PRÁCTICA 1 INTELIGENCIA ARTIFICIAL

Grupo 2301, Pareja 03

Alejandro Cabana Suárez y Aitor Arnaiz del Val

1 Similitud Coseno

1.1 Implementa una función que calcule la similitud coseno entre dos vectores, x e y, de dos formas:

1.1.1 De forma recursiva:

```
(defun dot-product-rec (u v)
      (if (or (null u) (null v))
            0
            (+ (* (first u) (first v))
                  (dot-product-rec (rest u) (rest v)))))
```

Esta función calcula el producto escalar de dos vectores de forma recursiva. Decidimos separar el cálculo del producto escalar y de la raiz de la norma-2 de los vectores por darle una mayor simplicidad y modularidad al problema y, por consiguiente, al código.

```
(defun 2-norm-rec (v)
    (sqrt (dot-product-rec v v)))
```

La función 2-norm-rec realiza el cálculo de la raiz del producto escalar consigo mismo, esto es, la norma-2 del vector. Se decidió implementar de esta forma para reaprovechar el código de la función anterior.

```
(defun sc-rec (x y)
    (if (or (null x)
            (null y)
            (every #'zerop x)
            (every #'zerop y))
    nil
    (/ (dot-product-rec x y)
            (* (2-norm-rec x)
                  (2-norm-rec y)))))
```

La función sc-rec hace uso de las dos funciones definidas anteriormente para hacer el cálculo de la similitud coseno entre dos vectores de forma recursiva, como puede apreciarse en el código.

1.1.2 Utilizando mapcar:

Al igual que en la implementación recursiva de la función que calcula la similitud coseno, hemos optado por simplificar y dividir el trabajo en dos funciones auxiliares: por un lado, dot-product-mapcar, que calcula el producto escalar de dos vectores utilizando la directiva mapcar de lisp, así como 2-norm-mapcar, que hace una llamada a la función anterior, al igual que lo hacía su versión recursiva. Por último, hemos implementado la función sc-mapcar, que devuelve la similitud coseno de dos vectores haciendo llamadas a las dos funciones descritas anteriormente:

```
(defun dot-product-mapcar (u v)
    (reduce #' + (mapcar #' * u v)))


(defun 2-norm-mapcar (v)
    (sqrt (dot-product-mapcar v v)))


(defun sc-mapcar (x y)
    (if (or (null x)
            (null y)
            (every #'zerop x)
            (every #'zerop y))
    nil
    (/ (dot-product-mapcar x y)
            (* (2-norm-mapcar x)
                    (2-norm-mapcar y)))))
```

1.2 Codifica una función que reciba un vector que representa a una categoría, un conjunto de vectores y un nivel de confianza que esté entre 0 y 1. La función deberá retornar el conjunto de vectores ordenado según mas se parezcan a la categoría dada si su semejanza es superior al nivel de confianza.

Para este apartado, hemos definido dos funciones: por una parte, sc-conf-ind hace el mismo trabajo que la función pedida en el enunciado de este apartado pero a efectos individuales, es decir, devuelve conses cuyo primer elemento es la similitud coseno entre el vector categoría y uno de los vectores pasados como argumento, y cuyo rest es este último vector vector, o NIL en caso de que esta similitud sea menor que la umbral pasada como parámetro;

```
(defun sc-conf-ind (cat v conf)
    (let ((sc (sc-rec cat v)))
            (if (or (null sc) (<= sc conf))
                    nil
                    (cons sc v))))
```

Por otra parte, hemos definido asímismo la función sc-conf, que es la que realiza el trabajo especificado en el enunciado mediante llamadas a sc-conf-ind.

```
(defun sc-conf (cat vs conf)
    (mapcar #'rest
            (sort
                    (remove nil
                            (mapcar #'(lambda (z) (sc-conf-ind cat z conf)) vs))
                    #'> :key #'first)))
```

El único problema que se nos planteaba en este punto era que si alguno de los vectores no tenia una similitud coseno con el vector categoría mayor de la umbral se metía un NIL en su posición de la lista, y por esta razón nos hemos visto obligados a aplicar la directiva remove para quitar estos nil de nuestro vector resultado.

Por otra parte, como teníamos que devolver el vector ordenado de mayoir similitud a menor, le aplicamos la función sort a esta lista, una vez desprovista de elementos NIL.

1.3 Clasificador por similitud coseno. Codifica una función que reciba un conjunto de categorías, cuyo primer elemento es su identificador, un conjunto de vectores que representan textos, cuyo primer elemento es su identificador, y devuelva una lista que contenga pares definidos por un identificador de la categoría que maximiza la similitud coseno con respecto a ese texto y el resultado de la similitud coseno. El tercer argumento es la función usada para evaluar la similitud coseno (mapcar o recursiva).

