

Programación para la Inteligencia Artificial

Lisp en la IA

Dr. Alejandro Guerra-Hernández

Instituto de Investigaciones en Inteligencia Artificial
Universidad Veracruzana
*Campus Sur, Calle Paseo Lote II, Sección Segunda No 112,
Nuevo Xalapa, Xalapa, Ver., México 91097*
mailto:aguerra@uv.mx
<https://www.uv.mx/personal/aguerra/pia>

Maestría en Inteligencia Artificial 2023



Universidad Veracruzana

- 1 Bioinformática: Significancia de los Codones en el Código Genético
- 2 Arboles de Decisión: CL-ID3
- 3 Filtrado colaborativo: Rankings y Recomendaciones



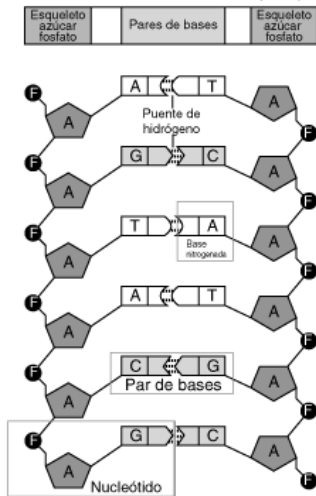
DNA, Codones y Nucleótidos

- ▶ El **Código Genético Universal** suele representarse como un mapa de 64 codones a 20 aminoácidos.
- ▶ Un **codón** es una triplete de nucleótidos $\langle N_1, N_2, N_3 \rangle$ donde $N_i \in \{A, C, G, T, \}$
- ▶ Los **nucleótidos** incluyen: Adenina (A), Citosina (C), Guanina (G), Timina (T).



Gráficamente

Ácido desoxiribonucleico (ADN)



Código Genético Universal

- Observen que el prompt ha cambiado al **paquete** :bioinfo.

```

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

```



Significancia

- ▶ La **significancia** de un nucleótido N_i , es fuertemente dependiente de su **posición** en el codón.
- ▶ Generalmente $N_2 > N_1 > N_3$.
- ▶ Con mayor probabilidad, mutaciones en N_3 producirán el mismo aminoácido, u otro con propiedades físico-químicas similares; no así en N_2 .
- ▶ La significancia también está relacionada con las propiedades **físico-químicas** de las bases.

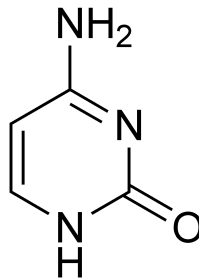
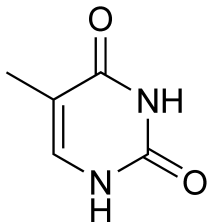


Propiedades Físico-Químicas

- ▶ Es posible representar el Código Genético como una 6-tupla de la forma $\langle C_1, H_1, C_2, H_2, C_3, H_3 \rangle$.
- ▶ C_i representa la **naturaleza química** del nucleótido N_i , es decir, si es una pirimidina (Y) o una purina (R).
- ▶ H_i representa la **fuerza del enlace de hidrógeno** del nucleótido N_i y su complemento, es decir, enlace fuerte (S) o débil (W).

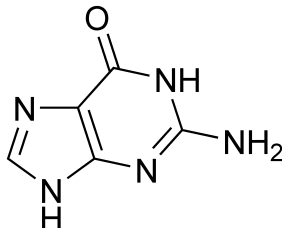
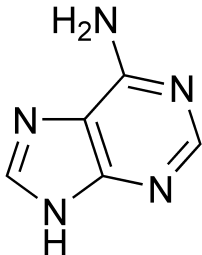


Pirimidinas: Timina y Citosina



Universidad Veracruzana

Purinas: Adenina y Guanina



Universidad Veracruzana

Fuerza de los Enlaces

$A=T$ enlace débil (doble enlace de hidrógeno)

$G\equiv C$ enlace fuerte (triple enlace de hidrógeno)



Universidad Veracruzana

Significancia de estas Propiedades

- ▶ M.A. Jiménez-Montaño propone el siguiente orden de significancia:

$$C_2 > H_2 > C_1 > H_1 > C_3 > H_3$$

- ▶ R. Swanson propone un orden alternativo:

$$C_2 > C_1 > H_2 > H_1 > C_3 > H_3$$

- ▶ ¿Cual es el bueno?
- ▶ Siguiendo a Mac Dónaill y Manktelow [3], Guerra-Hernández, Mora-Basáñez y Jiménez-Montaño [2] proponen un enfoque **cuantitativo** para establecer tales ordenamientos.



Cuantificando la Similitud

- ▶ Se propone **mutar** un nucleótido seleccionado ó una de sus propiedades y cuantificar el efecto de tal cambio..
- ▶ La matriz **PAM250** es una métrica ampliamente usada en análisis de similitud de secuencias genéticas.
- ▶ El elemento $s_{ij} \in PAM250$ es una cuantificación de la probabilidad de que un aminoácido i mute a j sobre un periodo evolutivo. Valores más pequeños (o negativos) reflejan disimilitud.
- ▶ Se pueden considerar otras matrices de similitud.



Significancia de la posición de los nucleótidos

1. Una posición $i \in \{1, 2, 3\}$ es seleccionada.
2. 12 mapeos son posibles: $T \rightarrow \{C, A, G\}$; $C \rightarrow \{A, G, T\}$;
 $A \rightarrow \{G, T, C\}$; $G \rightarrow \{T, C, A\}$.
3. Los mapeos se aplican a los 64 codones en la posición i , resultando en un código genético modificado $Gcode'$.
4. Los elementos s_{ij} de la **matriz de similitud** dan un valor numérico que mide la similitud de los aminoácidos i y j .



Continuación...

5. La **significancia global** S de un mapeo es computada a partir de los valores s_{ij} , sumados en los 64 codones::

$$S = \sum_{i,j=1\dots 64} S_{GCode_i Gcode'_j} \quad (1)$$

6. La **significancia promedio** $\sum_{i=1\dots 3} S_i/12$ se obtiene considerando los 12 posibles mapeos para cada posición de los nucleótidos.



