Programación para la Inteligencia Artificial
Tarea 3

Arturo Márquez Flores
Maestría en Inteligencia Artificial
Universidad Veracruzana
CIIA – Centro de Investigación en Inteligencia Artificial
Sebastián Camacho No 5, Xalapa, Ver., México 91000
arturomf94@gmail.com
https://github.com/arturomf94/pia-mia

7 de Diciembre del 2018

1. Pregunta 1

El código para responder la primera pregunta es el siguiente, en el Cuadro 1.

```
;;; Esta funcion tiene como atomo una lista
;;; Funciona con un cond, cuyo primer caso
;;; es cuando la lista esta vacia. En el
;;; segundo caso el cuerpo de la lisa esta
;;; vacia y el tercer caso se cumple siempre
;;; En el primero se regresa nil. En el
;;; segundo se regresa una lista que contiene
;;; la lista original. El tercero hace un loop
;;; sobre los atomos de la lista y para cada
;;; caso hace una llamada recursiva con el
;;; resto de la lista. Estos resultados se unen.

(defun perms (lst)
  (cond ((null lst) nil)
        ((null (cdr lst)) (list lst))
        (t (loop for atom in lst
             append (mapcar (lambda (l) (cons atom l))
                            (perms (remove atom lst)))))))
```

Cuadro 1: Implementación en Lisp para la Pregunta 1

En el Cuadro 2 podemos ver el resultado de su ejecución.

```
;;; CL-USER> (perms '(1 2 3))
;;;
;;; ((1 2 3) (1 3 2) (2 1 3) (2 3 1) (3 1 2) (3 2 1))
;;;
```

Cuadro 2: Ejecución de la Implementación en Lisp para la Pregunta 1

2. Pregunta 2

El código para responder la segunda pregunta es el siguiente, en el Cuadro 3.

```
;;; Esta funcion toma un atomo y una lista
;;; Se implementa un cond y se consideran
;;; los casos donde la lista esta vacia,
;;; donde la cabeza de la lista es igual
;;; al atomo y un caso que siempre se
;;; cumple. El primero regresa la lista
;;; inalterada. El segundo hace una llamada
;;; recursiva con solo el cuerpo de la lista.
;;; El tercero concatena la cabeza de la
;;; lista con una llamada recursiva limitada
;;; al cuerpo.

(defun eliminar (atom lst)
  (cond ((null lst) lst)
    ((eql (car lst) atom) (eliminar atom (cdr lst)))
    (t (append (list(car lst)) (eliminar atom (cdr lst))))))
```

Cuadro 3: Implementación en Lisp para la Pregunta 2

En el Cuadro 4 podemos ver el resultado de su ejecución.

```
;;; CL-USER> (eliminar 3 '(1 3 2 4 5 3 6 7))
;;;
;;; (1 2 4 5 6 7)
;;;
```

Cuadro 4: Ejecución de la Implementación en Lisp para la Pregunta 2

3. Pregunta 3

El código para responder la tercera pregunta es el siguiente, en los Cuadros 5 y 6.

```
1
2       ;;; En el primer caso, la segunda lista esta vacia
3       ;;; y el resultado es nil. En el segundo, la
4       ;;; primera lista esta vacia y se regresa el valor T
5       ;;; Por ultimo, si la cabeza de la primera es miembro
6       ;;; de la segunda se llama recursivamente la funcion
7       ;;; con el cuerpo de la primera y la segunda.
8
9       (defun subset (lst1 lst2)
10        (cond ((null lst2) nil)
11          ((null lst1) T)
12          ((member (car lst1) lst2) (subset (cdr lst1) lst2))))
13
14      ;;; En el primer caso si cualquier lista esta vacia
15      ;;; se regresa la lista vacia. En el segundo caso si
16      ;;; la cabeza de la primera lista esta en la segunda
17      ;;; entonces se juntan la cabeza de la primera y el
18      ;;; resultado de una llamada recursiva a inter con el
19      ;;; cuerpo de la primera lista y la segunda completa.
20      ;;; Por ultimo, en cualquier otro caso se hace la
21      ;;; llamada recursiva con el cuerpo de la primera y la
22      ;;; segunda completa.
23
24      (defun inter (lst1 lst2)
25        (cond ((or (null lst1) (null lst2)) '())
26          ((member (car lst1) lst2) (append (list(car lst1)) (inter (cdr lst1) lst2)))
27          (t (inter (cdr lst1) lst2))))
```

