7 LISP EN LA IA

En este capítulo revisaremos diversas aplicaciones de Lisp en problemas propios de la IA. Comenzaremos con un problema de bioinformática, consistente en establecer cuantitativamente la significancia [35] de los codones y sus propiedades físico-químiicas en el código genético. Este tema nos servirá para introducir el uso de paquetes e interfaces gráficas en Lisp, así como operaciones con matrices representadas como listas de listas. Continuaremos nuestras prácticas de Lisp con la implementación de nuestro bien conocido ID3 [69, 71]. Este tema ilustra también algunos aspectos no funcionales de Lisp introducidos en el capítulo anterior, como la lectura de lectura de archivos, el uso de librerías y la definición de sistemas ASDF. Finalmente, abordaremos el uso de Lisp para computar recomendaciones de productos y listas de recomendación [78] desde la perspectiva del filtrado colaborativo.

7.1 BIOINFORMÁTICA

Este proyecto de bioinformática tiene como objetivo cuantificar la significancia de los codones del código genético y sus propiedades físico-químicas [35]. La información genética se escribe, *grosso modo*, a partir de cuatro **bases**: adenina (A), guanina (G), timina (T) y citosina (C), agrupadas funcionalmente en tripletas $\langle N_1, N_2, N_3 \rangle$, donde $N_i \in \{A, G, T, C\}$, conocidas como **codones**. A las bases en un codón también se les conoce como **nucleótidos**. Por **propiedades fisico-químicas** entendemos aquí el tipo de base: Purinas (A,G) y pirimidinas (T, C); y el número de enlaces de hidrógeno: débil (2) y fuerte (3). Los codones traducen se traducen a **aminoácidos**, cuya identidad parece depender de la posición de los nucleótidos en el codón, así como de sus propiedades físico-químicas.

Base *Codón* *Nucleótido* *Propiedades* *Aminoácido*

La literatura propone diferentes ordenes de relevancia tanto para los nucleótidos, como para sus propiedades fisico-químicas. Este proyecto sigue la estrategia de utilizar matrices de similitud para cuantificar tal relevancia. La metodología resumida es la siguiente: Cada nucleótido tiene varios mapeos posibles que se aplican sistemáticamente al código genético estándar para cuantificar que posición resulta más afectada por estas **mutaciones**. Por más afectado queremos decir que las matrices de similitud nos dicen que tan parecido es el nuevo aminoácido al original. Se computa la **significancia** promedio en estos términos y se comparan los resultados para establecer la relevancia buscada.

Mutación *Significancia*

El resto de la sección está organizado como sigue: Introduciremos el concepto de **paquete** para organizar mejor nuestro código Lisp, ahora que abordamos una aplicación más compleja que las anteriores. Posteriormente resolveremos el problema planteado para después, diseñar una **interfaz gráfica** que facilite el uso de nuestro código.

Paquete *Interfaz gráfica*

7.1.1 Paquetes

Siendo ésta una aplicación más grande que los ejercicios previos del curso, conviene empaquetar esta aplicación. Los paquetes nos permiten organizar nuestros programas para evitar **conflictos** en los símbolos usados tanto en el REPL como por otras librerías. Además nos permiten configurar lo que sería la interfaz de programación de nuestra aplicación (**API**) de una librería.

Conflicto entre nombres

API

Antes de usar los paquetes es necesario entender sus limitaciones. Primero, los paquetes no proveen control directo sobre quién llama a qué función o accesa tal variable. El paquete sólo provee un control básico sobre espacios de nombres al controlar como es que el REPL convierte los nombres textuales en símbolos, pero más que el *reader* es el *evaluator* quien lleva a cabo esta tarea. Por eso no tiene sentido hablar de exportar una función o una variable desde un paquete. Podemos exportar símbolos para hacer ciertos nombres más fáciles de referenciar, pero el sistema de paquetes no permite restringir la manera en que esos nombres son usados.

Tomando esto en cuenta, podemos comenzar a experimentar. Los paquetes en Lisp se definen con la macro `defpackage` que permite no sólo crear un paquete nuevo, sino especificar que paquetes serán usados por el paquete nuevo; qué símbolos exporta; qué símbolos importa de otros paquetes; y crear símbolos sombreados *shadowed* para la resolución de conflictos.

Si queremos definir el paquete para nuestra aplicación bajo el nombre de `bioinfo` y especificar que usaremos el paquete para diseñar interfaces gráficas de de LispWorks (`CAPI`), entonces comenzaremos nuestro programa por:

```
1 (defpackage "BIOINFO"
2   (:add-use-defaults t)
3   (:use "CAPI"))
4
5 (in-package :bioinfo)
```

La expresión `(in-package :bioinfo)` hace que el paquete por default sea éste, en lugar de `CL-USER`. Si ustedes evalúan esta s-expresión en el REPL, verán que el prompt cambia a `BIOINFO>` para indicar que los símbolos definidos en el paquete están accesibles al usuario en el REPL. La variable especial `*package*` tiene el valor del paquete actual, por ejemplo:

```
1 CL-USER> *package*
2 #<The COMMON-LISP-USER package, 23/64 internal, 0/4 external>
3 CL-USER> (in-package "bioinfo")
4 #<The bioinfo package, 0/16 internal, 0/16 external>
5 BIOINFO> *package*
6 #<The bioinfo package, 0/16 internal, 0/16 external>
```

Un paquete puede verse como una tabla que estable una correspondencia entre cadenas de texto y símbolos de mi programa, aunque en realidad esta correspondencia es más flexible que una tabla. Los nombres de símbolos que hemos usado hasta ahora no están **calificados**. Para interpretarlos, el lenguaje convierte la cadena de texto a mayúsculas y se la pasa a la función `intern` que regresa el símbolo que el nombre representa en el paquete actual. Si el símbolo no existe, se crea uno nuevo con ese nombre. De manera que siempre que se escribe el mismo nombre en el mismo paquete, se recupera

Nombres calificados

el mismo símbolo. Los nombres calificados incluyen uno o dos símbolos de dos-puntos en su nombre. Cuando estos nombres son evaluados, la cadena de texto se parte en dos: Lo que está a la izquierda es el nombre de un paquete y lo de la derecha es el nombre del símbolo.

Símbolos externos

Un nombre con un solo símbolo de dos-puntos es un **símbolo externo**, uno que ha sido exportado explícitamente con la directiva `:export` como parte de la interfaz pública de un paquete. El símbolo dos-puntos se usa para referenciar cualquier símbolo de un paquete, aunque es altamente recomendable restringir su uso para respetar el diseño de interfaz pública provisto por el autor del paquete.

Los paquetes pueden organizarse en capas, usando `use-package` para heredar mapeos de otros paquetes, en nuestro ejemplo es el caso de `(:use CAPI)`. Para más información sobre paquetes, vean el capítulo 21 del libro de Seibel [80].

7.1.2 La resolución del problema

Código Genético Universal

Según Guerra-Hernández, Mora-Basáñez y Jiménez-Montaño [35], necesitamos representar el código genético estándar y las matrices de similitud en nuestro programa. Como éstas serán utilizadas en todo el programa, podemos definirlas como variables globales. La representación del **código genético universal**, puede ser una lista de listas donde cada listas miembro representa un par codón y aminoácido correspondiente:

```
1  (defvar *gc-tensor*
2    "The genetic code tensor A (universal genetic code)"
3    '(((t t t) phe) ((t c t) ser) ((t a t) tyr) ((t g t) cys)
4      ((t t c) phe) ((t c c) ser) ((t a c) tyr) ((t g c) cys)
5      ((t t a) leu) ((t c a) ser) ((t a a) ter) ((t g a) ter)
6      ((t t g) leu) ((t c g) ser) ((t a g) ter) ((t g g) trp)
7      ((c t t) leu) ((c c t) pro) ((c a t) his) ((c g t) arg)
8      ((c t c) leu) ((c c c) pro) ((c a c) his) ((c g c) arg)
9      ((c t a) leu) ((c c a) pro) ((c a a) gln) ((c g a) arg)
10     ((c t g) leu) ((c c g) pro) ((c a g) gln) ((c g g) arg)
11     ((a t t) ile) ((a c t) thr) ((a a t) asn) ((a g t) ser)
12     ((a t c) ile) ((a c c) thr) ((a a c) asn) ((a g c) ser)
13     ((a t a) ile) ((a c a) thr) ((a a a) lys) ((a g a) arg)
14     ((a t g) met) ((a c g) thr) ((a a g) lys) ((a g g) arg)
15     ((g t t) val) ((g c t) ala) ((g a t) asp) ((g g t) gly)
16     ((g t c) val) ((g c c) ala) ((g a c) asp) ((g g c) gly)
17     ((g t a) val) ((g c a) ala) ((g a a) glu) ((g g a) gly)
18     ((g t g) val) ((g c g) ala) ((g a g) glu) ((g g g) gly)))
```

Puesto que nuestro código define primero el paquete `:bioinfo` y nos movemos a él, `*gc-tensor*` es un símbolo interno de este paquete. Si regresamos al paquete `cl-user` y evaluamos `*gc-tensor*`, Lisp nos indicará que el símbolo no está acotado. Si queremos hacer referencia a él vía `bioinfo:*gc-tensor*`, Lisp nos avisará que este símbolo es interno al paquete `:bioinfo`. Si forzamos el asunto con `bioinfo::*gc-tensor*` obtendremos la matriz, pero con todos sus símbolos calificados explícitamente:

```
1  CL-USER> bioinfo::*gc-tensor*
2  (((T T T) BIOINFO::PHE) ((T BIOINFO::C T) BIOINFO::SER) ...
```

Si nos movemos a `:bioinfo`, la matriz puede accederse como se hace normalmente:

```
1  CL-USER> (in-package :bioinfo)
2  #<The BIOINFO package, 25/64 internal, 0/16 external>
3  BIOINFO> *gc-tensor*
4  (((T T T) PHE) ((T C T) SER) ...
```

Las matrices de similitud pueden definirse también como listas de listas, por ejemplo, La matriz de similitud de **Dayhoff** se codifica como sigue: *Matriz de Dayhoff*

```
1    (defvar *pam250*
2    ;;; Dayhoff PAM250 (percent accepted mutations) as reported
3    ;;; by Mac Donaill, Molecular Simulation 30(5) p.269
4    '(( 2 -2  0  0 -2  0  0  1 -1 -1 -2 -1 -1 -4  1  1  1 -6 -3  0)
5      (-2  6  0 -1 -4  1 -1 -3  2 -2 -3  3  0 -4  0  0 -1  2 -4 -2)
6      ( 0  0  2  2 -4  1  1  0  2 -2 -3  1 -2 -4 -1  1  0 -4 -2 -2)
7      ( 0 -1  2  4 -5  2  3  1  1 -2 -4  0 -3 -6 -1  0  0 -7 -4 -2)
8      (-2 -4 -4 -5 12 -5 -5 -3 -3 -2 -6 -5 -5 -4 -3  0 -2 -8  0 -2)
9      ( 0  1  1  2 -5  4  2 -1  3 -2 -2  1 -1 -5  0 -1 -1 -5 -4 -2)
10     ( 0 -1  1  3 -5  2  4  0  1 -2 -3  0 -2 -5 -1  0  0 -7 -4 -2)
11     ( 1 -3  0  1 -3 -1  0  5 -2 -3 -4 -2 -3 -5 -1  1  0 -7 -5 -1)
12     (-1  2  2  1 -3  3  1 -2  6 -2 -2  0 -2 -2  0 -1 -1 -3  0 -2)
13     (-1 -2 -2 -2 -2 -2 -2 -3 -2  5  2 -2  2  1 -2 -1  0 -5 -1  4)
14     (-2 -3 -3 -4 -6 -2 -3 -4 -2  2  6 -3  4  2 -3 -3 -2 -2 -1  2)
15     (-1  3  1  0 -5  1  0 -2  0 -2 -3  5  0 -5 -1  0  0 -3 -4 -2)
16     (-1  0 -2 -3 -5 -1 -2 -3 -2  2  4  0  6  0 -2 -2 -1 -4 -2  2)
17     (-3 -4 -3 -6 -4 -5 -5 -5 -2  1  2 -5  0  9 -5 -3 -3  0  7 -1)
18     ( 1  0  0 -1 -3  0 -1  0  0 -2 -3 -1 -2 -5  6  1  0 -6 -5 -1)
19     ( 1  0  1  0  0 -1  0  1 -1 -1 -3  0 -2 -3  1  2  1 -2 -3 -1)
20     ( 1 -1  0  0 -2 -1  0  0 -1  0 -2  0 -1 -3  0  1  3 -5 -3  0)
21     (-6  2 -4 -7 -8 -5 -7 -7 -3 -5 -2 -3 -4  0 -6 -2 -5 17  0 -6)
22     (-3 -4 -2 -4  0 -4 -4 -5  0 -1 -1 -4 -2  7 -5 -3 -3  0 10 -2)
23     ( 0 -2 -2 -2 -2 -2 -2 -1 -2  4  2 -2  2 -1 -1 -1  0 -6 -2  4)
24     ) "PAM250 matrix")
```

La matriz de similitud de Dayhoff `*pam250*` nos dice la similitud entre el aminoácido del renglón y la columna. Es necesario pués una función de **indexación** para buscar los aminoácidos en esta representación. Como los aminoácidos pueden ser referidos por una letra o tres letras, la función queda escrita como sigue: *Indexación*

```
1  (defun index (aa)
2    ;; Get the index in a matrix
3    (cond ((or (equal aa 'A)
4               (equal aa 'ALA)) 0)
5          ((or (equal aa 'R)
6               (equal aa 'ARG)) 1)
7          ((or (equal aa 'N)
8               (equal aa 'ASN)) 2)
9          ((or (equal aa 'D)
10              (equal aa 'ASP)) 3)
11         ((or (equal aa 'C)
12              (equal aa 'CYS)) 4)
13         ((or (equal aa 'Q)
14              (equal aa 'GLN)) 5)
15         ((or (equal aa 'E)
16              (equal aa 'GLU)) 6)
17         ((or (equal aa 'G)
18              (equal aa 'GLY)) 7)
```

```
19          ((or (equal aa 'H)
20               (equal aa 'HIS)) 8)
21          ((or (equal aa 'I)
22               (equal aa 'ILE)) 9)
23          ((or (equal aa 'L)
24               (equal aa 'LEU)) 10)
25          ((or (equal aa 'K)
26               (equal aa 'LYS)) 11)
27          ((or (equal aa 'M)
28               (equal aa 'MET)) 12)
29          ((or (equal aa 'F)
30               (equal aa 'PHE)) 13)
31          ((or (equal aa 'P)
32               (equal aa 'PRO)) 14)
33          ((or (equal aa 'S)
34               (equal aa 'SER)) 15)
35          ((or (equal aa 'T)
36               (equal aa 'THR)) 16)
37          ((or (equal aa 'W)
38               (equal aa 'TRP)) 17)
39          ((or (equal aa 'Y)
40               (equal aa 'TYR)) 18)
41          ((or (equal aa 'V)
42               (equal aa 'VAL)) 19)
43          ((equal aa 'ter) 20)))
```

de esta forma podemos recuperar los indices de los aminoácidos en las matrices de similitud:

```
1  BIOINFO> (index 'val)
2  19
3  BIOINFO> (index 'ter)
4  20
```

observen que estoy en el paquete `:bioinfo`.