Significancia de las propiedades de los nucleótidos

1. Una $i \in \{1, 2, 3\}$ y una propiedad del nucleótido C_i o H_i son seleccionados.
2. Dos mapeos son posibles, dependiendo de la propiedad seleccionada: $f_C : \{R \rightarrow Y, Y \rightarrow R\}$ o $f_H : \{W \rightarrow S, S \rightarrow W\}$. Básicamente, esto se corresponde con un cambio de bit de la propiedad seleccionada..
3. El mapeo correspondiente se aplica a los 64 codones en el Código Genético, resultando en un código modificado. table $GCode'$. Se procede como en los pasos 4 y 5 del experimento anterior.



Exp. 1: Similitud para las posiciones

- ▶ Como se esperaba $N_2 > N_1 > N_3$ para todas las matrices consideradas:

Matriz	N_1	N_2	N_3
PAM250	235.91	220.08	281.08
BLOSUM62	242.25	220.83	292.17
Prlić	157.48	142.38	185.40
Miyazawa	27.26	21.91	31.96
Sánchez	87.69	90.88	73.14



Exp. 2: Similitud para las propiedades físico-químicas

► Valores y ordenes obtenidos:

Matriz	C_1	H_1	C_2	H_2	C_3	H_3
PAM250	-16.00	16.00	-60.00	-32.00	122.00	276.00
BLOSUM62	-2.00	1.00	-76.00	-66.00	162.00	302.00
Prlić	18.14	29.34	-26.93	-30.83	111.54	189.84
Miyazawa	7.06	7.02	-18.71	-13.70	21.06	30.70
Sánchez	173.79	135.56	196.88	140.20	100.42	70.88

Matriz	Orden
PAM250	$C_2 > H_2 > C_1 > H_1 > C_3 > H_3$
BLOSUM62	$C_2 > H_2 > C_1 > H_1 > C_3 > H_3$
Prlić	$C_2 > H_2 > C_1 > H_1 > C_3 > H_3$
Miyazawa	$C_2 > H_2 > H_1 > C_1 > C_3 > H_3$
Sánchez	$C_2 > C_1 > H_2 > H_1 > C_3 > H_3$



Paquetes

► Definición y carga:

```
1 (defpackage :bioinfo
2   (:use :cl :ltk))
3
4 (in-package :bioinfo)
```

- Es necesario declarar el paquete y moverse dentro de él, **antes** de continuar con el resto de la implementación.
- `ltk` es la librería para el diseño de **interfaces gráficas** de basada en `tk`.
- Un paquete puede verse como un **mapeo** entre cadenas de texto y símbolos.



Símbolos y paquetes

No calificados. No contienen `:` en su nombre. Denotan al símbolo en el paquete actual. Ej. `*gc-tensor*`

Calificados. Contienen al menos un `:` en su nombre. Denotan al símbolo en un paquete dado. Pueden ser:

Externos. Explícitamente exportado por el paquete. Ej.
`bioinfo:*gc-tensor*`

Internos. No exportado por el paquete. Ej.
`bioinfo::*gc-tensor*`



Otras funciones sobre paquetes

- ▶ La variable especial `*package*` contiene el paquete actual:

```
1 > *package*  
2 #<The COMMON-LISP-USER package, 23/64 internal, 0/4 external>  
3 > (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>
```

- ▶ La directiva `:export` exporta símbolos.
- ▶ Los paquete pueden organizarse por capas , usando `use-package`.
- ▶ Vean el capítulo 21 del libro de Seibel [6].



Código Genético Universal

► Recuerden que estamos en el paquete :bioinfo

```

1 (defvar *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  "universal genetic code - tensor A")

```



Probando el paquete

- ▶ 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`, el acceso a los elementos de la matriz 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) ...
```



Matriz PAM250 I

- ▶ La distancia entre el aminoácido del renglón y la columna:

```

1  (defvar *pam250*
2  ;;; Dayhoff PAM250 (percent accepted mutations)
3  ;;; as reported by Mac Donaill, Molecular Simulation 30(5) p.269
4  (make-array
5  '(20 20)
6  :initial-contents
7  '(( 2 -2 0 0 -2 0 0 1 -1 -1 -2 -1 -1 -4 1 1 1 -6 -3 0)
8  (-2 6 0 -1 -4 1 -1 -3 2 -2 -3 3 0 -4 0 0 -1 2 -4 -2)
9  ( 0 0 2 2 -4 1 1 0 2 -2 -3 1 -2 -4 -1 1 0 -4 -2 -2)
10 ( 0 -1 2 4 -5 2 3 1 1 -2 -4 0 -3 -6 -1 0 0 -7 -4 -2)
11 (-2 -4 -4 -5 12 -5 -5 -3 -3 -2 -6 -5 -5 -4 -3 0 -2 -8 0 -2)
12 ( 0 1 1 2 -5 4 2 -1 3 -2 -2 1 -1 -5 0 -1 -1 -5 -4 -2)
13 ( 0 -1 1 3 -5 2 4 0 1 -2 -3 0 -2 -5 -1 0 0 -7 -4 -2)
14 ( 1 -3 0 1 -3 -1 0 5 -2 -3 -4 -2 -3 -5 -1 1 0 -7 -5 -1)
15 (-1 2 2 1 -3 3 1 -2 6 -2 -2 0 -2 -2 0 -1 -1 -3 0 -2)
16 (-1 -2 -2 -2 -2 -2 -2 -3 -2 5 2 -2 2 1 -2 -1 0 -5 -1 4)
17 (-2 -3 -3 -4 -6 -2 -3 -4 -2 2 6 -3 4 2 -3 -3 -2 -2 -1 2)
18 (-1 3 1 0 -5 1 0 -2 0 -2 -3 5 0 -5 -1 0 0 -3 -4 -2)
19 (-1 0 -2 -3 -5 -1 -2 -3 -2 2 4 0 6 0 -2 -2 -1 -4 -2 2)

```



Matriz PAM250 II

```
20      (-3 -4 -3 -6 -4 -5 -5 -5 -2  1  2 -5  0  9 -5 -3 -3  0  7 -1)
21      ( 1  0  0 -1 -3  0 -1  0  0 -2 -3 -1 -2 -5  6  1  0 -6 -5 -1)
22      ( 1  0  1  0  0 -1  0  1 -1 -1 -3  0 -2 -3  1  2  1 -2 -3 -1)
23      ( 1 -1  0  0 -2 -1  0  0 -1  0 -2  0 -1 -3  0  1  3 -5 -3  0)
24      (-6  2 -4 -7 -8 -5 -7 -7 -3 -5 -2 -3 -4  0 -6 -2 -5 17  0 -6)
25      (-3 -4 -2 -4  0 -4 -4 -5  0 -1 -1 -4 -2  7 -5 -3 -3  0 10 -2)
26      ( 0 -2 -2 -2 -2 -2 -2 -1 -2  4  2 -2  2 -1 -1 -1  0 -6 -2  4))))
```



Opciones como variable global

- ▶ Se puede definir una variable global para las **matrices** que podemos usar y la que usamos por defecto:

