Métodos de Aprendizaje en IA. Guía de prácticas.
Bloque 3: Representaciones Compuestas.

Capítulo 1

Tema 5: La Construcción de Listas de Decisión

En este tema se introducirán representaciones compuestas a partir de las estudiadas en el bloque 2. En particular, construiremos listas de decisión que se componen de conjunciones lógicas (tema 2) y/o conceptos umbral (tema 3). En este tema no profundizaremos en estructuras que incorporan conceptos de tipo competitivo (tema 4).

Se recomienda tener muy presente el documento [SRFI-1], ya que se hará un uso intensivo de las listas Scheme.

Durante el curso, se añadirá una plantilla de ejercicios para *normalizar* (de alguna forma) las entregas de todos vosotros. Para una mejor planificación del tiempo requerido por cada ejercicio, se recomienda que en la entrega incluyáis el tiempo real que os ha llevado realizar cada ejercicio. Así en el futuro podamos mejorar la planificación temporal de la asignatura. Con este objetivo se debe rellenar `(he-tardado <minutos><ejercicio>)` en cada plantilla[1]. **Importante:** el tiempo estimado no se tendrá en cuenta en ningún caso para evaluar el ejercicio.

1.1. Representación de los conceptos Listas de Decisión (LD)

Requisitos: estudio de la sección 5.1 [EML].

Para representar los conceptos LD de este tema utilizaremos listas de *reglas* cuya precondición ó Left Hand Side (LHS) estará compuesta por conceptos procedentes de los temas 2 y 3. Para facilitar la tarea de programación, utilizaremos una estructura como la siguiente:

```
( (<función match> <concepto> => <clase>) ...)
```

[1]Para poder cargar vuestro código sin errores, se añade a cada plantilla el código siguiente: `(define (he-tardado minutos ejercicio) ())`.

Por ejemplo,

```
((match-CL ((soleado)(30 40)(50 60)(*)) => +)
 (match-CL ((*)(*)(*)(*)) => -))
```

Se puede observar el parecido de esta estructura con las reglas de producción en cierto tipo de sistemas expertos. De hecho, los algoritmos que se incluyen en este tema suelen ser denominados habitualmente algoritmos de extracción de reglas o de aprendizaje de reglas. En cualquier caso, debe hacerse hincapié en que una *lista de decisión* no es un *conjunto de reglas* (ver pág 117, primer párrafo [EML]).

Ejercicio 1. ***[1h]*** *Construya la función* ***(LDi <concepto-LD><ejemplo-sin-clase>)*** *que devuelva la clase asociada a la* primera *regla en la lista de decisión en que la* ***<función match>*** *devuelva* ***#t*** *al ser aplicada sobre* ***<concepto>*** *y* ***<ejemplo-sin-clase>****. Si en dicha lista no se cumpliera nunca lo anterior, se devolvería el ejemplo sin clase. Por ejemplo,*

```
#(LDi '((match-CL ((soleado)(30 40)(50 60)(*)) => +)
        (match-CL ((*)(*)(*)(*)) => -))
      '(soleado 30 40 si))

(soleado 30 40 si -)

#(LDi `((match-LUU ,(list(car ejemplos)'(0 1 1 0 -30)) => +)
        (match-LUU ,(list(car ejemplos)'(0 1 -1 0 -20)) => -))
     '(soleado 30 40 si))

(soleado 30 40 si +)

#(LDi `((match-LUU ,(list(car ejemplos)'(0 1 1 0 -30)) => +)
        (match-LUU ,(list(car ejemplos)'(0 1 -1 0 -20)) => -))
     '(soleado 10 10 si))

(soleado 30 40 si)
```

Como se puede extraer del ejercicio anterior, una lista de decisión puede dejar al intérprete sin respuesta ante ciertos ejemplos. Para evitar este problema, es habitual que el último valor de la lista de decisión tenga un valor por defecto, como así se hace en el primer ejemplo del ejercicio anterior (ver pág 117, primer párrafo [EML]).