Cuadro 5: Ejecución de la Implementación en Lisp para la Pregunta 2

```
1
2       ;;; Si la primera lista esta vacia se regresa la segunda
3       ;;; lista. Si la segunda esta vacia se regresa la primera.
4       ;;; Si la cabeza de la primera esta en la segunda, entonces
5       ;;; se hace una llamada recursiva con el cuerpo de la
6       ;;; primera y la segunda. En cualquier otro caso se junta
7       ;;; la cabeza de la primera con una llamada recursiva
8       ;;; igual a la anterior.
9
10      (defun myunion (lst1 lst2)
11        (cond ((null lst1) lst2)
12          ((null lst2) lst1)
13          ((member (car lst1) lst2) (myunion (cdr lst1) lst2))
14          (t (append (list(car lst1)) (myunion (cdr lst1) lst2)))))
15
16      ;;; Si la primera lista esta vacia se regresa la lista
17      ;;; vacia. Si la segunda esta vacia se regresa la primera.
18      ;;; En el caso en el cual la cabeza de la primera esta
19      ;;; en la segunda, se hace la llamada recursiva con
20      ;;; el cuerpo de la primera y la segunda (eliminando la
21      ;;; cabeza de la primera). En cualquier otro caso se junta
22      ;;; la cabeza de la primera lista con la llamada recursiva
23      ;;; del cuerpo de la primera lista y la segunda.
24
25      (defun dif (lst1 lst2)
26        (cond ((null lst1) '())
27          ((null lst2) lst1)
28          ((member (car lst1) lst2) (dif (cdr lst1) (remove (car lst1) lst2)))
29          (t (append (list(car lst1)) (dif (cdr lst1) lst2)))))
```

Cuadro 6: Ejecución de la Implementación en Lisp para la Pregunta 2

En el Cuadro 7 podemos ver el resultado de su ejecución.

```
;;; CL-USER> (subset '(1 3) '(1 2 3 4))
;;; T
;;;
;;; CL-USER> (inter '(1 2 3) '(2 3 4))
;;; (2 3)
;;;
;;; CL-USER> (union '(1 2 3 4) '(2 3 4 5))
;;; (1 2 3 4 5)
;;;
;;; CL-USER> (dif '(1 2 3 4) '(2 3 4 5))
;;; (1)
;;;
;;; CL-USER> (dif '(1 2 3) '(1 4 5))
;;; (2 3)
```

Cuadro 7: Ejecución de la Implementación en Lisp para la Pregunta 3

4. Pregunta 4

El código para responder la tercera pregunta es el siguiente, en los Cuadros 8 y 9.

```
1
2       ;;; Regresa la cabeza de la lista tree
3       (defun root (tree)
4         (car tree))
5
6       ;;; Regresa la cabeza del cuerpo de la lista tree
7       (defun lbranch (tree)
8         (car (cdr tree)))
9
10      ;;; Regresa la cabesa del cuerpo del cuerpo de tree
11      (defun rbranch (tree)
12        (car (cdr (cdr tree))))
13
14      ;;; Esta funcion convierte una lista a un arbol.
15      ;;; Si la lista esta vacia regresa nil. Si el cuerpo
16      ;;; de la lista esta vacio, regresa una lista con la
17      ;;; cabeza de la lista y dos elementos nil. En cualquier
18      ;;; otro caso se regresa la lista compuesta de la cabeza
19      ;;; de la lista, nil y la llamada recursiva con el cuerpo
20      ;;; de la lista.
21      (defun list2tree (lst)
22        (cond ((null lst) nil)
23          ((null (cdr lst)) (list (car lst) nil nil))
24          (t (list (car lst) nil (list2tree (cdr lst))))))
25
26      ;;; Esta funcion agrega una hoja al arbol. Si la rama
27      ;;; derecha del arbol esta vacia entonces regresa el mismo
28      ;;; arbol pero con (elt nil nil) como rama derecha. En otro
29      ;;; caso regresa el mismo arbol pero la rama derecha se
30      ;;; sustituye con una llamada recursiva con la rama derecha
31      ;;; del arbol.
32      (defun add_elt_tree (elt tree)
33        (cond
34          ((null (rbranch tree))
35           (list (root tree) (lbranch tree) (list elt nil nil)))
36          (t
37           (list (root tree) (lbranch tree) (add_elt_tree elt (rbranch tree))))))
```