```

1 (defvar *options*
2   '(*pam250* *codegen* *robersy-grau* *miyazawa* *prlic* *blosum62*)
3   "Matrices")
4
5 (defvar *option* '*pam250* "Default option")

```

- ▶ Y aprovechando, otra para las posibles **mutaciones**:

```

1 (defvar *nucleotide-mappings*
2   ;;; The 12 possible nucleotide mappings for substitutions
3   '((T C) (T A) (T G)
4     (C A) (C G) (C T)
5     (A G) (A T) (A C)
6     (G T) (G C) (G A))
7   "the 12 possible nucleotide mappings")

```



Indexación de los aminoácidos I

- ▶ Regresa el índice para consultar la matriz de distancia:

```
1 (defun index (aa)
2   "Get the index of aminoacid aa in a matrix"
3   (case aa
4     ((A ALA) 0)
5     ((R ARG) 1)
6     ((N ASN) 2)
7     ((D ASP) 3)
8     ((C CYS) 4)
9     ((Q GLN) 5)
10    ((E GLU) 6)
11    ((G GLY) 7)
12    ((H HIS) 8)
13    ((I ILE) 9)
14    ((L LEU) 10)
15    ((K LYS) 11)
16    ((M MET) 12)
17    ((F PHE) 13)
18    ((P PRO) 14)
19    ((S SER) 15)
```



Indexación de los aminoácidos II

```
20      ((T THR) 16)
21      ((W TRP) 17)
22      ((Y TYR) 18)
23      ((V VAL) 19)
24      (ter 20))
```

▶ De manera que:

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



Distancias entre dos aminoácidos I

- Hay que poner especial cuidado con los terminadores.

```

1  (defun get-dist (aai aaj &optional (matrix *option*))
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    (cond ((and (equal aai 'ter) ;;; ter to ter transition case
7              (equal aaj 'ter))
8          (case matrix
9            (*pam250* 1)
10           (*codegen* 0)
11           (*miyazawa* 0)
12           (*prlic* 0)
13           (*blosum62* 0)
14           (*robersy-grau* (aref (eval matrix)
15                                (index aai)
16                                (index aaj)))
17           (otherwise (error "matrix not defined"))))
18    ((or (equal aai 'ter) ;;; aminoacid to ter or ter to aminoacid transit
19         (equal aaj 'ter))

```



Distancias entre dos aminoácidos II

```
20     (case matrix
21       (*pam250* -8)
22       (*codegen* 6)
23       (*miyazawa* -1.01)
24       (*prlic* 0)
25       (*blosum62* 0)
26       (*robersy-grau* (aref (eval matrix)
27                           (index aai)
28                           (index aaj)))
29       (otherwise (error "matrix not defined")))
30     (t (aref (eval matrix) (index aai) (index aaj))))
```

▶ De manera que:

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



Nucleótidos a aminoácidos y viceversa

► Funciones de conversión:

```

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*)))

```

► Corridas:

```

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)

```



Imprimir código genético universal

► Función para imprimir tensores en 4 columnas:

```
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) (caddrr tensor))
7              (print-tensor (cddddr tensor))))))
```



O usando solo format

```

1  BIOINFO> (format t "~{~a ~a ~a ~a ~%~}" *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  NIL

```



Experimento 1 I

```

1  (defun apply-mapping (pos mapping)
2    ;;; apply a mapping in the position pos
3    (mapcar #'(lambda(mmm)(cons mmm (cdr (nnn-to-aa mmm))))
4      (mapcar #'(lambda(nnn)(change pos (car nnn) mapping))
5        *gc-tensor*)))
6
7  (defun change (pos nnn mapping)
8    ;;; auxiliar for apply-mapping, it does only one change nnn
9    (let* ((nnnaux (copy-list nnn))
10      (from-n (first mapping))
11      (to-n (second mapping))
12      (elem (nth (- pos 1) nnnaux)))
13      (when (eql elem from-n)
14        (setf (nth (- pos 1) nnnaux) to-n))
15      nnnaux))
16
17  (defun apply-dist (tensor &optional (matrix *option*))
18    ;;; Tensor is intended to be tensor B, the modified one
19    ;;; The result is tensor C, the distance of exchanges (by default
20    ;;; *pam250*) between tensor B and *gc-tensor* A.

```



Experimento 1 II

```
21   ;;; Anyway (S (apply-pam250 *gc-tensor*)) = N (see below) = 317
22   (mapcar #'(lambda(i j)
23             (list (car i) (get-dist (cadr i) (cadr j) matrix)))
24             *gc-tensor* tensor))
```



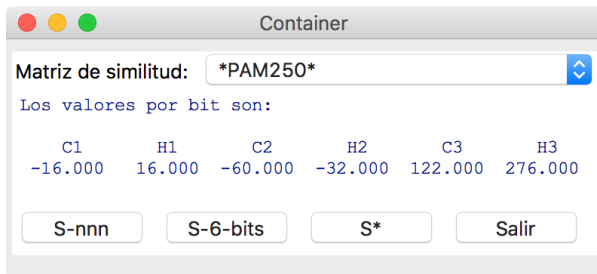
S-nnn