Ahora, dada una matriz de similitud como la de Dayhoff, podemos calcular la distancia entre dos aminoácidos, con el cruce renglón/columna correspondiente. El único detalle aquí es que las matrices de similitud no computan mutaciones que involucran el terminador (`ter`), ya que son en extremo improbables. De forma que necesitamos condicionar el cálculo de la distancia a los casos que no involucran terminadores:

```
1  (defun get-dist (aai aaj &optional matrix)
2    ;;; Get pam250 value in PAM for aminoacid aai substituted by aaj
3    ;;; aminoacids can be coded with one letter or three letters.
4    ;;; ter to ter = 1 and amino to ter (and viceversa) = -8
5
6    (let ((matrix (cond ((null matrix) *pam250*)
7                        (t matrix))))
8      (cond ((and (equal aai 'ter) ;; ter to ter transition
9                  (equal aaj 'ter))
10            (cond ((equal matrix *pam250*) 1)
11                  ((equal matrix *codegen*) 0)
12                  ((equal matrix *miyazawa*) 0)
13                  ((equal matrix *prlic*) 0)
14                  ((equal matrix *blosum62*) 0)
15                  ((equal matrix *robersy-grau*)
16                   (nth (index aaj)
17                        (nth (index aai) matrix)))
18                  (t (error "matrix not defined"))))
```

```
19            ((or (equal aai 'ter) ;;; amino to ter or ter to amino
20                 (equal aaj 'ter))
21             (cond ((equal matrix *pam250*) -8)
22                   ((equal matrix *codegen*) 6)
23                   ((equal matrix *miyazawa*) -1.01)
24                   ((equal matrix *prlic*) 0)
25                   ((equal matrix *blosum62*) 0)
26                   ((equal matrix *robersy-grau*)
27                    (nth (index aaj)
28                         (nth (index aai) matrix)))
29                   (t (error "matrix not defined"))))
30            (t (nth (index aaj)
31                    (nth (index aai) matrix))))))
```

De esta forma podemos saber que tan significativo es una mutación de `val` a `arg` según la matriz `*pam250*` (default) o usando otra matriz. El código completo define otras matrices como veremos a continuación. Si queremos saber la distancia entre dos aminoácidos usando la matriz `*blosum62*`, haremos una llamada como la segunda:

```
1 BIOINFO> (get-dist 'val 'arg)
2 -2
3 BIOINFO> (get-dist 'val 'arg *blosum62*)
4 -3
```

Ahora necesitamos funciones para manejar los nucleótidos y los aminoácidos. Esto es, dado un aminoácido, obtener sus tres nucleótidos y viceversa:

```
1  (defun aa-to-nnn (aa)
2    ;;; Gets the nucleotides of the aminoacid
3    (remove-if #'(lambda(aminoacid)
4                   (not (equal (cadr aminoacid) aa)))
5               *gc-tensor*))
6
7  (defun nnn-to-aa (nnn)
8    ;;; Gets the aminoacid from the nucleotides
9    (car (member-if #'(lambda(aminoacid)
10                       (equal (car aminoacid) nnn))
11                   *gc-tensor*)))
```

observen que estas funciones operan sobre la tabla `*gc-tensor*` por lo que explota su estructura para resolver la búsqueda. Por ejemplo:

```
1 BIOINFO> (aa-to-nnn 'val)
2 (((G T T) VAL) ((G T C) VAL) ((G T A) VAL) ((G T G) VAL))
3 BIOINFO> (nnn-to-aa '(g t t))
4 ((G T T) VAL)
```

El resto de las funciones están autodocumentadas en el código de esta sesión. Lo que resta es crear funciones para comunicar los resultados obtenidos: la interfaz con el usuario. Podemos hacer uso de `format`, por ejemplo, la siguiente función imprime un tensor:

```
1 (defun print-tensor (tensor)
2   (cond ((null tensor) t)
3         (t (progn
4              (format t "~a ~a ~a ~a ~%"
5                      (car tensor) (cadr tensor)
6                      (caddr tensor) (cadddr tensor))
7              (print-tensor (cddddr tensor))))))
```

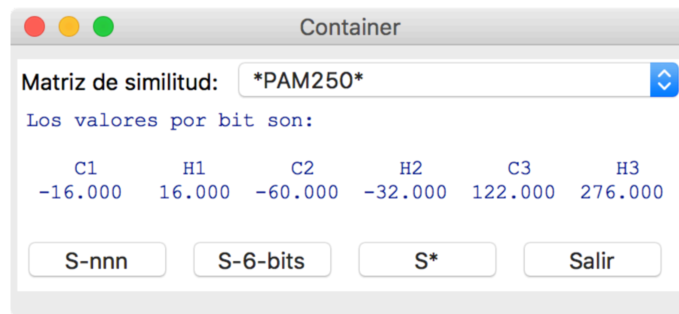


Figura 7.1: La interfaz gráfica de BIOINFO

de forma que:

```
1  BIOINFO> (print-tensor *gc-tensor*)
2  ((T T T) PHE) ((T C T) SER) ((T A T) TYR) ((T G T) CYS)
3  ((T T C) PHE) ((T C C) SER) ((T A C) TYR) ((T G C) CYS)
4  ((T T A) LEU) ((T C A) SER) ((T A A) TER) ((T G A) TER)
5  ((T T G) LEU) ((T C G) SER) ((T A G) TER) ((T G G) TRP)
6  ((C T T) LEU) ((C C T) PRO) ((C A T) HIS) ((C G T) ARG)
7  ((C T C) LEU) ((C C C) PRO) ((C A C) HIS) ((C G C) ARG)
8  ((C T A) LEU) ((C C A) PRO) ((C A A) GLN) ((C G A) ARG)
9  ((C T G) LEU) ((C C G) PRO) ((C A G) GLN) ((C G G) ARG)
10 ((A T T) ILE) ((A C T) THR) ((A A T) ASN) ((A G T) SER)
11 ((A T C) ILE) ((A C C) THR) ((A A C) ASN) ((A G C) SER)
12 ((A T A) ILE) ((A C A) THR) ((A A A) LYS) ((A G A) ARG)
13 ((A T G) MET) ((A C G) THR) ((A A G) LYS) ((A G G) ARG)
14 ((G T T) VAL) ((G C T) ALA) ((G A T) ASP) ((G G T) GLY)
15 ((G T C) VAL) ((G C C) ALA) ((G A C) ASP) ((G G C) GLY)
16 ((G T A) VAL) ((G C A) ALA) ((G A A) GLU) ((G G A) GLY)
17 ((G T G) VAL) ((G C G) ALA) ((G A G) GLU) ((G G G) GLY)
18 T
```

pero lo que realmente necesitamos es una interfaz gráfica que nos permita seleccionar la matriz de similitud que deseamos usar y el cómputo que queremos llevar a cabo.

7.1.3 Una interfaz gráfica

Después de trabajar con nuestro sistema en el REPL, lo ideal sería programar una interfaz gráfica (GUI) para esta aplicación. Aunque el diseño puede ser variado, la GUI debería permitirnos controlar la matriz de similitud a usar y el tipo de computo a llevar a cabo. Supongan que la interfaz gráfica deseada es como se muestra en la figura 7.1. ¿Cómo podemos implementar esta GUI? Eso depende de la implementación de Lisp utilizada y la librería de gráficos elegida. En lo que sigue utilizaremos LispWorks 6.1 y su CAPI (*Common Application Programmer's Interface*).

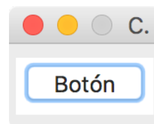
El CAPI es una librería para implementar interfaces de aplicaciones basadas en ventanas, portables a diferentes sistemas operativos. La interfaz se modela usando CLOS y para ello el CAPI provee cuatro clases de objetos básicos que incluyen interfaces, menus, paneles y formatos (*layouts*. La ayuda de Lispworks provee una descripción detallada de estas clases de objetos.

Para comenzar, necesitamos tener acceso a los símbolos de la librería CAPI, por eso al declarar el paquete `BIOINFO`, le pedímos que haga uso de esta librería con la línea `(:use "CAPI")`.

Una primera aproximación para crear le ventana principal y sus objetos es usando `contain`. En realidad lo ideal es definir interfaces con `define-interface` y desplegarlas con `display`, pero para el desarrollo rápido y pruebas puede usarse `contain`. Para crear una ventana con un botón se puede escribir:

```
1 (make-instance 'capi:push-button
2                :data "Button")
3
4 (capi:contain *)
```

y `LispWorks` desplegará la ventana:



`contain` provee un formato por default para cada elemento del CAPI que se especifique. En este caso se crea un botón y éste es desplegado dentro de la interfaz. Para que el botón tenga funcionalidad es necesario especificar el código que ejecutará mediante el slot `:callback`. Si queremos que nuestro botón despliegue en un panel el mensaje `Hola todos`, podemos escribir:

```
1 (make-instance 'capi:push-button
2                :data "Hello"
3                :callback
4                #'(lambda (&rest args)
5                          (capi:display-message
6                             "Hello World")))
7 (capi:contain *)
```

De esta forma podemos definir los botones de nuestra interfaz:

```
1  (setq run-s-nnn (make-instance 'push-button
2                            :text "S-nnn"
3                            :callback 's-nnn-message
4                            :visible-min-width '(:character 7)))
5
6  (setq run-s-6-bits (make-instance 'push-button
7                            :text "S-6-bits"
8                            :callback 's-6-bits-message
9                            :visible-min-width '(:character 7)))
10
11 (setq run-s* (make-instance 'push-button
12                               :text "S*"
13                               :callback 's*-message
14                               :visible-min-width '(:character 7)))
15
16 (setq exit (make-instance 'push-button
17                             :text "Salir"
18                             :callback #'(lambda (data interface)
19                                       (quit-interface interface))
20                             :visible-min-width '(:character 7)))
```

donde las funciones que ejecutal los `callback` están definidas por:

```
(defun s-nnn-message (data interface)
  (declare (ignore data interface))
  (apply-in-pane-process
   output
   #'(setf display-pane-text)
   (format nil  "Los valores por codón son: ~%~%~9:@<S1~>~9
                 :@<S2~>~9:@<S3~>~%~9:@<~6,3F~>~9:@<~6,3F~>~9:@<~6,3F~>"
           (S-nnn 1 (eval *option*))
           (S-nnn 2 (eval *option*))
           (S-nnn 3 (eval *option*)))
   output))

(defun s-6-bits-message (data interface)
  (declare (ignore data interface))
  (apply-in-pane-process
   output
   #'(setf display-pane-text)
   (format nil "Los valores por bit son:~%~%~9:@<C1~>~9:@<H1~>~9:
                @<C2~>~9:@<H2~>~9:@<C3~>~9:@<H3~>~%~9:@<~6,3F~>~9:
                @<~6,3F~>~9:@<~6,3F~>~9:@<~6,3F~>~9:@<~6,3F~>~9:
                @<~6,3F~>"
           (S-6bits 1 (eval *option*))
           (S-6bits 2 (eval *option*))
           (S-6bits 3 (eval *option*))
           (S-6bits 4 (eval *option*))
           (S-6bits 5 (eval *option*))
           (S-6bits 6 (eval *option*)))
   output))

(defun s*-message (data interface)
  (declare (ignore data interface))
  (apply-in-pane-process
   output
   #'(setf display-pane-text)
   (format nil  "Los valores por codón son: ~%~%~9:@<S1~>~9:@<S2~>
                ~9:@<S3~>~%~9:@<~6,3F~>~9:@<~6,3F~>~9:@<~6,3F~>"
           (S* 1 (eval *option*))
           (S* 2 (eval *option*))
           (S* 3 (eval *option*)))
   output))
```

Ahora podemos formatear los botones en un renglón:

```
(setq buttons
      (make-instance 'row-layout
         :description (list run-s-nnn run-s-6-bits run-s* exit)))
```

Para elegir la matriz de similitud que se usará, podemos usar un panel de opciones:

```
(defun set-option (data interface)
  (setq *option* data))

(setq options (make-instance 'option-pane
                             :items *options*
                             :selected-item *pam250*
                             :selection-callback 'set-option
                             :title "Matriz de similitud: "))
```

y para desplegar los resultados un panel de display llamado `output` porque es ahí donde las acciones de los botones despliegan sus resultados:

```
(setq output
   (make-instance 'display-pane
                   :font (gp:make-font-description
                   :family "Courier New"
                   :size 12)
                   :foreground :navy
                   :text '("Bienvenido a Bioinfo UV/CIIA aguerra")
                   :visible-min-height '(:character 5)))
```

Finalmente `contain` despliega toda la interfaz:

```
(contain
 (make-instance 'column-layout
                :description (list options output buttons)))
```

7.1.4 Creando un ejecutable

Una vez que hemos programado la interfaz gráfica de nuestra aplicación, y si contamos con la versión profesional de LispWorks, podemos crear un ejecutable de nuestra aplicación. Para ello es necesario escribir un script dependiente del sistema operativo que estamos usando. En el caso de macOS, el script es como sigue:

```
;;; Automatically generated delivery script

(in-package "CL-USER")

(load-all-patches)

;;; Load the application:

(compile-file "bioinfo.lsp")
(load "bioinfo")

;;; Load the example file that defines WRITE-MACOS-APPLICATION-BUNDLE
;;; to create the bundle.

(compile-file (sys:example-file
                       "configuration/macos-application-bundle.lisp") :load t)

(deliver 'bioinfo::start
         (when (save-argument-real-p)
           (write-macos-application-bundle
            "bioinfo.app"))
         0
         :interface :capi)
```

Al llamar a este script desde LispWorks, creamos una aplicación ejecutable que incluye la interfaz gráfica.