Cuadro 8: Ejecución de la Implementación en Lisp para la Pregunta 2

```
1
2       ;;; Las funciones in_order, pre_order y post_order
3       ;;; imprimen los elementos del arbol en ordenes diferentes.
4
5
6       ;;; in_order imprime primero la rama izquierda del arbol
7       ;;; luego la raiz y finalmente la rama derecha. Este orden
8       ;;; se repite en las llamadas recursivas para imprimir las
9       ;;; ramas.
10      (defun in_order (tree)
11        (if (not (null (lbranch tree))) (in_order (lbranch tree)))
12        (print (root tree))
13        (if (not (null (rbranch tree))) (in_order (rbranch tree))))
14
15      ;;; pre_order imprime primero la raiz del arbol
16      ;;; luego la rama izquierda y finalmente la rama derecha. Este orden
17      ;;; se repite en las llamadas recursivas para imprimir las
18      ;;; ramas.
19      (defun pre_order (tree)
20        (print (root tree))
21        (if (not (null (lbranch tree))) (pre_order (lbranch tree)))
22        (if (not (null (rbranch tree))) (pre_order (rbranch tree))))
23
24      ;;; post_order imprime primero la rama izquierda del arbol
25      ;;; luego la rama derecha y finalmente la raiz. Este orden
26      ;;; se repite en las llamadas recursivas para imprimir las
27      ;;; ramas.
28      (defun post_order (tree)
29        (if (not (null (lbranch tree))) (post_order (lbranch tree)))
30        (if (not (null (rbranch tree))) (post_order (rbranch tree)))
31        (print (root tree)))
```

Cuadro 9: Ejecución de la Implementación en Lisp para la Pregunta 2

En el Cuadro 10 podemos ver el resultado de su ejecución.

```
;;; CL-USER> (setq tree '(1 (2 nil nil) (3 nil nil)))
;;; (1 (2 NIL NIL) (3 NIL NIL))
;;;
;;; CL-USER> (root tree)
;;; 1
;;;
;;; CL-USER> (lbranch tree)
;;; (2 NIL NIL)
;;;
;;; CL-USER> (rbranch tree)
;;; (3 NIL NIL)
;;;
;;; CL-USER> (setq tree (list2tree '(1 2 3)))
;;; (1 NIL (2 NIL (3 NIL NIL)))
;;;
;;; CL-USER> (setq tree (add_elt_tree 4 tree))
;;; (1 NIL (2 NIL (3 NIL (4 NIL NIL))))
;;;
;;; CL-USER> (in_order tree)
;;; 1
;;; 2
;;; 3
;;; 4
;;; NIL
;;;
;;; CL-USER> (pre_order tree)
;;; 1
;;; 2
;;; 3
;;; 4
;;; NIL
;;;
;;; CL-USER> (post_order tree)
;;; 4
;;; 3
;;; 2
;;; 1
;;; 1
```