```
1 (defun S-nnn (pos &optional (matrix *option*))
2   ;;; computes average S for a given position in the codon
3   (/ (float (apply #'(+
4           (mapcar #'(lambda(ldists)
5                       (S-aux ldists matrix))
6                       (mapcar #'(lambda (lmaps)
7                                   (apply-mapping pos lmaps))
8                                   *nucleotide-mappings*))))))
9     12.0))
10
11 (defun S-aux (tensor &optional (matrix *option*))
12   ;;; Computes S = Sum Cijk
13   ;;; The overall significance of a selected mapping
14   (apply #'(+ (mapcar #'(lambda(l) (cadr l))
15                       (apply-dist tensor matrix))))
```



Interfaz gráfica

- ▶ ¿Como podemos programar una interfaz como la siguiente?

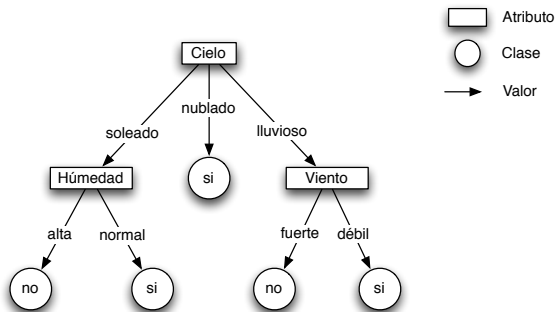


- ▶ Con ltk, ver gui/0.



Árboles de Decisión

- ▶ Los árboles de decisión representan hipótesis sobre una clase, como una **conjunción de disyunciones** de atributos.
- ▶ Recordando un buen día para jugar tenis [4]:



Ejemplos de entrenamiento

► Representación proposicional (atributo-valor)

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



Arboles de Decisión

- Podemos representarlos como una lista de listas:

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



Funciones de acceso al árbol

► Algunas funciones básicas de acceso:

```
1 (defun root (tree)
2   "Returns the root of the decision tree"
3   (car tree))
4
5 (defun sub-tree (tree value)
6   "Returns the sub-tree under the branch labeled as value"
7   (second (assoc value (cdr tree))))
8
9 (defun leaf-p (tree)
10  "Verifies if tree is a leaf"
11  (atom tree))
```



Ejemplos

► Dado nuestro árbol inicial tenemos que:

```
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 (sub-tree arbol 'soleado))
8 NIL
9 CL-USER> (leaf (sub-tree arbol 'nublado))
10 T
```



¿Cómo funciona assoc?

- ▶ Regresa el primer cons de su segundo argumento (una lista), cuyo car es igual a su primer argumento.
- ▶ Hay variantes assoc-if y assoc-if-not.

```
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)  
5 CL-USER> (assoc 'lluvia (cdr arbol))  
6 (LLUVIA (VIENTO (FUERTE NO) (DEBIL SI)))
```



Imprimiendo un árbol de decisión

- Observen el uso de loop y terpri:

```

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
10        (terpri)
11        (print-tree (second subtree) (+ depth 5))))))
12
13 (defun mytab (n)
14   (loop for i from 1 to n do (format t " ")))

```



Ejemplo

► Imprimiendo nuestro árbol:

```
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
```



Ejemplos de entrenamiento

- ▶ Aunque los ejemplos también pueden representarse como una lista de listas, esto no es buena idea.
- ▶ Necesitamos una representación que nos permita **relacionar** el valor de un atributo en un ejemplo dado.
- ▶ **Ejemplo:** Supongan que `*examples*` es mi repositorio de ejemplos:

```
1 CL-USER> (get-value 'cielo (car *examples*))  
2 LLUVIA
```



Ejemplos, atributos y valores

- ▶ Las bases para la **indexación** son las siguientes:

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

- ▶ Ejemplo:

```
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
```



Archivos de ejemplos: ARFF y CSV

► Un archivo `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}
```

```
@DATA
```

```
soleado, calor, alta, debil, no
```

```
soleado, calor, alta, fuerte, no
```

```
nublado, calor, alta, debil, si
```

```
lluvia, templado, alta, debil, si
```

```
...
```



With a little help from your friends

- ▶ ASFD2 (*Another System Definition Facility 2*) es una herramienta para describir como están **organizados** los **archivos fuente** de un sistema Lisp.
- ▶ En particular sus **dependencias**.
- ▶ Quicklisp es una herramienta que nos permite **instalar** librerías que han sido definidas usando ASDF2:

`http://www.quicklisp.org`

- ▶ **Librerías** curadas a mano:

`https://awesome-cl.com`



Instalando librerías: trivial-shell

- ▶ trivial-shell permite ejecutar comandos del shell de Unix.
- ▶ Su instalación con quicklisp es como sigue:

```
1 CL-USER> (ql:quickload "trivial-shell")
2 To load "trivial-shell":
3   Install 1 Quicklisp release:
4     trivial-shell
5   ; Fetching #<QL-HTTP:URL
6   "http://beta.quicklisp.org/archive/trivial-shell/
7   2011-05-22/trivial-shell-20110522-http.tgz">
8   ; 13.61KB
9   =====
10  13,937 bytes in 0.00 seconds (13610.35KB/sec)
11  ; Loading "trivial-shell"
12  [package com.metabang.trivial-timeout].....
13  [package trivial-shell]..
14  ("trivial-shell")
```



Ejecutando comandos shell: grep

- ▶ De forma que podemos usar egrep para extraer líneas del archivo ARFF:

```
1 CL-USER> (trivial-shell:shell-command
2           "egrep \"@ATTR\" tenis.arff")
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 NIL
10 0
```



Partiendo cadenas: split-sequence

- Podemos llamar a `split-sequence` para partir cadenas de texto:

```
1 CL-USER> (ql:quickload "split-sequence")
2 To load "split-sequence":
3   Load 1 ASDF system:
4     split-sequence
5 ; Loading "split-sequence"
6
7 ("split-sequence")
8 CL-USER> (split-sequence:split-sequence #\, "hola, que, tal")
9 ("hola" " que" " tal")
10 14
```



Ensamblando cajitas

- Podemos usar estas facilidades para leer las líneas del archivo ARFF:

```
1 CL-USER> (split-sequence:split-sequence #\Newline
2                                     (trivial-shell:shell-command
3                                     "egrep \"@ATTR\" tenis.arff"))
4 ("@ATTRIBUTE cielo {soleado,nublado,lluvia}"
5  "@ATTRIBUTE temperatura {calor,templado,frio}"
6  "@ATTRIBUTE humedad {alta,normal}" "@ATTRIBUTE viento {debil,fuerte}"
7  "@ATTRIBUTE jugar-tenis {si,no}" "")
8 184
```



Variables globales

- ▶ Definamos variables globales para nuestro ambiente de trabajo:

```
1 (defvar *attributes* nil "The attributes of the problem")
2 (defvar *domains* nil "The domain of the attributes")
3 (defvar *target* nil "The target concept")
4 (defvar *examples* nil "The training set")
5 (defvar *trace* nil "Trace the computations")
```