Por último, algunos de los algoritmos que aparecen más adelante necesitarán conocer la función "match" asociada a cada algoritmo. Por tanto, construiremos una función que se lo facilite:

Ejercicio 2. ***[30']*** *Empezaremos por definir una variable global que mantenga esta información:*

```
(define *funciones-match* '((HGS . match-CL)(HTC . match-TC)...))
```

Implemente la función `(funcion-match <algoritmo>)` *que devuelve la función match correspondiente. Por ejemplo,*

```
# (eval `(,(funcion-match HGS) '((soleado)(*)) '(soleado si)))
 #t
```

1.2. Aprendizaje No Incremental de Listas de Decisión

Requisito: estudio de la sección 5.2 [EML].

Ejercicio 3. ***[1h]*** *A partir de la tabla 5-1 [EML] implemente la función* `(NSC0 <algoritmo><PSET><NSET><DNF>)`*, la cual será iniciada mediante* `(NSC0 <algoritmo>ejemplos+ ejemplos- ())` *desde la función* `(NSC <algoritmo>ejemplos)`*. La variable* `<algoritmo>` *debe sustituirse por una de las funciones procedentes del tema 2 ó tema 3. En particular, se estudiará como ejemplo la función* `HGS`*. Note que se debe devolver una lista de conceptos y no una lista de decisión.*

```
# (NSC HGS ejemplos)
(((soleado)(30 40)(10 20)(si)) ((soleado)(30)(10 20)(no)) ...)
```

Ejercicio 4. ***[1h]*** *Estudie la salida de la función* `NSC` *a partir de los datos "buen día para salir al campo" y los datos "agaricus-lepiota" e "ionosphere" procedentes de UCI mediante los algoritmos* `HGS`*,* `HTC`*,* `PCP` *y* `LMS`*. Comente las diferencias, por ejemplo ¿qué configuración devuelve listas de mayor tamaño? ¿cuáles considera que podrían ofrecer mayor precisión?.*

Hasta ahora, gran parte de los algoritmos estudiados, exceptuando los de tipo competitivo, sólo permitían la introducción directa de ejemplos de entrenamiento etiquetados por 2 tipos de clase; las representaciones compuestas admiten datos con múltiples clases.

Ejercicio 5. ***[1h]*** *A partir de la tabla 5-2 [EML], implemente la función* `(MSC0 <algoritmo>ejemplos)` *que admita un conjunto de ejemplos con clases múltiples y devuelva una* lista de decisión*. Note que* `MSC0` *debe implementarse a través de* `NSC`*. Finalmente, implemente la función* `(MSC <algoritmo>ejemplos)` *que devuelva la lista de decisión que devuelve* `MSC0` *junto con un valor por defecto; algo como lo siguiente:*

```
(append (MSC0 <algoritmo> ejemplos)
        (list
          `(match-CL ,(make-list (- (length (car ejemplos)) 1) '(*))
                     => ,(A0 ejemplos))))
```

Ejercicio 6. *[3h] Estudie la eficiencia de* `MSC` *mediante distintas configuraciones variando el algoritmo introducido como parámetro:* `HGS`*,* `HTC`*,* `PCP` *y* `LMS`*. Para ello, denomine cada nueva función como* `MSC-<subalgoritmo>` *(p.ej.* `MSC-HGS`*). Tras ello, compruebe la eficiencia de cada uno mediante ten-fold-crossvalidation y los datos "buen día para salir al campo", "agaricus-lepiota", "ionosphere", "poker" y "lymphography". ¿Qué configuración produce mejores resultados? ¿Por qué cree que es así?.*

Capítulo 2

Tema 6: Revisión y Extensión de Redes de Inferencia

En este tema se plantean algoritmos para la revisión y extensión de redes de conceptos básicos. Aquí, el autor de [EML] plantea los tres tipos de redes fundamentales: redes de conceptos lógicos, redes de conceptos umbral y redes de conceptos probabilísticos. Cada una de estas redes ha dado lugar a campos muy extensos en el aprendizaje automático, respectivamente: Programación Lógica Inductiva / Minería de Datos Multirelacional (Inductive Logic Programming (ILP) / Multirelational Data-Mining), Redes Neuronales / Reconocimiento de Patrones (Neural Networks / Pattern Recognition), y aprendizaje de Redes Bayesianas. Tal es su extensión, que muchos autores los consideran campos distintos.