7.2 ARBOLES DE DECISIÓN

Recordemos la estructura de un árbol de decisión con la ayuda de la figura 7.2. Cada nodo del árbol está conformado por un **atributo** y puede verse como la pregunta: ¿Qué valor tiene este atributo en el caso que vamos a clasificar? Para este ejemplo en particular, la **raíz** del árbol pregunta ¿Cómo está

Atributo

Raíz

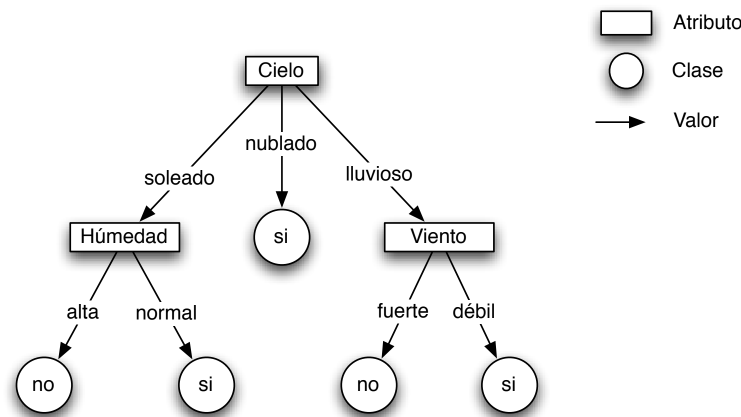


Figura 7.2: Un árbol para decidir si juego tenis o no. Adaptado de Mitchell [59], p. 59.

el cielo hoy? Las ramas que salen de cada nodo corresponden a los posibles valores del atributo en cuestión. Los nodos que no tienen hijos, se conocen como **hojas** y representan un valor de la **clase** que se quiere predecir; en este caso, si juego tenis o no.

Hoja/Clase

De esta forma, un árbol de decisión representa una hipótesis sobre el atributo clase, expresada como una disyunción de conjunciones del resto de los atributos proposicionales [1], usados para describir nuestro problema de decisión. Cada rama del árbol representa una conjunción de pares atributo-valor y el árbol completo es la **disyunción de esas conjunciones**.

Hipótesis disyntiva conjuntiva

Ejemplo 7.1. *La rama izquierda del árbol mostrado en la figura 7.2, expresa que no se juega tenis cuando el cielo está soleado y la humedad es alta.*

Los árboles de decisión pueden **representarse** naturalmente en Lisp como una lista de listas. A continuación definimos el árbol del ejemplo:

Representación

```
1  CL-USER> (setq arbol '(cielo
2                        (soleado (humedad
3                                  (normal-gc si)
4                                  (alta no)))
5                        (nublado si)
6                        (lluvia (viento
7                                 (fuerte no)
8                                 (debil si)))))
9  (CIELO (SOLEADO (HUMEDAD (NORMAL SI) (ALTA NO))) (NUBLADO SI)
10 (LLUVIA (VIENTO (FUERTE NO) (DEBIL SI))))
```

Una vez que hemos adoptado esta representación, es posible definir funciones de **acceso** y predicados adecuados para la manipulación del árbol.

Acceso

```
1  (defun root (tree)
2    (car tree))
3
4  (defun sub-tree (tree value)
5    (second (assoc attribute (cdr tree))))
6
```

1 Una versión extendida para utilizar representaciones de primer orden, propuesta por Blockeel y De Raedt [4], se conoce como árbol lógico de decisión.

```
7 (defun leaf-p (tree)
8   (atom tree))
```

De forma que podemos consultar la raíz de un árbol, el sub-árbol por debajo de cierta rama dado el valor de ésta y si un árbol es hoja:

```
1  CL-USER> (root arbol)
2  CIELO
3  CL-USER> (sub-tree arbol 'soleado)
4  (HUMEDAD (NORMAL SI) (ALTA NO))
5  CL-USER> (sub-tree arbol 'nublado)
6  SI
7  CL-USER> (leaf-p (sub-tree arbol 'soleado))
8  NIL
9  CL-USER> (leaf-p (sub-tree arbol 'nublado))
10 T
```

La función predefinida `assoc` busca el subárbol asociado al valor del atributo en cuestión, cuyo segundo elemento es el subárbol propiamente dicho. *assoc*

Ejemplo 7.2. *Consideren el siguiente ejemplo de su uso:*

```
1  CL-USER> (assoc 'uno '((uno 1) (dos 2) (tres 3)))
2 (UNO 1)
3  CL-USER> (assoc 'dos '((uno 1) (dos 2) (tres 3)))
4 (DOS 2)
```

Aunque la representación elegida del árbol de decisión resulta natural en Lisp, dificulta la su lectura por parte del usuario. Para resolver este problema, podemos escribir una función para **desplegar** el árbol de manera más legible (Observen el uso de la macro `loop` y `terpri` que produce una línea nueva): *Despliegue*

```
1  (defun print-tree (tree &optional (depth 0))
2    (mytab depth)
3    (format t "~A~%" (first tree))
4    (loop for subtree in (cdr tree) do
5          (mytab (+ depth 1))
6          (format t "- ~A" (first subtree))
7          (if (atom (second subtree))
8        (format t " -> ~A~%" (second subtree))
9        (progn (terpri)(print-tree (second subtree) (+ depth 5))))))
10
11 (defun mytab (n)
12   (loop for i from 1 to n do (format t " ")))
```

De forma que:

```
1  CL-USER> (print-tree arbol)
2  CIELO
3   - SOLEADO
4      HUMEDAD
5       - NORMAL -> SI
6       - ALTA -> NO
7   - NUBLADO -> SI
8   - LLUVIA
9      VIENTO
10      - FUERTE -> NO
11      - DEBIL -> SI
12 NIL
```

Día	Cielo	Temperatura	Humedad	Viento	Jugar-tenis?
1	soleado	calor	alta	débil	no
2	soleado	calor	alta	fuerte	no
3	nublado	calor	alta	débil	si
4	lluvia	templado	alta	débil	si
5	lluvia	frío	normal	débil	si
6	lluvia	frío	normal	fuerte	no
7	nublado	frío	normal	fuerte	si
8	soleado	templado	alta	débil	no
9	soleado	frío	normal	débil	si
10	lluvia	templado	normal	débil	si
11	soleado	templado	normal	fuerte	si
12	nublado	templado	alta	fuerte	si
13	nublado	calor	normal	débil	si
14	lluvia	templado	alta	fuerte	no

Cuadro 7.1: *Conjunto de ejemplos de entrenamiento para la clase jugar-tenis? Adaptado de Mitchell [59], p.59.*

7.2.1 Ejemplos de entrenamiento

Ahora bien, lo que queremos es **inducir** los árboles de decisión a partir de un conjunto de **ejemplos de entrenamiento**. Cada ejemplo en este conjunto representa un caso cuyo valor de clase conocemos. Se busca que el árbol inducido explique los ejemplos vistos y pueda predecir casos nuevos, cuyo valor de clase desconocemos. Para el árbol mostrado en la sección anterior los ejemplos con que fue construido se muestran en el Cuadro 7.1.

Inducción

Ejemplos de entrenamiento

Basados en la representación elegida para los árboles de decisión, podríamos representar los ejemplos como una lista de listas de valores para sus atributos:

```
1  CL-USER> (setq ejemplos
2                 '((SOLEADO CALOR ALTA DEBIL NO)
3                   (SOLEADO CALOR ALTA FUERTE NO)
4                   (NUBLADO CALOR ALTA DEBIL SI)
5                   (LLUVIA TEMPLADO ALTA DEBIL SI)
6                   (LLUVIA FRIO NORMAL DEBIL SI)
7                   (LLUVIA FRIO NORMAL FUERTE NO)
8                   (NUBLADO FRIO NORMAL FUERTE SI)
9                   (SOLEADO TEMPLADO ALTA DEBIL NO)
10                  (SOLEADO FRIO NORMAL DEBIL SI)
11                  (LLUVIA TEMPLADO NORMAL DEBIL SI)
12                  (SOLEADO TEMPLADO NORMAL FUERTE SI)
13                  (NUBLADO TEMPLADO ALTA FUERTE SI)
14                  (NUBLADO CALOR NORMAL DEBIL SI)
15                  (LLUVIA TEMPLADO ALTA FUERTE NO)))
```

Pero sería más útil identificar cada ejemplo por medio de una **llave** y poder acceder a sus atributos por nombre, por ejemplo, preguntar por el valor del `cielo` en el ejemplo 7. Las siguientes funciones son la base para esta estrategia:

Indexación

```
1  (defun put-value (attr inst val)
2    (setf (get inst attr) val))
3
```

```
4 (defun get-value (attr inst)
5   (get inst attr))
```

Su uso se ejemplifica en la siguiente sesión:

```
1  CL-USER> (put-value 'nombre 'ej1 'alejandro)
2  ALEJANDRO
3  CL-USER> (get-value 'nombre 'ej1)
4  ALEJANDRO
5  CL-USER> (setq *ejemplos* nil)
6  NIL
7  CL-USER> (push 'ej1 *ejemplos*)
8  (EJ1)
9  CL-USER> (get-value 'nombre (car *ejemplos*))
10 ALEJANDRO
```

Normalmente, los ejemplos de entrenamiento están almacenados en un archivo que Lisp no puede leer directamente, como una hoja de cálculo, una base de datos relacional o un archivo de texto con un formato definido. La primer decisión con respecto a los ejemplos de entrenamiento, es como serán cargados en Lisp.

Formatos de los archivos

El formato `ARFF` es usado por algunas herramientas de aprendizaje automático como Weka [93]. Esencialmente Weka usa un formato separado por comas y utiliza etiquetas para definir atributos y sus dominios (attribute, la clase (relation), comentarios (%) y el inicio de los datos (data). El conjunto de entrenamiento sobre jugar tenis quedaría representado en este formato como un archivo con extensión `.arff`:

```
1  @RELATION jugar-tenis
2
3  @ATTRIBUTE cielo {soleado,nublado,lluvia}
4  @ATTRIBUTE temperatura {calor,templado,frio}
5  @ATTRIBUTE humedad {alta,normal}
6  @ATTRIBUTE viento {debil,fuerte}
7  @ATTRIBUTE jugar-tenis {si,no}
8
9  @DATA
10
11 soleado, calor, alta, debil, no
12 soleado, calor, alta, fuerte, no
13 nublado, calor, alta, debil, si
14 lluvia, templado, alta, debil, si
15 lluvia, frio, normal, debil, si
16 lluvia, frio, normal, fuerte, no
17 nublado, frio, normal, fuerte, si
18 soleado, templado, alta, debil, no
19 soleado, frio, normal, debil, si
20 lluvia, templado, normal, debil, si
21 soleado, templado, normal, fuerte, si
22 nublado, templado, alta, fuerte, si
23 nublado, calor, normal, debil, si
24 lluvia, templado, alta, fuerte, no
```

También sería deseable que nuestro programa pudiera cargar conjuntos de entrenamiento en formato `CSV` (*comma separate values*), usado ampliamente en este contexto. En ese caso, el archivo anterior luciría como:

```
cielo, temperatura, humedad, viento, jugar-tenis
soleado, calor, alta, debil, no
soleado, calor, alta, fuerte, no
nublado, calor, alta, debil, si
lluvia, templado, alta, debil, si
...
```

Ambiente de aprendizaje

Primero, necesitamos declarar algunas variables globales, para identificar ejemplos, atributos y sus dominios, datos, etc. Esto configura nuestro ambiente de aprendizaje. Muchas de las funciones que definiremos necesitan tener acceso a estos valores, por ello, se definen usando `defvar` y la notación usual (por convención) de Lisp entre asteriscos:

```
;;; Global variables

(defvar *examples* nil "The training set")
(defvar *attributes* nil "The attributes of the problem")
(defvar *data* nil "The values of the atributes of all *examples*")
(defvar *domains* nil "The domain of the attributes")
(defvar *target* nil "The target concept")
(defvar *trace* nil "Trace the computations")
```

la semántica de estas variables es auto explicativa.

Lectura de archivos

Ahora tenemos dos opciones, leer el archivo usando alguna utilería del sistema operativo en el que estamos ejecutando Lisp, por ejemplo `grep` en Unix; ó programar directamente la lectura de archivos. Primero veremos la versión programada enteramente programada en Lisp. Comencemos por una función que nos permita obtener una lista de cadenas de caracteres, donde cada cadena corresponde con una línea del archivo.

```
(defun read-lines-from-file (file)
  (remove-if (lambda (x) (equal x ""))
             (with-open-file (in file)
               (loop for line = (read-line in nil 'end)
                  until (eq line 'end) collect line))))
```

Observen el uso de `loop` combinado con sus formas `until` y `collect`.