Datos crudos

- ▶ Leyendo los datos del conjunto de entrenamiento (líneas que no comienzan con salto de línea o arroba):

```
1 (defun get-data (arff)
2   (mapcar #'(lambda(x)
3     (mapcar #'read-from-string
4       (split-sequence:split-sequence #\, x)))
5     (butlast
6       (split-sequence:split-sequence
7         #\Newline
8         (trivial-shell:shell-command
9           (concatenate 'string
10            "egrep -v \"~$|[@%]\" "
11            arff))))))
```



Ejemplo

► Leyendo los ejemplos de entrenamiento:

```
1 CL-USER> (get-data "tenis.arff")
2 ((SOLEADO CALOR ALTA DEBIL NO) (SOLEADO CALOR ALTA FUERTE NO) (NUBLADO
3 CALOR ALTA DEBIL SI) (LLUVIA TEMPLADO ALTA DEBIL SI) (LLUVIA FRIO
4 NORMAL DEBIL SI) (LLUVIA FRIO NORMAL FUERTE NO) (NUBLADO FRIO NORMAL
5 FUERTE SI) (SOLEADO TEMPLADO ALTA DEBIL NO) (SOLEADO FRIO NORMAL DEBIL
6 SI) (LLUVIA TEMPLADO NORMAL DEBIL SI) (SOLEADO TEMPLADO NORMAL FUERTE
7 SI) (NUBLADO TEMPLADO ALTA FUERTE SI) (NUBLADO CALOR NORMAL DEBIL SI)
8 (LLUVIA TEMPLADO ALTA FUERTE NO))
```



Clase a predecir

► Recuperando la clase a predecir (*target*):

```

1 (defun get-target (arff)
2   (read-from-string
3     (cadr (split-sequence:split-sequence
4           #\Space
5           (car
6             (butlast
7               (split-sequence:split-sequence
8                 #\Newline
9                 (trivial-shell:shell-command
10                  (concatenate 'string
11                               "egrep \"@rel|@REL\" "
12                               arff))))))))))

```

```

1 CL-USER> (get-target "tenis.arff")
2 JUGAR-TENIS
3 11

```



Dominios

► Obteniendo el dominio de un atributo:

```
1 (defun get-domain (attribute)
2   (nth (position attribute *attributes*)
3       *domains*))
```



Todo junto

► Leyendo un ARFF:

```
1 (defun id3-load (arff)
2   (let ((data (get-data arff))
3         (attribs-doms (get-attrs-doms arff)))
4     (setf *gensym-counter* 1)
5     (setf *attributes* (mapcar #'car attribs-doms))
6     (setf *domains* (mapcar #'cadr attribs-doms))
7     (setf *target* (get-target arff))
8     (loop for d in data do
9       (let ((ej (gensym "ej")))
10        (push ej *examples*)
11        (loop for attrib in *attributes*
12              as v in d do
13              (put-value attrib ej v))))
14     (format t "Training set initialized"))
```



Ejemplo

► Cargando los ejemplos de tenis:

```
1 CL-USER> (id3-load "tenis.arff")
2 Training set initialized
3 NIL
4 CL-USER> *examples*
5 (#:lej14| #:lej13| #:lej12| #:lej11| #:lej10| #:lej9| #:lej8| #:lej7|
6 #:lej6| #:lej5| #:lej4| #:lej3| #:lej2| #:lej1|)
7 CL-USER> *attributes*
8 (CIELO TEMPERATURA HUMEDAD VIENTO JUGAR-TENIS)
9 CL-USER> *domains*
10 ((SOLEADO NUBLADO LLUVIA) (CALOR TEMPLADO FRIO) (ALTA NORMAL)
11 (DEBIL FUERTE) (SI NO))
```



Y el reseteo

▶ Limpiando las variables globales:

```
1 (defun id3-reset ()
2   (setf *examples* nil
3         *target* nil
4         *attributes* nil
5         *domains* nil
6         *trace* nil
7         *root* nil)
8   (format t "The ID3 setting has been reset."))
```



Definición de un sistema ASDF: cl-id3.asd

```
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-cross-validation"
10                      :depends-on ("cl-id3-package"
11                                   "cl-id3-algorithm")))
12                (:file "cl-id3-classify"
13                      :depends-on ("cl-id3-package"
14                                   "cl-id3-algorithm"
15                                   "cl-id3-load"))))
```



Definiendo un paquete: cl-id3-package

```
1 (defpackage :cl-id3
2   (:use :cl :split-sequence)
3   (:export :load-file
4            :induce
5            :print-tree
6            :classify
7            :classify-new-instance
8            :cross-validation))
```



Leyendo archivos: cl-id3-load

```
1 (in-package :cl-id3)
2
3 ;;; Auxiliar functions
4
5 (defun read-lines-from-file (file)
6   "It reads the FILE into a list of strings"
7   (remove-if (lambda (x) (equal x ""))
8             (with-open-file (in file)
9               (loop for line = (read-line in nil 'end)
10                  until (eq line 'end) collect line))))
```



Portabilidad: Bye, bye grep

```
1 (defun arff-get-data (lines)
2   "It extracts the value for *data* from the lines of a ARFF file"
3   (mapcar #'(lambda(x)
4             (mapcar #'read-from-string
5                   (split-sequence #\, x)))
6         (remove-if
7         (lambda (x) (string-equal "@" (subseq x 0 1)))
8         lines)))
9
10 (defun csv-get-data (lines)
11   "It extracts the value for *data* from the lines of a CSV file"
12   (mapcar #'(lambda(x)
13             (mapcar #'read-from-string
14                   (split-sequence #\, x))) (cdr lines)))
```



Ejemplo:

▶ Ejecutando arff-get-data (más cajitas):

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



Load I

```

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-attrs-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))

```



Load II

```
21         (get-examples *data*)
22         (format t "Training setting 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        ...
27        (t (error "File's ~s extension can not be determined." file))))
28 (error "File ~s does not exist.~%" file)))
```



Ejemplo

► Cargando un archivo:

```
1 CL-ID3> (load-file "tenis.arff")
2 The ID3 setting has been reset.
3 Training setting 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 (#:lej14| #:lej13| #:lej12| #:lej11| #:lej10| #:lej9| #:lej8|
11 #:lej7| #:lej6| #:lej5| #:lej4| #:lej3| #:lej2| #:lej1|)
```



Algoritmo principal

```

1 (defun id3 (examples attribs)
2   "It induces a decision tree runing ID3 over EXAMPLES and ATTRIBS)"
3   (let ((class-by-default (get-value *target* (car examples))))
4     (cond
5       ;; Stop criteria
6       ((same-class-value-p *target*
7                            class-by-default
8                            examples) class-by-default)
9       ;; Failure
10      ((null attribs) (target-most-common-value examples))
11      ;; Recursive call
12      (t (let* ((partition (best-partition attribs examples))
13              (node (first partition)))
14          (cons node
15                (loop for branch in (cdr partition) collect
16                      (list (first branch)
17                            (id3 (cdr branch)
18                                (remove node attribs))))))))))

```



¿Todos somos de la misma clase?