Por todo ello, resulta muy interesante leer el tema 6 de [EML] (al menos la sección 6.1 y "Historical and bibliographical remarks"), ya que ofrece una visión global y conjunta de estos tres campos a la vez. No obstante, no se pedirá ningún ejercicio de este tema, ya que cada una de estas 3 partes requeriría —al menos— el tiempo de una nueva asignatura[1].

[1] De hecho, la asignatura Métodos Neuronales Bioinspirados sería una de ellas.

Capítulo 3

Tema 7: La Formación de Jerarquías de Conceptos

En este tema se reúnen las técnicas que organizan los conceptos del bloque 2 en *redes jerárquicas*, en contraposición a otras redes como las del tema anterior. Como cabría esperar, este tema es muy amplio y ha dado lugar a muchos algoritmos, entre ellos algunos de clustering o de agrupación de conceptos que entrarían en la clasificación de algoritmos de aprendizaje de tipo *no supervisado*. No obstante, aunque estudiaremos la estructura general de esta representación, por requisitos de espacio y tiempo, nos centraremos fundamentalmente en la construcción de su instanciación más popular: los árboles de decisión.

3.1. Representación de las Jerarquías de Conceptos (JC)

Requisito: Estudio de la sección 7.1 [EML]

Para describir los conceptos `JC` de manera genérica utilizaremos una definición recursiva; así, definimos una *jerarquía de conceptos* `JC` como:

```
JC=(<metodo-eleccion-de-rama>
    (<descriptor-de-rama> -> JC)
    (<descriptor-de-rama> -> JC)
    ...
   )

JC=(=> <clase>)

ó

JC=() ;clase desconocida
```

```
ó

JC=(=> <interprete> <concepto>)
```

Particularizando lo anterior al caso de árboles de decisión clásicos `adc`:

```
<metodo-eleccion-de-rama>=adc
<descriptor-de-rama>=<concepto-CL>

Así tenemos:

JC=(adc
    (<concepto-CL> -> JC)
    (<concepto-CL> -> JC)
    ...
   )

ó

JC=(=> <clase>)
```

Un ejemplo de estructura JC sería el siguiente:

```
(adc
  (((soleado)(*)) ->
   (adc
     (((*)(#-1/0 30)) -> (=> +))
     (((*)((30) #1/0)) -> (=> -))
  ))
  (((nublado)(*)) -> (=> -))
  (((lluvioso)(*)) ->
   (adc
     (((*)(#-1/0 10)) -> (=> -))
     (((*)((10) #1/0)) -> (=> +))
  ))
)
```

Durante la *interpretación* de una estructura JC de tipo `adc`, el acceso a una determinada rama se da cuando, para la instancia a considerar, ésta cumple el concepto-CL que inicia dicha rama. Es importante notar que las ramas tienen un orden de preferencia: en el caso de que dos ó más conceptos CL se cumplan, siempre se considerará prioritaria la rama que se encuentra más a la izquierda del árbol. En otras palabras, con `adc` la búsqueda de la clase se realizará mediante *primero el mejor* (best-first). Siguiendo esta misma