Algunas de las funciones que definiremos hacen uso de `split-sequence` que está definida en una librería no estándar de Lisp. Esto significa que ustedes tendrán que instalar la librería antes de poder compilar las definiciones listadas a continuación. La próxima sección aborda la instalación y uso de librerías.

```
CL-USER> (read-lines-from-file "tenis.arff")
("@RELATION jugar-tenis" "@ATTRIBUTE cielo {soleado,nublado,lluvia}"
"@ATTRIBUTE temperatura {calor,templado,frio}" "@ATTRIBUTE humedad
{alta,normal}" "@ATTRIBUTE viento {debil,fuerte}" "@ATTRIBUTE
jugar-tenis {si,no}" "soleado, calor, alta, debil, no" "soleado,
calor, alta, fuerte, no" "nublado, calor, alta, debil, si" "lluvia,
templado, alta, debil, si" "lluvia, frio, normal, debil, si" "lluvia,
frio, normal, fuerte, no" "nublado, frio, normal, fuerte, si"
"soleado, templado, alta, debil, no" "soleado, frio, normal, debil,
```

```
10 si" "lluvia, templado, normal, debil, si" "soleado, templado, normal,
11 fuerte, si" "nublado, templado, alta, fuerte, si" "nublado, calor,
12 normal, debil, si" "lluvia, templado, alta, fuerte, no")
```

Ahora necesitamos manipular la lista de cadenas obtenida, para instanciar adecuadamente las variables de nuestro ambiente de aprendizaje. Para el caso de los archivos ARFF estás son las funciones básicas:

```
1  (defun arff-get-target (lines)
2    "It extracts the value for *target* from the lines of a ARFF file"
3    (read-from-string
4     (cadr (split-sequence
5            #\Space
6            (car (remove-if-not
7                  (lambda (x) (or (string-equal "@r" (subseq x 0 2))
8                                  (string-equal "@R" (subseq x 0 2))))
9                  lines))))))
10
11 (defun arff-get-data (lines)
12   "It extracts the value for *data* from the lines of a ARFF file"
13   (mapcar #'(lambda(x)
14               (mapcar #'read-from-string
15                       (split-sequence #\, x)))
16           (remove-if
17            (lambda (x) (string-equal "@" (subseq x 0 1)))
18            lines)))
19
20 (defun arff-get-attribs-doms (lines)
21   " It extracts the list (attibutes domains) from an ARFF file"
22   (mapcar #'(lambda(x)
23               (list (read-from-string (car x))
24                     (mapcar #'read-from-string
25                             (split-sequence
26                              #\,
27                              (remove-if (lambda(x)
28                                           (or (string-equal "{" x)
29                                               (string-equal "}" x)))
30                                         (cadr x))))))
31           (mapcar #'(lambda(x)
32                       (cdr (split-sequence
33                             #\Space x)))
34                   (remove-if-not
35                    (lambda (x)
36                      (or (string-equal "@a" (subseq x 0 2))
37                          (string-equal "@A" (subseq x 0 2))))
38                    lines))))
```

Así, podemos extraer la clase y los datos del archivo ARFF como se muestra a continuación:

```
1  CL-ID3> (arff-get-target (read-lines-from-file "tenis.arff"))
2  JUGAR-TENIS
3  11
4  CL-ID3> (arff-get-data (read-lines-from-file "tenis.arff"))
5  ((SOLEADO CALOR ALTA DEBIL NO) (SOLEADO CALOR ALTA FUERTE NO) (NUBLADO
6  CALOR ALTA DEBIL SI) (LLUVIA TEMPLADO ALTA DEBIL SI) (LLUVIA FRIO
7  NORMAL DEBIL SI) (LLUVIA FRIO NORMAL FUERTE NO) (NUBLADO FRIO NORMAL
8  FUERTE SI) (SOLEADO TEMPLADO ALTA DEBIL NO) (SOLEADO FRIO NORMAL DEBIL
9  SI) (LLUVIA TEMPLADO NORMAL DEBIL SI) (SOLEADO TEMPLADO NORMAL FUERTE
10 SI) (NUBLADO TEMPLADO ALTA FUERTE SI) (NUBLADO CALOR NORMAL DEBIL SI)
11 (LLUVIA TEMPLADO ALTA FUERTE NO))
```

Evidentemente, necesitamos implementar versiones de estas funciones para el caso de que el archivo de entrada esté en formato CSV.

```
1  (defun csv-get-target (lines)
2    "It extracts the value for *target* from the lines of a CSV file"
3    (read-from-string
4     (car (last (split-sequence #\, (car lines))))))
5
6  (defun csv-get-data (lines)
7    "It extracts the value for *data* from the lines of a CSV file"
8    (mapcar #'(lambda(x)
9                (mapcar #'read-from-string
10                       (split-sequence #\, x)))
11           (cdr lines)))
12
13 (defun csv-get-attribs-doms (lines)
14   "It extracts the list (attibutes domains) from an CSV file"
15   (labels ((csv-get-values (attribs data)
16              (loop for a in attribs collect
17                   (remove-duplicates
18                    (mapcar #'(lambda(l)
19                                (nth (position a attribs) l))
20                            data)))))
21     (let* ((attribs (mapcar #'read-from-string
22                             (split-sequence #\, (car lines))))
23            (data (csv-get-data lines))
24            (values (csv-get-values attribs data)))
25       (mapcar #'list attribs values))))
```

La función principal para cargar un archivo e inicializar el ambiente de aprendizaje es la siguiente:

```
1  (defun load-file (file)
2    "It initializes the learning setting from FILE"
3    (labels ((get-examples (data)
4               (loop for d in data do
5                    (let ((ej (gensym "ej")))
6                      (setf *examples* (cons ej *examples*))
7                      (loop for attrib in *attributes*
8                         as v in d do
9                           (put-value attrib ej v))))))
10     (if (probe-file file)
11         (let ((file-ext (car (last (split-sequence #\. file))))
12               (file-lines (read-lines-from-file file)))
13           (reset)
14           (cond
15             ((equal file-ext "arff")
16              (let ((attribs-doms (arff-get-attribs-doms file-lines)))
17                (setf *attributes* (mapcar #'car attribs-doms))
18                (setf *domains* (mapcar #'cadr attribs-doms))
19                (setf *target* (arff-get-target file-lines))
20                (setf *data* (arff-get-data file-lines))
21                (get-examples *data*)
22                (format t "Training set initialized after ~s.~%" file)))
23             ((equal file-ext "csv")
24              (let ((attribs-doms (csv-get-attribs-doms file-lines)))
25                (setf *attributes* (mapcar #'car attribs-doms))
26                (setf *domains* (mapcar #'cadr attribs-doms))
27                (setf *target* (csv-get-target file-lines))
28                (setf *data* (csv-get-data file-lines))
29                (get-examples *data*)
```

```
30                (format t "Training set initialized after ~s.~%" file)))
31             (t (error "File's ~s extension can not be determined." file))))
32          (error "File ~s does not exist.~%" file))))
```

La función `reset` reinicia los valores de las variables globales que configuran el ambiente de aprendizaje:

```
1 (defun reset ()
2   (setf *data* nil
3         *examples* nil
4         *target* nil
5         *attributes* nil
6         *domains* nil
7         *root* nil
8         *gensym-counter* 1)
9   (format t "The ID3 setting has been reset.~%"))
```

En la distribución del sistema todas estas definiciones se encuentran en el archivo `cl-id3-load.lisp`. Con el código cargado en Lisp, podemos hacer lo siguiente:

```
1  CL-ID3> (load-file "tenis.arff")
2  The ID3 setting has been reset.
3  Training set initialized after "tenis.arff".
4  NIL
5  CL-ID3> *target*
6  JUGAR-TENIS
7  CL-ID3> *attributes*
8  (CIELO TEMPERATURA HUMEDAD VIENTO JUGAR-TENIS)
9  CL-ID3> *examples*
10 (#:|ej14| #:|ej13| #:|ej12| #:|ej11| #:|ej10| #:|ej9| #:|ej8| #:|ej7|
11 #:|ej6| #:|ej5| #:|ej4| #:|ej3| #:|ej2| #:|ej1|)
```

Como pueden observar, el ambiente de aprendizaje ha sido inicializado con los datos guardados en `tenis.arff`. Lo mismo sucedería para `tenis.csv`.

7.2.2 Clasificación a partir de un árbol de decisión

El procedimiento para clasificar un caso nuevo utilizando un árbol de decisión consiste en **filtrar** el ejemplo de manera ascendente, hasta encontrar una hoja que corresponde a la clase buscada. Consideren el proceso de clasificación del siguiente caso:

Filtrar

$$\langle Cielo = soleado, Temperatura = caliente, Humedad = alta, Viento = fuerte \rangle$$

Como el atributo `Cielo`, tiene el valor `soleado` en el ejemplo, éste es filtrado hacía abajo del árbol por la rama de la izquierda. Como el atributo `Humedad`, tiene el valor `alta`, el ejemplo es filtrado nuevamente por rama de la izquierda, lo cual nos lleva a la hoja que indica la clasificación del este nuevo caso: `Jugar-tenis? = no`. El Algoritmo 7.1 define la función `clasifica` para árboles de decisión.

La implementación en Lisp de este algoritmo, dada la representación del árbol adoptada y las funciones de acceso definidas, es la siguiente:

```
1 (defun classify (instance tree)
2   (let* ((val (get-value (root tree) instance))
3          (branch (sub-tree tree val)))
```

Algoritmo 7.1 Clasifica ejemplo E dado un árbol de decisión A.

```
1: function CLASIFICA(E: ejemplo, A: árbol)
2:     Clase ← valor-atributo(raíz(A),E);
3:     if es-hoja(raíz(A)) then
4:         return Clase
5:     else
6:         clasifica(E, sub-árbol(A,Clase));
7:     end if
8: end function
```

```
4      (if (leaf branch)
5          branch
6          (classify instance branch))))
```

7.2.3 Definiendo una librería ASDF: cl-id3

En la medida que vayamos completando nuestra implementación de ID3, el código se irá haciendo más grande. Será conveniente separarlo en varios archivos y definir las dependencias de compilación entre estos usando `ASDF`. El siguiente listado corresponde al archivo `cl-id3.asd` incluido en la distribución final de nuestro programa.

```
1  (asdf:defsystem :cl-id3
2      :depends-on (:split-sequence)
3      :components ((:file "cl-id3-package")
4       (:file "cl-id3-algorithm"
5        :depends-on ("cl-id3-package"))
6       (:file "cl-id3-load"
7        :depends-on ("cl-id3-package"
8               "cl-id3-algorithm"))
9       (:file "cl-id3-classify"
10       :depends-on ("cl-id3-package"
11              "cl-id3-algorithm"
12              "cl-id3-load"))
13      (:file "cl-id3-cross-validation"
14       :depends-on ("cl-id3-package"
15              "cl-id3-algorithm"
16              "cl-id3-load"
17              "cl-id3-classify"))
18      (:file "cl-id3-gui"
19       :depends-on ("cl-id3-package"
20              "cl-id3-algorithm"
21              "cl-id3-load"
22              "cl-id3-classify"
23              "cl-id3-cross-validation"))))
```

Esta definición de sistema establece que nuestra librería `cl-id3` depende de la librería `split-sequence`, de forma que ésta última se cargará en Lisp antes de intentar compilar y cargar nuestras fuentes. El orden en que nuestras fuentes son compiladas y cargadas se establece en los `:depends-on` de la definición.

7.2.4 Paquetes: cl-id3

Observen que la función `split-sequence` está definida dentro del paquete del mismo nombre, de forma que si queremos llamar a está función desde la consola de Lisp, debemos indicar el paquete al que pertenece. Esto es necesario porque por defecto estamos ubicados en el paquete `cl-user`.

Es conveniente definir un paquete para nuestra aplicación de forma que las funciones relevantes no se confundan con las definidas en el paquete `cl-user`. A continuación listamos el archivo `cl-id3-package.lisp`:

```
1  ;;; cl-id3-package
2  ;;; The package for cl-id3
3
4  (defpackage :cl-id3
5    (:use :cl :capi :split-sequence)
6    (:export :load-file
7       :induce
8       :print-tree
9       :classify
10      :classify-new-instance
11      :cross-validation
12      :gui))
```

Esta definición le indica a Lisp que el paquete `cl-id3` hace uso de los paquetes `common-lisp`, `capi` y `split-sequence`, por lo que no será necesario indicarlos al invocar sus funciones en nuestro código (observen que las funciones de carga de archivos llaman a `split-sequence`, sin indicar a que paquete pertenece). Los símbolos definidos bajo `:export` son visibles desde otros paquetes.

Es necesario que el resto de nuestros archivos fuente, incluyan como primer línea lo siguiente:

```
1  (in-package :cl-id3)
```

Una vez que el sistema `cl-id3` es compilado y cargado en Lisp, se puede usar como se muestra a continuación:

```
1  CL-USER> (cl-id3:load-file "tenis.arff")
2  The ID3 setting has been reset.
3  Training set initialized after "tenis.arff".
4  NIL
5  CL-USER> (cl-id3:induce)
6  (CIELO (SOLEADO (HUMEDAD (NORMAL SI) (ALTA NO))) (NUBLADO SI)
7  (LLUVIA (VIENTO (FUERTE NO) (DEBIL SI))))
```

Volvamos a la definición del algoritmo ID3.

7.2.5 ¿Qué atributo es el mejor clasificador?

La decisión central de ID3 consiste en seleccionar qué atributo colocará en cada nodo del árbol de decisión. En el algoritmo presentado, esta opción la lleva a cabo la función `mejor-partición`, que toma como argumentos un conjunto de ejemplos de entrenamiento y un conjunto de atributos, regresando la partición inducida por el atributo, que sólo, clasifica mejor los ejemplos de entrenamiento.