► Criterio terminal:

```
1 (defun same-class-value-p (attrib value examples)
2   "Do all EXAMPLES have the same VALUE for a given ATTRIB ?"
3   (every #'(lambda(e)
4             (eq value
5               (get-value attrib e)))
6         examples))
```

► Ejemplo:

```
1 CL-ID3> (same-class-value-p *target* 'si *examples*)
2 NIL
3 CL-ID3> (same-class-value-p *target* 'no *examples*)
4 NIL
```



Valor mayoritario de Clase

```
1 (defun target-most-common-value (examples)
2   "It gets the most common value for *target* in EXAMPLES"
3   (let ((domain (get-domain *target*)))
4     (vals (mapcar #'(lambda(x) (get-value *target* x))
5                 examples)))
6   (caar (sort (loop for v in domain collect
7                 (list v (count v vals)))
8               #'(lambda(x y) (>= (cadr x)
9                                   (cadr y)))))))
```

```
1 CL-ID3> (target-most-common-value *examples*)
2 SI
```



Partición

```

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)))

```

```

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

```



Entropía

```

1 (defun entropy (examples attrib)
2   "It computes the entropy of EXAMPLES with respect to an ATTRIB"
3   (let ((partition (get-partition attrib examples))
4         (number-of-examples (length examples)))
5     (apply #'+
6           (mapcar #'(lambda(part)
7                       (let* ((size-part (count-if #'atom
8                                                    (cdr part)))
9                              (proportion
10                               (if (eq size-part 0) 0
11                                   (/ size-part
12                                     number-of-examples))))
13                               (* -1.0 proportion (log proportion 2))))
14         (cdr partition))))))

```

```

1 CL-ID3> (entropy *examples* 'jugar-tenis)
2 0.9402859

```



Ganancia de Información

```

1 (defun information-gain (examples attribute)
2   "It computes information-gain for an ATTRIBUTE in EXAMPLES"
3   (let ((parts (get-partition attribute examples))
4         (no-examples (count-if #'atom examples)))
5     (- (entropy examples *target*)
6        (apply #'+
7              (mapcar
8                #'(lambda(part)
9                  (let* ((size-part (count-if #'atom
10                                             (cdr part)))
11                       (proportion (if (eq size-part 0) 0
12                                       (/ size-part
13                                           no-examples))))
14                    (* proportion (entropy (cdr part) *target*))))
15              (cdr parts))))))

```

```

1 CL-ID3> (information-gain *examples* 'cielo)
2 0.2467497

```



Mejor partición

```

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

```

```

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

```



Induciendo un árbol

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

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



Y con print-tree

```
1 CL-ID3> (print-tree (induce))
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
```



Cargando el sistema ASDF

- ▶ Copiar la carpeta del proyecto `cl-id3` en el directorio `common-lisp` en el `home` del usuario.
- ▶ En SBCL ejecutar:

```
1 CL-USER> (asdf:load-system :cl-id3)
```



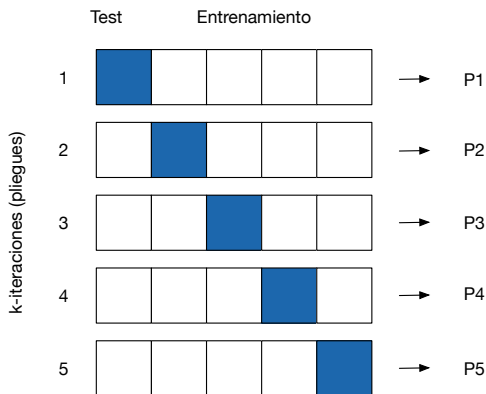
Prueba

```
1 CL-USER> (cl-id3:load-file "tenis.csv")
2 The ID3 setting has been reset.
3 Training set initialized after "tenis.csv".
4 NIL
5 CL-USER> cl-id3::*attributes*
6 (CIELO HUMEDAD TEMPERATURA VIENTO JUGAR-TENIS)
7 CL-USER> (cl-id3:print-tree (cl-id3:induce))
8 CIELO
9 - SOLEADO
10   TEMPERATURA
11     - NORMAL -> SI
12     - ALTA -> NO
13 - NUBLADO -> SI
14 - LLUVIA
15   VIENTO
16     - FUERTE -> NO
17     - DEBIL -> SI
18 NIL
```



Gráficamente

- Precisión promedio, para $k = 5$: $\frac{1}{5} \sum_{i=1}^5 P_i$



Funciones auxiliares

```
1 (defun square (n)
2   (* n n))
3
4 (defun mean (data)
5   (/ (reduce #' + data) (length data)))
6
7 (defun std-dev (data)
8   (let ((data-mean (mean data)))
9     (sqrt (/ (reduce #' + (mapcar
10                #'(lambda (x)
11                    (square (- data-mean x)))
12                data))
13            (length data))))))
```



Cross-validation

```

1 (defun cross-validation (k)
2   "Cross-validations with k sets of size 1 - (/ 1 k)"
3   (let* ((long (length *examples*))
4          (n-training (- long (floor long k)))
5          (n-testing (- long n-training))
6          (results (loop repeat k
7                        collect
8                          (let* ((training-data (folding n-training long))
9                                 (test-data (set-difference *examples* trainin
10                                 (tree (induce training-data)))
11                                 (/ (report tree test-data) n-testing))))))
12   ;; Estadísticas sobre todos los folds
13   (format t "~% Precision promedio: ~S~% Desviacion std: ~S~%"
14           (* 1.0 (mean results))
15           (std-dev results))))
16

```



Folding

```
1 (defun folding (n size)
2   (let ((buffer nil))
3     (loop repeat n
4       collect
5         (nth (let ((r (random size)))
6               (progn
7                 (while (member r buffer)
8                   (setf r (random size)))
9                 (push r buffer)
10                r)) *examples*))))
```



Prueba I

```
1 CL-USER> (cl-id3:load-file "~/common-lisp/cl-id3/tenis.csv")
2 The ID3 setting has been reset.
3 Training set initialized after "~/common-lisp/cl-id3/tenis.csv".
4 NIL
5 CL-USER> (cl-id3:cross-validation 2)
6 CIELO
7   - NUBLADO -> SI
8   - SOLEADO -> NO
9   - LLUVIA
10      VIENTO
11     - FUERTE -> NO
12     - DEBIL -> SI
13
14 Ejemplos bien clasificados: 5
15 Ejemplos mal clasificados: 2
16
17 CIELO
18   - SOLEADO -> NO
19   - LLUVIA
20      VIENTO
```



Prueba II

```
21         - FUERTE -> NO
22         - DEBIL  -> SI
23     - NUBLADO -> SI
24
25     Ejemplos bien clasificados: 5
26     Ejemplos mal clasificados: 2
27
28
29     Precision promedio: 0.71428573
30     Desviacion std: 0.0
```



Amazon, Reddit & Co.