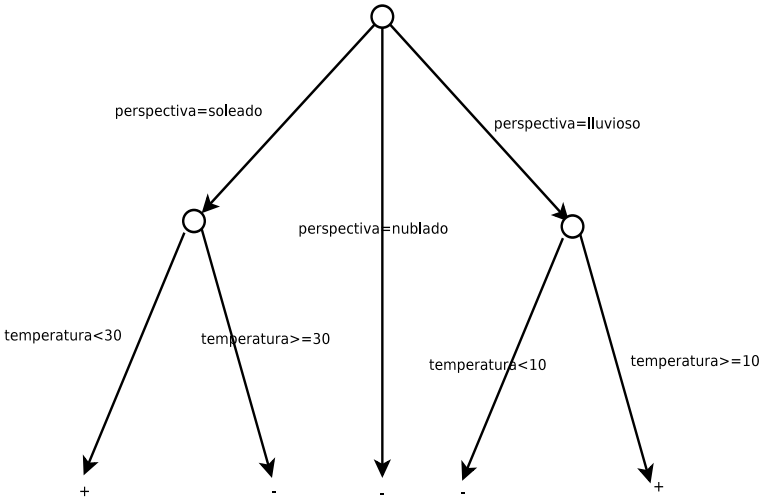
política, si ninguna rama cumple una condición válida, se volverá jerárquicamente al nodo superior. Note que con otros métodos de elección de rama la búsqueda puede realizarse de forma diferente (ver [EML] pag 189, 2º párrafo).

Ejercicio 7. *Construya el* `<metodo-eleccion-de-rama>` `adc` *como una función que acepta como primer parámetro un* `<ejemplo-sin-clase>` *y como segundo parámetro una lista con un número indefinido de ramas (listas) de una estructura JC como la del ejemplo anterior. Esta función debe devolver la* estructura JC *(la parte que aparece tras* `->`*) de una sola de las ramas: la* primera *en la que se cumpla* `(match-CL <concepto><ejemplo-sin-clase>)`*. Si dicha condición no se cumple en ninguna rama,* `adc` *debe devolver* `()`*. Por ejemplo,*

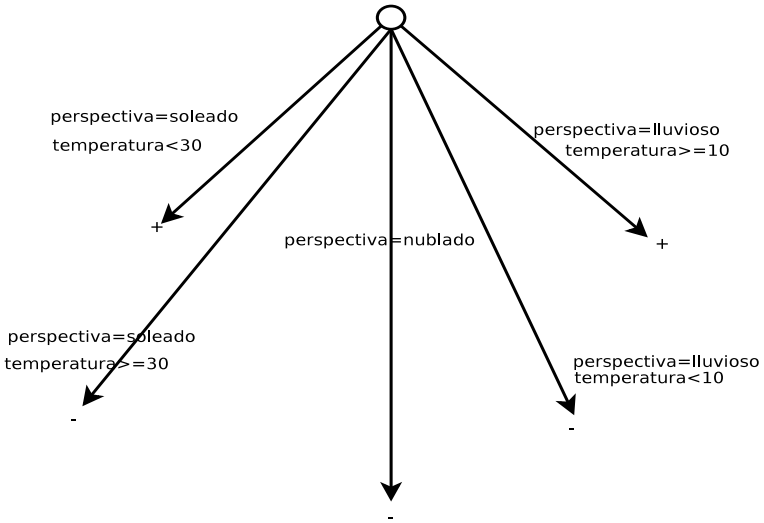
```
#(adc '(soleado 20)
 '((((soleado)(*)) ->
   (adc
     (((*)(#-1/0 30)) -> (=> +))
     (((*)((30) #1/0)) -> (=> -))
  ))
  (((nublado)(*)) -> (=> -))
  (((lluvioso)(*)) ->
   (adc
     (((*)(#-1/0 10)) -> (=> -))
     (((*)((10) #1/0)) -> (=> +))
  )))
)

>> (adc
     (((*)(#-1/0 30)) -> (=> +))
     (((*)((30) #1/0)) -> (=> -))
   )

#(adc '(soleado 20)
   '(
     (((*)(#-1/0 30)) -> (=> +))
     (((*)((30) #1/0)) -> (=> -))
    )
  )

>> (=> +)
```

La estructura JC que estamos estudiando en estos ejemplos es un caso especial dentro de los `adc`; es un *árbol de decisión univariado* (sección 7.1 [EML]). De hecho, la forma más habitual de representar este árbol es la

siguiente:



Ejercicio 8. *A partir del árbol anterior, genere una estructura JC equivalente, pero que se corresponda con un árbol de decisión* multivariado *(sección 7.1 [EML]). Como ayuda, observe el siguiente dibujo:*



Aunque las estructuras JC con el método `adc` son las más conocidas, existen otros casos en los que el acceso a las ramas del árbol está regulado por otro tipo de conceptos (p.ej. conceptos-UU, etc.).