Cuadro 10: Ejecución de la Implementación en Lisp para la Pregunta 3

5. Pregunta 5

El código para responder la primera pregunta es el siguiente, en el Cuadro 11.

```
;;; Esta macro esta basada en el macro for
;;; de las notas largas del curso. Para
;;; modificarlo se fijo la variable start
;;; en 1 y la variable stop ahora es n.
(defmacro nreps (x n &body body)
  (let ((gstop (gensym)))
    `(do ((,x 1 (1+ ,x))
          (,gstop ,n))
         ((> ,x ,gstop))
       (print ,@body))))
```

Cuadro 11: Implementación en Lisp para la Pregunta 5

En el Cuadro 12 podemos ver el resultado de su ejecución.

```
;;; CL-USER> (nreps x 4 (incf x))
;;; 2
;;; 4
;;; NIL
```

Cuadro 12: Ejecución de la Implementación en Lisp para la Pregunta 5

En este caso el resultado difiere con aquel mostrado en el documento de la tarea. Esto es porque en cada valor de x (comenzando en 1) la expresión *incf* incrementa el valor en uno. Por lo tanto el primer output es 2. A este valor es incrementado también por *nreps* en una unidad. Sin embargo en el ejemplo del Cuadro 13 podemos observar que *nreps* sí funciona con otro tipo de expresiones, como *cos x*.

```
;;; CL-USER> (nreps x 4 (cos x))
;;;
;;; 0.54030230
;;; -0.41614684
;;; -0.9899925
;;; -0.6536436
;;; NIL
```

Cuadro 13: Ejecución de la Implementación en Lisp para la Pregunta 5

6. Pregunta 6

Para esta pregunta se modificó la implementación del algoritmo ID3 en Lisp de la misma manera en que se modificó en la tarea pasada. Como se menciona en [1], en la sección 5, una de las limitaciones del modelo del ID3 presentado es el hecho de que el algoritmo no puede procesar datos que sean conflictivos; es decir, que contengan ruido.

Una de las soluciones planteadas por el autor es la posibilidad de generalizar la noción de clase como un número entre 0 y 1. Con esto, podríamos interpretar el resultado como la probabilidad de que una observación con los atributos correspondientes pertenezca a dicha clase.

En esta sección se muestran las modificaciones pertinentes al archivo *cl-id3-algorithm.lisp* para poder lograr este cambio en la noción de clase. Para tener un punto de referencia fijo, utilizamos los mismos ejemplos vistos en clase y tratados en [1]. En la Figura 1 observamos estos ejemplos.

1	cielo	temperatura	humedad	viento	jugarTenis
2	soleado	alta	alta	debil	no
3	soleado	alta	alta	fuerte	no
4	nublado	alta	alta	debil	si
5	lluvioso	templada	alta	debil	si
6	lluvioso	fresca	normal	debil	si
7	lluvioso	fresca	normal	fuerte	no
8	nublado	fresca	normal	fuerte	si
9	soleado	templada	alta	debil	no
10	soleado	fresca	normal	debil	si
11	lluvioso	templada	normal	debil	si
12	soleado	templada	normal	fuerte	si
13	nublado	templada	alta	fuerte	si
14	nublado	alta	normal	debil	si
15	lluvioso	templada	alta	fuerte	no

Figura 1: Ejemplos Originales

En los Cuadros 14 y 15 observamos los cambios al código relevantes.

```
1
2       ;;; Se agrego la funcion count-instance
3       ;;; para contar todas las instancias de un
4       ;;; atomo en una lista.
5       (defun count-instance (a L)
6         (cond
7          ((null L) 0)
8          ((equal a (car L)) (+ 1 (count-instance a (cdr L))))
9          (t (count-instance a (cdr L)))))
10
11      ;;; La funcion count-instance-prop expresa
12      ;;; el conteo de count-instance como proporcion
13      ;;; de la longitud de la lista.
14      (defun count-instance-prop (a L)
15        (/ (count-instance a L) (list-length L)))
16
17      ;;; La funcion list-to-string formatea
18      ;;; una lista para poder expresarla como
19      ;;; cadena.
20      (defun list-to-string (lst)
21        (format nil "~A~%" lst))
```