Para este apartado hemos definido una sola función, la función sc-classifier, que por así decirlo, nos "indexa" por el vector de textos la categoría que maximiza la similitud coseno para cada uno de esos textos, devolviendo así una lista de pares (identificador de categoría, similitud coseno de esa categoría con respecto al texto actual)

```
(defun sc-classifier (cats texts func)
    (if (null texts)
        nil
        (cons (reduce #'(lambda (x y)
                (if (> (first (rest x)) (first (rest y)))
                x
                y))
        (mapcar #'(lambda (cat)
                (list (first cat)
                        (funcall func (rest cat) (rest (first texts)))))
        cats))
    (sc-classifier cats (rest texts) func))))
```

Como aspectos reseñables en la implementación de esta función podemos señalar los siguientes:

Por un lado, que es una función recursiva, que se llama a sí misma texto a texto, hasta que no queden más textos que comparar con las categorías en cats (vector de categorías).

Por otra parte, cabe destacar que como estamos accediendo a listas de números al aplicar la directiva reduce con la función "mayor que" para acceder a los números y al no tratarse de conses los elementos sobre los que trabajamos (como así era hasta el cambio de enunciado) debemos acceder a los números guardados en estas listas aplicando la función first, como se ve en la quinta línea de la función

Por último, podemos destacar el uso de funcall para llamar a la función pertinente (la cual se pasa como parámetro) dependiendo de si queremos utilizar la versión recursiva o con mapcar de obtención de la similitud coseno.

1.4 Haz pruebas llamando a sc-classifier con las distintas variantes de la distancia coseno y para varias dimensiones de los vectores de entrada. Verifica y compara tiempos de ejecución. Documenta en la memoria las conclusiones a las que llegues.

Ejecutando la función sc-classifier para varios tamaños de sublistas y y distinto número de sublistas por lista podemos apreciar que para un número pequeño de sublistas y pocos elementos por sublista funciona mejor la versión sc-mapcar, aunque esta mejora de tiempos es prácticamente inapreciable, mientas que según vamos incrementando tanto el tamaño de las sublistas como el número de sublistas podemos apreciar una mejora del rendimiento de la versión recursiva (sc-rec) con respecto a la que usa mapcar

2 Raíces de una función.

2.1 Implemente una función bisect que aplique el método de bisección para encontrar una raíz de una función en un intervalo dado;

Implementamos la función lógica xnor, que devuelve T si y sólo si sus dos argumentos son T o los dos son NIL. Esto nos servirá para comprobar si dos números tienen el mismo signo, ya que los signos los representaremos como booleanos (T: +, NIL: -).

```
(defun xnor (a b)
    (or (and a b)
        (and (not a)
             (not b))))
```

El cuerpo de bisect es un gran cond que comprueba primero si a o b ya son 0, o si da la casualidad de que su punto medio sea 0, en cuyo caso ya hemos encontrado la raíz.

Si ninguno lo es, entonces mira qué extremo del intervalo tiene distinto signo que el punto medio al aplicarles la función, y se llama a sí misma recursivamente con el punto medio y el extremo elegido, como nuevos extremos de intervalo.

```
(defun bisect (f a b tol)
    (let* ((m (/ (+ a b) 2))
           (fa (funcall f a))
           (fb (funcall f b))
           (fm (funcall f m))
           (siga (> fa 0))
           (sigb (> fb 0))
           (sigm (> fm 0)))
    (cond ((xnor siga sigb)
           nil)
           ((= fa 0)
           a)
           ((= fb 0)
           b)
           ((= fm 0)
```

```
m)
((< (- b a) tol)
m)
((xnor sigm sigb)
(bisect f a m tol))
(t
(bisect f m b tol)))))
```

2.2 Implemente una función allroot que recibe una función, una lista de valores y una tolerancia. Si entre los elementos l[i] y l[i + 1] de la lista hay una raíz, la función devuelve el valor de la raíz. El resultado es una lista con todas las raíces encontradas.

Ahora queremos hacer lo mismo que en la función del apartado anterior, solo que para todos los elementos de una lista, así que el primer impulso es usar mapcar para conseguirlo. Sin embargo, si en alguno de estos intervalos no se encuentra raíz, bisect devolverá NIL. Como no queremos que aparezcan NIL en la solución, en lugar de usar mapcar recorremos la lista recursivamente, comprobamos el resultado de bisect para cada par y sólo lo añadimos si no es NIL.

```
(defun allroot (f lst tol)
    (if (null (rest lst))
        nil
        (let ((root (bisect f (first lst) (second lst) tol)))
            (if (null root)
                (allroot f (rest lst) tol)
                (cons root
                    (allroot f (rest lst) tol))))))
```

2.3 Implemente una función allind que recibe una funcion, dos límites del intervalo, un valor entero N y la tolerancia. La función divide el intervalo en 2 N secciones y busca en cada una una raíz de la función (dentro de la tolerancia). Devuelve una lista con las raices encontradas.