7.2.6 Particiones

Como pueden observar en la descripción del algoritmo ID3 (Algoritmo 4.2, página 109), una operación común sobre el conjunto de entrenamiento es la de partición con respecto a algún atributo. La idea es tener una función que tome un atributo y un conjunto de ejemplos y los particione de acuerdo a los valores observados del atributo, por ejemplo:

```
1  CL-USER> (in-package :cl-id3)
2  #<The CL-ID3 package, 148/512 internal, 7/16 external>
3  CL-ID3> (load-file "tenis.arff")
4  The ID3 setting has been reset.
5  Training set initialized after "tenis.arff".
6  NIL
7  CL-ID3> (get-partition 'temperatura *examples*)
8  (TEMPERATURA (FRIO #:|ej5| #:|ej6| #:|ej7| #:|ej9|)
9  (CALOR #:|ej1| #:|ej2| #:|ej3| #:|ej13|)
10 (TEMPLADO #:|ej4| #:|ej8| #:|ej10| #:|ej11| #:|ej12| #:|ej14|))
```

Lo que significa que el atributo `temperatura` tiene tres valores diferentes en el conjunto de entrenamiento: `frío`, `calor` y `templado`. Los ejemplos 5,6,7 y 9 tienen como valor del atributo `temperatura= frío`, etc. La definición de la función `get-partition` es como sigue:

```
1  (defun get-partition (attrib examples)
2    "It gets the partition induced by ATTRIB in EXAMPLES"
3    (let (result vlist v)
4      (loop for e in examples do
5            (setq v (get-value attrib e))
6            (if (setq vlist (assoc v result))
7                ;;; value v existed, the example e is added
8                ;;; to the cdr of vlist
9                (rplacd vlist (cons e (cdr vlist)))
10               ;;; else a pair (v e) is added to result
11               (setq result (cons (list v e) result))))
12     (cons attrib result)))
```

el truco está en el `if` de la línea 6, que determina si el valor del atributo en el ejemplo actual es un nuevo valor o uno ya existente. Si se trata de un nuevo valor lo inserta en `result` como una lista `(valor ejemplo)`. Si ya existía, `rplacd` se encarga de remplazar el `cdr` de la lista `(valor ejemplo)` existente, agregando el nuevo ejemplo: `(valor ejemplo ejemplo-nuevo)`.

Necesitaremos una función `best-partition` que encuentre el atributo que mejor separa los ejemplos de entrenamiento de acuerdo a la clase buscada ¿En qué consiste una buena medida cuantitativa de la bondad de un atributo? Para contestar a esta cuestión, definiremos una propiedad estadística llamada ganancia de información.

Entropía y ganancia de información

Recordemos que una manera de cuantificar la bondad de un atributo en este contexto, consiste en considerar la cantidad de información que proveerá este atributo, tal y como ésto es definido en la teoría de información de Shannon y Weaver [81]. Un bit de información es suficiente para determinar el valor de un atributo booleano, por ejemplo, si/no, verdader/falso, 1/0, etc., sobre el cual no sabemos nada. En general, si los posibles valores del atributo

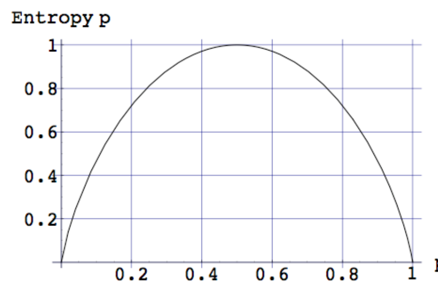


Figura 7.3: *Gráfica de la función entropia para clasificaciones booleanas.*

v_i, ocurren con probabilidades $P(v_i)$, entonces en contenido de información, o **entropía**, E de la respuesta actual está dado por:

Entropía

$$E(P(v_i), \ldots, P(v_n)) = \sum_{i=1}^{n} -P(v_i) \ \log_2 \ P(v_i)$$

Consideren nuevamente el caso booleano, aplicando esta ecuación a un volado con una moneda confiable, tenemos que la probabilidad de obtener aguila o sol es de $1/2$ para cada una:

$$E(\frac{1}{2}, \frac{1}{2}) = -\frac{1}{2} \ \log_2 \ \frac{1}{2} - \frac{1}{2} \ \log_2 \ \frac{1}{2} = 1$$

Ejecutar el volado nos provee 1 bit de información, de hecho, nos provee la clasificación del experimento: si fue aguila o sol. Si los volados los ejecutamos con una moneda cargada que da 99 % de las veces sol, entonces $E(1/100, 99/100) = 0{,}08$ bits de información, menos que en el caso de la moneda justa, porque ahora tenemos más evidencia sobre el posible resultado del experimento. Si la probabilidad de que el volado de sol es del 100 %, entonces $E(0, 1) = 0$ bits de información, ejecutar el volado no provee información alguna. La gráfica de la función de entropía se muestra en la figura 7.3.

Consideren nuevamente los ejemplos de entrenamiento del cuadro 7.1 (página 202). De 14 ejemplos, 9 son positivos (si es un buen día para jugar tenis) y 5 son negativos. La entropia de este conjunto de entrenamiento es:

$$E(\frac{9}{14}, \frac{5}{14}) = 0{,}940$$

Si todos los ejemplos son positivos o negativos, por ejemplo, pertencen todos a la misma clase, la entropia será 0. Una posible interpretación de ésto, es considerar la entropia como una medida de ruido o desorden en los ejemplos. Definimos la **ganancia de información** como la reducción de la entropía causada por particionar un conjunto de entrenamiento S, con respecto a un atributo a:

Ganancia de información

$$Ganancia(S, a) = E(S) - \sum_{v \in a} \frac{|S_v|}{|S|} E(S_v)$$

Observen que el segundo término de *Ganancia*, es la entropía con respecto al atributo a. Al utilizar esta medida en ID3, sobre los ejemplos del cuadro 7.1, deberíamos obtener algo como:

```
CL-USER> (best-partition (remove *target* *attributes*)
                         *examples*)
(CIELO (SOLEADO #:|ej1| #:|ej2| #:|ej8| #:|ej9| #:|ej11|)
       (NUBLADO #:|ej3| #:|ej7| #:|ej12| #:|ej13|)
       (LLUVIA #:|ej4| #:|ej5| #:|ej6| #:|ej10| #:|ej14|))
```

Esto indica que el atributo con mayor ganancia de información fue *cielo*, de ahí que esta parte del algoritmo genera la partición de los ejemplos de entrenamiento con respecto a este atributo. Si particionamos recursivamente los ejemplos que tienen el atributo *cielo* = *soleado*, obtendríamos:

```
(HUMEDAD (NORMAL #:|ej9| #:|ej11|)
         (ALTA #:|ej1| #:|ej2| #:|ej8|))
```

Lo cual indica que en el nodo debajo de *soleado* deberíamos incluir el atributo *humedad*. Todos los ejemplos con *humedad* = *normal*, tienen valor *si* para el concepto objetivo. De la misma forma, todos los ejemplos con valor *humedad* = *alta*, tiene valor *no* para el concepto objetivo. Así que ambas ramas descendiendo de nodo *humedad*, llevarán a clases terminales de nuestro problema de aprendizaje. El algoritmo terminará por construir el árbol de la figura 7.2.

Esto nos da las pistas necesarias para programar ID3. Primero, tenemos la función que computa entropía:

```
(defun entropy (examples attrib)
  "It computes the entropy of EXAMPLES with respect to an ATTRIB"
  (let ((partition (get-partition attrib examples))
        (number-of-examples (length examples)))
    (apply #'+
           (mapcar #'(lambda(part)
                       (let* ((size-part (count-if #'atom
                                                   (cdr part)))
                              (proportion
                               (if (eq size-part 0) 0
                                   (/ size-part
                                      number-of-examples))))
                         (* -1.0 proportion (log proportion 2))))
                   (cdr partition)))))
```

Si queremos ahora saber la entropía del conjunto de `*examples*` con respecto a la clase `jugar-tenis`, tenemos que:

```
CL-USER> (entropy *examples* 'jugar-tenis)
0.9402859
```

Ahora, para computar ganancia de información, definimos:

```
(defun information-gain (examples attribute)
  "It computes information-gain for an ATTRIBUTE in EXAMPLES"
  (let ((parts (get-partition attribute examples))
        (no-examples (count-if #'atom examples)))
    (- (entropy examples *target*)
       (apply #'+
              (mapcar
               #'(lambda(part)
                   (let* ((size-part (count-if #'atom
                                               (cdr part)))
                          (proportion (if (eq size-part 0) 0
                                          (/ size-part
                                             no-examples))))
```

```
14                      (* proportion (entropy (cdr part) *target*))))
15                (cdr parts))))))
```

de forma que la ganancia de información del atributo `cielo`, con respecto a la clase `jugar-tenis`, puede obtenerse de la siguiente manera:

```
1 CL-USER> (information-gain *examples* 'cielo 'jugar-tenis)
2 0.24674976
```

Ahora podemos implementar la función para encontrar la mejor partición con respecto a la ganancia de información:

```
1  (defun best-partition (attributes examples)
2    "It computes one of the best partitions induced by ATTRIBUTES
3    over EXAMPLES"
4    (let* ((info-gains
5            (loop for attrib in attributes collect
6                 (let ((ig (information-gain examples attrib))
7                       (p (get-partition attrib examples)))
8                   (when *trace*
9                     (format t "Partición inducida por el atributo
10                              ~s:~%~s~%"
11                            attrib p)
12                    (format t "Ganancia de información: ~s~%"
13                            ig))
14                  (list ig p))))
15           (best (cadar (sort info-gains
16                         #'(lambda(x y) (> (car x) (car y)))))))
17      (when *trace* (format t "Best partition: ~s~%-------------~%"
18                            best))
19     best))
```

Si queremos encontrar la mejor partición inicial, tenemos:

```
1 CL-ID3> (best-partition (remove *target* *attributes*) *examples*)
2 (CIELO (SOLEADO #:|ej1| #:|ej2| #:|ej8| #:|ej9| #:|ej11|)
3        (NUBLADO #:|ej3| #:|ej7| #:|ej12| #:|ej13|)
4        (LLUVIA #:|ej4| #:|ej5| #:|ej6| #:|ej10| #:|ej14|))
```

Si queremos más información sobre cómo se obtuvo esta partición, podemos usar la opción de `(setf *trace* t)`:

```
1  CL-ID3> (setf *trace* t)
2  T
3  CL-ID3> (best-partition (remove *target* *attributes*)
4        *examples*)
5  Partición inducida por el atributo CIELO:
6  (CIELO (SOLEADO #:|ej1| #:|ej2| #:|ej8| #:|ej9| #:|ej11|)
7  (NUBLADO #:|ej3| #:|ej7| #:|ej12| #:|ej13|)
8  (LLUVIA #:|ej4| #:|ej5| #:|ej6| #:|ej10| #:|ej14|))
9  Ganancia de información: 0.2467497
10 Partición inducida por el atributo TEMPERATURA:
11 (TEMPERATURA (FRIO #:|ej5| #:|ej6| #:|ej7| #:|ej9|)
12 (CALOR #:|ej1| #:|ej2| #:|ej3| #:|ej13|)
13 (TEMPLADO #:|ej4| #:|ej8| #:|ej10| #:|ej11| #:|ej12| #:|ej14|))
14 Ganancia de información: 0.029222489
15 Partición inducida por el atributo HUMEDAD:
16 (HUMEDAD (NORMAL #:|ej5| #:|ej6| #:|ej7| #:|ej9| #:|ej10|
17 #:|ej11| #:|ej13|)
18 (ALTA #:|ej1| #:|ej2| #:|ej3| #:|ej4| #:|ej8| #:|ej12| #:|ej14|))
19 Ganancia de información: 0.15183544
20 Partición inducida por el atributo VIENTO:
```

```
21 (VIENTO (DEBIL #:|ej1| #:|ej3| #:|ej4| #:|ej5| #:|ej8| #:|ej9|
22 #:|ej10| #:|ej13|)
23 (FUERTE #:|ej2| #:|ej6| #:|ej7| #:|ej11| #:|ej12| #:|ej14|))
24 Ganancia de información: 0.048126936
25 Best partition: (CIELO (SOLEADO #:|ej1| #:|ej2| #:|ej8| #:|ej9|
26 #:|ej11|) (NUBLADO #:|ej3| #:|ej7| #:|ej12| #:|ej13|)
27 (LLUVIA #:|ej4| #:|ej5| #:|ej6| #:|ej10| #:|ej14|))
28 -------------
29 (CIELO (SOLEADO #:|ej1| #:|ej2| #:|ej8| #:|ej9| #:|ej11|) (NUBLADO
30 #:|ej3| #:|ej7| #:|ej12| #:|ej13|) (LLUVIA #:|ej4| #:|ej5| #:|ej6|
31 #:|ej10| #:|ej14|))
```

id3 llama recursivamente a `best-partition` :

```
1  (defun id3 (examples attribs)
2    "It induces a decision tree running id3 over EXAMPLES
3    and ATTRIBS)"
4    (let ((class-by-default (get-value *target*
5                                       (car examples))))
6      (cond
7        ;; Stop criteria
8        ((same-class-value-p *target* class-by-default examples)
9          class-by-default)
10       ;; Failure
11       ((null attribs) (target-most-common-value examples))
12       ;; Recursive call
13       (t (let ((partition (best-partition attribs
14                                          examples)))
15            (cons (first partition)
16                  (loop for branch in (cdr partition) collect
17                        (list (first branch)
18                              (id3 (cdr branch)
19                                   (remove (first partition)
20                                           attribs)))))))))))
21
22 (defun same-class-value-p (attrib value examples)
23   "Do all EXAMPLES have the same VALUE for a given ATTRIB ?"
24   (every #'(lambda(e)
25              (eq value
26                  (get-value attrib e)))
27          examples))
28
29 (defun target-most-common-value (examples)
30   "It gets the most common value for *target* in EXAMPLES"
31   (let ((domain (get-domain *target*))
32         (values (mapcar #'(lambda(x) (get-value *target* x))
33                         examples)))
34     (caar (sort (loop for v in domain collect
35                       (list v (count v values)))
36                 #'(lambda(x  y) (>= (cadr x)
37                                     (cadr y)))))))
38
39 (defun get-domain (attribute)
40   "It gets the domain of an ATTRIBUTE"
41   (nth (position attribute *attributes*)
42        *domains*))
```

La implementación del algoritmo termina con una pequeña función de interfaz, para ejecutar `id3` sobre el ambiente de aprendizaje por defecto. Aprovecho esta función para verificar si la clase está incluida en el archivo `ARFF`,

ya que WEKA puede eliminar ese atributo del archivo . El símbolo induce es exportado por el paquete cl-id3:

```
1 (defun induce (&optional (examples *examples*))
2   "It induces the decision tree using learning sertting"
3   (when (not (member *target* *attributes*))
4     (error "The target is defined incorrectly: Maybe Weka modified your ARFF"))
5   (id3 examples (remove *target* *attributes*)))
```

De forma que para construir el árbol de decisión ejecutamos:

```
1 CL-ID3> (induce)
2 (CIELO (SOLEADO (HUMEDAD (NORMAL SI) (ALTA NO))) (NUBLADO SI) (LLUVIA
3 (VIENTO (FUERTE NO) (DEBIL SI))))
```

De forma que:

```
1  CL-ID3> (print-tree *)
2  CIELO
3   - SOLEADO
4       HUMEDAD
5         - NORMAL -> SI
6         - ALTA -> NO
7   - NUBLADO -> SI
8   - LLUVIA
9       VIENTO
10        - FUERTE -> NO
11        - DEBIL -> SI
12 NIL
```

Aunque en realidad lo que necesitamos es una interfaz gráfica.