Cuadro 14: Modificaciones al Código

```
;;; id3

(defun id3 (examples attribs)
  "It induces a decision tree running id3 over EXAMPLES and ATTRIBS)"
  ;;; Se agrego la variable vals que crea la lista de todos los
  ;;; valores de *target* en examples
  (let ((class-by-default (get-value *target*
                                     (car examples)))
        (vals (mapcar #'(lambda(x) (get-value *target* x)) examples)))
    (cond
      ;; Stop criteria
      ;;; Se modifico este criterio para que regresara la clase en
      ;;; cuestion y la propocion de clasificaciones de esa clase.
      ;;; En este caso la proporcion siempre sera 1.
      ((same-class-value-p *target*
                           class-by-default
                           examples) (list-to-string
                     (list class-by-default
                           (count-instance-prop class-by-default vals))))
      ;; Failure
      ;;; Tambien se modifico este criterio para que regresara
      ;;; la clase en cuestion y la proporcion que representa esa
      ;;; clase en ese nodo. En este caso esa proporcion siempre es
      ;;; menor que 1.
      ((null attribs) (list-to-string
              (list class-by-default
                    (count-instance-prop class-by-default vals))))
      ;; Recursive call
      (t (let* ((partition (best-partition attribs examples))
            (node (first partition)))
       (cons node
             (loop for branch in (cdr partition) collect
                   (list (first branch)
                         (id3 (cdr branch)
                              (remove node attribs)))))))))
```

Cuadro 15: Modificaciones al Código

Con estas modificaciones, los resultados (en la interfaz gráfica) de utilizar la base original se muestran en la Figura 2.

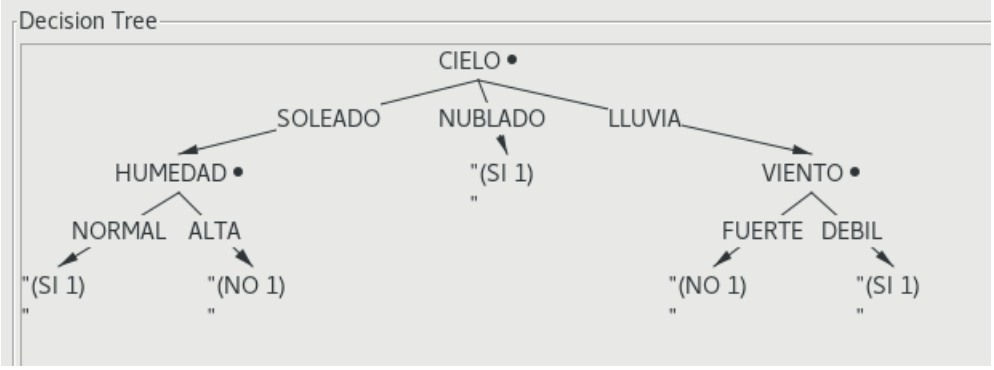

Figura 2: Resultado con Datos Originales

Los ejemplos en 2, siguiendo el ejemplo en [1], modifican la observación 1 en la columna *cielo*, creando un conflicto entre la observación 1 y la observación 3.

1	cielo	temperatura	humedad	viento	jugarTenis
2	nublado	alta	alta	debil	no
3	soleado	alta	alta	fuerte	no
4	nublado	alta	alta	debil	si
5	lluvioso	templada	alta	debil	si
6	lluvioso	fresca	normal	debil	si
7	lluvioso	fresca	normal	fuerte	no
8	nublado	fresca	normal	fuerte	si
9	soleado	templada	alta	debil	no
10	soleado	fresca	normal	debil	si
11	lluvioso	templada	normal	debil	si
12	soleado	templada	normal	fuerte	si
13	nublado	templada	alta	fuerte	si
14	nublado	alta	normal	debil	si
15	lluvioso	templada	alta	fuerte	no

Figura 3: Ejemplos Modificados

Utilizando el algoritmo modificado podemos procesar los ejemplos de 3. En los Cuadros 4 y 5 podemos ver los resultados en la rama izquierda y derecha, respectivamente.