- ▶ La idea es usar las **preferencias** de los usuarios, para dar **recomendaciones** a otros usuarios.
- ▶ **Ejemplo:** Amazon, Netflix, iTunes Store, Reddit, etc.

Customers Who Bought This Item Also Bought

Page 1 of 18

The screenshot shows a carousel of five Haskell-related books. Each book card includes a cover image, title, author, star rating, and price. Navigation arrows are visible on the left and right sides of the carousel.

Book Title	Author	Rating	Price
Learn You a Haskell for Great Good!: A Beginner...	Miran Lipovaca	★★★★★ (22)	\$25.57
An Introduction to Functional Programming Thro...	Greg Michaelson	★★★★★ (1)	\$15.18
Purely Functional Data Structures	Chris Okasaki	★★★★☆ (10)	\$46.05
The Haskell Road to Logic, Maths and Programming (T...	Kees Doets	★★★★★ (9)	\$22.29
Real World Haskell	Bryan O'Sullivan	★★★★☆ (29)	\$37.23

- ▶ Seguiremos la presentación de Seagaran [5].



Universidad Veracruzana

Filtrado Colaborativo

- ▶ Se trata de encontrar usuarios con gustos afines y usar sus preferencias para emitir recomendaciones.
- ▶ El término fue acuñado en Xerox PARC, ¿Donde más?, en 1992 por Goldberg et al. [1].



Preferencias

- ▶ Asumamos 7 críticos para 5 películas.
- ▶ La escala de calificación va de 0 a 5.

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

- ▶ Observen que no todos los críticos revisan todas la películas.



Representando preferencias

- ▶ Usaremos listas de propiedades:

```
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  ...)
```



Funciones de acceso

```
1 (defun get-critic (critic)
2   (find critic *critics*
3     :test #'(lambda(name reg)
4       (string= name (getf reg :critic))))))
5
6 (defun get-film-rating (critic film)
7   (cadr (find film (getf (get-critic critic) :critics)
8     :test #'(lambda(film reg)
9       (string= film (car reg))))))
10
11 (defun get-films-rated-by (person)
12   (mapcar #'car (fourth (get-critic person))))
```



Ejemplos de uso

```
1 CL-USER> (get-critic "Toby")
2 (:CRITIC "Toby" :CRITICS ((("Snakes on a Plane" 4.5) ("Superman Returns" 4.0)
3 ("You, Me and Dupree" 1.0)))
4 CL-USER> (get-film-rating "Toby" "Superman Returns")
5 4.0
6 CL-USER> (get-films-rated-by "Toby")
7 ("Snakes on a Plane" "Superman Returns" "You, Me and Dupree")
```



Productos en común

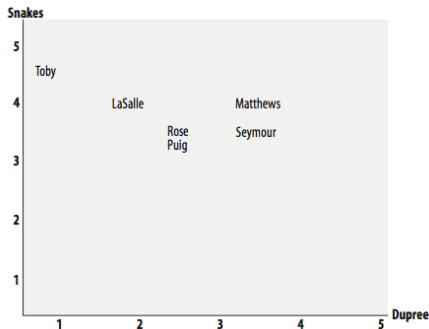
```
1 (defun shared-items (critic1 critic2)
2   (intersection (get-films-rated-by critic1)
3                 (get-films-rated-by critic2)
4                 :test #'string= ))
```

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



Distancia Euclidiana y Espacio de Preferencias

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



► Normalizaremos la distancia en $[0, \dots, 1]$.



Computando

```
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))
```

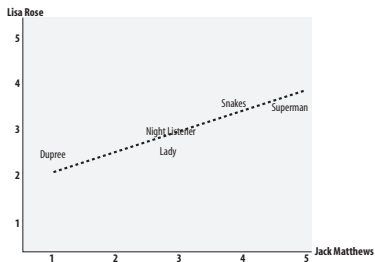
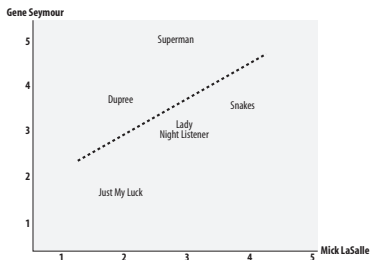
```
1 CL-USER> (sim-euclidean "Lisa Rose" "Gene Seymour")
2 0.29429805
3 CL-USER> (sim-euclidean "Lisa Rose" "Claudia Puig")
4 0.38742587
```

Puig tiene preferencias más cercanas a las de Rose que Seymour.



Correlación de Pearson

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



- Resuelve **corrimientos** en la escala.