7.2.7 Interfaz gráfica para cl-id3

Una vez que definimos la dependencia entre los archivos de nuestro sistema vía ASDF y el paquete para nuestra aplicación vía defpackage, podemos pensar en definir una interfaz gráfica para :cl-id3. Lo primero es incluir un archivo cl-id3-gui.lisp en la definición del sistema. El paquete :cl-id3 ya hace uso de :capi, la librería para interfaces gráficas de Lispworks. La interfaz finalizada se muestra en la figura 7.4. En esta sección abordaremos la implementación de la interfaz.

Como es usual, la interfaz incluye una barra de menús, una ventana principal para desplegar el árbol inducido e información sobre el proceso de inducción; y varias ventanas auxiliares para desplegar ejemplos, atributos y demás información adicional. En el escritorio puede verse el icono cl-id3 (un bonsái) que permite ejecutar nuestra aplicación.

El código de la interfaz se puede dividir conceptualmente en dos partes: una que incluye la definición de los elementos gráficos en ella y otra que define el comportamiento de esos elementos en la interacción con el usuario.

Definiendo la interfaz

Los componentes gráficos de la interfaz y la manera en que estos se despliegan, se define haciendo uso de la función define-interface del capi como se muestra a continuación:

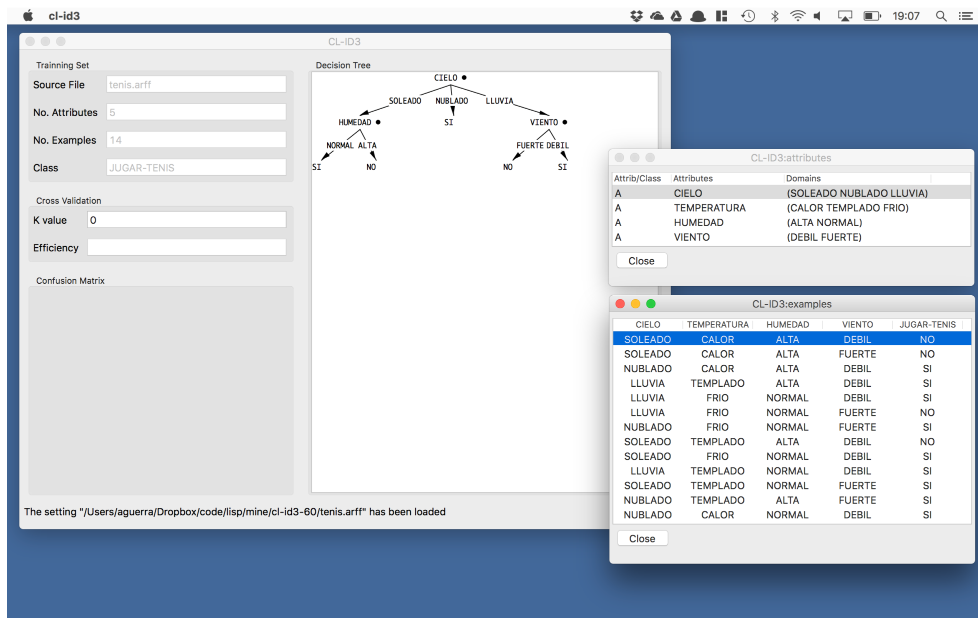


Figura 7.4: La interfaz gráfica de la aplicación `cl-id3`.

```
1  (define-interface cl-id3-gui ()
2    ()
3    (:panes
4     (source-id-pane text-input-pane
5                     :accessor source-id-pane
6                     :text ""
7                     :enabled nil)
8     (num-attributes-pane text-input-pane
9                          :accessor num-attributes-pane
10                         :text ""
11                         :enabled nil)
12    (num-examples-pane text-input-pane
13                       :accessor num-examples-pane
14                       :text ""
15                       :enabled nil)
16    (class-pane text-input-pane
17                :accessor class-pane
18                :text ""
19                :enabled nil)
20    (efficiency-pane text-input-pane
21                     :text ""
22                     :enabled nil)
23    (k-value-pane text-input-pane
24                  :text "0")
25    (tree-pane graph-pane
26               :title "Decision Tree"
27               :title-position :frame
28               :children-function 'node-children
29               :edge-pane-function
30               #'(lambda(self from to)
31                   (declare (ignore self from))
32                   (make-instance
33                    'labelled-arrow-pinboard-object
34                    :data (princ-to-string (node-from-label to))))
35               :visible-min-width 450
36               :layout-function :top-down)
```

```
37     (state-pane title-pane
38                 :accessor state-pane
39                 :text "Welcome to CL-ID3."))
40    (:menus
41     (file-menu
42      "File"
43      (("Open" :selection-callback 'gui-load-file
44               :accelerator #\o)
45       ("Quit" :selection-callback 'gui-quit
46               :accelerator #\q)))
47     (view-menu
48      "View"
49      (("Attributes" :selection-callback 'gui-view-attributes
50                     :accelerator #\a
51                     :enabled-function #'(lambda (menu) *attributes-on*))
52       ("Examples" :selection-callback 'gui-view-examples
53                   :accelerator #\e
54                   :enabled-function #'(lambda (menu) *examples-on*))))
55     (id3-menu
56      "id3"
57      (("Induce" :selection-callback 'gui-induce
58                 :accelerator #\i
59                 :enabled-function #'(lambda (menu) *induce-on*))
60       ("Classify" :selection-callback 'gui-classify
61                   :accelerator #\k
62                   :enabled-function #'(lambda (menu) *classify-on*))
63       ("Cross-validation" :selection-callback 'gui-cross-validation
64                           :accelerator #\c
65                           :enabled-function
66                           #'(lambda (menu) *cross-validation-on*))))
67     (help-menu
68      "Help"
69      (("About" :selection-callback 'gui-about))))
70    (:menu-bar file-menu view-menu id3-menu help-menu)
71    (:layouts
72     (main-layout column-layout '(panes state-pane))
73     (panes row-layout '(info-pane tree-pane))
74    (matrix-pane row-layout '(confusion)
75                  :title "Confusion Matrix" :x-gap '10 :y-gap '30
76                  :title-position :frame :visible-min-width '200)
77     (info-pane column-layout '(setting-pane id3-pane matrix-pane))
78     (setting-pane grid-layout
79       '("Source File" source-id-pane
80         "No. Attributes" num-attributes-pane
81         "No. Examples" num-examples-pane
82         "Class" class-pane)
83                   :y-adjust :center
84                   :title "Trainning Set"
85                   :title-position :frame :columns '2)
86     (id3-pane grid-layout '("K value" k-value-pane
87           "Efficiency" efficiency-pane)
88               :y-adjust :center
89               :title "Cross Validation"
90               :title-position :frame :columns '2))
91    (:default-initargs
92     :title "CL-ID3"
93     :visible-min-width 840
94     :visible-min-height 600))
```

Hay cuatro elementos a especificar en una interfaz: paneles (elementos de la interfaz que en otros lenguajes de programación se conocen como *widgets*), menús, barra de menús y la composición gráfica de esos elementos (*layout*). Técnicamente, una interfaz es una clase con ranuras especiales para definir estos elementos y por lo tanto, la función `define-interface` es análoga a `defclass`. Estamos en el dominio del sistema orientado a objetos de Lisp, conocido como `CLOS` [5, 42].

En la línea 3, inicia la definición de los paneles de la interfaz con la ranura `:panes`. Aquí se definen los elementos gráficos que se utilizarán en nuestras ventanas. Por ejemplo `source-id-pane` (línea 4) es un panel de entrada de texto que inicialmente despliega una cadena vacía y está deshabilitado (el usuario no puede escribir en él). El panel `tree-pane` (línea 25) es un panel gráfico que nos permitirá visualizar el árbol inducido. La función `node-children` se encargará de computar los hijos de la raíz del árbol, para dibujarlos. Como deseamos que los arcos entre nodos estén etiquetados con el valor del atributo padre, redefinimos el tipo de arco en la ranura `:edge-pane-function` (línea 29). Como podrán deducir de la función anónima ahí definida, necesitaremos cambiar la representación interna del árbol inducido para hacer más sencilla su visualización. La línea 37 define un panel de tipo título para implementar una barra de estado del sistema.

A partir de la línea 40 definimos los menús del sistema. Todos ellos tienen *shortcuts* definidos en las ranuras `:accelerator`. La ranura `:enabled-function` me permite habilitar y deshabilitar los menús según convenga, con base en las siguientes variables globales:

```
1 (defvar *examples-on* nil "t enables the examples menu")
2 (defvar *attributes-on* nil "t enables the attributes menu")
3 (defvar *induce-on* nil "t enables the induce menu")
4 (defvar *classify-on* nil "t enables the classify menu")
5 (defvar *cross-validation-on* nil "t enables the cross-validation menu")
```

Si el valor de las variables cambia a `t`, el menú asociado se habilita.

El comportamiento de los menús está definido por la función asociada a la ranura `:selection-callback`. Las funciones asociadas a esta ranura se definen más adelante. La línea 69 define la barra de menús, es decir, el orden en que aparecen los menús definidos.

Ahora solo nos resta definir la disposición gráfica de todos estos elementos en la interfaz. Esto se especifica mediante la ranura `:layout` a partir de la línea 70. Usamos tres tipos de disposiciones: en columna (`column-layout`), en renglón (`row-layout`) y en rejilla (`grid-layout`). La columna y el renglón funcional como pilas de objetos, horizontales o verticales respectivamente. La rejilla nos permite acomodar objetos en varias columnas. El efecto es similar a definir un rengón de columnas.

Finalmente la ranura `default-initargs` permite especificar valores iniciales para desplegar la interfaz, por ejemplo el título y su tamaño mínimo, tanto horizontal como vertical.

Definiendo el comportamiento de la interfaz

En esta sección revisaremos las funciones asociadas a las ranuras `callback` de los componentes de la interfaz. Estas funciones definen el comportamien-

to de los componentes. La siguiente función se hace cargo de leer archivos que definen conjuntos de entrenamiento para `:cl-id3`:

```
(defun gui-load-file (data interface)
  (declare (ignore data))
  (let ((file (prompt-for-file
               nil
               :filter "*.arff"
               :filters '("WEKA files" "*.arff"
                          "Comme Separated Values" "*.csv"))))
    (when file
      (let* ((path (princ-to-string file))
             (setting (car (last (split-sequence #\/ path)))))
        (load-file path)
        (setf (text-input-pane-text (source-id-pane interface))
              setting)
        (setf (text-input-pane-text (num-attributes-pane interface))
              (princ-to-string (length *attributes*)))
        (setf (text-input-pane-text (num-examples-pane interface))
              (princ-to-string (length *examples*)))
        (setf (text-input-pane-text (class-pane interface))
              (princ-to-string *target*))
        (setf (title-pane-text (state-pane interface))
              (format nil "The setting ~s has been loaded"
                      path))
        (setf *examples-on* t *attributes-on* t *induce-on* t)))))
```

Por defecto, estas funciones reciben como argumentos `data` e `interface` cuyo contenido son los datos en el objeto de la interfaz, por ejemplo el texto capturado; y la interfaz que hizo la llamada a la función. La función predefinida `prompt-for-file` abre un panel para seleccionar un archivo y regresa un *path* al archivo seleccionado. Podemos seleccionar el tipo de archivo que nos interesa entre las opciones `ARFF` y `CSV`. Posteriormente convertimos el camino al archivo en una cadena de texto con la función predefinida `princ-to-string` y extraemos el nombre del archivo. La serie de `setf` cambia el texto asociado a los objetos en la interfaz. La última asignación habilita los menús que permiten visualizar archivos y atributos, así como inducir el árbol de decisión.