Ejercicio 9. *Construya el* `<metodo-eleccion-de-rama> adg` *(árbol de decisión generalizado) de manera similar al método* `adc`*. No obstante, en este caso las ramas y estructuras JC contendrán como* `<descriptor-de-rama>` *una lista de dos elementos: los conceptos junto con su correspondiente función* `match` *(ver ejemplo más adelante). Así, esta función debe devolver la*

estructura JC *(la parte que aparece tras ->) de la* primera *rama en la que se cumpla (<funcion-match><concepto><ejemplo-sin-clase>). Si dicha condición no se cumple en ninguna rama,* `adg` *debe devolver (). Por ejemplo,*

```
#(adg '(soleado 20)
 '(
  ((match-CL ((soleado)(*))) ->
   (adg
     ((match-LUU (,(car ejemplos)(1 2 -10))) -> (=> +))
     ((match-LUU (,(car ejemplos)(1 2 10))) -> (=> -))
   ))
  ((match-CL ((nublado)(*))) -> (=> -))
  )
)

>> (adg
     ((match-LUU ((...)(1 2 -10))) -> (=> +))
     ((match-LUU ((...)(1 2 10))) -> (=> -))
   )

#(adg '(soleado 20)
   `(((match-LUU (,(car ejemplos)(1 2 -10))) -> (=> +))
     ((match-LUU (,(car ejemplos)(1 2 10))) -> (=> -)))
  )

>> (=> -)
```

Ejercicio 10. *Construya el intérprete (JCi <concepto-JC><ejemplo-sin-clase>). Note que este intérprete debe ser lo suficientemente general como para que acepte cualquier tipo de <metodo-eleccion-de-rama> dentro de un <concepto-JC>. En caso de que el intérprete no encuentre ninguna rama válida y no pudiese responder, se devolverá el ejemplo sin clase. Por ejemplo:*

```
#(JCi '(adg
         ((match-CL ((soleado)(*))) ->
          (adg
            ((match-CL ((*)(#-1/0 30))) -> (=> +))
            ((match-CL ((*)((30) #1/0))) -> (=> -))
           ))
          ((match-CL ((nublado)(*))) -> (=> -))
          ((match-CL ((lluvioso)(*))) ->
           (adg
              ((match-CL ((*)(#-1/0 10))) -> (=> -))
              ((match-CL ((*)((10) #1/0))) => (=> +))
```

```
            )
          )
        )

   '(soleado 20)
 )

>> (soleado 20 +)

#(JCi '(adc
        (((soleado)(*)) ->
         (adc
           (((*)(#-1/0 30)) -> (=> +))
           (((*)((30) #1/0)) -> (=> -))
          ))
        (((nublado)(*)) -> (=> -))
        (((lluvioso)(*)) ->
         (adc
           (((*)(#-1/0 10)) -> (=> -))
           (((*)((10) #1/0)) -> (=> +))
          ))
        )

      '(lluvioso 50)
)

>> (lluvioso 50 +)

#(JCi '(adc
        (((soleado)(*)) ->
         (adc
           (((*)(#-1/0 30)) -> (=> +))
           (((*)((30) #1/0)) -> (=> -))
          ))
        (((nublado)(*)) -> (=> -))
        (((lluvioso)(*)) ->
         (adc
           (((*)(#-1/0 10)) -> (=> -))
           (((*)((10) #1/0)) -> ())
          ))
        )

      '(lluvioso 50)
)
```

```
>> (lluvioso 50)
```

Observe que, al igual que con las listas de decisión, en el caso más general debería existir un valor por defecto —al menos en el nodo superior o raíz— para no dejar nunca al intérprete sin una respuesta válida. Sin embargo, en el caso particular de árboles de decisión univariados clásicos veremos que las ramas cubrirán todas y cada una de las posibilidades.