Como nos piden dividir el intervalo en 2^N trozos, pensamos en dividir recursivamente el intervalo en mitades N veces.
Utilizando el propio argumento N de la función como contador, sabemos cuándo parar la recursión. Cuando llega a 0, aplicamos bisec al fragmento de intervalo resultante, y se irán concatenando los raíces encontradas. De nuevo, para evitar que se cuelen NIL, los filtramos en este paso.

```
(defun allind (f a b N tol)
    (if (= N 0)
        (remove nil (list (bisect f a b tol)))
        (let ((m (/ (+ a b) 2)))
            (append (allind f a m (- N 1) tol)
                (allind f m b (- N 1) tol)))))
```

3 Combinación de listas

3.1 Define una función que combine un elemento dado con todos los elementos de una lista:

Para solucionar este ejercicio, basta con aplicar (list elt e) a cada elemento e de la lista. Esto lo hacemos utilizando mapcar con una lambda que cumple esa función.

```
(defun combine-elt-lst (ele lst)
    (mapcar #'(lambda (list1)
                (list ele list1)) lst))
```

3.2 Diseña una función que calcule el producto cartesiano de dos listas:

Tenemos que hacer lo que en la funcion del apartado anterior, pero para cada elemento de una lista; así que utilizamos mapcan para conseguirlo (ya que queremos los resultados de cada operación concatenados).

```
(defun combine-lst-lst (lst1 lst2)
    (mapcan #'(lambda (e)
                (combine-elt-lst e lst2))
        lst1))
```

3.3 Diseña una función que calcule todas las posibles disposiciones de elementos pertenecientes a N listas de forma que en cada disposición aparezca únicamente un elemento de cada lista:

Lo primero que se nos pasa por la cabeza al pensar este ejercicio es utilizar reduce para reducir la lista con combine-lst-lst.

Imaginemos que implementamos la función así y se la aplicamos a ((a b) (+ -) (1 2)). La primera iteración de reduce daría como resultado ((a +) (a -) (b +) (b -)), y la siguiente tendría que volver a aplicar combine-lst-lst a este resultado y (1 2). A primera vista parece que saldrá lo que queremos, pero lo que obtenemos es (((a +) 1) ((a +) 2) ((a -) 1) ...).

Nosotros querríamos exactamente este comportamiento, solo que en lugar de que se cree una nueva lista al combinar dos elementos, si el primero es una lista debería añadirse el segundo elemento al final.

Para conseguir esto, creamos una versión alternativa de combine-lst-lst y combine-elt-lst. La diferencia única diferencia entre estas versiones y las originales es que combine-elt-lst-alt comprueba si el elemento que se le pasa es una lista bien formada, y si lo es, le concatenamos el elemento oportuno al final.

```
(defun combine-elt-lst-alt (ele lst)
    (mapcar #'(lambda (list1)
                (if (proper-list ele)
                        (append ele (list list1))
                (list ele list1)))
            lst))


(defun combine-lst-lst-alt (lst1 lst2)
    (mapcan #'(lambda (e)
                (combine-elt-lst-alt e lst2))
            lst1))
```

Por supuesto, para poder hacer esto, hemos tenido que implementar una función proper-list que comprobara si algo es una lista bien formada (ya que listp en realidad hace lo mismo que consp solo que devolviendo T para NIL).

```
(defun proper-list(l)
    (or (null l)
            (and (consp l)
                    (proper-list (rest l)))))
```

Además, añadimos una condición extra en combine-list-of-lsts para que actúe correctamente cuando la lista de listas sólo contiene una lista.

```
(defun combine-list-of-lsts (lstolsts)
    (if (null (rest lstolsts))
                (mapcar #'list (first lstolsts))
            (reduce #'combine-lst-lst-alt lstolsts)))
```

4 Notas

En cuanto a los casos de prueba realizados, hemos probado en todas las funciones todos aquellos casos que las harían fallar, como puede ser el introducir vectores nulos en el ejercicio 1 o una lista con listas vacías dentro en el ejercicio 3. Todos estos casos de error están manejados mediante el control de errores de cada función, donde devolvemos nil en caso de que se produzca uno de estos errores en las funciónes, salvo en el ejercicio 1 apartado 1, donde devolvemos 0 en caso de que uno de los vectores que nos pasen sea null, para reflejar que su similitud coseno es nula.