))) 1.1 2.1 0.001) ;;---> NIL
```

Pseudocódigo

```
Entrada: f (función), a (número real), b (número real mayor que a),
tol (número real)
Salida: c (solución de f en [a b] con error máximo tol)

Procesamiento:
bisect (f a b tol):
    si f(a) es 0
        devuelve a
    si f(b) es 0
        devuelve b
    si dist(a b) menor tol
```

```
        devuelve mid(a b)
    en caso contrario
        si f(a)*f(mid(a b)) menor 0
            devuelve bisect(f a mid(a b) tol)
        en caso contrario
            devuelve bisect(f mid(a b) b tol)
fin bisect
```

Comentarios sobre implementación

Puesto que se exige un número real a la salida, pero la condición de salida se mide con la norma del intervalo, hemos decidido que el output de la función sea el punto medio del primer intervalo que cumple dicha condición. También cabe destacar que en el enunciado se pide explícitamente que bisect devuelva NIL si $f(a)f(b) >= 0$, pero sería más correcto comprobar $f(a)f(b) > 0$.

2.2. Función allroot

Batería de pruebas

```
» (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) --> (0.50027466 1.0005188 1.5007629 2.001007)
» (allroot #'(lambda(x) (cos (* 6.28 x))) '(0.00 0.50 1.00 1.50 2.00)
0.0001) --> (0.2501526 0.7503967 1.2506409 1.750885)
» (allroot #'(lambda(x) (cos (* 6.28 x))) '(0.00 0.50 1.00 1.50 2.00 2.50)
0.0001) --> (0.2501526 0.7503967 1.2506409 1.750885 2.2511292)
```

Pseudocódigo

```
Entrada: f (función), lst (lista de números reales),
tol (número real)
Salida: lsol (lista con las soluciones de f encontradas
mediante bisect entre
los puntos de lst)

Procesamiento:
```

```
allroot (f lst tol):
    si lst vacía
        devuelve NIL
    si rest(lst) vacía
        devuelve NIL
    en caso contrario
        devuelve concatenar(bisect(f first(lst) first(rest(lst)) tol)
                            allroot(f rest(lst) tol))
fin allroot
```

Comentarios sobre implementación

Se implementa fácilmente con una llamada recursiva y la función bisect.

2.3. Función allind

Batería de pruebas

```
» (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)
--> (0.50027084 1.0005027 1.5007347 2.0010324)
» (allind #'(lambda(x) (cos (* 6.28 x))) 0.0 2.14 3 0.0001)
--> (0.25009555 0.7503519 1.2506084 1.7508645)
» (allind #'(lambda(x) (cos (* 6.28 x))) 0.0 2.15 4 0.0001)
--> (0.2501488 0.75038075 1.2506126 1.7509103)
```

Pseudocódigo

```
Entrada: f (función), a (número real), b (número real mayor que a),
N (número natural), tol (número real)
Salida: lsol (lista con las soluciones de f encontradas en la
partición de [a b] en 2^N intervalos)

Procesamiento:
allind (f a b N tol):
    si N es 1
        devuelve allroot(f list(a mid(a b) b) tol)
    en caso contrario
```

```
        devuelve concatenar(allind(f a mid(a b) N-1 tol)
                            allind(f mid(a b) b N-1 tol))
fin allind
```

Comentarios sobre implementación

Hemos decidido implementar esta función basándonos en allroot, ya que se adapta perfectamente a la funcionalidad exigida y no requiere mayor esfuerzo. Para conseguir la lista de 2^N elementos basta con partir el intervalo por la mitad de forma recursiva N veces.

3. Combinación de listas

3.1. Combinar elemento con lista

Batería de pruebas

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

Pseudocódigo

```
Entrada: elt (elemento a combinar), lst (lista con la que
combinar el elemento)
Salida: lsol (lista de listas con todas las combinaciones
posibles del elemento con los elementos
de la lista)

Procesamiento:
combine-elt-lst (elt lst):
    para cada el en lst:
        lis <-- lista(elt el)
        add(lsol lis)
    devuelve lsol
fin combine-elt-lst
```

Comentarios sobre implementación

La única decisión de diseño que hemos realizado ha sido la de usar (list) en vez de (cons), y esto ha sido simplemente, con el objetivo de conseguir la salida pedida en el enunciado de la práctica ya que (cons) añadía un punto al par (A . B), mientras que (list) no lo hacía (A B).

3.2. Combinar lista con lista

Batería de pruebas

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

Pseudocódigo

```
Entrada: lst1 (lista a combinar), lst2 (lista con la que combinar)
Salida: lsol (lista de listas con todas las combinaciones posibles
de ambas listas)

Procesamiento:
combine-lst-lst (lst1 lst2):
    para cada el en lst1:
        lis <-- combine-elt-lst(el lst2)
        add(lsol lis)
    devuelve lsol
fin combine-lst-lst
```

Comentarios sobre implementación

Hemos decidido aprovechar la implementación del apartado anterior, y por lo tanto, hemos separado el problema en N combinaciones de elemento con lista donde N es el número de elementos de la primera lista.

3.3. Función de disposiciones

Batería de pruebas

```
» (combine-list-of-lsts '()) --> (NIL)
» (combine-list-of-lsts '(() (+ -) (1 2 3 4))) --> NIL
» (combine-list-of-lsts '((a b c) () (1 2 3 4))) --> NIL
» (combine-list-of-lsts '((a b c) (1 2 3 4) ())) --> NIL
» (combine-list-of-lsts '((1 2 3 4))) --> ((1) (2) (3) (4))

» (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))

» (combine-list-of-lsts '((a b c) (+ -) (1 2 3 4) (p o) (i r s)))
-->((A + 1 P I) (A + 1 P R) (A + 1 P S)
    (A + 1 O I) (A + 1 O R) (A + 1 O S)
    (A + 2 P I) (A + 2 P R) (A + 2 P S)
    (A + 2 O I) ...)
```

Pseudocódigo

```
Entrada: lstolsts (lista de listas a combinar)
Salida: lsol (lista con todas las combinaciones posibles de las listas)

combine-list-of-lsts (lstolsts):
    si lstolsts vacía
        devuelve (lista NIL)
    en caso contrario
        para cada el en (primera lista de lstolsts):
            lis-aux <-- (combine-list-of-lsts(rest lstolsts))
            lis-res <-- combine-elt-lst(el lis-aux)
            add(lsol lista-res)
        devuelve lsol
fin combine-list-of-lsts
```

Comentarios sobre implementación

Hemos implementado una primera versión, que funcionaba pero era muy larga (y poco elegante). Pero, después, hemos conseguido llegar a una solución mucho más elegante, para la que hemos tenido que ultilizar una función çombine-elt-lst-c"que es como la que hemos tenido que implementar en el apartado 3.1 salvo por el retorno (en vez de hacer (list), empleamos (cons) al combinar el elemento con la lista). A parte de eso, lo más difícil ha sido darnos cuenta del caso base.