3.2. Formación no incremental de Jerarquías de Conceptos mediante el paradigma divide-y-vencerás

Requisitos: Estudio de las secciones 7.2.1 y 7.2.2 [EML]

Antes de comenzar los ejercicios, debemos definir ciertos conceptos nuevos. Así, definimos *discriminante* como una estructura de datos basada en cierto atributo de un dataset que permite dividir dicho dataset en varias partes en función de dicho atributo.

En concreto, un discriminante basado en un atributo nominal se representará del siguiente modo: `(<nombre atributo><posición><lista de valores posibles>)`. Por ejemplo, `(perspectiva 0 (soleado lluvioso nublado))`. En el caso de un atributo numérico, éste se representará del siguiente modo: `(<nombre atributo><posición>numerico <umbral de discriminación>)`. Por ejemplo, `(temperatura 1 numerico 25)`.

Ejercicio 11. *Construya la función* `(dividir-ejemplos <discriminante><ejemplos (sin metadatos)>)` *que devuelva lo siguiente:*

En el caso de un discriminante nominal, la función devolverá una lista con varios conjuntos de ejemplos: uno por cada valor posible del atributo nominal correspondiente. El primer elemento de cada nuevo conjunto/lista de ejemplos comenzará por el valor (del atributo) con el que han sido discriminados. Por ejemplo:

```
 (dividir-ejemplos
   '(perspectiva 0 (soleado lluvioso))
   '((soleado 10 -)(soleado 25 +)(lluvioso 30 -)))

>> (
    (soleado (soleado 10 -)(soleado 25 +))
    (lluvioso (lluvioso 30 -))
   )
```

En el caso de un discriminante numérico, la función devolverá una lista con dos *conjuntos de ejemplos: uno en el que se cumpla que los valores numéricos sean superiores y otro en que los valores sean inferiores al* umbral de discriminación. *El primer elemento de cada nuevo conjunto/lista de ejemplos comenzará por* `(>= <umbral>)` *o* `(< <umbral>)` *según haya sido discriminado. Por ejemplo:*

```
(dividir-ejemplos
  '(temperatura 1 numerico 25)
  '((soleado 10 -)(soleado 25 +)(lluvioso 30 -)))

>> (
    ((>= 25) (soleado 25 +)(lluvioso 30 -))
    ((< 25) (soleado 10 -))
   )
```

Ejercicio 12. *Construya la función* `(generar-discriminantes <metadatos><ejemplos>)` *que devuelva un* discriminante nominal *por cada atributo nominal y un* discriminante numérico *por cada atributo y valor numérico diferente de todo el conjunto de ejemplos de entrenamiento. Por ejemplo,*

```
(generar-discriminantes
  '((perspectiva (soleado lluvioso))
    (temperatura numerico))
  '((soleado 10 -)(soleado 25 +)(lluvioso 30 -)))

>> ((perspectiva 0 (soleado lluvioso))
    (temperatura 1 numerico 10)
    (temperatura 1 numerico 25)
    (temperatura 1 numerico 30))
```

Ejercicio 13. *A partir del ejercicio 11 y el algoritmo* `A0` *construya la función* `(capacidad-de-discriminacion1 <discriminante><ejemplos-disponibles>)` *mediante la fórmula que aparece al final de la página 194 [EML]. Por ejemplo,*

```
(capacidad-de-discriminacion1
  '(perspectiva 0 (soleado lluvioso))
  '((soleado 10 -)(soleado 25 +)(lluvioso 30 -)))

>> (1+1)/3=2/3

(capacidad-de-discriminacion1
  '(temperatura 1 numerico 25)
  '((soleado 10 -)(soleado 25 +)(lluvioso 30 -)))
```

```
>> (1+1)/3=2/3

 (capacidad-de-discriminacion1
   '(temperatura 1 numerico 10)
   '((soleado 10 -)(soleado 25 +)(lluvioso 30 -)))

>> (2+0)/3=2/3
```

Tras ello, defina la siguiente función:

```
(define capacidad-de-discriminacion capacidad-de-discriminacion1)
```