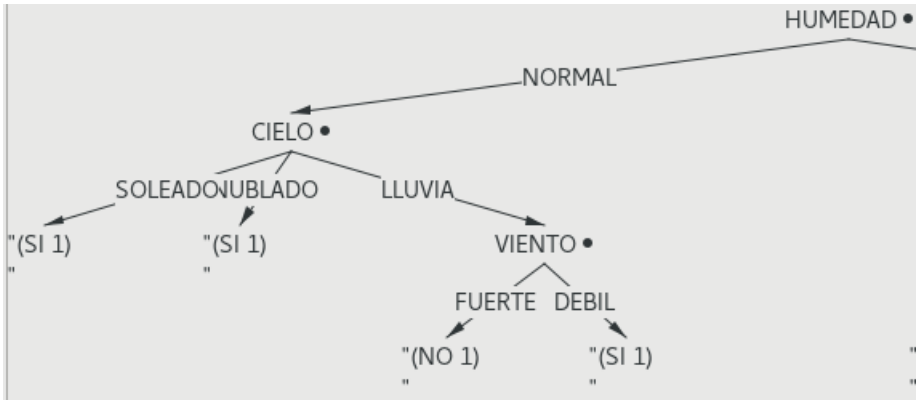

Figura 4: Resultado con Datos Modificados

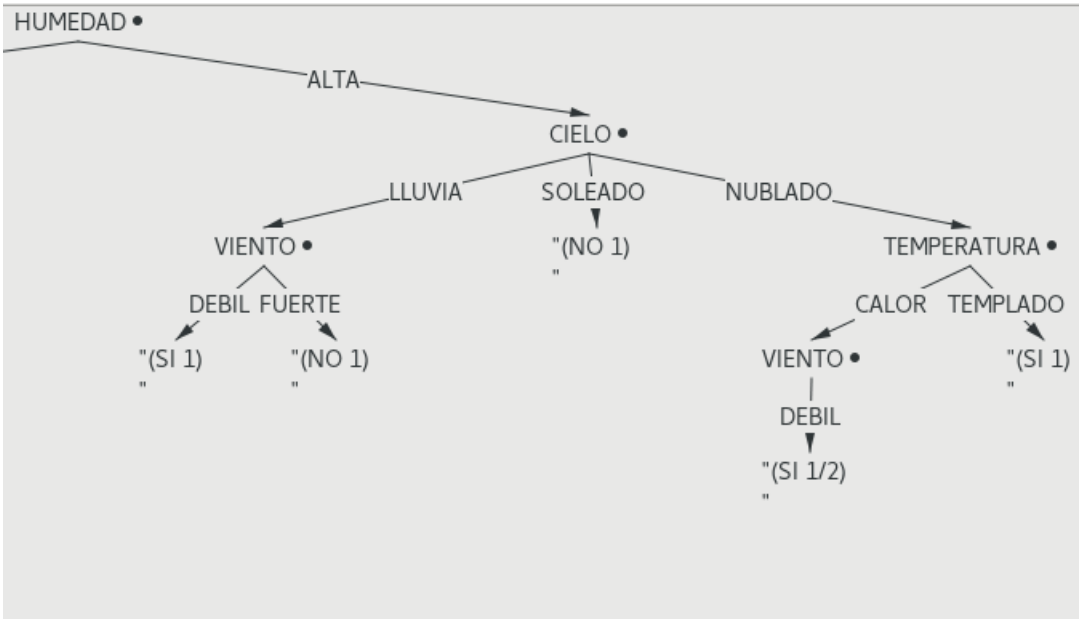

Figura 5: Resultado con Datos Modificados

Podemos observar que ahora el árbol de decisión expresa todas las clases con un número del 0 al 1. En particular, el caso conflictivo ahora se expresa como un *SI* con probabilidad $\frac{1}{2}$.

Referencias

[1] Quinlan, *Induction of Decision Trees*, Machine Learning 1: 81-106, 1986;