La siguiente función destruye la interfaz que la llama, esta asociada a las opciones `quit` de la aplicación:

```
(defun gui-quit (data interface)
  (declare (ignore data))
  (quit-interface interface))
```

El desplegado de los atributos y sus dominios se lleva a cabo ejecutando la siguiente función:

```
(defun gui-view-attributes (data interface)
  (declare (ignore data interface))
  (let* ((max-length-attrib (apply #'max
                                   (mapcar #'length
                                           (mapcar #'princ-to-string
                                                   *attributes*))))
         (pane-total-width (list 'character
                                 (* max-length-attrib
                                    (+ 1 (length *attributes*))))))
    (define-interface gui-domains () ()
```

CL-ID3:attributes

Attrib/Class	Attributes	Domains
A	CIELO	(SOLEADO NUBLADO LLUVIA)
A	TEMPERATURA	(CALOR TEMPLADO FRIO)
A	HUMEDAD	(ALTA NORMAL)
A	VIENTO	(DEBIL FUERTE)

Close

Figura 7.5: Atributos y sus dominios desplejados en la interfaz.

```
11        (:panes
12         (attributes-pane multi-column-list-panel
13                          :columns '((:title "Attrib/Class"
14                                      :adjust :left
15                                      :visible-min-width (character 10))
16                                     (:title "Attributes" :adjust :left
17                                      :visible-min-width (character 20))
18                                     (:title "Domains" :adjust :left
19                                      :visible-min-width (character 20)))
20                          :items (loop for a in *attributes* collect
21              (list (if (eql *target* a) 'c 'a )
22              a
23              (get-domain a)))
24                          :visible-min-width pane-total-width
25                          :visible-min-height :text-height
26                          :vertical-scroll t)
27         (button-pane push-button
28                      :text "Close"
29                      :callback 'gui-quit))
30        (:default-initargs
31         :title "CL-ID3:attributes"))
32      (display (make-instance 'gui-domains))))
```

En este caso usamos un `multi-column-list-panel` que nos permite definir columnas con encabezado. La ejecución de esta función genera una ventana como la que se muestra en la figura 7.5.

La función para inducir el árbol de decisión y desplegarlo gráficamente es la siguiente:

```
1 (defun gui-induce (data interface)
2   "It induces the decisicion tree and displays it in the INTERFACE"
3   (declare (ignore data))
4   (setf *current-tree* (induce))
5   (display-tree (make-tree *current-tree*)
6                 interface))
```

Primero induce el árbol y después cambia su representación para poder dibujarlo. El cambio de representación hace uso de nodos que incluyen la etiqueta del arco que precede a cada nodo, excepto claro la raíz del árbol:

```
1 (defstruct node
2   (inf nil)
3   (sub-trees nil)
4   (from-label nil))
5
6 (defun make-tree (tree-as-lst)
7   "It makes a tree of nodes with TREE-AS-LST"
8   (make-node :inf (root tree-as-lst)
9              :sub-trees (make-sub-trees (children tree-as-lst))))
10
```

```
11 (defun make-sub-trees (children-lst)
12   "It makes de subtrees list of a tree with CHILDREN"
13   (loop for child in children-lst collect
14         (let ((sub-tree (second child))
15               (label (first child)))
16           (if (leaf-p sub-tree)
17               (make-node :inf sub-tree
18                          :sub-trees nil
19                          :from-label label)
20             (make-node :inf (root sub-tree)
21                        :sub-trees (make-sub-trees (children sub-tree))
22                        :from-label label)))))
```

Opcionalmente podríamos cambiar nuestro algoritmo `id3` para que generará el árbol con esta representación. Las funciones para visualizar el árbol a partir de la nueva representación son:

```
1  (defmethod print-object ((n node) stream)
2    (format stream "~s " (node-inf n)))
3
4  (defun display-tree (root interface)
5  "It displays the tree with ROOT in its pane in INTERFACE"
6  (with-slots (tree-pane) interface
7      (setf (graph-pane-roots tree-pane)
8            (list root))
9      (map-pane-children tree-pane ;;; redraw panes
10                        (lambda (item)
11                          (update-pinboard-object item)))))
12
13 (defun node-children (node)
14   "It gets the children of NODE to be displayed"
15   (let ((children (node-sub-trees node)))
16     (when children
17       (if (leaf-p children) (list children)
18         children))))
```

La línea 9 es necesaria para acomodar los nodos una vez que el árbol ha sido dibujado totalmente, de otra forma los desplazamientos ocurridos al dibujarlo incrementalmente, pueden desajustar su presentación final. La función `node-children` es la función asociada al panel `tree-pane` para computar los hijos de un nodo. La función `print-object` especifica como queremos etiquetar los nodos del árbol.

La siguiente función nos permite desplegar la interfaz gráfica que hemos definido, después de limpiar la configuración de la herramienta:

```
1 (defun gui ()
2   (reset)
3   (display (make-instance 'cl-id3-gui)))
```

La llamada a esta función despliega la interfaz que se muestra en la figura 7.4.

7.3 FILTRADO COLABORATIVO

Inteligencia colectiva

Recomendaciones

En esta sección abordaremos una aplicación en el dominio de la **inteligencia colectiva**, que consiste en utilizar las preferencias de un grupo de personas, para hacer **recomendaciones** a otras personas. Las recomdaciones pueden

ser acerca de productos a comprar en una tienda *on-line*, sitios web de interés, música, películas, etc. Nos concentraremos en las funciones necesarias para 1) encontrar personas con gustos similares y 2) elaborar recomendaciones automáticas con base en lo que a otra gente le gusta.

Quizás ya conozcan los sistemas de recomendación al estilo de `Amazon` (Ver figura 7.6). Esta compañía rastrea los habitos de compra de sus clientes, y cuando uno entra a su página web, recibe recomendaciones sobre productos que nos podrían gustar, con base en esa información. `Amazon` puede incluso sugerirte películas, aún cuando tu solo hayas comprado libros. Sitios como `http://reddit.com` te permiten votar ligas a otras páginas web y recibir recomendaciones con base en tus votos, sobre ligas que te pudiesen interesar. De estos ejemplos podemos deducir que los datos a recolectar son diversos tanto en contenido, como en su origen y representación.

Figura 7.6: El sistema de recomendación de Amazon.

La forma pre-tecnológica de obtener recomendaciones, digamos sobre películas, es preguntarle a nuestros amigos su opinión. Sabemos además, que algunos de nuestros amigos tienen mejores "criterios" cinematográficos que otros, algo que aprendemos con base a la experiencia –Observamos cuando les gusta lo que a nosotros nos gusta. Al aumentar la oferta de películas, es más difícil resolver este problema consultando a un pequeño grupo de amigos, esta es la razón de ser del **filtrado colaborativo**, donde el *modus operandi* es el siguiente: Buscar en un gran grupo de personas, un grupo más pequeño afin a nuestros gustos. Usar los gustos de ese sub grupo de personas para generar una lista ordenada de recomendaciones sobre productos que yo no he consumido, en este caso, películas que no he visto. El nombre de filtrado colaborativo se le debe a Goldberg y col. [31].

Filtrado colaborativo

7.3.1 Preferencias

Es necesario registrar las preferencias de diferentes personas. El cuadro 7.2 muestra un conjunto de preferencias expresadas por diferentes personas con respecto a ciertas películas. Sus preferencias están expresadas como una calificación entre cero y cinco. Como se puede observar, no todos los críticos han visto todas las películas.

Podemos usar diversas representaciones para estos datos. La primera que se nos viene a la mente es una lista de listas. Sin embargo, observen que la tabla sugiere información relacional, p. ej., seguramente nos interesará acceder a la calificación que Puig le dió a Superman Returns. De manera que representaremos estos datos como una **lista de propiedades** y así explotar

Información relacional

Lista de propiedades

Película	Rose	Seymour	Phillips	Puig	LaSalle	Mathews	Toby
Lady in Water	2.5	3.0	2.5	-	3.0	3.0	-
Snakes on Plane	3.5	3.5	3.0	3.5	4.0	4.0	4.5
Just my Luck	3.0	1.5	-	3.0	2.0	-	-
Superman returns	3.5	5.0	3.5	4.0	3.0	5.0	4.0
The Night Listener	3.0	3.0	4.0	4.5	3.0	3.0	-
You, Me and Dupree	2.5	3.5	-	2.5	2.0	3.5	3.5

Cuadro 7.2: Preferencias sobre películas. Adaptado de Seagaran [78].

algunas funciones predefinidas en Lisp, como `find`, `setf` y `getf`. La tabla 7.2 quedaría codificada como sigue:

```
1  (defvar *critics*
2    '((:critic "Lisa Rose"
3       :critics (("Lady in the Water" 2.5)
4           ("Snakes on a Plane" 3.5)
5           ("Just My Luck" 3.0)
6           ("Superman Returns" 3.5)
7           ("You, Me and Dupree" 2.5)
8           ("The Night Listener" 3.0)))
9      (:critic "Gene Seymour"
10      :critics (("Lady in the Water" 3.0)
11          ("Snakes on a Plane" 3.5)
12          ("Just My Luck" 1.5)
13          ("Superman Returns" 5.0)
14          ("The Night Listener" 3.0)
15          ("You, Me and Dupree" 3.5)))
16 ...)
```

Ahora es necesario implementar algunas funciones de acceso a estos datos. La primera de ellas, nos permite recuperar el registro de un critico de nuestra base de datos `*critics*`:

```
1 (defun get-critic (critic)
2   (find critic *critics*
3         :test #'(lambda(name reg)
4                   (string= name (getf reg :critic)))))
```

observen que es necesario indicarle a `find` que las comparaciones se hacen con base a `string=`, es decir, igualdad entre cadenas de texto. La razón para introducir esto en una función anónima es que el objeto donde busco el nombre del crítico (`name`) es un objeto compuesto, el registro correspondiente, del cual hay que extraer el nombre con `(getf reg :critic)`. Esto ilustra el uso de las funciones anónimas para cambiar el criterio de comparación de una función y acceder a elementos específicos de un objeto compuesto.

De manera que si queremos recuperar el registro de `Toby`, tenemos:

```
1 CL-USER> (get-critic "Toby")
2 (:CRITIC "Toby" :CRITICS (("Snakes on a Plane" 4.5)
3 ("Superman Returns" 4.0) ("You, Me and Dupree" 1.0)))
```

También es deseable acceder a la calificación que un crítico le ha dado a una película determinada:

```
1 (defun get-film-rating (critic film)
2   (cadr (find film (getf (get-critic critic) :critics)
```

```
3                :test #'(lambda(film reg)
4                           (string= film (car reg))))))
```

donde el uso de la función anónima es análogo al caso anterior. De manera que:

```
1 CL-USER> (get-film-rating "Toby" "Superman Returns")
2 4.0
```

También nos será muy útil saber qué películas a evaluado un crítico dado:

```
1  (defun get-films-rated-by (person)
2   (mapcar #'car (fourth (get-critic person))))
```

Observen el uso de `fourth` en esta función. El registro de un crítico es, a fin de cuentas, una lista de cuatro elementos, donde el cuarto de entre ellos es la lista de películas que ha evaluado representada como una lista de pares película-calificación. Si obtenemos esa lista, debemos recuperar el primer elemento de cada par con la ayuda de `mapcar`. Su uso es como sigue:

```
1 CL-USER> (get-films-rated-by "Toby")
2 ("Snakes on a Plane" "Superman Returns" "You, Me and Dupree")
```

Finalmente, implementaremos la siguiente función para saber que películas han visto dos críticos dados:

```
1 defun shared-items (critic1 critic2)
2   (intersection (get-films-rated-by critic1)
3                 (get-films-rated-by critic2)
4                 :test #'string= ))
```

observen que en este caso `:test` puede recibir la función `#'string=` directamente, puesto que los objetos a comparar son todos cadenas de texto. Las funciones de intersección y unión de conjuntos están predefinidas en Lisp, aunque por default usan `eq` para hacer comparaciones. Su uso es el siguiente:

```
1 CL-USER> (shared-items "Toby" "Lisa Rose")
2 ("Snakes on a Plane" "Superman Returns" "You, Me and Dupree")
```

7.3.2 Usuarios similares

Para poder establecer que usuarios se parecen a un usuario dado, es necesario adoptar una **medida de similitud**. Existen una gran diversidad de ellas, López Iñesta [50] hace una revisión exhaustiva de las medidas de similitud en el contexto que nos interesa, es decir, las listas de recomendaciones. Comenzaremos por utilizar la distancia euclidiana, expresada por:

Medidas de similitud

$$dist((x_1, y_1)(x_2, y_2)) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

La figura 7.7 muestra como podemos usar esta distancia para configurar un **espacio de preferencias** para dos películas, pero el concepto se puede generalizar a n películas. El crítico `Toby` da un valor de 4.5 a `Snakes` y 1.0 a `Dupree`. Entre más cerca estén en este espacio dos personas, más similares son sus preferencias.