Ejercicio 14. ***[1h]*** *Construya la función* ***(mayor-discriminante <lista de discriminantes><ejemplos-disponibles>)*** *que devuelva una lista en la que el primer elemento sea el discriminante que mayor valor obtuvo en la función* ***capacidad-de-discriminacion****. El resto de la lista serán el resto de discriminantes.*

```
 (mayor-discriminante
  '((perspectiva 0 (soleado lluvioso))
    (temperatura 1 numerico 10)
    (temperatura 1 numerico 25)
    (temperatura 1 numerico 30))

  '((soleado 10 -)(soleado 25 +)(lluvioso 30 -)))
 )

>> '((perspectiva 0 (soleado lluvioso))
   (temperatura 1 numerico 10)
   (temperatura 1 numerico 25)
   (temperatura 1 numerico 30))
; en este caso todos los atributos poseen
; la misma capacidad de discriminación
```

Ejercicio 15. ***[4h]*** *Construya la función* ***(DDT0 <lista de discriminantes disponibles><ejemplos-disponibles>)*** *inspirándose en la tabla 7-2 [EML], pero tratando de construir una estructura JC como la estudiada en la sección anterior. Más concretamente, un árbol de decisión clásico* **adc***. Considere que lo que es un atributo en [EML] es un discriminante en estas prácticas [P3]. En particular, como ejemplo de una llamada a DDT0 tendríamos lo siguiente:*

```
(DDT0
   '((perspectiva 0 (soleado lluvioso nublado))
     (temperatura 1 numerico 10)
```

```
     (temperatura 1 numerico 25)
     (temperatura 1 numerico 30))

   '((soleado 10 +)(lluvioso 25 -)(nublado 30 +)))

;Imaginando que se ha escogido "perspectiva" como
;más discriminante, entonces lo que devolvería esta llamada
;sería algo como lo siguiente, recursivamente...
`(adc
  (((soleado)(*)) ->
   ,(DDT0 '((temperatura 1 numerico 10)
            (temperatura 1 numerico 25)
            (temperatura 1 numerico 30))
       (filter (lambda(ej) (match-CL '((soleado)(*)) (drop-right ej 1)))
               ejemplos-disponibles)))
  (((lluvioso)(*)) -> ...)
  (((nublado)(*)) -> ...)
 )
```

Finalmente, construya la función `(DDT ejemplos)` *que utiliza* `DDT0` *como subrutina.*

Además de la ya estudiada, existen otras métricas para evaluar la capacidad de discriminación en un atributo para un árbol de decisión univariado clásico. Quizá la más extendida y popular es la denominada *ganancia de información*, la cual se emplea en los muy conocidos algoritmos de aprendizaje ID3, c4.5 y c5.0. Dicha métrica está basada en la entropía que posee un conjunto de ejemplos de entrenamiento.

La entropía de un conjunto de ejemplos de entrenamiento E previamente etiquetados en un conjunto de *clases* se define de la siguiente manera:

$$entropia(E) = \sum_{c \in clases} -\frac{|E_c|}{|E|} \times \log_2 \frac{|E_c|}{|E|} \tag{3.1}$$

La ganancia de información o reducción de entropía que se produce cuando dichos ejemplos se separan mediante un determinado atributo A se mide mediante la fórmula de *ganancia* siguiente (ver detalles en páginas 55-59 [ML]):

$$ganancia(E, A) = entropia(E) - \sum_{v \in valores\_de(A)} \frac{|E_v|}{|E|} \times entropia(E_v) \tag{3.2}$$

Ejercicio 16. ***[2h]*** *Construya la función* `(capacidad-de-discriminacion2 <discriminante><ejemplos-disponibles>)` *considerando que ésta viene determinada por la fórmula 3.2 anterior.*

Ejercicio 17. *[2h] Adjunte y compare los árboles de decisión generados por* `DDT` *a partir de las métricas de los dos ejercicios 13 y 16 para los ejemplos "buen día para salir al campo". ¿Qué árbol considera que es más complejo? . Tras ello, compare (utilice la función* `cross-validation` *con* `folds` *a 10) los dos algoritmos anteriores en los datos "agaricus-lepiota" y "lymphography".*

3.2.1. Transformación de estructuras JC tipo `adc` en estructuras DL

Uno de los atractivos de los árboles de decisión clásicos `adc` es la posibilidad de transformarlos en un conjunto de reglas, o —con más precisión— en una lista de decisión. De esta forma, sus defensores suelen clamar que el conocimiento extraído de un árbol de decisión es *explícito*, ya que puede ser interpretado directamente por un ser humano (en forma de reglas explícitas), en contraposición al conocimiento extraído por otros (habitualmente aquellos procedentes del paradigma conexionista, p.ej. `PCP`/`LMS`) cuya interpretación resulta lejos de ser trivial.