Computando

```

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) (* x y)) prefs1 prefs2)))
16               (num (- psum (/ (* sum1 sum2) n)))
17               (den (sqrt (* (- sum1sq (/ (sqr sum1) n))
18                               (- sum2sq (/ (sqr sum2) n))))))
19         (if (zerop den)
20             1
21             (/ num den))))))

```



Ejemplos

```
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
```



Y el top de los críticos es...

```
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=))
```

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



Emitiendo recomendaciones

```

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 (apply #'(lambda (critic)
14                                  (mapcar #'(lambda (critic)
15                                              (let ((rating
16                                                  (get-film-rating critic film))
17                                                  (sim
18                                                    (funcall similarity person critic)))
19                                                  (if rating
20                                                    (progn
21                                                      (setf sim-sums (+ sim-sums sim))
22                                                      (* rating sim))
23                                                    0)))
24                                  other-critics))))
25            (list (/ total sim-sums) film)))
26     #'(lambda(x y) (> (car x) (car y))))))

```



Ejemplos

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



Invirtiendo la lista de preferencias

- ▶ Se trata de indexar ahora por película:

```
1 (defun transform-prefs ()
2   "Inverts the dictionary *critics*, indexing it by film"
3   (let* ((critics (loop for critic in *critics* collect
4                       (getf critic :critic)))
5          (films (reduce #'(lambda (x y) (union x y :test #'string=))
6                          (loop for critic in critics collect
7                              (get-films-rated-by critic))))
8     (loop for film in films collect
9           (list :critic film
10                :critics (remove-if #'(lambda(x) (null (cadr x)))
11                                   (mapcar #'(lambda (c)
12                                           (list c (get-film-rating c film)))
13                                           critics))))))
```



Transformación

► El diccionario luce ahora así:

```

1 CL-USER> (transform-prefs)
2 ((:CRITIC "Lady in the Water" :CRITICS (("Lisa Rose" 2.5) ("Gene
3 Seymour" 3.0) ("Michael Phillips" 2.5) ("Mick LaSalle" 3.0) ("Jack
4 Matthews" 3.0))) (:CRITIC "Snakes on a Plane" :CRITICS (("Lisa Rose"
5 3.5) ("Gene Seymour" 3.5) ("Michael Phillips" 3.0) ("Claudia Puig"
6 3.5) ("Mick LaSalle" 4.0) ("Jack Matthews" 4.0) ("Toby" 4.5)))
7 (:CRITIC "Just My Luck" :CRITICS (("Lisa Rose" 3.0) ("Gene Seymour"
8 1.5) ("Claudia Puig" 3.0) ("Mick LaSalle" 2.0))) (:CRITIC "Superman
9 Returns" :CRITICS (("Lisa Rose" 3.5) ("Gene Seymour" 5.0) ("Michael
10 Phillips" 3.5) ("Claudia Puig" 4.0) ("Mick LaSalle" 3.0) ("Jack
11 Matthews" 5.0) ("Toby" 4.0))) (:CRITIC "You, Me and Dupree" :CRITICS
12 (("Lisa Rose" 2.5) ("Gene Seymour" 3.5) ("Claudia Puig" 2.5) ("Mick
13 LaSalle" 2.0) ("Jack Matthews" 3.5) ("Toby" 1.0))) (:CRITIC "The Night
14 Listener" :CRITICS (("Lisa Rose" 3.0) ("Gene Seymour" 3.0) ("Michael
15 Phillips" 4.0) ("Claudia Puig" 4.5) ("Mick LaSalle" 3.0) ("Jack
16 Matthews" 3.0)))

```

► Así podemos usar las funciones ya programadas.



Corrida

► ¿Qué películas se parecen a Superman Returns?

```

1 CL-USER> (setf *critics* (transform-prefs))
2 ((:CRITIC "Lady in the Water" :CRITICS (("Lisa Rose" 2.5) ("Gene
3 Seymour" 3.0) ("Michael Phillips" 2.5) ("Mick LaSalle" 3.0) ("Jack
4 Matthews" 3.0))) (:CRITIC "Snakes on a Plane" :CRITICS (("Lisa Rose"
5 3.5) ("Gene Seymour" 3.5) ("Michael Phillips" 3.0) ("Claudia Puig"
6 3.5) ("Mick LaSalle" 4.0) ("Jack Matthews" 4.0) ("Toby" 4.5)))
7 (:CRITIC "Just My Luck" :CRITICS (("Lisa Rose" 3.0) ("Gene Seymour"
8 1.5) ("Claudia Puig" 3.0) ("Mick LaSalle" 2.0))) (:CRITIC "Superman
9 Returns" :CRITICS (("Lisa Rose" 3.5) ("Gene Seymour" 5.0) ("Michael
10 Phillips" 3.5) ("Claudia Puig" 4.0) ("Mick LaSalle" 3.0) ("Jack
11 Matthews" 5.0) ("Toby" 4.0))) (:CRITIC "You, Me and Dupree" :CRITICS
12 (("Lisa Rose" 2.5) ("Gene Seymour" 3.5) ("Claudia Puig" 2.5) ("Mick
13 LaSalle" 2.0) ("Jack Matthews" 3.5) ("Toby" 1.0))) (:CRITIC "The Night
14 Listener" :CRITICS (("Lisa Rose" 3.0) ("Gene Seymour" 3.0) ("Michael
15 Phillips" 4.0) ("Claudia Puig" 4.5) ("Mick LaSalle" 3.0) ("Jack
16 Matthews" 3.0))))
17 CL-USER> (top-matches "Superman Returns" 3)
18 (("You, Me and Dupree" 0.6579514) ("Lady in the Water" 0.48795068)
19 ("Snakes on a Plane" 0.111803316))

```



Otra corrida

▶ ¿Quién recomienda Just My Luck?

```
1 CL-USER> (get-recommendations "Just My Luck")
2 ((3.8716946 "Jack Matthews") (2.9610002 "Toby")
3  (2.2872024 "Michael Phillips"))
```

▶ ¿Y The Night Listener?

```
1 CL-USER> (get-recommendations "The Night Listener")
2 ((3.5313943 "Toby"))
```



Referencias I

- [1] D Goldberg et al. "Using collaborative filtering to weave an information tapestry". En: *Communications of the ACM* 35.12 (1992), págs. 61-71.
- [2] A Guerra-Hernández, CR de-la Mora-Basáñez y MA Jiménez-Montaño. "The significance of nucleotides within DNA codons: a quantitative approach.". En: *Avances en la Ciencia de la Computación: VI Encuentro Internacional de Computación ENC'05*. Ed. por L Villaseñor-Pineda y A Martínez-García. Sociedad Mexicana de Ciencias de la Computación (SMCC). Puebla, Pue., México: Benemérita Universidad Autónoma de Puebla, 2005, págs. 167-169.
- [3] DA Mac Dónaill y M Manktelow. "Molecular informatics: quantifying information patterns in the genetic code". En: *Molecular Simulation* 30.5 (2004), págs. 267-272.
- [4] T Mitchell. *Machine Learning*. Computer Science Series. Singapore: McGraw-Hill International Editions, 1997.
- [5] T Seagaran. *Programming Collective Intelligence*. Sebastopol, CA, USA: O'Reilly Media, 2008.
- [6] P Seibel. *Practical Common Lisp*. New York, NY, USA: Apress, 2005.