Espacio de preferencias

Como en realidad no nos interesa saber la distancia entre dos usuarios, sino si son parecidos o no, normalizaremos la distancia de manera que una

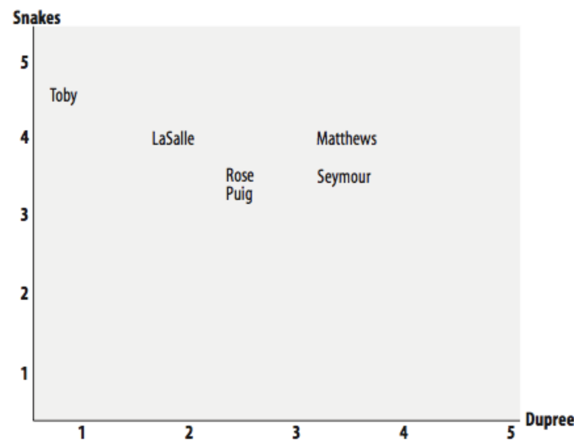


Figura 7.7: Espacio de preferencias basado en distancia euclidiana.

distancia de uno índica preferencias idénticas; y uno de cero, lo contrario. La implementación en Lisp de la función de distancia euclidiana es como sigue:

```
1  (defun sim-euclidean (person1 person2)
2    "1 means identical preferences, 0 the opposite"
3    (let ((shared (shared-items person1 person2)))
4      (if (null shared)
5          0
6          (/ 1
7             (+ 1
8                (sqrt (loop for film in shared sum
9                           (sqr (- (get-film-rating person1 film)
10                                  (get-film-rating person2 film)))))))))))
11
12 (defun sqr (x) (* x x))
```

De forma que podemos computar distancias entre críticos:

```
1 CL-USER> (sim-euclidean "Lisa Rose" "Gene Seymour")
2 0.29429805
3 CL-USER> (sim-euclidean "Lisa Rose" "Claudia Puig")
4 0.38742587
```

Sin embargo, la distancia euclidiana tiene algunos problemas. Por ejemplo, si se comparan dos críticos y uno de ellos califica sistemáticamente más alto, esta distancia dirá que son lejanos, aún cuando sus preferencias reflejen lo mismo. Para corregir estos corrimientos en la escala, podemos adoptar la **correlación de Pearson**:

Correlación de Pearson

$$r_{xy} = \frac{n \sum x_i y_i - \sum x_i \sum y_i}{\sqrt{n \sum x_i^2 - (\sum x_i)^2} \sqrt{n \sum y_i^2 - (\sum y_i)^2}}$$

La figura 7.8 muestra una comparación entre dos críticos basada en la correlación de Pearson. La línea punteada se conoce como **línea de mejor ajuste** porque se acerca tanto como es posible a los objetos en el espacio. Si la correlación fuese perfecta, la línea de mejor ajuste sería diagonal y tocaría todos los puntos en el espacio.

Línea de mejor ajuste

La implementación de la similitud basada en la correlación de Pearson en Lisp es como sigue:

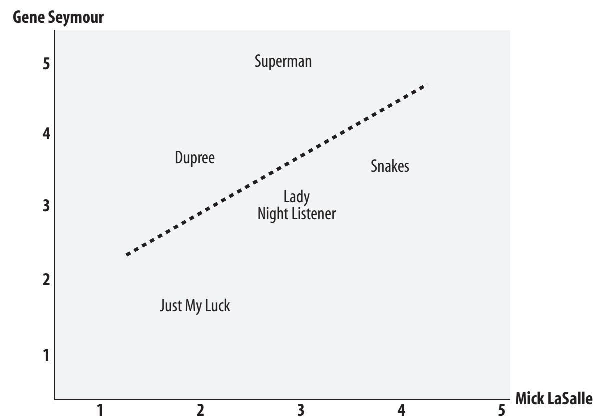


Figura 7.8: Comparación entre dos críticos con base en la correlación de Pearson.

```
 1 (defun sim-pearson (person1 person2)
 2   "1 means identical preferences, -1 the opposite"
 3   (let ((si (shared-items person1 person2)))
 4     (if (null si)
 5         0
 6         (let* ((n (length si))
 7                (prefs1 (loop for film in si collect
 8                              (get-film-rating person1 film)))
 9                (prefs2 (loop for film in si collect
10                              (get-film-rating person2 film)))
11                (sum1 (sum prefs1))
12                (sum2 (sum prefs2))
13                (sum1sq (sum (mapcar #'sqr prefs1)))
14                (sum2sq (sum (mapcar #'sqr prefs2)))
15                (psum (sum (mapcar #'(lambda (x y)
16                                       (* x y)) prefs1 prefs2)))
17                (num (- psum (/ (* sum1 sum2) n)))
18                (den (sqrt (* (- sum1sq (/ (sqr sum1) n))
19                              (- sum2sq (/ (sqr sum2) n))))))
20           (if (zerop den)
21               1
22               (/ num den))))))
```

Observen que esta medida esta normalizada entre -1 y 1, y éste último valor refleja preferencias idénticas. Algunas consultas posibles incluyen:

```
1 CL-USER> (sim-pearson "Lisa Rose" "Gene Seymour")
2 0.39605904
3 CL-USER> (sim-pearson "Mick LaSalle" "Gene Seymour")
4 0.4117648
5 CL-USER> (sim-pearson "Lisa Rose" "Jack Matthews")
6 0.74701745
```

Ahora podemos computar el listado de los críticos que más se parecen a una persona dada. La definición de esta función es como sigue:

```
1 (defun top-matches (person &optional (n 5) (sim #'sim-pearson))
2   (subseq (sort (mapcar #'(lambda(c)
3                              (list c (funcall sim person c)))
4                          (get-other-critics person))
5                 #'(lambda(x y) (> (cadr x) (cadr y))))
6           0 n))
7
```

```
 8 (defun get-other-critics (critic)
 9   (remove critic
10           (mapcar #'(lambda(x) (getf x :critic))
11                   *critics*)
12           :test #'string=))
```

De forma que el top-3 para `Toby` puede computarse como sigue:

```
1 CL-USER> (top-matches "Toby" 3)
2 (("Lisa Rose" 0.9912409) ("Mick LaSalle" 0.92447347)
3  ("Claudia Puig" 0.8934049))
```

7.3.3 Recomendaciones

Ahora que podemos computar la lista de críticos más cercanos a las preferencias de una persona, podemos usarlos para hacer recomendaciones. Para ello, necesitamos generar una calificación ponderada:

```
 1 (defun get-recommendations (person &optional (similarity #'sim-pearson))
 2   (let* ((other-critics (get-other-critics person))
 3          (films-not-seen (set-difference
 4                           (reduce #'(lambda (x y)
 5                                       (union x y :test #'string=))
 6                                   (mapcar #'get-films-rated-by
 7                                           other-critics))
 8                           (get-films-rated-by person)
 9                           :test #'string=)))
10     (sort
11      (loop for film in films-not-seen collect
12           (let* ((sim-sums 0)
13                  (total
14                   (apply #'+
15                          (mapcar #'(lambda(critic)
16                                      (let ((rating
17                                             (get-film-rating critic film))
18                                            (sim
19                                             (funcall similarity person critic)))
20                                        (if rating
21                                            (progn
22                                              (setf sim-sums (+ sim-sums sim))
23                                              (* rating sim))
24                                            0)))
25                                  other-critics))))
26             (list (/ total sim-sums) film)))
27      #'(lambda(x y) (> (car x ) (car y))))))
```

De manera que las recomendaciones para `Toby` son:

```
1 CL-USER> (get-recommendations "Toby")
2 ((3.1192015 "The Night Listener") (3.0022347 "Lady in the Water")
3  (2.5309806 "Just My Luck"))
```

7.3.4 Emparejando productos

Si queremos recomendar una películas que podrían gustarme si me gusta `Superman` es necesario invertir nuestra base de datos `*critics*`. Esto lo hace la siguiente función:

```
(defun transform-prefs ()
  "Inverts the dictionary *critics*, indexing it by film"
  (let* ((critics (loop for critic in *critics* collect
                      (getf critic :critic)))
         (films (reduce #'(lambda (x y) (union x y :test #'string=))
                        (loop for critic in critics collect
                             (get-films-rated-by critic)))))
    (loop for film in films collect
         (list :critic film
               :critics (remove-if #'(lambda(x) (null (cadr x)))
                                   (mapcar #'(lambda (c)
                                               (list c (get-film-rating c film)))
                                           critics))))))
```

De forma que:

```
CL-USER> (transform-prefs)
((:CRITIC "Lady in the Water" :CRITICS (("Lisa Rose" 2.5) ("Gene
Seymour" 3.0) ("Michael Phillips" 2.5) ("Mick LaSalle" 3.0) ("Jack
Matthews" 3.0))) (:CRITIC "Snakes on a Plane" :CRITICS (("Lisa Rose"
3.5) ("Gene Seymour" 3.5) ("Michael Phillips" 3.0) ("Claudia Puig"
3.5) ("Mick LaSalle" 4.0) ("Jack Matthews" 4.0) ("Toby" 4.5)))
(:CRITIC "Just My Luck" :CRITICS (("Lisa Rose" 3.0) ("Gene Seymour"
1.5) ("Claudia Puig" 3.0) ("Mick LaSalle" 2.0))) (:CRITIC "Superman
Returns" :CRITICS (("Lisa Rose" 3.5) ("Gene Seymour" 5.0) ("Michael
Phillips" 3.5) ("Claudia Puig" 4.0) ("Mick LaSalle" 3.0) ("Jack
Matthews" 5.0) ("Toby" 4.0))) (:CRITIC "You, Me and Dupree" :CRITICS
(("Lisa Rose" 2.5) ("Gene Seymour" 3.5) ("Claudia Puig" 2.5) ("Mick
LaSalle" 2.0) ("Jack Matthews" 3.5) ("Toby" 1.0))) (:CRITIC "The Night
Listener" :CRITICS (("Lisa Rose" 3.0) ("Gene Seymour" 3.0) ("Michael
Phillips" 4.0) ("Claudia Puig" 4.5) ("Mick LaSalle" 3.0) ("Jack
Matthews" 3.0))))
```

Observen que las críticas están indexadas ahora por película y cada una de ellas es una lista de las calificaciones que los distintos críticos dan a la película. Ahora podemos usar las mismas funciones de recomendación con base en películas, no personas.

```
CL-USER> (setf *critics* (transform-prefs))
((:CRITIC "Lady in the Water" :CRITICS (("Lisa Rose" 2.5) ("Gene
Seymour" 3.0) ("Michael Phillips" 2.5) ("Mick LaSalle" 3.0) ("Jack
Matthews" 3.0))) (:CRITIC "Snakes on a Plane" :CRITICS (("Lisa Rose"
3.5) ("Gene Seymour" 3.5) ("Michael Phillips" 3.0) ("Claudia Puig"
3.5) ("Mick LaSalle" 4.0) ("Jack Matthews" 4.0) ("Toby" 4.5)))
(:CRITIC "Just My Luck" :CRITICS (("Lisa Rose" 3.0) ("Gene Seymour"
1.5) ("Claudia Puig" 3.0) ("Mick LaSalle" 2.0))) (:CRITIC "Superman
Returns" :CRITICS (("Lisa Rose" 3.5) ("Gene Seymour" 5.0) ("Michael
Phillips" 3.5) ("Claudia Puig" 4.0) ("Mick LaSalle" 3.0) ("Jack
Matthews" 5.0) ("Toby" 4.0))) (:CRITIC "You, Me and Dupree" :CRITICS
(("Lisa Rose" 2.5) ("Gene Seymour" 3.5) ("Claudia Puig" 2.5) ("Mick
LaSalle" 2.0) ("Jack Matthews" 3.5) ("Toby" 1.0))) (:CRITIC "The Night
Listener" :CRITICS (("Lisa Rose" 3.0) ("Gene Seymour" 3.0) ("Michael
Phillips" 4.0) ("Claudia Puig" 4.5) ("Mick LaSalle" 3.0) ("Jack
Matthews" 3.0))))
CL-USER> (top-matches "Superman Returns" 3)
(("You, Me and Dupree" 0.6579514) ("Lady in the Water" 0.48795068)
("Snakes on a Plane" 0.111803316))
```

También podemos computar quién recomienda `Just My Luck`:

```
CL-USER> (get-recommendations "Just My Luck")
```

```
2 ((3.8716946 "Jack Matthews") (2.9610002 "Toby") (2.2872024 "Michael
3 Phillips"))
```

7.4 LECTURAS Y EJERCICIOS SUGERIDOS

La programación funcional (y la lógica) es usada con regularidad en bioinformática. El proyecto sobre la relevancia de los codones en el código genético fue desarrollada originalmente por Guerra-Hernández, Mora-Basáñez y Jiménez-Montaño [35], con la idea de verificar si la propuesta de Mac Dónaill y Manktelow [51] generaba ordenes de relevancia iguales, independientemente de la matriz de similitud usada. Como se sospechaba, el resultado fue negativo, diferentes matrices inducen diferentes ordenes a nivel de propiedades físico-químicas, aunque el mismo orden para los codones. Khomtchouk, Weitz y Wahlestedt [41] discuten las facilidades de Lisp para construir modelos bioinformátics complejos y flexibles.

El algoritmo ID3 ya fue comentado ampliamente en el capítulo 4. Es interesante comprar la implementación en Lisp de este algoritmo y la solución en Prolog que se describe en ese capítulo. Utilizamos este tema además para introducir el manejo de sistemas ASDF en Lisp, el uso de librerías instalables vía `Quicklisp` y la implementación de una interfaz gráfica más compleja usando el `CAPI` de `Lispworks`.

La inteligencia colectiva [78] es un tema muy en boga, que puede abordarse adecuadamente desde una perspectiva funcional. Goldberg y col. [31] introdujeron el concepto de filtrado colaborativo que implementamos en este capítulo. Por supuesto, el tema está ligado a todo lo que tenga que ver con Ciencia de Datos. López Iñesta [50] provee un estudio muy interesante sobre la representación de medidas de similitud entre pares de objetos para establecer recomendaciones y rankings. El trabajo es además interesante porque los objetos en cuestión pueden ser imágenes.

Ejercicios

Ejercicio 7.1. *¿Puede mejorarse la definición de `get-dist` del proyecto de bioinformática, con el objetivo de hacerla más legible?*

Ejercicio 7.2. *La funciones propuestas está diseñada para trabajar con la representación propuesta para las matrices de similitud. ¿Pueden pensar en otra representación para las matrices? ¿Cómo sería entonces `get-dist`?*

Ejercicio 7.3. *Implementen una de las siguientes mejoras a la implementación en Lisp de ID3. La salida debe dirigirse a la interfaz gráfica que construímos.*

- *Agregar otra medida de información al sistema para usarla opcionalmente.*
- *Obtener el bosque de árboles posibles en lugar de sólo un árbol.*
- *Ejecutar* cross-validation *para evaluar las hipótesis obtenidas.*

Ejercicio 7.4. *Implemente otras medidas de similitud para el filtrado colaborativo.*

Ejercicio 7.5. *Implemente una interfaz gráfica para el filtrado colaborativo, donde las preferencias puedan leerse de un archivo* `CSV`*. Utilize la interfaz de* `CL-ID3` *como guía.*