Requisito: estudio de la sección 7.6 [EML]

Ejercicio 18. *[1h] Construya la función* `(union-CL <concepto-CL1><concepto-CL2>)` *que devuelve el* `<concepto-CL1>` *pero sustituyendo cada test* `(*)` *por aquel que aparezca en* `<concepto-CL2>`*. Por ejemplo,*

```
#(union-CL '((soleado)(*)) '((*)(10 30)))
>> ((soleado)(10 30))

#(union-CL '((*)(*)) '((*)(10 30)))
>> ((*)(10 30))
```

Ejercicio 19. *[3h] Mediante la función anterior, construya el algoritmo* `(DTL <arbol-decision-clasico>)` *a partir de la tabla 7-8 [EML] que admita un árbol de decisión clásico* `adc` *y que devuelva una lista de decisión en la forma en que aparecen en la sección 1.1. Por ejemplo,*

```
# (DTL '(adc
        (((soleado)(*)) ->
         (adc
           (((*)(#-1/0 30)) -> (=> +))
           (((*)((30) #1/0)) -> (=> -))
          ))
        (((nublado)(*)) -> (=> -))
        (((lluvioso)(*)) ->
         (adc
           (((*)(#-1/0 10)) -> (=> -))
           (((*)((10) #1/0)) -> (=> +))
          ))
```

```
        ))

>> ((match-CL ((soleado)(#-1/0 30)) => +)
    (match-CL ((soleado)((30) #1/0)) => -)
    (match-CL ((nublado)(*)) => -)
    (match-CL ((lluvioso)(#-1/0 10)) => -)
    (match-CL ((lluvioso)((10) #1/0)) => +))
;Nota: el orden de cada regla en la lista de decisión
; viene marcado por el propio algoritmo (tabla 7-8)
```

Termine definiendo el algoritmo `(DTL-DDT ejemplos)` *mediante el siguiente código:*

```
(define (DTL-DDT ejemplos)
  (DTL (DDT ejemplos)))
```

Ejercicio 20. ***[2h]*** *Realice las mismas pruebas que en el ejercicio 17 pero esta vez comparando listas de decisión en lugar de árboles. ¿Qué lista de decisión contiene más términos?. Compare (utilice la función* `cross-validation` *con* `folds` *a 10) los algoritmos* `DDT` *con sus parientes* `DTL-DDT` *mediante los datos "agaricus-lepiota" y "lymphography" ¿existen diferencias de precisión? ¿por qué cree que es así?.*

Ejercicio 21. ***[3h]*** *Para finalizar este bloque y cerrar la asignatura, compare en una tabla (utilice la función* `stratified-cross-validation` *con* `folds` *a 10) la eficiencia de los algoritmos de* conceptos simples*: A0, A1, HGS, HCT, PCP, LMS, IB y NB y los algoritmos de* conceptos compuestos*: MSC-HGS, MSC-HTC, MSC-PCP, MSC-LMS, DDT(1), DDT(2), DTL-DDT(1) y DTL-DDT(2) sobre todos los dominios tratados en estas prácticas: "buen día para salir al campo", "agaricus-lepiota", "ionosphere", "poker" y "lymphography" ¿Existe algún algoritmo que destaque en especialmente en cada uno de los dominios? Escriba, a modo de conclusión de esta asignatura, los resultados obtenidos en este ejercicio. Entre otras cosas, ¿considera que existen diferencias significativas entre los conceptos de representación simple y los conceptos de representación compuesta?.*